Université Paris VII — Denis Diderot
UFR d'Informatique

École doctorale de Sciences mathématiques de Paris Centre

DOCTORAT
Spécialité : **Informatique**

Gilles PESKINE

Types abstraits dans les systèmes répartis

Directeur : Jean-Jacques LÉVY

Thèse soutenue le 12 juin 2008
devant le jury composé de

M.	Jean-Jacques LÉVY,	directeur
M.	Robert HARPER,	rapporteur
M.	Jacques GARRIGUE,	rapporteur
M.	James LEIFER,	examinateur
M.	Didier RÉMY,	examinateur
M.	Peter SEWELL,	examinateur
M.	Roberto DI COSMO,	Président du jury

Types abstraits dans les systèmes répartis

Résumé

Soit un réseau de machines exécutant des programmes ML qui s'échangent des données. Comment peut-on garantir le typage des communications en présence de types abstraits ? Une approche sûre est de traiter des types abstraits définis sur des machines différentes comme distincts. En pratique, cela est bien trop restrictif, en particulier lorsqu'un type abstrait sert à garantir un invariant.

Les principales contributions de cette thèse sont les suivantes : je définis une notion d'empreinte de type abstrait : deux types abstraits sont réputés compatibles s'ils ont la même empreinte ; je propose une sémantique opérationnelle pour un système de module qui préserve les types, y compris abstraits ; je décris un système de module nouveau, mieux adapté aux applications réparties.

L'empreinte d'un type abstrait doit refléter sa sémantique attendue, qui n'est pas toujours apparente au vu du code source. Une approximation sûre est de donner la même empreinte à deux modules lorsqu'ils ont le même code. Des modules composés sont compatibles si leurs composants le sont.

Les sémantiques opérationnelles habituelles des modules de ML effacent les barrières d'abstraction. J'utilise des crochets colorés qui matérialisent ces barrières et évitent la perte d'information. J'étudie deux calculs ainsi équipés, un lambda-calcul simplement typé et un système de module expressif pour ML.

J'exprime des équivalences de modules de signature arbitraire, pas seulement de types, au moyen de signatures singleton. Un système d'effets simple préserve le typage statique et permet de distinguer des foncteurs applicatifs et génératifs. Je discute de formes statiques et dynamiques du scellage de modules.

Mots clés : langages de programmation, lambda-calcul, ML, théorie des types, types abstraits, modules, foncteurs, générativité, sortes singletons, sérialisation, programmation distribuée, programmation répartie

Abstract Types in Distributed Systems

Abstract

Consider a network of nodes running ML programs that exchange data. How can data which has an abstract type on one node be accepted on another node? A safe approach is to treat abstract types as distinct whenever they are defined on different nodes. However this is too restrictive in practice, for example in the common case where an abstract type enforces a semantic invariant.

The main contributions of this thesis are threefold: I define a notion of hash of an abstract type, whereby abstract types that have the same hash are deemed compatible; I give an operational semantics for a module system that preserves types, including abstract types; I also propose a new, more general module system that is well-suited to distributed applications.

The hash of an abstract type must reflect its intended semantics, which is often not apparent from the program's code. In practice, two modules have the same hash if they have the same code. Compound modules are compatible when they are built from compatible components.

Existing operational semantics for ML modules lose information as they erase abstraction boundaries. I use coloured brackets to track the visibility of abstract types. I study two calculi equipped with brackets, a simply-typed lambda-calculus and a rich ML module calculus.

I use singleton signatures to keep track of not only type but also code sharing, so that module equivalence is defined at arbitrary signatures. A simple effect system limits type constraint to a statically checkable fragment, while permitting both applicative and generative functors. I discuss static and dynamic forms of module sealing.

Keywords: programming languages, lambda calculus, ML, type theory, abstract types, modules, functors, generativity, singleton kinds, marshalling, serialisation, distributed programming

Les travaux décrits dans cette thèse ont été réalisés au sein du projet MOSCOVA de l'INRIA Rocquencourt.

Remerciements

Je tiens à remercier avant tout Jean-Jacques Lévy, directeur de thèse hors pair, tant pour ses qualités scientifiques qu'humaines. Je salue particulièrement sa culture et sa patience, sans lesquelles cette thèse n'aurait jamais pu avoir lieu, et surtout n'aurait jamais pu être achevée.

Je salue également l'environnement chaleureux et stimulant du projet MOSCOVA et de ses projets frères, le « bâtiment 8 » virtuel. Ses habitants sont trop nombreux pour les énumérer tous : la qualité du groupe tient à celle de tous ses membres.

Je remercie Peter Sewell pour sa suggestion d'utiliser les « crochets colorés » pour garder une trace des barrières d'abstraction. C'est cette idée qui est à l'origine de ma thèse.

Sur cette base, James Leifer et moi-même avons élaboré le système HAT, qui est décrit au chapitre III de la présente thèse. Le système HAT est le résultat d'une collaboration qui va bien au-delà d'un simple encadrement.

Outre Jean-Jacques Lévy, James Leifer et Peter Sewell, je remercie toutes les personnes qui ont bien voulu consacrer du temps à répondre à mes interrogations ou à suggérer des pistes de recherche. À l'INRIA, je remercie notamment Georges Gonthier, Xavier Leroy, Didier Rémy et Francesco Zappa Nardelli. Robert Harper, en plus d'être un des géants sur l'épaule duquel je suis monté, et d'avoir bien voulu rapporter cette thèse, m'a grâce à son attention au détail permis de débusquer une erreur dans le traitement des crochets colorés en présence de foncteurs génératifs.

Cette thèse est trop longtemps restée « presque » achevée. Je remercie Jean-Jacques Lévy et Peter Sewell de m'avoir patiemment supporté durant cette période.

Je remercie également Emmanuelle Grousset, Sylvie Loubressac et Michèle Wasse pour leur soutien administratif largement au-delà de simples obligations professionnelles.

Durant mon séjour à l'INRIA Rocquencourt, j'ai été financé par l'École Normale Supérieure, puis une allocation de recherche du Ministère de l'Éducation nationale, de l'Enseignement supérieur et de la Recherche par l'intermédiaire de l'Université Paris VII — Denis Diderot, puis directement par l'INRIA Rocquencourt.

I acknowledge the support of EPSRC grants GR/T11715 and EP/C510712.

Table des matières

Introduction **15**

I Préliminaires **19**

I.1 Abstraction 19
I.1.1 Perspective historique 19
I.1.1.1 Procédures 19
I.1.1.2 Modularité 20
I.1.1.3 Classes et grappes 21
I.1.1.4 Types abstraits 21
I.1.1.5 Interface d'un type abstrait 22
I.1.1.6 Spécifications 23
I.1.2 Usages des types abstraits 23
I.1.2.1 Invariants 23
I.1.2.2 Confidentialité 25
I.1.2.3 Estampillage 25
I.1.2.4 Contrôle de ressources 27
I.1.2.5 Terminologie : types de données algébriques et types abstraits . . . 28
I.1.2.6 Bilan 28
I.2 Systèmes de modules pour ML 29
I.2.1 Anatomie d'un système de modules 29
I.2.1.1 Introduction 29
I.2.1.2 Modules de base 30
I.2.1.3 Familles de modules 30
I.2.1.4 Constructions avancées 32
I.2.1.5 Calculs de signatures 33
I.2.1.6 Modules et types abstraits 34
I.2.2 Partage de types 35
I.2.2.1 Sommes fortes, sommes faibles 35
I.2.2.2 Sommes translucentes et types manifestes 36
I.2.2.3 Foncteurs applicatifs 37
I.2.2.4 Cohabitation de foncteurs applicatifs et génératifs 40
I.2.2.5 Modules de première classe 40
I.2.2.6 Modules arguments et problème de l'évitement 42
I.3 Typage statique et dynamique 43
I.3.1 Du typage statique 43
I.3.1.1 Séparation des phases 43
I.3.1.2 Les limites de la paramétricité 43
I.3.1.3 Les limites du typage statique 44
I.3.2 Analyse et vérification de type à l'exécution 45
I.3.2.1 Type dynamique 45
I.3.2.2 Dynamiques polymorphes 46
I.3.2.3 Dynamiques et existentiels 48
I.3.2.4 Typage dynamique et abstraction 49

II Empreintes pour les types abstraits distribués 51
II.1 Introduction . . . 51
II.2 Étude de cas . . . 53
II.2.1 Types concrets . . . 54
II.2.2 Types abstraits : respect des invariants . . . 54
II.2.2.1 Modules et types abstraits . . . 54
II.2.2.2 Compatibilité inter-machines . . . 55
II.2.2.3 Concret n'est pas abstrait . . . 56
II.2.2.4 Abstrait n'est pas concret . . . 56
II.2.2.5 Abstrait n'est pas abstrait . . . 57
II.2.2.6 Un problème indécidable . . . 59
II.2.3 De l'importance, ou non, des noms . . . 59
II.2.3.1 Noms des variables locales . . . 59
II.2.3.2 Noms des points d'entrée . . . 60
II.2.3.3 Nom(s) du module . . . 61
II.3 Empreintes . . . 62
II.3.1 Empreintes de modules et désérialisation . . . 62
II.3.1.1 Concept d'empreinte . . . 62
II.3.1.2 Définition . . . 63
II.3.1.3 Récapitulation des exemples . . . 64
II.3.1.4 Empreinte et signature . . . 64
II.3.2 Dépendances . . . 65
II.3.2.1 Un exemple . . . 65
II.3.2.2 Définition . . . 66
II.3.2.3 Dépendance faible . . . 67
II.3.3 Implémentation . . . 67
II.3.3.1 Réification . . . 67
II.3.3.2 Représentation compacte : compression . . . 69
II.3.3.3 Représentation compacte : empreintes cryptographiques . . . 69
II.4 Autres utilisations des empreintes . . . 70
II.4.1 Analyse dynamique de type . . . 70
II.4.1.1 Introduction . . . 70
II.4.1.2 Du test au filtrage . . . 70
II.4.1.3 Analyse de type . . . 71
II.4.1.4 Modules de première classe . . . 72
II.4.1.5 Exemple : affichage du type . . . 73
II.4.1.6 Analyse de signature . . . 73
II.4.2 Négociation de code . . . 73
II.4.2.1 Problématique . . . 73
II.4.2.2 Stratégies de propagation de code . . . 74
II.5 Foncteurs . . . 75
II.5.1 Problématique . . . 75
II.5.1.1 Introduction . . . 75
II.5.1.2 Stratégies . . . 76
II.5.2 Exemples . . . 77
II.5.2.1 Empreinte d'un module non scellé . . . 77
II.5.2.2 Paramétrisation . . . 78
II.5.2.3 Compositionnalité . . . 79
II.5.2.4 Paramètre inutile . . . 80
II.5.2.5 Foncteurs scellés . . . 81
II.5.2.6 Calculs sur les modules . . . 82
II.5.3 Sémantiques . . . 82
II.5.3.1 Empreintes de valeurs . . . 82
II.5.3.2 Empreintes systématiques . . . 83

II.5.3.3 Une sémantique hybride . . . 85
II.6 Compatibilité d'empreintes . . . 85
II.6.1 Générativité . . . 85
II.6.1.1 Introduction . . . 85
II.6.1.2 Empreintes singularisées . . . 86
II.6.1.3 Choix du type d'empreinte . . . 87
II.6.1.4 Appel distant . . . 87
II.6.1.5 Foncteurs génératifs . . . 88
II.6.1.6 Empreintes de valeurs . . . 89
II.6.1.7 Empreintes de code . . . 90
II.6.2 Respect de l'abstraction . . . 90
II.6.2.1 Compatibilité d'empreintes et rupture d'abstraction . . . 90
II.6.2.2 Paramétricité . . . 91
II.6.3 Conversions entre empreintes . . . 92
II.6.3.1 Problématique . . . 92
II.6.3.2 Sous-empreinte vérifiée . . . 92
II.6.3.3 Sous-empreinte affirmée . . . 93

III HAT : Un premier calcul d'empreintes **95**
III.1 Concrétiser l'abstraction : les crochets colorés . . . 95
III.1.1 Introduction . . . 95
III.1.1.1 Valeurs de types abstraits . . . 95
III.1.1.2 Scellage . . . 96
III.1.1.3 Crochets colorés . . . 97
III.1.2 Crochets et évaluation . . . 98
III.1.2.1 Types de base . . . 98
III.1.2.2 Poussée de crochets . . . 99
III.1.2.3 Couleur ambiante . . . 99
III.2 Le langage HAT . . . 100
III.2.1 Introduction . . . 100
III.2.2 Syntaxe . . . 101
III.2.2.1 Syntaxe du langage . . . 101
III.2.2.2 Métathéorie . . . 102
III.2.3 Typage . . . 103
III.2.3.1 $\zeta \notin \mathbf{dom}\,\Gamma$ Absence de conflit de variables . . . 104
III.2.3.2 $\vdash \mathsf{c}\,\mathsf{ok}$ Correction d'une couleur . . . 104
III.2.3.3 $\Gamma \vdash_{\mathsf{c}} \mathsf{ok}$ Correction de l'environnement . . . 104
III.2.3.4 $\Gamma \vdash_{\mathsf{c}} \mathsf{K}\,\mathsf{ok}$ Correction d'une sorte . . . 104
III.2.3.5 $\Gamma \vdash_{\mathsf{c}} \mathsf{K} \equiv \mathsf{K}'$ Équivalence de sortes . . . 104
III.2.3.6 $\Gamma \vdash_{\mathsf{c}} \mathsf{K} <: \mathsf{K}'$ Sous-sortage . . . 104
III.2.3.7 $\Gamma \vdash_{\mathsf{c}} \mathsf{T} : \mathsf{K}$ Correction d'un type . . . 105
III.2.3.8 $\Gamma \vdash_{\mathsf{c}} \mathsf{T} \equiv \mathsf{T}'$ Équivalence de types . . . 105
III.2.3.9 $\Gamma \vdash_{\mathsf{c}} \mathsf{S}\,\mathsf{ok}$ Correction d'une signature . . . 106
III.2.3.10 $\Gamma \vdash_{\mathsf{c}} \mathsf{S} <: \mathsf{S}'$ Sous-signaturage . . . 106
III.2.3.11 $\Gamma \vdash_{\mathsf{c}} \mathsf{E} : \mathsf{T}$ Type d'une expression . . . 106
III.2.3.12 $\Gamma \vdash_{\mathsf{c}} \mathsf{M} : \mathsf{S}$ Signature d'une structure . . . 107
III.2.3.13 $\Gamma \vdash_{\mathsf{c}} \mathsf{U} : \mathsf{S}$ Signature d'un nom de module . . . 107
III.2.3.14 $\Gamma \vdash_{\mathsf{c}} \mathsf{L} : \mathsf{T}$ Correction d'une machine . . . 108
III.2.3.15 $\vdash \mathsf{N}\,\mathsf{ok}$ Correction d'un réseau . . . 108
III.2.4 Compilation . . . 108
III.2.4.1 $\mathsf{L} \Rightarrow \mathsf{L}'$ Compilation sur une machine . . . 108
III.2.5 Exécution . . . 108
III.2.5.1 $\mathsf{E} \longrightarrow_{\mathsf{c}} \mathsf{E}'$ Réduction des expressions . . . 109
III.2.5.2 $\mathsf{N} \longrightarrow \mathsf{N}'$; $\mathsf{N} \equiv \mathsf{N}'$ Réduction des réseaux . . . 110
III.2.6 Un exemple . . . 111

III.2.7 Résultats . . . 113
III.2.7.1 Correction de l'exécution . . . 113
III.2.7.2 Correction de la compilation . . . 114
III.2.7.3 Décidabilité du typage . . . 114
III.2.7.4 Effacement des crochets colorés . . . 114
III.2.7.5 Correspondance entre le typage statique et le typage dynamique . . 115

IV TOPHAT : un calcul de modules adapté aux environnements répartis 117
IV.1 Introduction . . . 117
IV.2 Un calcul de modules [$\mathcal{B}$] . . . 118
IV.2.1 Fondements . . . 118
IV.2.1.1 Structures . . . 118
IV.2.1.2 Signatures de structures . . . 119
IV.2.1.3 Foncteurs . . . 121
IV.2.1.4 Liaisons locales . . . 121
IV.2.2 Du langage de base . . . 122
IV.2.2.1 Exigences . . . 122
IV.2.2.2 Un langage unique . . . 122
IV.2.3 Présentation formelle du noyau . . . 123
IV.2.3.1 Syntaxe . . . 123
IV.2.3.2 Variables . . . 123
IV.2.3.3 Environnements . . . 124
IV.2.4 Typage . . . 124
IV.2.4.1 Introduction . . . 124
IV.2.4.2 $\Gamma \vdash \mathsf{ok}$ Correction de l'environnement . . . 125
IV.2.4.3 $\Gamma \vdash \mathsf{T}\,\mathsf{ok}$ Correction des types . . . 125
IV.2.4.4 $\Gamma \vdash \mathsf{E} : \mathsf{T}$ Typage des expressions . . . 125
IV.2.4.5 $\langle \mathsf{T} \rangle$, $\mathsf{Typ}\,\mathsf{E}$ Champs types . . . 126
IV.2.5 Exécution . . . 126
IV.2.5.1 $\mathsf{E} \longrightarrow \mathsf{E}'$ Réduction des expressions . . . 126
IV.3 Singletons [$\mathcal{S}$] . . . 127
IV.3.1 Motivation . . . 127
IV.3.1.1 Types abstraits, types concrets . . . 127
IV.3.1.2 Partage de types . . . 129
IV.3.1.3 Singletons de valeurs . . . 131
IV.3.1.4 Singletons d'ordre supérieur . . . 132
IV.3.1.5 Un exemple pratique . . . 132
IV.3.2 Propriétés . . . 134
IV.3.2.1 Nature syntaxique . . . 134
IV.3.2.2 Sémantique de l'appartenance au singleton . . . 134
IV.3.2.3 Correction d'un singleton . . . 135
IV.3.3 Règles de typage . . . 136
IV.3.3.1 $\Gamma \vdash \mathsf{T} <: \mathsf{T}'$; ... Sous-typage . . . 136
IV.3.3.2 $\mathsf{S}(\mathsf{E})$ Singletons . . . 137
IV.3.3.3 $\Gamma \vdash \mathsf{E} : \mathsf{T}$; $\Gamma \vdash \mathsf{T}_1 <: \mathsf{T}_2$ Typage des expressions . . . 137
IV.3.3.4 $\Gamma \vdash \mathsf{T} \equiv \mathsf{T}'$; $\Gamma \vdash \mathsf{E} \equiv \mathsf{E}'$ Équivalences de convertibilité . . . 138
IV.3.3.5 $\Gamma \vdash \mathsf{T} \longrightarrow \mathsf{T}'$ Conversion des types . . . 138
IV.3.3.6 $\Gamma \vdash \mathsf{E} \longrightarrow \mathsf{E}'$ Conversion des expressions . . . 139
IV.3.3.7 Extensionnalité . . . 140
IV.4 Scellage [$\mathcal{E}$] . . . 140
IV.4.1 Scellage . . . 140
IV.4.1.1 Types concrets inconnus . . . 140
IV.4.1.2 Types existentiels . . . 141
IV.4.1.3 Scellage . . . 142
IV.4.1.4 L'effet du scellage . . . 144

IV.4.1.5 Comparaison entre types existentiels et scellage 144
IV.4.2 Un système d'effets 145
IV.4.2.1 Introduction 145
IV.4.2.2 Pureté 147
IV.4.2.3 Projetabilité, séparabilité et comparabilité 148
IV.4.3 Présentation formelle 149
IV.4.3.1 Syntaxe 149
IV.4.3.2 $\mathsf{E} \longrightarrow \mathsf{E}'$ Exécution 150
IV.4.3.3 $\Gamma \vdash \ldots$ Typage : correction, équivalences, sous-typage 150
IV.4.3.4 $\Gamma \vdash \mathsf{E} :^{\gamma} \mathsf{T}$ Typage des expressions 152
IV.4.4 Applicativité 153
IV.4.4.1 Foncteurs applicatifs 153
IV.4.4.2 Scellage statique : formalisation [$\mathcal{W}$] 154
IV.4.4.3 Équivalences en présence de scellage statique 156
IV.4.4.4 Autres formes de scellage 157
IV.4.4.5 Encodages mutuels des scellages statique et dynamique 158
IV.4.4.6 Observations sur l'applicativité par scellage de foncteur 159
IV.5 Couleurs et crochets [$\mathcal{C}$] 160
IV.5.1 Empreintes 160
IV.5.1.1 Génération d'apax 160
IV.5.1.2 Lexiques 161
IV.5.1.3 Du scellage aux crochets 162
IV.5.1.4 Types abstraits 162
IV.5.1.5 Renforcement 163
IV.5.2 Couleurs 165
IV.5.2.1 Colorisation 165
IV.5.2.2 Sémantique d'un type et dépendances d'un apax 166
IV.5.2.3 Couleurs de variables 168
IV.5.2.4 Crochets absolus, crochets additifs 169
IV.5.3 Polymorphisme 170
IV.5.3.1 Coloration d'un type 170
IV.5.3.2 Sortage des types 171
IV.5.3.3 Crochets et application de fonction ; fonctions polymorphes 172
IV.5.3.4 Types et valeurs polymorphes 173
IV.5.3.5 Fusion des couleurs 174
IV.5.3.6 Foncteurs génératifs 175
IV.5.4 Évaluation 175
IV.5.4.1 Syntaxe 175
IV.5.4.2 Valeurs et composantes abstraites 177
IV.5.4.3 $\mathsf{B} \vdash \mathsf{E} \longrightarrow_{\mathsf{c}} \mathsf{B}' \vdash \mathsf{E}'$ Réduction 178
IV.5.5 Typage 180
IV.5.5.1 $\mathsf{B}; \Gamma \vdash_{\mathsf{c}} \mathsf{ok}$ Formation de l'environnement 181
IV.5.5.2 $\mathsf{B}; \Gamma \vdash_{\mathsf{c}} \mathsf{T} : \mathsf{K}$ Sortage des types 181
IV.5.5.3 $\mathsf{B}; \Gamma \vdash_{\mathsf{c}} \mathsf{c}_0 \, \mathsf{transparent}$ Transparence d'une couleur 182
IV.5.5.4 $\mathsf{B}; \Gamma \vdash_{\mathsf{c}} \mathsf{A} \triangleright \mathsf{E} : \mathsf{T}$; ... Composantes 182
IV.5.5.5 $\mathsf{B}; \Gamma \vdash_{\mathsf{c}} \mathsf{E} :^{\gamma} \mathsf{T}$ Coloration des expressions 183
IV.5.5.6 $\mathsf{B}; \Gamma \vdash_{\mathsf{c}} \mathsf{E} \longrightarrow \mathsf{E}'$ Conversion et crochets colorés 183
IV.6 Typage dynamique et distribution [$\mathcal{D}$] 184
IV.6.1 Typage dynamique 184
IV.6.1.1 Introduction 184
IV.6.1.2 Dynamiques 185
IV.6.1.3 Correspondance entre le typage statique et le typage dynamique . . 185
IV.6.1.4 Dynamiques et couleurs 186
IV.6.2 Formalisation 187
IV.6.2.1 Syntaxe 187

IV.6.2.2 Réduction 188
IV.6.2.3 Typage 188
IV.6.3 Communication inter-machines 189
IV.6.3.1 Introduction 189
IV.6.3.2 Communication et couleurs 190
IV.6.3.3 Universels 190
IV.6.3.4 Partage d'empreintes 192
IV.6.3.5 Scellage statique et empreintes structurelles 192
IV.7 Conclusion 193

V Conclusion **195**
V.1 Bilan 195
V.2 Autres approches 196
V.2.1 Origines théoriques 196
V.2.2 Pratique 196
V.2.3 Acute et HashCaml 197
V.2.4 Alice ML 198
V.3 Perspectives 198
V.3.1 Raffinements théoriques 198
V.3.1.1 Stratification 198
V.3.1.2 Niveaux de langue 199
V.3.1.3 Analyse d'effets 200
V.3.1.4 Couleurs et crochets 200
V.3.1.5 Décidabilité du typage 201
V.3.1.6 Paramétricité 201
V.3.2 Fonctionnalités supplémentaires 201
V.3.2.1 Nommage des champs et sous-typage en largeur 201
V.3.2.2 Vers un langage de programmation 202
V.3.2.3 Programmation générique 203
V.3.2.4 Sécurité 203
V.3.3 Implémentation 203
V.3.3.1 Calcul d'empreintes 203
V.3.3.2 Typage de TOPHAT 204
V.3.3.3 Intégration à Objective Caml : le système de modules 204
V.3.4 Applications du typage dynamique 204
V.3.4.1 JoCaml : Serveur de noms 205

A Tables récapitulatives de TOPHAT **207**

B Sûreté de TOPHAT **217**
B.1 Typage 217
B.1.1 Correction 217
B.1.2 Transparence 219
B.1.3 Affaiblissement 220
B.1.4 Substitution 222
B.1.5 Validité 223
B.2 Conversion 228
B.2.1 Confluence 229
B.2.2 Cohérence de l'équivalence de types 234
B.2.3 Sous-typage 234
B.2.4 Analyse du typage d'une expression 236
B.2.5 Typage des valeurs 238
B.3 Sûreté 240

B.3.1 Opérations sur les types 240
B.3.2 Préservation du typage par réduction 242
B.3.3 Progrès 246

Bibliographie **249**

Index **257**

Introduction

Problématique

Notre objectif dans la présente thèse est d'étendre un langage de la famille ML pour l'adapter aux systèmes répartis. Plus précisément, nous nous intéressons aux demandes que le caractère distribué de l'environnement impose au système de types — nous ne nous préoccuperons pas des aspects liés à l'exécution concurrente, ou aux pannes potentielles.

Considérons deux machines *A* et *B* qui exécutent chacun un programme. À un moment dans leur exécution, *A* et *B* se mettent à échanger des données. Le problème qui est au cœur de la présente thèse peut se résumer ainsi : comment s'assurer que *A* et *B* se comprennent ?

Une connexion réseau permet de transférer des suites de bits. Lorsque *A* envoie une donnée à *B*, cette donnée doit donc être transformée en une suite de bits ; cette opération est appelée **sérialisation**[1] (*serialization*, *pickling*, *marshaling*). *B* doit réaliser l'opération inverse, appelée **désérialisation**. De nombreux langages offrent une représentation standard des données sous forme de chaîne de caractères : s-expressions de Lisp, bibliothèque `Marshal` de Objective Caml [L$^+$] ou `Pickle` de Standard ML [PSL], interface `Serializable` en Java [Sun]... Il existe également des normes de description des données pour la communication indépendants d'un langage, comme ASN.1 et XML. Qu'il s'agisse d'une bibliothèque de sérialisation d'un langage ou d'une norme de représentation de données, il s'agit au minimum de spécifier comment représenter des nombres (entiers sur n bits, gros ou petit-boutiens, ou bien écriture décimale), des chaînes de caractères (jeux de caractères et encodages : ASCII, Unicode, ...), des séquences, etc.

Pour sérialiser, il faut savoir transformer une donnée en une suite de bits non ambiguë. La désérialisation pose quand à elle deux problèmes : il faut non seulement transformer la suite de bits en une donnée, mais aussi être capable de vérifier que la donnée en question correspond bien au type attendu. Par exemple, si le programme sur la machine *B* attend un nombre, et que le programme sur la machine *A* envoie la chaîne de caractères `"toto"`, une erreur doit être détectée. L'approche habituelle dans les langages de la famille ML est de détecter de telles erreurs le plus tôt possible, c'est-à-dire dès l'écriture du programme (dans la phase de typage lors de la compilation). Il paraît naturel d'exprimer en ML l'exigence du programme sur *B* sous forme d'une contrainte de type ; mais comment imposer cette contrainte ?

Suivant la démarche habituelle de ML, c'est lors de la compilation de *A* ou de *B* que l'incompatibilité doit être détectée. Ainsi, dans le programme tournant sur *A*, on définira un canal de type `string` (sur lequel il est correct d'envoyer `"toto"`), et sur la machine *B*, on définira un canal de type `int` (sur lequel on est assuré de ne recevoir que des nombres). Mais ceci ne fait que repousser le problème : ce n'est que lors de l'établissement de la communication entre *A* et *B* que l'on peut détecter qu'on est en train de mettre en relation un canal à entiers et un canal à chaînes de caractères.

Cette constatation nous pousse à souhaiter une vérification de types durant l'exécution des

[1]Le vocable « sériation » (de « sérier ») n'a pas été retenu par les informaticiens francophones.

programmes, spécifiquement *lors de l'établissement d'un canal de communication entre deux programmes qui n'ont pas encore communiqué.* (Si les programmes ont déjà communiqué, la vérification n'est pas forcément nécessaire, puisqu'ils ont pu se mettre d'accord sur le type des canaux de communications qui seront établis dans le futur. Ainsi JoCaml [MM01] dispose d'un système de types statique [FLMR97] ; cependant, si deux programmes JoCaml veulent communiquer, ils doivent au début passer par un « serveur de noms », qui n'est pas correctement typé.)

Bien que ML soit conçu pour être typé statiquement, et que les compilateurs effacent en général les types de sorte qu'ils ne sont plus présents lors de l'exécution, il existe des systèmes permettant de vérifier à l'exécution qu'une valeur a un certain type. Cependant ces systèmes ne savent pas gérer les types abstraits : chacun permet de déclarer qu'une valeur a un certain type prédéfini, ou un certain type construit par des moyens reproductibles à partir des types prédéfinis (liste d'entiers, arbre n-aire contenant des chaînes de caractères aux feuilles, ...), mais les types abstraits ne disposent pas d'une telle représentation compatible sur les différentes machines d'un système réparti.

Une solution est d'interdire la sérialisation des valeurs de type abstrait. Une autre est d'obliger l'auteur du type abstrait à fournir des fonctions de sérialisation et de désérialisation ; mais ceci ne résoud en fait pas notre problème : un format de sérialisation peut en général être obtenu automatiquement en utilisant la représentation interne du type abstrait, mais le véritable problème est de déterminer si le type d'envoi et le type de réception sont les mêmes, lorsque tous deux sont abstraits. C'est bien là le cœur problème : quand deux types abstraits sont-ils les mêmes ?

Il y a deux intuitions principales quant à la nature d'un type abstrait. D'une part, un type abstrait est *caché* : il possède une *implémentation*, qui est un type « concret » (l'implémentation peut faire intervenir d'autres types abstraits, mais en remontant la chaîne on finit par tomber sur des types prédéfinis) ; un type abstrait est donc un type concret, on ne sait juste pas lequel. D'autre part, un type abstrait est un *nouveau* type, distinct de tous les autres types (en particulier il est différent de tous les types concrets, et il est différent de son implémentation, au sens où l'on ne peut pas convertir librement de l'un à l'autre).

Quand deux types cachés sont-ils les mêmes ? Un pré-requis qui vient immédiatement à l'esprit est que les implémentations (les parties cachées) soient les mêmes. Cette condition n'est pourtant ni nécessaire ni suffisante. On peut souhaiter considérer deux types cachés comme les mêmes dès lors que leurs implémentations ont des comportements identiques, même si elles n'ont pas exactement le même code. Inversement, ce n'est pas parce que l'on cache deux fois le même type que l'on veut que les deux types cachés soient compatibles — par exemple un type `Euro` ne doit pas être confondu avec un type `Dollar`, même si leurs implémentations se trouvent être identiques. L'abstraction peut avoir plusieurs rôles différents ; pour chaque usage, le degré idéal de compatibilité est différent.

Quand deux types nouveaux sont-ils les mêmes ? Le modèle le plus simple consiste à répondre « quand ils ont été créés en même temps ». Cette approche a souvent été raffinée, en proposant des constructions du langage qui suivant les circonstances créent ou non de nouveaux types. En ML, le contrôle de la génération de nouveaux types s'effectue via le langage des *modules*, que nous serons donc amené à étudier.

Plan général de cette thèse

Le chapitre I présente les bases sur lesquelles cette thèse s'appuie. Nous nous penchons d'abord sur la notion d'abstraction, ses usages et ses moyens d'expression. Dans les langages de la famille ML, la source essentielle d'abstraction est le système de modules, et celui-ci a une histoire riche dont nous retraçons les grandes lignes. Nous évoquons également le thème de l'ajout de typage dynamique à un langage statiquement typé.

Le chapitre II est consacré à la notion d'*empreinte*. L'empreinte d'un composant logiciel identifie l'abstraction qu'il fournit. Nous étudions des exemples de programmes afin de dégager le degré de compatibilité souhaitable dans différentes conditions, et nous examinons comment calculer les empreintes de façons à ce qu'avoir la même empreinte signifie être compatible.

Le chapitre III présente un premier langage équipé d'un calcul d'empreintes, le langage HAT. Ce langage est une extension du lambda-calcul simplement typé, avec une notion simple de module. Nous proposons de conserver une trace des domaines d'abstraction au cours de l'exécution du programme en délimitant ces domaines par des *crochets colorés*. Nous incluons également une communication dynamiquement typée qui utilise les empreintes pour tester l'égalité des types abstraits.

Le chapitre IV décrit un nouveau *système de modules* pour ML, adapté aux systèmes collaboratifs, nommé TOPHAT. Ce langage incorpore les fonctionnalités centrales des systèmes de modules, notamment les foncteurs et le scellage. Des types singletons sans restriction de signatures permettent d'exprimer le partage de code, généralisant le partage de types habituel dans les dialectes actuels de ML. Nous montrons de plus comment exprimer différentes notions de scellage, suivant le degré de générativité attendu. Le langage TOPHAT utilise comme HAT des empreintes pour tester dynamiquement l'égalité entre types abstraits, et des crochets colorés pour préserver les frontières d'abstraction.

Nous concluons en comparant notre approche avec d'autres, et en signalant des perspectives de travaux complémentaires.

L'annexe A récapitule la définition formelle du langage TOPHAT présenté au chapitre IV. L'annexe B est consacrée à une démonstration de la sûreté de TOPHAT.

Note sur les exemples de code

Nous donnons en général les exemples de code dans la syntaxe d'Objective Caml. Nous ne ferons pas appel de la part du lecteur à la connaissance de points de détail du langage (en particulier concernant la sémantique des modules). Lorsque la discussion concerne des fonctionnalités qui ne sont pas disponibles en Objective Caml, ou qui y ont une sémantique différente, nous utiliserons souvent par cohérence une syntaxe basée sur celle d'Objective Caml et intégrant les constructions en cours d'étude (plutôt que de donner la syntaxe concrète du langage dont il est question). Le lecteur habitué à Standard ML pourra consulter une table de correspondance entre les deux langages [Ros].

Chapitre I

Préliminaires

Dans ce chapitre, nous adopterons quelquefois un point de vue historique. Nous ne prétendons nullement l'exhaustivité dans notre exposition — cette thèse n'est pas une thèse d'histoire des sciences. Notre présentation est résolument orientée vers les aspects sous-jacents au travail technique subséquent. Nous privilégierons les développements proches de notre ligne directrice sur ceux établissant l'antériorité d'une notion. Nous invitons le lecteur intéressé par une étude proprement historique à se reporter aux ouvrages appropriés cités.

I.1 Abstraction

I.1.1 Perspective historique

I.1.1.1 Procédures

Depuis que les logiciels sont trop complexes pour être appréhendés directement dans leur ensemble, s'est posé le problème de comprendre comment les séparer en morceaux gérables. Le logiciel a ceci de particulier par rapport à d'autres systèmes complexes qu'il peut grandir sans limite en changeant de destination mais non de nature. La fabrication d'un parpaing, la construction d'une maison ou la gestion d'une ville relèvent de corps de métiers différents ; les compétences requises pour être vétérinaire, berger ou ministre de l'agriculture sont assurément distinctes ; en revanche les millions de lignes pouvant constituer un logiciel sont le fait de personnes ayant suivi des études similaires.

Le logiciel n'admet donc pas une structuration physique telle que celle entre mur, maison et ville ou entre mouton, troupeau et politique agricole nationale. Or un programme de dix millions de lignes n'est pas appréhendable directement comme peut l'être un programme de dix lignes. Il apparaît donc la nécessité de trouver une structuration propre aux logiciels.

Historiquement, la première unité de structure largement utilisée est la procédure. Ainsi M. V. Wilkes note dès 1950 [Wil80] que[1] « la constitution d'une bibliothèque de procédures [est] de première importance. Non seulement parce que l'accès à une telle bibliothèque réduit l'effort demandé au programmeur, mais [...] qu'elle lui permettent de travailler à un niveau plus élevé qu'un ordinateur binaire. » La procédure permet de décharger la concrétisation d'une sous-tâche, en l'accomplissant au moyen d'un composant déjà prévu à cet effet ; ce faisant elle libère l'esprit du programmeur pour aborder des problèmes plus avancés.

[1]Les traductions sont de l'auteur. Nous avons volontairement adopté un vocabulaire cohérent dans cette discussion, quitte à ce que la terminologie employée en français soit par endroits anachronique.

M. Wilkes souligne également l'avantage de procédures linéaires[2], c'est-à-dire ayant un unique point d'entrée et un unique canal de retour (par opposition à un bloc de code dans lequel on entrerait et sortirait par des sauts multiples). Une procédure n'est pas seulement un bloc de code réutilisable : elle a aussi l'intérêt d'etre compréhensible comme un tout, puisque son utilisateur n'a pas besoin de connaître le flot de contrôle interne, et peut l'utiliser comme un composant atomique dans son propre programme. L'utilisateur d'une procédure fait abstraction de certains éléments constitutifs de celle-ci.

Nous voyons apparaître là une progression entre deux notions :
- La **modularité** consiste à découper un problème en sous-tâches (l'assemblage de ces sous-tâches pouvant lui-même être une tâche).
- L'**abstraction** est l'idée que la compréhension d'une solution à un sous-problème est indépendante de la compréhension de la manière d'utiliser cette solution.

I.1.1.2 Modularité

Si la procédure était d'emploi courant, celui-ci était encore loin d'être généralisé dans les années 1960 (pour plus de précisions sur cette période, nous suggérons la lecture de M. Jackson [Jac06] (§3) et des ouvrages qu'il cite). La systématisation de l'utilisation de procédures linéaires est au centre de la programmation structurée qui se répand alors.

La programmation structurée ne se limite pas à la modularisation du flot de contrôle. Il est également souhaitable que les procédures soient fermées au sens habituel du terme en théorie des langages, c'est-à-dire ne font référence qu'aux variables qu'elles déclarent elles-mêmes (y compris leurs paramètres) — en d'autres termes, il s'agit d'éviter l'utilisation de variables globales. L'intérêt est que le comportement d'une procédure qui ne dépend pas de paramètres extérieurs cachés tels que la valeur de variable globale est considérablement plus simple à décrire puisqu'il est reproductible. Il se crée ainsi une couche d'abstraction plus forte entre la procédure et le reste du programme.

La procédure en tant que composant logiciel est particulièrement adaptée à la structure de contrôle du programme. Elle est de plus fréquemment un composant facilement identifiable en termes de maintenance. Cependant il arrive que la structuration cognitive d'un programme demande des composants dont le flot de contrôle n'est pas linéaire. Le modèle de la procédure n'est alors plus adapté. Ceci est d'autant plus fréquent que la taille du code correspondant au composant cognitif est grande.

En 1968 eut lieu le premier symposium consacré à la programmation modulaire [BC68]. (Nous suivons ici le compte rendu de J. B. Dennis [Den72].) Un point important dégagé alors est que la modularité (ainsi que l'abstraction, telle que nous avons défini ces notions plus haut) est une propriété qui caractérise un langage ou un système dans lequel sont décrits des programmes. Citons la définition de J. B. Dennis (op. cit.) :

> Un système informatique est modulaire si l'on peut identifier dans le langage de description sous-jacent une classe d'objets correspondant aux composants dans la représentation des programmes. Ces objets sont appelé modules de programmes. Le langage doit fournir un moyen de combiner ces modules en modules plus grands sans nécessiter de modifier les modules constituants. De plus, le sens d'un module ne doit pas dépendre du contexte dans lequel il est utilisé.

La dernière phrase de J. Dennis caractérise ce que nous appelons ici abstraction.

En dehors de la procédure, J. Dennis identifie une autre forme naturelle de module : on peut voir un programme comme une collection de modules reliés par des canaux de communication et

[2] L'adjectif original est « closed » ; le sens moderne de ce mot, et de sa traduction naturelle « fermées » (ou « closes »), étant différent, nous lui substituons un terme ne prêtant pas à confusion.

qui opèrent concurremment. Chaque module est alors modélisable par l'indication du ou des flots en sur chaque canal qui en sort en fonction des données arrivant sur les canaux d'entrées. Cette notion de module s'accommode bien d'entrées et sorties multiples et étalées dans le temps.

I.1.1.3 Classes et grappes

Les dimensions de modularité que nous venons d'aborder sont décrites par référence au flot de contrôle du programme. Une autre famille de notions de modularité apparaît vers la même époque, centrée sur les données manipulées : l'on regroupe le code qui agit sur les mêmes données. Cette famille est composée de deux genres frères : les classes d'objets, et les types abstraits. Nous allons esquisser les débuts de ces deux familles.

La notion de classe naît véritablement en 1966 avec le langage SIMULA 67 [ND78] ; une importante contribution à la maturation de l'idée est le langage Smalltalk [Kay93]. En terminologie moderne, un objet regroupe au sein d'une même unité linguistique des données et du code agissant sur ces données (les méthodes). L'objet passe ainsi au cœur de la compréhension du programme. Le développement d'un programme passe alors par l'identification des structures de données intéressantes, et l'élaboration relativement indépendante de chacune d'entre elles.

Une autre approche consiste à associer les procédures agissant sur des données du même type non pas à l'objet manipulé, mais au type lui-même. Dans cette optique, un module réunit au sein d'une même entité linguistique un type et des procédures. Convenons d'employer le mot **grappe**[3] pour désigner ce concept de module dans ce sens (le terme usuel de *module* étant ici employé dans son sens plus général). Une grappe associe des procédures qui agissent sur des objets de même type, tandis qu'une classe associe des procédures qui agissent sur le même objet.

Classes et grappes sont des sources de modularité, mais pas encore d'abstraction. En effet, dans un cas comme dans l'autre, il n'y a pas de barrière entre l'objet et le reste du monde. Une méthode peut accéder librement aux champs d'un autre objet ; une procédure d'une grappe peut modifier des objets d'un type extérieur. Il ne suffit pas de lire le code de la seule classe ou grappe pour comprendre le comportement des objets concernés.

I.1.1.4 Types abstraits

Ce manque ressenti au début des années 1970 est une des principales motivations qui vont conduire au langage CLU [Lis93, LSAS77]. CLU est le premier langage implémenté qui exprime directement l'abstraction. Pour assurer l'indépendance des modules, deux aspects doivent être traités : l'encapsulation et la spécification.

L'encapsulation consiste à cacher certains aspects d'un module, de manière à ce qu'ils ne soient visibles qu'à l'intérieur de ce module. Typiquement, pour un module implémentant une structure de données, la représentation interne de ces données n'importe pas à l'utilisateur dudit module. Celle-ci peut donc être changée (par exemple pour rendre le code plus efficace) sans que le code utilisant le module doive être modifié.

La spécification est la description des aspects du module qui doivent être connus par son utilisateur. La spécification est un pendant indispensable de l'encapsulation : pour que l'utilisateur n'ait pas à connaître les aspects cachés, il faut lui fournir une description des propriétés qu'il est en droit d'attendre. S'agissant d'une structure de donnée, la spécification indique la présence d'un type (sans expliciter sa représentation) et liste les opérations sur ce type ; elle peut également préciser des relations entre ces opérations.

[3] D'après les « *clusters* » de CLU.

Le langage CLU est nommé pour les grappes (*clusters*) qu'il fournit. Donnons un exemple de squelette d'une grappe, qui implémente des ensembles d'entiers.

```
ensemble = cluster is vide, membre, ajoute, retire
  rep array[int]
  ... % (définitions des opérations omises)
end intset
```

Ce fragment de code déclare un type appelé `intset`. Ce type est **abstrait** : en-dehors de la définition de la grappe, le type `intset` est distinct de tous les autres types du programme, et en particulier l'utilisation d'un objet de type `intset` là où le type `array[int]` est attendu ou vice versa déclenchera une erreur de typage. Cette possibilité de définir des types abstraits est une contribution majeure du langage CLU.

Sur le plan théorique, une idée importante est que « les types ne sont pas des ensembles » (titre d'un article de J. Morris en 1973 [Mor73b]). Le typage d'un programme doit assurer deux types de propriétés : l'authenticité des valeurs, c'est-à-dire le fait qu'une valeur de type `T` ait forcément été produite par un constructeur reconnu du type `T` (fourni par le même module que le type), et la confidentialité, c'est-à-dire le fait que les détails d'implémentation d'un module restent cachés à l'intérieur de ce module. J. Morris présente un mécanisme de *scellage* [Mor73a, Mor73b], dans lequel une clé est associée à chaque type abstrait, clé qui est utilisée pour sceller et désceller les valeurs de ce type.

Dans un langage de haut niveau, un type abstrait ressemble beaucoup à un type prédéfini [LZ73], en ce qu'un certain nombre d'opérations sur ce type sont fournies, et l'utilisateur ne peut utiliser qu'elles pour manipuler les valeurs de ce type, ne disposant en particulier pas d'information sur la représentation en mémoire des valeurs.

Depuis le début des années 1970, les travaux sur la modularité conceptuelle des programmes se répartissent pour l'essentiel en deux lignées : l'approche par classes et l'approche par grappes. Au-delà d'une différence d'outil, les deux approches sont souvent le fruit de communautés, voire de philosophies différentes. Les premiers mettent l'accent sur la facilité d'écriture du code, les seconds sur la facilité de compréhension. L'abstraction appartient à la seconde communauté : puisqu'elle limite la vue sur l'intérieur d'une abstraction, elle interdit l'écriture de certains programmes ; mais les programmes qu'elle permet d'écrire sont plus clairs puisqu'ils peuvent être compris morceau par morceau.[4]

Pour l'aspect historique, citons sans détailler le langage Alphard [WLS76], qui offre une notion de « *formulaire* » (*form*) similaire à la grappe de CLU. Ses auteurs ont particulièrement porté leur attention sur l'utilisation des formulaires pour implémenter des structures de données, en présentant notamment une notion de générateurs [SWL77] (CLU offre une fonctionnalité similaire sous le nom d'itérateur).

I.1.1.5 Interface d'un type abstrait

Le caractère nouveau du type abstrait soulève une difficulté : comment créer ou utiliser des valeurs de ce type, à l'intérieur du module qui le fournit ? La question peut être reformulée en ces termes : comment passer du type représentation au type abstrait (et réciproquement) ? Deux types de réponses existent suivant le langage : certains fournissent une paire de primitives de conversion dont la portée est restreinte au module (méthode explicite), d'autres rendent les types équivalents à l'intérieur du module.

[4] Pour une comparaison plus détaillée des deux approches, on pourra par exemple lire l'analyse de W. Cook [Coo90].

La proposition de J. Morris [Mor73b] comprend les deux méthodes. Une valeur d'un type abstrait $\mathtt{i}$ est créée par l'opération de scellage $\mathtt{Seal}_\mathtt{i}$, et détruite par l'opération réciproque $\mathtt{Unseal}_\mathtt{i}$; ces deux opérations sont en fait deux fonctions fournies par une opération primitive `Createseal`, chaque appel à celle-ci produisant une valeur différente de $\mathtt{i}$. Le programmeur gère lui-même la restriction de l'accès aux opérations $\mathtt{Seal}_\mathtt{i}$ et $\mathtt{Unseal}_\mathtt{i}$. Les premières versions du langage ML [GMM+78] incluant des types abstraits utilisent une approche similaire : lors de la définition d'un type abstrait $\mathtt{T}$, deux identificateurs `absT` et `repT` sont ajoutés à l'environnement, fournissant deux isomorphismes réciproques entre le type abstrait et le type représentation.

J. Morris propose également un système moins expressif dans lequel les opérations de scellage et de déscellage sont implicites (et invisibles dans le code généré par le compilateur), les annotations de typage sur les procédures du module permettant d'indiquer qu'un argument ou une valeur retournée doit être convertie (virtuellement) vers ou depuis le type abstrait. Les langages Modula-3 [CDG+92], Ada [USG83], et ML [MTH90, L+] après l'introduction des modules [Mac84] utilisent des approches similaires : ce sont des annotations de types (l'interface du module) fournies par le programmeur qui déterminent les points où une valeur du type abstrait est formée ou détruite, cette conversion étant correcte parce que les deux types sont considérés comme équivalents à l'interface du module.

I.1.1.6 Spécifications

En CLU comme en Alphard, une même construction du langage définit à la fois l'interface d'un type abstrait (la liste et le type des fonctions fournies) et l'implémentation sous-jacente (la représentation du type, le code des fonctions). Or il s'agit là d'aspects bien séparés. L'un des avantages des types abstraits est d'ailleurs qu'ils permettent de substituer une implémentation par une autre ayant la même interface [Mor73b]. J. Guttag définit l'expression « type de données abstrait » (*abstract data type*) comme « une classe d'objets définie par une spécification indépendante de leur représentation » [Gut77].

Des travaux subséquents (par exemple une extension de FP [GHW81], ou le langage Ada [IBFW83]) font des définitions d'interfaces et des définitions d'implémentation des constructions séparés. À partir de là, la théorie des types abstraits recouvre plusieurs aspects : comment les spécifier, comment les implémenter, et la correspondance entre implémentations et spécifications. Le deuxième point tient de la programmation en général ; le dernier est assuré par le typage ou autre forme d'analyse du programme. Une fois les bases posées, la théorie des types abstraits se penche donc particulièrement sur les spécifications.

I.1.2 Usages des types abstraits

I.1.2.1 Invariants

L'usage le plus courant des types abstraits est le maintien d'invariants pour une structure de données que le langage ne saurait exprimer autrement. Par exemple, le langage ML permet de de définir un type des listes d'entiers, mais non un type des listes triées d'entiers, ou des listes d'entiers de longueur première, ou des ensembles finis d'entiers. La figure I.1 présente une implémentation (naïve) de ce dernier type.

Le type `Ensemble.t` correspond aux ensembles finis d'entiers parce que les opérations proposées vérifient les propriétés algébriques qui définissent la notion d'ensemble fini :

$$\forall x \in \mathtt{int}, \quad \neg\mathtt{membre}\ x\ \mathtt{vide}$$
$$\forall x \in \mathtt{int}, \forall y \in \mathtt{int}, \forall s \in \mathtt{Ensemble.t}, \quad \mathtt{membre}\ y\ (\mathtt{ajoute}\ x\ s) \iff \mathtt{membre}\ y\ s \vee x = y$$
$$\cdots$$

```
module Ensemble = (struct
  type t = int list
  let vide = []
  let rec membre x s =
    match s with [] -> false | h::t -> x = h || membre x t
  let ajoute x s = if membre x s then s else x::s
  let rec retire x s =
    match s with [] -> [] | h::t -> if x = h then t else h::(retire x t)
  ...
end : sig
  type t
  val vide : t
  val membre : int -> t -> bool
  val ajoute : int -> t -> t
  val retire : int -> t -> t
  ...
end)
```

Fig. I.1 – Un type abstrait pour les ensembles d'entiers

Ces propriétés algébriques ne peuvent pas être décrites dans le langage des interfaces ; elles ne sont mentionnées au mieux que dans la documentation du module, sinon seulement dans l'esprit de l'auteur. Les propriétés de cette nature appartiennent plutôt au domaine des langages de spécification ou à celui des assistants de preuve. En particulier, la vérification de telles propriétés nécessite une assistance manuelle, contrairement à la vérification de typage qu'effectue un compilateur ML pour assurer que l'implémentation du module ensemble (`struct ... end`) respecte bien son interface (`sig ... end`).

Pour l'instant, nous avons décrit une grappe au sens de la section I.1.1.3 — le caractère abstrait du type n'est pas intervenu. Voyons en quoi celui-ci est bénéfique.

L'une des propriétés algébriques attendues d'un type d'ensembles est que `retire` supprime effectivement l'élément de l'ensemble :

$$\forall x \in \texttt{int}, \forall s \in \texttt{Ensemble.t}, \texttt{membre}\ x\ \texttt{(retire}\ x\ s\texttt{)} = \texttt{false}$$

Or pour que le code que nous avons montré ci-dessus vérifie cette propriété, il ne faut pas que la liste s contienne plus d'un élément égal à x. Le code est correct parce que l'absence de doublon dans la liste est un *invariant* du type `Ensemble.t`, c'est-à-dire une propriété vraie de toute valeur de ce type. Deux facteurs se combinent pour assurer cela : d'une part, tous les ensembles construits par les fonctions du module `Ensemble` respectent l'invariant ; d'autre part, une valeur de type `Ensemble.t` est forcément produite par une fonction du module `Ensemble`, parce que le type `Ensemble.t` est abstrait (l'abstraction assure l'authenticité de toute valeur de type abstrait). Si le type était concret, la vérification de l'invariant nécessiterait l'examen du programme entier ; le type étant abstrait, la seule lecture du module `Ensemble` suffit.

De même que les propriétés algébriques extérieures, les langages de programmation ne permettent en général pas au programmeur d'exprimer les invariants des structures de données courantes ; la vérification de ces invariants est indécidable en général, et même dans des cas particuliers relève actuellement de la démonstration assistée et non de la programmation. En ce sens, les types abstraits permettent de dépasser les limites du langage — le type `Ensemble.t` peut être vu comme le type des listes finies d'entiers sans doublons.

I.1.2.2 Confidentialité

Outre leur intérêt pour vérifier la correction du programme, les types abstraits peuvent faciliter la maintenance. En effet, le caractère abstrait du type interdit au programme de dépendre de la représentation des valeurs, ce qui permet de substituer une autre implémentation sans avoir à modifier le reste du programme. Ainsi nous pourrions remplacer l'implémentation du module `Ensemble` ci-dessus par du code plus efficace, utilisant par exemple un arbre de recherche plutôt qu'une liste, sans que le reste du programme soit affecté. L'implémentation alternative doit bien sûr vérifier les propriétés algébriques promises par la documentation du module.

L'abstraction ne permet bien entendu pas d'éviter toutes les erreurs de programmation. Par exemple, ajoutons au module `Ensemble` ci-dessus une fonction `choix : t -> int`, spécifiée comme renvoyant un élément arbitraire de l'ensemble (et levant une exception si on l'applique à l'ensemble vide), et codée par `let choix s = List.hd s` ; l'élément choisi est alors le dernier inséré, alors qu'une implémentation à base d'arbre de recherche tendrait à être indépendante de l'ordre d'insertion. L'utilisateur du module ne doit pas faire l'une ou l'autre hypothèse. Néanmoins, l'obligation pour l'utilisateur d'un module de passer par l'interface fournie incite celui-ci à respecter la spécification que la documentation doit fournir. Il ne s'agit ici après tout que des limites usuelles des propriétés de typage décidables, que l'abstraction a permis de repousser.

Le respect d'invariants nécessite de limiter les constructeurs du type abstrait à ceux énumérés dans l'interface par l'auteur de l'abstraction, le caractère abstrait du type assurant l'authenticité des valeurs. Dualement, l'indépendance vis-à-vis de l'implémentation est garantie par la restriction des destructeurs du type abstrait. L'abstraction fournit donc une forme de confidentialité : le type représentation est tenu secret.

En pratique, l'authenticité est souvent vécue comme plus importante que la confidentialité. Ce n'est pas sans justification : l'authenticité aide plus à la correction du programme, tandis que la confidentialité aide plus à sa maintenabilité. Nous avons de plus décrit le problème de la dépendance accidentelle, qui n'a pas vraiment de dual (l'utilisateur dispose en général d'une implémentation de l'abstraction, alors que l'auteur de l'abstraction ne sait pas dans quelles circonstances son code sera utilisé). Le système de scellés proposé par J. Morris [Mor73b] propose à la fois des scellés transparents, qui certifient une valeur sans empêcher de la lire, et des scellés opaques, qui obligent l'utilisation du déscellage correspondant pour lire la valeur. De même, le langage Objective Caml [L$^+$] permet de déclarer un type comme semi-abstrait avec des constructeurs privés mais des destructeurs publics, avec le mot-clé `private`, mais ne dispose pas de la notion duale.

I.1.2.3 Estampillage

Nous avons introduit un type abstrait comme étant un type différent de tous les autres types du programme. Nous avons vu des avantages à cette différence via l'authenticité et la confidentialité qu'elle peut apporter. Mais il y a également des cas où l'on ne cherche pas à cacher la représentation du type, seulement à obtenir un type distinct mais manifestement isomorphe au type initial. Par exemple, on pourra souhaiter un type `mètres` visiblement représenté comme le type `float`, mais sans autoriser les conversions silencieuses entre l'un et l'autre, afin notamment de ne pas risquer de confondre une valeur du type `mètres` avec une valeur du type `pieds`. De même, on peut définir des types `latin1` et `utf8` tous deux manifestement représentés comme le type prédéfini `string`, mais incompatibles pour éviter de mélanger des textes encodés différemment[5]. Dans ces exemples, c'est par son nom, et non par ses propriétés, que l'on souhaite distinguer un type.

Le langage ML offre plusieurs possibilités pour définir un tel type. L'une est d'utiliser des *types fantômes* (*phantom types*).

```
module Mètres = (struct type t = unit end : sig type t end)
module Pieds = (struct type t = unit end : sig type t end)
module Grandeur = (struct
  type 'a t = float
  let u x = x
  let n x = x
  let plus x y = x +. y
end : sig
  type 'a t
  val u : float -> 'a t
  val n : 'a t -> float
  val plus : 'a t -> 'a t -> 'a t
end)
type mètres = Mètres.t Grandeur.t
type pieds = Pieds.t Grandeur.t
let longueur = Grandeur.plus (Grandeur.u 3.0 : mètres) (Grandeur.u 2.0 : mètres)
let () = print_float (Grandeur.n longueur)
let longueur' = (*rejetée au typage*)
               Grandeur.plus (Grandeur.u 3.0 : mètres) (Grandeur.u 4.0 : pieds)
```

Le type `Grandeur.t` est paramétré ; deux types paramètres incompatibles (comme par exemple `Mètres.t` et `Pieds.t`) produisent des types incompatibles `mètres` et `pieds`. Les types abstraits `Mètres.t` et `Pieds.t` sont des types fantômes, ainsi nommés car aucune valeur de ces types n'est jamais créée — seule l'existence de ces types importe. Les fonctions `Grandeur.u` et `Grandeur.n` permettent respectivement de construire et de détruire une grandeur avec unité ; ce sont des conversions de type explicites. Le polymorphisme nous permet en outre de définir des fonctions de manipulation des grandeurs avec unité, comme `Grandeur.plus` qui ajoute deux grandeurs de même unité, le résultat étant encore une grandeur de la même unité. Une tentative d'ajouter des mètres et des pieds résulte en une erreur de typage puisque `mètres` et `pieds` sont incompatibles.

Plutôt que de faire appel à un type abstrait muni de deux fonctions de conversion triviale, nous pouvons utiliser un type génératif construit.

```
module Mètres = (struct type t = unit end : sig type t end)
module Pieds = (struct type t = unit end : sig type t end)
type 'a grandeur = Grandeur of float
type mètres = Mètres.t grandeur
type pieds = Pieds.t grandeur
let nombre (Grandeur x) = x
let plus (Grandeur x : 'a grandeur) (Grandeur y : 'a grandeur) : 'a grandeur =
    Grandeur (x +. y)
let longueur = plus (Grandeur 3.0 : mètres) (Grandeur 2.0 : mètres)
```

[5]On pourra toutefois préférer des types en lecture seule comme les types « `private` » d'Objective Caml, puisque certaines suites d'octets sont invalides dans ces encodages.

```
let () = print_float (nombre longueur)
(* let longueur' = plus longueur (Grandeur.u 4.0 : pieds) *) (*mal typé*)
```

Comme auparavant, nous définissons un type paramétré par des types fantômes. Dans cette version, le type `'a grandeur` n'est pas abstrait, mais il est génératif : il est distinct de tous les autres types, mais sa représentation est manifeste. En particulier, le constructeur `Grandeur` est partout disponible ; le destructeur associé l'est aussi, même si nous définissons une fonction `nombre` par commodité syntaxique. Remarquons que pour que la fonction `plus` ne permette pas d'ajouter des mètres et des pieds, il faut restreindre son type ; ici il n'y a pas de module dont nous pouvons restreindre l'interface, donc nous annotons directement la définition.

Notons que les types génératifs sont d'usage plus général en ML : hors extensions au langage de base, ils sont le seul moyen de définir des types sommes ou récursifs[6]. Sur un plan théorique, nous pouvons modéliser un type génératif par un type abstrait muni de constructeurs et de destructeurs ayant une syntaxe particulière[7].

I.1.2.4 Contrôle de ressources

Jusqu'à présent, nous avons utilisé des types abstraits pour modéliser une structure de données caractérisée par une représentation et une sémantique particulière dont la spécification complète dépasse les capacités du langage. Mais tous les types ne sont pas des types de données. Nous avons utilisé le typage pour contrôler l'accès aux données, mais il peut aussi servir à contrôler l'accès aux ressources.

Considérons une table d'atomes, c'est-à-dire une table de correspondance entre des entiers machine et des noms[8]. Le but de cette structure de données est de passer de noms externes (des chaînes de caractères) à une représentation plus maniable ; la seule opération définie sur les noms est le test d'égalité. Des avantages des entiers machine sont une occupation mémoire faible et constante et un test d'égalité plus rapide.

```
module Atomes = struct
  type atome = int
  let dernier = ref 0
  let table = Hashtbl.create 42
  let internalise s = try Hashtbl.find table s with Not_found ->
                      incr dernier; Hashtbl.add table s dernier; !dernier
  let égal a b = (a = b)
end : sig
  type atome
  val internalise s : string -> atome
  val égal : atome -> atome -> bool
end
```

Un entier est un atome lorsque la table `table` du module `Atomes` contient une association de lui vers une chaîne. L'invariant associé au type abstrait n'est pas une pure propriété algébrique : il concerne également la structure de données modifiable `table` (qui n'est pas accessible en dehors

[6] La générativité n'est pas nécessaire, comme le prouve l'existence des variants polymorphes d'Objective Caml, mais elle facilite l'écriture du compilateur et la compréhension des rapports d'erreurs.

[7] Ainsi qu'un traitement particulier par le compilateur, notamment vis-à-vis du filtrage.

[8] Si nous associions d'autres données à chaque nom, nous obtiendrions une table de symboles telle qu'on trouve fréquemment dans les systèmes Lisp.

du module `Atomes`). L'abstraction protège non seulement les atomes (effet direct du type abstrait) mais aussi la table, qui est ici une ressource avec des contraintes de bonne utilisation.

La bibliothèque standard d'Objective Caml définit le type abstrait `Unix.file_descr` des descripteurs de fichiers. Sous Unix, la représentation de ce type est `int` ; mais la validité d'un entier comme descripteur de fichier n'est pas décidée par un invariant algébrique : il doit s'agir du numéro d'un fichier ouvert et pas refermé. Le fait qu'un entier soit ou non un descripteur de fichier évolue au cours de l'exécution du programme. L'utilisation d'un type abstrait protège les descripteurs de fichiers en assurant leur authenticité et leur confidentialité. Un système de types concrets aussi sophistiqué qu'il soit ne ferait pas l'affaire car la validité d'un descripteur de fichiers est définie extérieurement au programme.

Dans ces deux exemples, les valeurs du type abstrait sont des jetons d'accès à une ressource partagée. L'abstraction assure l'authenticité des jetons, et sert aussi à interdire l'accès direct à la ressource contrôlée par le module.

I.1.2.5 Terminologie : types de données algébriques et types abstraits

On trouve dans la littérature différentes expressions dont le sens exact est sujet à variation : « type algébrique » (*algebraic type*), « type de données algébrique » (*algebraic datatype*), « type abstrait » (*abstract type*), « type de données abstrait » (*abstract datatype*). L'acronyme anglais *ADT* se développe suivant les auteurs en « *abstract data type* » ou « *algebraic data type* ».

Il existe au moins deux définitions à l'expression « type (de données) algébrique ». Elle peut désigner un type décrit par une construction algébrique, c'est-à-dire une solution d'un système d'équations polynomiales (par exemple, le type des listes finies d'entiers est la plus petite solution de l'équation $\mathsf{T} = \{\mathtt{nil}\} + \mathtt{int} \times \mathsf{T}$). L'expression est aussi quelquefois employée pour désigner un type qui peut être caractérisé par des relations algébriques entre les opérations définies sur ce type (et d'autres) : en ce sens, le type des listes finies triées d'entiers est un type algébrique.

L'expression « type de données abstrait » (et aussi, mais à un degré moindre, « type abstrait ») est encore plus polysémique, et les nuances de sens peuvent parfois prêter à confusion. D'une part, par métonymie, un ADT peut désigner tantôt le type lui-même dans une instance précise (le type `Ensemble.t` de la figure I.1), tantôt une implémentation complète d'un type et de ses opérations (la définition complète du module `Ensemble`), tantôt la seule interface et les spécifications externes (la signature de `Ensemble`, ainsi que les équations caractérisant la structure logique d'ensemble). D'autre part, la nature que peuvent avoir les types en question varie : si, en général, le qualificatif « abstrait » implique que la représentation des objets est cachée, une idée sous-jacente est souvent qu'un système de types plus puissant pourrait en principe caractériser le type, ce qui exclut par exemple les descripteurs de fichiers. L'acronyme ADT est encore plus ambigu puisqu'il peut recouvrir tous les sens donnés à « type de données algébrique » ou à « type de données abstrait ».

Même si cela est quelque peu inhabituel, nous distinguerons dans cette thèse les expressions « type abstrait » et « type de données abstrait ». Nous utiliserons le premier terme, sans surprise, pour désigner n'importe quel type dont la représentation est cachée (au moins formellement). Nous utiliserons le second pour désigner un type abstrait servant à définir une structure de données, ce qui est le sens le plus courant donné à l'acronyme ADT.

I.1.2.6 Bilan

Nous avons décrit dans cette section trois catégories d'utilisation de types abstraits.

- Un type abstrait peut désigner un type dont la caractérisation algébrique est en principe possible mais dépasse l'expressivité du système de types (et souvent dépasse l'expressivité

de n'importe quel langage de programmation actuel). C'est le cas des exemples présentés en I.1.2.1 et I.1.2.2. La nouveauté (fraîcheur) du type assure l'authenticité et la confidentialité des valeurs. Ce sont là les types de données abstraits.
- Dans les cas discutés en I.1.2.3, les types abstraits sont utilisés pour distinguer des types isomorphes. Ils servent à donner des noms aux types ; un type peut être caractérisé par la donnée de sa représentation et de son nom.
- Un type abstrait peut également protéger une ressource précise, comme décrit en I.1.2.4. Dans ce cas, le caractère unique du type est essentiel — deux implémentations ayant le même code contrôlent néanmoins chacune une ressource et doivent donc fournir des types incompatibles.

I.2 Systèmes de modules pour ML

Dans cette section, nous examinons les principales idées mises en œuvre dans les systèmes de modules développés pour ML. Notre présentation est centrée sur les concepts qui interviendront dans la suite de cette thèse. Pour une vision moins orientée, nous renvoyons le lecteur notamment au chapitre idoine d'« ATTAPL » [HP05] ou à la bibliographie en ligne de J. Bender [Ben].

Sauf précision contraire, nous utilisons dans les exemples une syntaxe basée sur celle d'Objective Caml, plutôt que de demander au lecteur de se familiariser avec chacun des langages exhibant le comportement qui est l'objet de la discussion. Le texte donnera la sémantique du code équivalent dans le ou les langages en question.

I.2.1 Anatomie d'un système de modules

I.2.1.1 Introduction

Les premières versions du langage ML [GMM+78] permettaient de définir un type abstrait par une construction analogue à celle de CLU, en donnant la représentation du type et le code des opérations sur ce type. Voici par exemple un type abstrait codant des ensembles finis d'entiers. Les fonctions `absensemble` et `repensemble` sont fournies aux clauses « `with ... and ...` » ; elles ont respectivement les types `int list -> ensemble` et `ensemble -> int list`.

```
abstype ensemble = int list
with vide = absensemble []
and membre x s = (* ... *)
and ajoute x s = if membre x s then s else absensemble (x::(repensemble s))
retire x s = (* ... *)
```

D. MacQueen propose en 1984 un système de modules pour ML [Mac84]. À partir de là, la définition d'un type abstrait passe par la définition d'un module muni d'une interface qui cache l'implémentation du type. La définition d'un type d'ensemble d'entiers analogue à celle ci-dessus serait à quelques détails syntaxiques près identique à celle présentée en Objective Caml dans la figure I.1.

La norme de Standard ML [MTH90] définit un langage de module. Ce langage permet de décrire et de manipuler des composants logiciels. Plusieurs implémentations de Standard ML, ainsi que Objective Caml, disposent de systèmes de modules plus sophistiqués. Le développement d'un système de modules s'articule autour de plusieurs objectifs parmis lesquels :
- un langage plus expressif, au niveau des possibilités qu'il fournit pour assembler des modules ;
- un typage plus fin, assurant un contrôle plus précis de l'abstraction d'un module ;
- la compilation séparée, voire la liaison dynamique.

Le sujet principal de la présente thèse étant l'abstraction, nous nous pencherons particulièrement sur les travaux concernant les types abstraits. Comme les difficultés liées à ceux-ci sont principalement dues aux interactions avec certaines manières d'assembler des modules, nous allons d'abord passer en revue les principales méthodes d'assemblage.

Conformément à la tradition, nous désignerons par *noyau* (*core*) du langage le langage des valeurs et des types par opposition au langage des modules.

I.2.1.2 Modules de base

Le composant logiciel de base en ML est la **structure**. Une structure rassemble les différents types d'entités que l'on peut trouver dans un programme ML : types, valeurs, exceptions, ainsi que dans certains dialectes modules imbriqués, signatures, classes... (Dans cette thèse, nous nous limiterons aux types, aux valeurs et aux modules imbriqués, les autres natures d'entités pouvant se comprendre à partir de celles-ci.) Chaque composant, appelé **champ**, est désigné par un nom, qui est utilisable à la fois dans la définition des entités subséquentes dans la structure et dans le code qui fait appel à la structure. La figure I.1 présente un exemple de structure.

De même que le noyau de ML est typé, le langage des modules est muni de types, usuellement appelés signatures. La figure susmentionnée propose une signature possible pour la structure citée. Pour chaque champ qui est une valeur, la signature énonce un type ; pour un champ type, la signature peut juste mentionner l'existence du type ou donner plus d'informations (nous reviendrons sur ce point à la section I.2.1.6).

Les modules servent en ML d'espaces de noms. Deux structures `A` et `B` peuvent toutes deux comporter un champ type `t` ; le champ `t` de `A` est accessible sous le nom `A.t` tandis que le champ `t` de `B` a le nom `B.t`. L'identificateur `t` non qualifié désigne le champ `t` de la structure en train d'être définie, ou éventuellement celui d'une structure qui a été « ouverte » par une directive `open` (cette directive n'a d'effet que sur la transformation des identificateurs non qualifiés en noms résolus).

L'unité de compilation naturelle en ML est une structure. La plupart des implémentations de ML proposent au moins la compilation incrémentale : une unité peut être compilée sans que le code qui l'utilise soit disponible (une structure peut donc former une bibliothèque, ou une partie d'icelle). De nombreuses implémentations permettent même la compilation séparée [AM94] : on peut compiler une unité en ne disposant que des signatures des unités auxquelles elle fait appel, les différentes unités constituant un programme pouvant donc être compilées dans n'importe quel ordre.

I.2.1.3 Familles de modules

Le code implémentant des ensembles finis d'entiers dépend très peu du fait que les éléments de l'ensemble soient des nombres entiers. Il est facile d'isoler les dépendances dans le code, de façon à obtenir un module qui convient pour n'importe quel type d'éléments muni d'une fonction d'égalité. Le résultat présenté ici est un **foncteur**, c'est-à-dire un module paramétrisé par un autre module — en d'autres termes, un foncteur est une fonction au niveau des modules. Le module argument contient un type et une fonction (implémentant l'égalité) ; le foncteur peut être appliqué à plusieurs arguments (par exemple, les entiers, et les chaînes de caractères comparées sans distinguer la casse des lettres).

```
module EnsembleF = functor (A : sig type t val egal : t->t->bool end) ->
  (struct
    type t = A.t list
    let vide = []
```

```
    let membre x s =
      match s with [] -> false | h::t -> A.egal x h || membre x t
    let ajoute x s = if membre x s then s else x::s
    let rec retire x s =
      match s with [] -> [] | h::t -> if A.egal x h then t else h::(retire x t)
    ...
  end : sig
    type t
    val vide : t
    val membre : A.t -> t -> bool
    val ajoute : A.t -> t -> t
    val retire : A.t -> t -> t
    ...
  end)
module Entiers = struct type t = int let egal x y = (x=y) end
module EnsembleEntiers = EnsembleF(Entiers)
module Mots = struct
    type t = string
    let egal x y = (String.uppercase x = String.uppercase y)
  end
module EnsembleMots = EnsembleF(Mots)
```

On notera que la signature donnée au résultat fait appel à l'argument, puisque les spécifications des fonctions `membre`, `ajoute` et `retire` utilisent le nom `A.t` (le seul dont elles disposent) pour désigner le type des éléments de l'ensemble. Le type du foncteur `EnsembleF` est un type de fonction dépendant (aussi appelé type **produit dépendant**).

```
EnsembleF : functor (A : sig type t val egal : t->t->bool end) -> sig
  type t
  val vide : t
  val membre : A.t -> t -> bool
  val ajoute : A.t -> t -> t
  val retire : A.t -> t -> t
  ...
end
```

Nous avons vu ci-dessus un programme comme une liste de modules (des structures), ou peut-être comme un paquet de modules ordonnés partiellement par la relation de dépendance. En présence de compilation séparée, ce modèle n'est pas vraiment satisfaisant. Si le module `B`, qui fait appel au module `A`, peut être compilé en ne connaissant que la signature de `A`, il est naturel de présenter `B` comme un foncteur qui prend `A` comme argument. Des précurseurs de tels foncteurs sont présents dans la première proposition de D. MacQueen [Mac84] ; ils sont incorporés à Standard ML [MTH90].

En Standard ML, les foncteurs sont limités au premier ordre : un foncteur ne peut avoir comme argument que des structures, et non d'autres foncteurs. Cette limitation est levée dans la plupart des implémentations actuelles. Les difficultés tiennent principalement à la propagation de types en présence d'abstraction, ce qui sera l'objet de notre section I.2.2.

I.2.1.4 Constructions avancées

Le langage des modules décrit la structure des programmes. Nous avons présenté ses objets de base, les structures, et une méthode pour assembler des modules entre eux, les foncteurs. D'autres fonctionnalités se rencontrent ; nous en présentons ici quelques unes. Dans la recherche de langages de modules plus riches, un facteur limitant est que ce langage doit être compris par le compilateur (au niveau à la fois du typage et de la génération de code). Par exemple, si l'on souhaite garantir que la compilation d'un programme termine toujours, le langage des modules ne peut pas être Turing-complet.

Modules imbriqués, modules locaux De nombreux dialectes de ML permettent la définition d'un module comme champ d'un autre module : un module imbriqué dans un autre. Ceci est d'un intérêt double. D'une part, au niveau de la pure structuration logicielle, il est souvent utile de voir un composant comme un assemblage de composants à l'échelle inférieure ; ceci peut se réaliser par le biais de modules dépendant d'autre modules, mais l'imbrication facilite l'organisation. D'autre part, les modules sont la seule manière en ML d'obtenir certains effets, comme les espaces de noms et l'agglutination de types et de valeurs.

Une autre extension courante est la possibilité définir un module localement, c'est-à-dire qu'une construction comme `let module M = ... in E` est autorisée là où une expression E l'est. Ceci permet en particulier de définir un module à l'intérieur d'une fonction. L'intérêt (et la difficulté) de cette fonctionnalité est qu'elle permet de définir localement des types (par exemple en appliquant un foncteur).

Modules récursifs Nous avons décrit précédemment (I.2.1.3) un programme ML comme une liste ordonnée de modules, chaque module pouvant dépendre des précédents mais non des suivants. Certains langages de programmation permettent des dépendances arbitraires, y compris récursives, entre modules. On peut aborder le problème de deux manières : soit en permettant des définitions de modules mutuellement récursifs, soit en définissant une opération de liaison qui autorise les dépendances cycliques.

Dans tous les cas, les définitions récursives de modules permettent d'exprimer des définitions de types ou de valeurs qui sont normalement rejetées. Considérons par exemple les modules suivants :

```
module F = functor (A : sig  type t  val x : int  end) -> struct
  type t = A.t -> int
  let x = A.x + 1
end
module rec R = F(R)
```

Le foncteur `F` ne pose aucun problème, pourtant la définition récursive `R = F(R)` n'a généralement pas de sens : elle conduit à définir un type `t` vérifiant l'équation $\texttt{t} = \texttt{t} \to \texttt{int}$ (qui n'a pas de solution en ML), et une valeur x vérifiant $x = x + 1$ (qui n'a pas de solution dans une arithmétique raisonnable).

D'autres complications surgissent en présence de modules récursifs. Si l'initialisation des modules en jeu nécessite des effets de bord, il peut être difficile ou impossible de donner un sens à la définition. Une autre famille de difficultés est liée aux composantes génératives dans les modules : les types génératifs (`datatype`), les types abstraits, les exceptions ; le degré de générativité utile n'est pas encore objet de consensus.

Il existe plusieurs théories et implémentations de modules récursifs, qui diffèrent notamment par leurs critères de correction des définitions ; nous ne les détaillerons pas ici.

Modules de première classe Si de nombreuses implémentations de ML permettent de définir un module localement à l'intérieur d'une fonction, il est plus délicat d'étendre le langage pour pouvoir écrire une fonction qui renvoie un module ; une fonctionnalité qui a sensiblement le même pouvoir expressif est de stocker un module dans une variable. Cela implique essentiellement de traiter les modules comme des objets de première classe.

L'ajout de modules de première classe est bénin au niveau de la représentation des données (ceci est orthogonal à la possibilité de définitions récursives de structures, qui ne l'est pas). En revanche, le typage devient nettement plus délicat, comme nous pourrons le constater à la section IV.3.

I.2.1.5 Calculs de signatures

Nous avons longuement traité de la construction de programmes en assemblant des composants. Il est également utile de pouvoir modulariser les spécifications. En ML, cela signifie qu'une signature doit pouvoir être le résultat d'un calcul : au-dessus du langage des modules (lui-même au-dessus du noyau), il faut définir un langage des signatures.

Une première approche est de traiter les signatures comme on a traité les modules, et de définir un lambda-calcul typé pour les manipuler. Le langage des signatures doit en outre être muni de primitives pour construire des signatures : ajouter, renommer ou retirer des champs, recopier tout ou partie d'une signature, etc. Un exemple de tel langage est proposé par N. Ramsey, K. Fischer et P. Govereau [RFG05]. On peut également considérer des signatures paramétrées par des modules, comme le fait M. Jones [Jon96]. Si elle est très expressive, cette approche a l'inconvénient d'ajouter une nouvelle couche au langage, dont la complexité est relativement grande par rapport au bénéfice.

Un lambda-calcul de signatures consiste en une approche *paramétrique* des familles de signatures ; les implémentations actuelles de ML utilisent plutôt une approche *fibrée* (nettement moins expressive), consistant à décrire une famille de signature à l'aide d'une signature-type, un élément de la famille étant obtenu en modifiant une partie de la signature-type.

L'usage typique de familles de signatures en ML est de préciser un champ type dans une signature, c'est-à-dire de transformer une signature ayant un champ type `t` abstrait en une signature identique sauf pour ce qui est du champ `t` qui est devenu concret. Ceci permet une forme de polymorphisme au niveau des modules. Par exemple, le code Objective Caml[9] suivant déclare une signature `ENSEMBLE` qui peut convernir à n'importe quel module implémentant une structure d'ensemble ; le type des éléments est désigné par la composante `elt`, qui reste abstraite. En modifiant cette composante, on peut spécialiser cette signature à un type particulier d'éléments.

```
module type ENSEMBLE = sig
    type elt
    type t
    val vide : t
    val membre : elt -> t -> bool
    val ajoute : elt -> t -> t
    val retire : elt -> t -> t
    ...
  end
module type ENSEMBLE_ENTIERS = ENSEMBLE with type elt = int
module Ensemble_bool = (struct
  type elt = bool
  type t = bool * bool
```

[9]Un « `module type` » d'Objective Caml est une « `signature` » en Standard ML.

```
  val vide = (false, false)
  val membre b (f,t) = if b then t else f
  val ajoute b (f,t) = if b then (f,true) else (true,t)
  val retire b (f,t) = if b then (f,false) else (false,t)
end : ENSEMBLE with type elt = bool)
module EnsembleF = (functor (A : sig type t val egal : t->t->bool end) -> struct
    type elt = A.t
    ... (*voir I.2.1.3*)
  end : functor (A : sig type t val egal : t->t->bool end) ->
        ENSEMBLE with type elt = A.t)
```

I.2.1.6 Modules et types abstraits

La définition d'un type abstrait en ML s'effectue en deux étapes. La première est la définition d'une structure M déclarant le type représentation (par exemple sous le nom `t`) et fournissant les opérations sur le type. La seconde étape est le *scellage* du module ainsi défini par une signature S dans laquelle le type `t` est abstrait.

En Standard ML, l'opération de scellage, aussi appelée « contrainte opaque », se note `M :> S`. Au niveau du typage, les seules informations disponibles sur le module `M :> S` sont celles fournies par la signature S. Cette opération se distingue de l'ascription, ou « contrainte transparente », notée habituellement `M : S` : de même que l'annotation de type dans le noyau `(E : T)`, celle-ci peut exclure certains programmes si la signature contrainte est moins générale que la signature inferée, mais elle n'empeche pas l'utilisation de la connaissance de M pour typer le programme. Illustrons la différence sur un exemple simple :

```
structure M = struct type t = int end
signature S = sig type t end
structure Transparente = (M : S)
structure Opaque = (M :> S)
val _ = (3 : Transparente.t)    (*correct*)
val _ = (3 : Opaque.t)          (*erreur : types incompatibles*)
```

L'ascription n'est pas nécessaire dans la mesure où les langages de modules disposent de signatures principales (c'est-à-dire que tout module admet une signature qui est plus générale que toutes les autres) : une contrainte à la signature la plus générale a la même sémantique qu'elle soit transparente ou opaque. Le langage Objective Caml ne dispose que de la contrainte opaque, et celle-ci est notée `M : S`. Nous renvoyons à la section I.1.2 pour des exemples de modules scellés définissant des types abstraits, en syntaxe d'Objective Caml.

L'interaction entre le scellage et d'autres fonctionnalités du langage des modules est quelquefois subtile. En particulier, le scellage crée de nouveaux types ; or il n'est pas toujours désirable, dans des constructions complexes, de créer systématiquement des types incompatibles. En particulier, des signatures complexes doivent pouvoir exprimer des dépendances internes, signalant que tel champ type est égal à tel autre champ type.

Le problème du partage des types — ou, vu depuis l'autre côté, le problème de la générativité des types — est au cœur de nombreux travaux sur les systèmes de modules. Dans la section suivante, nous présentons les principales avancées sur ce sujet.

I.2.2 Partage de types

I.2.2.1 Sommes fortes, sommes faibles

Dans les années 1980, deux approches coexistaient quant au typage des modules en ML : l'approche *transparente* et l'approche *opaque* (la terminologie est de M. Lillibridge [Lil97]).

Certains systèmes de modules spécifient entièrement tous les composantes types des modules : les signatures sont systématiquement transparentes. Éventuellement, l'utilisateur peut écrire une signature de la forme `sig type t end`, mais dans ce cas l'apposition de la signature à un module ne cache rien quant au contenu du module, si bien que la déclaration

```
module M = struct  type t = int  let x = 3  end
         : sig     type t        val x : t  end
```

n'empêche aucunement le type `M.t` d'être égal à `int`, et l'expression `M.x + 1` d'être correcte. Des exemples de tels langages sont λ^{ML} de R. Harper, J. Mitchell et E. Moggi [HMM90] ou DL de D. MacQueen [Mac86].

Les systèmes de modules à signatures transparentes ne répondent en apparence pas à nos besoins, puisqu'ils ne permettent pas de créer des types abstraits. Pourtant il existe une manière de mettre en jeu des types inconnus. Pourvu que le système autorise des foncteurs « polymorphes », dont les champs types de l'argument ne sont pas spécifiés, la signature résultat de ces foncteurs peut mentionner le type de l'argument : le foncteur a donc un type produit dépendant (voir la section I.2.1.3.

```
module Double = functor (A : struct type t end) -> struct type t = A.t * A.t end
              : functor (A : struct type t end) -> sig type t = A.t * A.t end
```

Lorsque l'on écrit le corps d'un tel foncteur, les champs types de l'argument (comme `A.t` dans l'exemple ci-dessus) sont inconnus : ils se comportent comme des types abstraits.

La limite de l'approche transparente tient au fait que l'abstraction doit être créée à l'endroit où elle est utilisée, et non comme il se devrait à l'endroit où l'implémentation de l'abstraction est fournie. En d'autres termes, le créateur d'une abstraction ne peut rendre celle-ci effectivement abstraite ; c'est l'utilisateur qui doit respecter volontairement les règles attendues par le créateur, alors que l'abstraction vise à éviter cette situation.

L'approche opaque est inverse de l'approche transparente. Elle consiste à ne disposer pour spécifier un champ type d'une signature que de l'indication « `type t` » (« ceci est un type »), sans précision de la nature du type en question. Tout type fourni par un module est alors abstrait. Par exemple, le type `M.t` défini ci-dessus est incompatible avec tout autre type, et l'expression `M.x + 1` n'est pas correcte puisque `M.x` n'a pas le type `int`.

L'exemple séminal de langage suivant l'approche opaque est SOL de J. Mitchell et G. Plotkin [MP88]. Ces auteurs ont notamment remarqué que les types abstraits sont fondamendalement des types existentiels [CW85] : la signature `sig type t val x : t end` peut se lire « il existe un type `t` tel que `x` ait le type `t` », ce qui se note mathématiquement $\exists t.\{\mathtt{x} : \mathtt{t}\}$ (en notant classiquement $\{\mathtt{x1} : T_1; \ldots ; \mathtt{x}n : T_n\}$ un n-uplet avec champs nommés). D. MacQueen [Mac86] contraste les types existentiels, ou **sommes faibles** (dans lesquelles la première composante n'est pas spécifiée), avec les tuples habituels, ou **sommes fortes** (dans lesquelles la première composante est spécifiée).

Dans la signature `sig type t val x : t end`, le seul moyen d'exprimer le type de `x` est d'utiliser le nom `t` que l'on vient de définir. Les sommes faibles sont donc fondamentalement des *sommes dépendantes* (aussi appelées *paires dépendantes*.)

Les types existentiels doivent être « ouverts » avant utilisation. En effet, savoir qu'un module a la signature $\exists \mathtt{t}.\mathsf{T}$ ne donne aucune information sur le type `t` : `t` est ici une variable liée, et deux modules qui ont la signature $\exists \mathtt{t}.\mathsf{T}$ n'ont pas forcément le même « champ `t` ». Si M est un module de signature (correspondant à) $\exists \mathtt{t}.\mathsf{T}$, on l'utilise en l'ouvrant grâce à la construction `open` M `as X in e` ; le champ type peut alors être désigné par `X.t`, mais seulement dans la limite de la portée de la variable `X`, c'est-à-dire dans `e`. La modélisation d'un programme ML avec des sommes fortes [CL90] demande une ouverture dont la portée est « tout le reste du programme », ce qui s'accomode mal à la compilation séparée.

Un défaut majeur de l'approche opaque est qu'elle rend très vite des types incompatibles, si bien que de nombreux programmes *a priori* corrects, ne violant aucune abstraction voulue, sont rejetés. Le manque est particulièrement criant lorsque l'on considère des foncteurs ; il n'est par exemple pas possible d'écrire un foncteur identité qui transforme un module en un module équivalent : les champs types du résultat ne peuvent pas être compatibles avec ceux de l'argument.

I.2.2.2 Sommes translucentes et types manifestes

Les sommes fortes sont complètement opaques, cachant systématiquement les types ; les sommes faibles sont complètement transparentes, ne permettant pas l'abstraction. Les dialectes actuels de ML fournissent des **sommes translucentes**, dans lesquels les champs types peuvent être spécifés (ce sont alors des **types manifestes** ; on dit aussi « types concrets ») ou non (**types abstraits**) au choix du programmeur. Le terme « somme translucente » a été proposé par R. Harper et M. Lillibridge [HL94] ; un système similaire a été présenté simultanément par X. Leroy sous le nom de « types manifestes » [Ler94].

Il est possible, au sein d'un même module, de spécifier certains champs comme concrets et d'autres comme abstraits.

```
module M = struct        sig
  type t = int             type t = int
  type u = int     :       type u
  let x = 3                val x : t
  let y = 3                val y : u
end                      end
let n = x + 1      (*correct, M.t = int*)
let o = y + 1      (*mal typé, M.u est abstrait*)
```

Les sommes translucentes permettent un contrôle fin du partage des types entre modules, remplaçant les contraintes de partage (`sharing`) proposées initialement pour Standard ML [Mac84][10]. L'intérêt des sommes translucentes est qu'elles permettent d'exprimer l'égalité d'un champ type avec n'importe quelle expression de types, tandis que les contraintes de partage de Standard ML sont limitées aux champs de structures.

Une propriété importante des sommes translucentes est qu'elles fournissent des signatures principales aux modules, c'est-à-dire que tout module bien typé possède une signature plus générale que toutes les autres. Cela signifie que l'on peut toujours apposer une signature à un module (par

[10]Nous nous limitons ici à des systèmes décrivant le partage via des contraintes d'égalité ; une autre approche, plus opérationnelle, consiste à imposer la génération d'un nouveau tampon (*stamp*) [AM91, MTHM97] lorsqu'il n'y a pas égalité. L'utilisation de contraintes d'égalité peut exprimer un partage équivalent [Ler96] et est plus simple à manipuler.

exemple, pour le compiler séparément, ou pour le passer en argument à un foncteur) sans perte d'information au niveau du typage. Lorsque l'on scelle un module par une signature non transparente, celle-ci n'est pas principale, comme le montre l'exemple suivant :

```
module A  =   struct  type t = int  let x = 3  end
module B  = (A : sig  type t        val x : t  end)
module C  = (B : sig  type t        val x : t  end)
module B' = (B : sig  type t = B.t  val x : t  end)
```

Les types `B.t` et `C.t` sont incompatibles, puisque le champ type `t` est abstrait dans `C`. La signature principale du module `B` est celle apposée ci-dessus pour former le module `B'` : elle indique que le champ type `t` du module `B` est justement `B.t`. Ce principe est un exemple de *renforcement* (*strengthening*) par la règle « self » [Ler94, Ler95] (connue sous le nom de règle VALUE outre-Atlantique [HL94]) ; nous proposons le terme d'« **auto-typage** » pour traduire « *selfification* » [DCH03]. On note que l'auto-typage permet de s'affranchir de la contrainte de portée limitée des sommes fortes mentionnée à la section I.2.2.1.

Les contraintes de partage de types restent utiles au niveau du langage des signatures, pour réutiliser une signature dans laquelle un champ type est laissé abstrait en spécifiant désormais ce type. En Objective Caml ceci s'écrit

```
sig type t end with type t = int
```

L'intérêt d'une telle notation est de permettre l'utilisation de signatures nommées pour décrire des modules pouvant être obtenus par application de foncteurs ; nous en avons vu un exemple à la section I.2.1.5.

I.2.2.3 Foncteurs applicatifs

Les sommes translucentes ne décrivent pas complètement le comportement attendu dans un langage équipé de foncteurs d'ordre supérieur[11]. Les problèmes sont dus à un manque de contrôle sur la propagation des types dès qu'un foncteur peut en prendre un autre en argument.

Le paradigme de foncteur d'ordre 2 est le foncteur `Applique`, qui prend commes arguments un foncteur et un argument potentiel à celui-ci. Conformément aux usages d'Objective Caml, nous l'écrivons ici sous forme curryfiée.

```
module type S = sig type t end
module A = struct type t = int end
module Ident = functor(X : S) -> X
module Applique = functor(F : functor(X:S)->S) -> functor(X : S) -> F(X)
module B = Applique(Ident, A)
```

Ce fragment de code ne produit jamais de type abstrait, il est par conséquent naturel de typer `Applique(Ident)(A)` comme `Ident(A)` pour obtenir finalement l'égalité `B.t` = `A.t`. Or les sommes translucentes ne permettent pas d'attribuer une signature suffisamment précise au foncteur `Applique` : la signature principale, dans la présentation originale [Ler94, HL94], est `functor(F : functor(X:S)->S) -> functor(X : S) -> S`, qui ne permet pas de faire le lien entre le champ `t` du résultat et celui de l'argument `X`.

[11] La compilation séparée conduit à monter d'un cran dans l'échelle des foncteurs, donc la simple introduction de foncteurs explicites fait passer à l'ordre 2 dans la théorie (voir la section I.2.1.3).

On rencontre un manque analogue en observant la signature d'un foncteur qui crée un type abstrait. Considérons par exemple une bibliothèque fournissant une implémentation générique d'ensembles et de dictionnaires indexés par un type ordonné[12]. Ces deux implémentations sont liées par la fonction `domaine` qui renvoie l'ensemble des clés présentes dans un dictionnaire.

```
module type ORDRE = sig type t val compare : t->t->int end
module type ENSEMBLE = sig
    type t
    type element
    val membre : element -> t -> bool
    ...
  end
module Ensemble = functor (Element : ORDRE) -> (struct
    type t = ...
    type element = Element.t
    ...
  end : ENSEMBLE with type element = Element.t)
module Dict = functor (Cle : ORDRE) -> (struct
    type 'a t = ...
    ...
    module Domaine = Ensemble(Cle)
    let domaine = ...
  end : sig
    type 'a t
    ...
    module Domaine : ENSEMBLE with type element = Element.t
    val domaine : 'a t -> Domaine.t
  end)
module Mots = Ensemble(String)
module Table = Dict(String)
```

Telle que décrite ci-dessus, la fonction `domaine` est inutilisable : un appel à `Table.domaine` renvoie une valeur de type `Table.Domaine.t`, qui n'est pas le même type que `Mots.t`. Rien ne spécifie en effet dans la signature de `Dict` que c'est le module `Ensemble` qui a été utilisé pour construire `Domaine`.

Une solution envisageable dans ce cas est de faire de `Domaine` un deuxième argument au foncteur `Dict`. On écrira alors

```
module Dict' = functor (Cle : ORDRE) ->
               functor (Domaine : ENSEMBLE with type element = Element.t) -> ...
module Mots = Ensemble(String)
module Table = Dict'(String)(Mots)
```

L'indication du domaine infecte toutes les utilisations du foncteur `Dict`. Outre la lourdeur syntaxique, qui empirerait au fur et à mesure que d'autres sous-modules viendraient s'ajouter à `Domaine`, ceci rendrait plus difficile la réutilisation de code. Par exemple, la bibliothèque standard d'Objective Caml fournit des foncteurs analogues à nos `Ensemble` et `Dict`, mais sans la fonction

[12] Nos foncteurs `Ensemble` et `Dict` sont basés respectivement sur `Set.Make` et `Map.Make` d'Objective Caml.

`domaine` ; il serait bon que ce celle-ci puisse être ajoutée sans que cela implique la modification du code déjà existant.

La solution proposée par X. Leroy [Ler95] est d'autoriser dans les « noms de types » — plus précisément appelés **chemins de types** — non seulement des noms de modules, mais aussi des appels de foncteurs. On peut alors décrire les signatures des foncteurs `Applique` et `Dict` d'une manière qui répond aux besoins que nous avons évoqué :

```
Applique : functor(F : functor(X:S)->S) -> functor(X : S) -> sig
             type t = F(X).t
           end
Dict : functor(Cle : ORDRE) -> sig
         module Domaine : ENSEMBLE with type element = Element.t
                                   with type t = Ensemble(Element).t
       end
```

Le pouvoir typant supplémentaire vient de la généralisation de la règle d'auto-typage aux résultats d'applications de foncteurs : le module `F(A)` a la signature `sig type t = F(A).t end`. Cette généralisation rend les foncteurs **applicatifs** : appliquer deux fois le même foncteur au même argument donne le même résultat.

```
module F = functor(X : S) -> (... : sig type t end)
module B1 = F(A)
module B2 = F(A)
```

Dans l'interprétation applicative des foncteurs, `B1.t` et `B2.t` sont compatibles (il s'agit dans les deux cas d'une abréviation de `F(A).t`). Dans l'interprétation **générative**, en revanche, `B1.t` et `B2.t` sont distincts.

Les approches applicative et générative ont chacune leurs points forts et leurs points faibles.

- La générativité généralise plus directement la formation de types abstraits, qui repose sur la création de nouveaux types. L'applicativité ne brise pas pour autant l'abstraction : elle ne fait que réduire les lieux où celle-ci entre en jeu.
- Dans l'approche générative, si l'on souhaite obtenir plusieurs instances incompatibles d'une même spécification, il suffit que chaque instance soit obtenue par une application de foncteur distincte ; on écrira ainsi en Standard ML[13] :

  ```
  structure Linge : sig type t ... end = functor() -> ...
  structure Torchon = Abstrait()
  structure Serviette = Abstrait()
  ```

  Les types `Torchon.t` et `Serviette.t` sont incompatibles. Ceci peut se simuler aisément avec des foncteurs applicatifs en ajoutant un argument « inutile » au foncteur :

  ```
  module Linge' = functor(Un : sig end) -> (... : sig type t ... end)
  module Torchon = Abstrait(struct end)
  module Serviette = Abstrait(struct end)
  ```

  Le type `Torchon.t` n'est rien de plus que `Torchon.t`, puisque `struct end` est anonyme si bien que l'on ne peut pas lui appliquer la règle d'auto-typage.

[13] Rappelons que « `structure` » en Standard ML est synonyme de « `module` » en Objective Caml.

– L'applicativité modélise mieux les définitions de structures de données. Elle est plus douteuse lorsque l'application du foncteur engendre des effets de bord. Nous renvoyons à ce sujet aux sections IV.4.2.3 et IV.4.4.1. Notons juste que X. Leroy [Ler95] signale que tant que les modules restent de seconde classe, l'approche applicative ne pose pas de problème de cohérence.

En Standard ML, les foncteurs sont génératifs, bien que certains dialectes permettent également des foncteurs applicatifs. En Objective Caml, les foncteurs sont systématiquement applicatifs.

I.2.2.4 Cohabitation de foncteurs applicatifs et génératifs

Plusieurs calculs ont été proposés dans lesquels foncteurs applicatifs et génératifs cohabitent. Dans tous les cas, une difficulté est de gérer la différence, qui représente au moins une annotation supplémentaire dans les signatures de foncteurs.

C. Russo [Rus98] laisse le programmeur décider si chaque foncteur doit être applicatif ou génératif. On pourra par exemple choisir de déclarer le foncteur `Ensemble` comme applicatif, mais laisser `Linge` génératif. La cohabitation est toutefois d'un intérêt limité, car la simple introduction dans le langage de foncteurs applicatifs peut retirer de l'abstraction d'un programme qui n'utilise que des foncteurs génératifs. Il est en effet possible de « redéfinir » un foncteur génératif en applicatif par le biais d'une êta-expansion : on peut écrire `functor()->Linge()` en annotant le foncteur comme applicatif.

Le langage Moscow ML [RRS] est basé sur la théorie de C. Russo. Il offre une possibilité supplémentaire de qualifier les signatures de foncteurs comme applicatives ou génératives, ce qui pourrait permettre d'interdire la requalification d'un foncteur génératif en applicatif. Malheureusement, D. Dreyer [Dre02] a montré que cette extension casse le système de types.

Z. Shao [Sha99] (§2.4) sépare les foncteurs applicatifs des foncteurs génératifs par leur signature, et restreint les foncteurs applicatifs à ne pas sceller leur argument. Ainsi, dans nos exemples ci-dessus, `Applique` est applicatif, mais `Ensemble` doit rester génératif. Z. Shao remarque que cette restriction n'empêche aucunement un foncteur applicatif de créer un type abstrait : il reste en effet possible de sceller le foncteur lui-même plutôt que son résultat. On pourra donc définir un foncteur `Ensemble` applicatif de la manière suivante :

```
module Ensemble =
   ((functor(Element : ORDRE) -> struct type t = ... ... end) :
    (functor(Element : ORDRE) -> sig    type t        ... end))
```

D. Dreyer, K. Crary et R. Harper [DCH02] proposent un système de modules équipé des deux notions de foncteurs, la distinction s'effectuant comme chez Z. Shao au niveau des signatures. Ils munissent également leur langage de deux formes de scellage, une forme « forte » qui force un foncteur à être génératif lorsqu'elle est utilisée pour former son résultat et une forme « faible » qui permet le comportement applicatif. La permission pour un foncteur d'être signé applicativement est contrôlée par un système d'effets.

I.2.2.5 Modules de première classe

En ML, les modules sont en général des objets de seconde classe : ils ne peuvent pas être mélangés librement avec les valeurs du noyau. En particulier, on ne peut pas passer un module en argument à une fonction du noyau, ni faire renvoyer un module par une fonction du noyau. Le langage des modules forme en fait une couche au-dessus du langage de noyau, et est dans une certaine mesure indépendante de celui-ci [Rus98, Ler00]. Cette indépendance n'empêche cependant pas la stricte stratification de ML d'être relâchée dans différentes extensions.

Le langage de M. Lillibridge [HL94], basé sur les sommes translucentes, n'opère pas de distinction fondamentale entre les modules et les objets du noyau : les modules sont donc des objets de première classe. Lorsque le calcul d'un module demande celui d'une expression, le système peut être amené à oublier l'identité des types de ce module. Considérons par exemple un programme qui manipule des ensembles finis d'entiers en utilisant soit une implémentation générique d'ensembles finis, soit une représentation particulière plus compacte lorsque le domaine est suffisamment petit.

```
module DictTronqué =
  if minimum >= 0 && maximum <= 30
  then struct type t = int       let membre n v = (v lsr n <> 0)       ... end
  else struct type t = int list  let membre n v = List.find ((=) n) v  ... end
```

Le type `DictTronqué.t` ne peut pas être déterminé à la compilation, puisque le choix entre `int` et `int list` demande la connaissance des valeurs de `minimum` et `maximum`. Les modules de première classe introduisent donc un point où le typage peut perdre de l'information. Malheureusement ceci rend le typage indécidable [Lil97].

C. Russo [Rus99] propose un langage dans lequel la séparation entre noyau et modules est conservée au niveau syntaxique, mais un module peut être *emballé* (*packed*) pour former une valeur du noyau qui peut ensuite être déballée en un module. Si M est un module de signature S, l'expression `pack` M `as` S a le type `Pack` S ; si m est une expression de type `Pack` S, la construction `open` m `as X : S in` e recouvre le module qui a servi a fabriquer m et le rend accessible sous le nom `X` dans e. Ceci revient à peu près à ajouter au langage du noyau des types existentiels [MP88] (voir la section I.2.2.1). L'avantage majeur de conserver la stratification est que le typage du langage proposé par C. Russo est décidable. Cette forme de modules de première classe s'avère robuste vis-à-vis d'extensions du langage des modules [DCH02, Dre05].

Dans l'exemple de `DictTronqué`, le besoin des modules de première classe résulte d'un enchaînement de dépendances : la nature exacte du type `DictTronqué` dépend de l'évaluation d'une expression, et la portée de ce type dépasse celle d'un choix dynamique (ici `if ...`). Certains dialectes de ML, par exemple Moscow ML [RRS] et Objective Caml [L+], permettent la définition locale de modules. Par exemple, le code de la fonction `f` suivante manipule des dictionnaires indexés par chaînes de caractères dont seuls les `n` premiers caractères sont significatifs (le foncteur `Dict` fabrique une structure de dictionnaire comme dans la section I.2.2.3).

```
let f n =
  let module M = struct
    type t = string
    let compare x y = String.compare (String.sub x 0 n) (String.sub y 0 n)
  end in
  let module E = Dict(M) in ...
```

Le type abstrait `E.t` dépend de l'argument `n` (ajouter `"bonjour"` et chercher `"bonsoir"` n'a pas la même sémantique suivant si `n > 3` ou non). La restriction qui rend le typage possible est que la fonction `f` retourne forcément une valeur du noyau, donc ne contenant pas de champ type ; qui plus est le type abstrait `E.t` ne doit pas apparaître dans le type du résultat (cette seconde contrainte est à rapprocher du caractère limité de la portée du déballage des types existentiels mentionnée à la section I.2.2.1 : ici encore, le nom de type `E.t` est local, il n'y a pas de nom global). Ces restrictions font que le typeur n'a pas besoin d'oublier la nature des types (sauf lorsque le programmeur l'a explicitement demandé par scellage), si bien que la liaison locale de modules est un ajout relativement anodin au langage[14].

[14] En fait, elle interfère avec le polymorphisme implicite, mais ceci n'interviendra pas dans cette thèse.

I.2.2.6 Modules arguments et problème de l'évitement

La difficulté des modules de première classe est de faire renvoyer un module par une fonction du noyau, ou plus généralement d'avoir un module (donc potentiellement un type) qui dépend d'un calcul du noyau. La situation duale est très différente : passer un module en argument à une fonction est beaucoup plus simple. En effet, cela peut se voir comme du polymorphisme explicite [Lil97, DCH02]. Par exemple, permettre d'écrire `f (struct type t = int let x = 3 end)` revient essentiellement à permettre d'écrire une fonction `f` polymorphe et de spécifier que le type paramètre est `int` lors de l'appel.

Le typage de l'application d'une fonction à un module pose cependant un problème que l'on ne rencontre pas lors de l'application d'un foncteur. Nous avons vu à la section I.2.1.3 qu'un foncteur peut avoir une signature dépendante. Si c'est le cas, l'application de foncteur $\mathsf{F}(\mathsf{M})$ a une signature qui mentionne M. Par exemple, considérons le module suivant, qui définit une structure de donnée pour stocker des chaînes de caractères en ignorant la casse.

```
module Mots =
  Ensemble(struct type t = string
                  let compare x y =
                      String.compare (String.lowercase x) (String.lowercase y)
           end)
```

Le type abstrait créé par l'application du foncteur `Ensemble` doit être nommé en fonction du module anonyme qui lui est passé en argument. L'auto-typage permet de contourner le problème : ce type s'appelle `Mots.t`. Le langage du noyau étant plus riche, on ne dispose pas en général d'un nom à attribuer. Supposons par exemple que l'on puisse définir la fonction suivante, qui prend un module `M` en argument :

```
let f M = Ensemble(M).singleton "coucou"
```

On peut attribuer à `f` le type dépendant qui se noterait dans une syntaxe ressemblant à celle d'Objective Caml `function(M : ORDRE) -> Ensemble(M).t`. En revanche, appliquer `f` au module anonyme qui a servi à la construction du module `Mots` est impossible : il n'y a pas de moyen de nommer le type abstrait résultant — du moins tant que l'on ne veut pas faire apparaître des termes arbitraires du langage du noyau dans le typage.

Que l'on applique un foncteur ou une fonction à un module : deux cas sont gérables. Si le type du foncteur ou de la fonction est non dépendant, l'application peut se faire suivant la règle usuelle. Un autre cas favorable est celui où l'argument est connu statiquement ; dans ce cas cet argument peut apparaître dans le type du résultat. La définition de « connu statiquement » dépend du système considéré ; en Objective Caml, par exemple, elle correspond aux chemins (voir la section I.2.2.3). En général, les arguments problématiques sont ceux dont la fabrication entraîne des effets de bord (il est donc impossible de les connaître statiquement) ; on demande donc à l'argument d'être pur, la pureté étant une approximation jugée gérable par le concepteur du système du caractère sans effet de bord.

Le rôle d'un typeur, lorsqu'il rencontre une application de foncteur ou de fonction $\mathsf{F}(\mathsf{M})$, est de décider si celle-ci est bien typée. Si l'expression de module M n'est pas jugée pure, et si le type déclaré ou inféré pour F est dépendant, que faire ? Une solution sûre est de rejeter le programme comme mal typé. Cependant ceci peut être trop restrictif, car il est fort possible que dans le cas particulier F admette un type (ou signature) non dépendant.

Le problème de la recherche d'un type non dépendant pour une fonction prenant un module en argument est appelé **problème de l'évitement** (*avoidance problem*) [DCH02]. Le même problème se pose lorsque l'on cherche à typer le déballage d'un paquet existentiel (ou d'un module de première classe) : il s'agit d'attribuer à l'expression e dans `open` m `as X : S in` e un type évitant la variable `X`. Il n'y a en général pas de meilleur type possible ; ce défaut peut rendre le typage indécidable [GP98].

I.3 Typage statique et dynamique

I.3.1 Du typage statique

I.3.1.1 Séparation des phases

Les langages de la famille ML sont typés statiquement. Cela signifie que leur développement s'effectue en deux phases : d'abord une phase de compilation, dite aussi phase statique, puis une phase d'exécution, dite aussi phase dynamique (la terminologie a été introduite par L. Cardelli [Car88]). La phase de compilation contient notamment une phase de typage, qui décide si le programme est bien typé ou non et rejette les programmes mal typés.

L'intérêt primordial du typage est qu'il assure que le comportement du programme à l'exécution est bien défini : dans les termes de R. Milner, « un programme bien typé ne génère pas d'erreur » [Mil78]. Bien sûr, la notion d'erreur varie selon les langages, et un programme peut toujours être confronté à des erreurs venues du monde extérieur (fichier introuvable, manque de mémoire...) ; techniquement, l'exécution est en général spécifiée partiellement, et le typage a pour but d'éviter d'exécuter un programme dont le comportement n'est pas spécifié, tel que `2 + true` ou `3(4)`.

La distinction entre typage et exécution est le plus souvent claire : le langage est divisé en deux catégories syntaxiques, les types et les expressions, et le typage consiste à attribuer des types aux expressions, tandis que l'évaluation opère sur les expressions. Toutefois, certains langages incorporent des *types dépendants* ; c'est notamment le cas des systèmes de modules pour ML (voir la section I.2.1), le langage des modules mélangeant types et expressions. Le problème de recouvrer la séparation entre les deux aspects, lorsque la syntaxe ne le fait plus, a été notamment décrit par R. Harper, J. Mitchell et E. Moggi [HMM90], et est l'une des questions centrales lors de la conception d'un système de modules pour un langage de la famille ML.

I.3.1.2 Les limites de la paramétricité

Le fait que la phase de typage précède la phase d'exécution permet de limiter certaines erreurs à la première phase où elles ne sont pas dangereuses. Cela est permis par l'absence d'expressions dans le monde des types. La séparation des deux mondes a une conséquence duale : elle fait qu'il n'y a pas de types dans le monde des expressions. Ceci n'est pas vrai dans un langage comme ML, car le polymorphisme introduit une dépendance des expressions envers les types. Néanmoins, le polymorpisme de ML est *paramétrique*, ce qui limite considérablement l'influence de cette dépendance : du code qui dépend paramétriquement d'un type s'exécute toujours de la même manière quelle que soit le type en question. Les bénéfices de cette paramétricité sont de deux ordres. D'une part, elle permet d'effacer les informations de typage lors de la compilation, si bien que l'exécution ne concerne que les expressions, ce qui a le double avantage de la rendre plus compréhensible et plus efficace. D'autre part, un programme écrit dans un langage où le polymorphisme est purement paramétrique vérifie des propriétés mathématiques fortes, mises en lumière notamment par P. Wadler [Wad89].

Le problème de la paramétricité est qu'elle empêche d'écrire certains programmes — les programmes qui ne vérifient pas les théorèmes satisfaits dans un monde paramétrique. Par exemple, un tel théorème est qu'une fonction de type $\forall\alpha, \alpha * \alpha \to \mathsf{T}$ (où T ne fait pas intervenir α) est forcément constante, ce qui empêche d'écrire une fonction `equal :` $\forall\alpha, \alpha * \alpha \to$ `bool` qui renvoie `true` si ses deux arguments sont égaux et `false` sinon. Or les différents dialectes de ML fournissent une telle fonction (éventuellement limitée à une certaine classse de types, excluant notamment les types fonctionnels, mais ceci n'invaliderait pas le théorème de paramétricité). Cette fonction d'égalité polymorphe n'est pas paramétrique, et ne peut pas être écrite en ML.

Il est possible en ML d'écrire une fonction d'égalité spécialisée à un type donné, et le polymorphisme paramétrique permet de composer de telles fonctions pour agir sur des structures de données (par exemple, une fonction d'égalité sur les listes prendra comme paramètre une fonction d'égalité à appliquer aux éléments de la liste). Une manière de réaliser cela est de fabriquer des modules ayant la signature `sig type t val equal : t*t->bool end`; mais cela nécessite l'utilisation (et éventuellement la construction préalable) du module adéquat, ce qui est d'autant plus malcommode que les types en jeu sont complexes.

De nombreuses extensions à ML ont été proposées pour pouvoir écrire l'égalité polymorphe, en choisissant la bonne fonction spécialisée à appliquer pour le type effectif des arguments. Elles s'articulent en deux natures : soit la fonction d'égalité est déterminée statiquement par la classe du type, soit elle est choisie dynamiquement par analyse sur le type. Dans l'approche des classes de types, un objet sur lequel on veut appliquer la fonction d'égalité doit avoir un type de la forme `Equal` α, et chaque type α qui est instance de la classe `Equal` a une fonction d'égalité associée. Dans l'approche dynamique, une fonction maîtresse analyse le type de son argument et appelle la fonction auxiliaire adéquate. Dans la section I.3.2.1, nous passerons en revue les principaux travaux concernant l'approche dynamique, d'une part parce que celle-ci a reçu plus d'attention dans les langages de la famille ML (la communauté Haskell utilise plus souvent les classes de types), d'autre part parce que l'analyse dynamique de type est fortement liée à la vérification dynamique de type qui est un objectif principal de la présente thèse.

I.3.1.3 Les limites du typage statique

Comme la paramétricité, le typage statique est une bonne chose, mais limite l'expressivité des programmes. Les bénéfices du typage statique sont plus importants, et les cas d'insuffisance sont plus rares ; aussi est-ce avec précautions qu'il convient de s'en priver. En revanche, les conséquences de sa rupture sont plus limitées : alors que la présence d'une seule fonction polymorphe non paramétrique fournie par la bibliothèque peut invalider tous les théorèmes de paramétricité, une fonction effectuant une vérification de typage à l'exécution peut apparaître de l'extérieur comme une boîte noire sans signe particulier, pourvu que l'erreur de typage potentielle soit rattrapée.

La théorie montre que le typage statique ne permet pas d'écrire tous les programmes. En effet, dès lors le typage est décidable, il y a nécessairement des programmes corrects qui sont mal typés, c'est-à-dire que le typage est incomplet[15]. En pratique, le typage statique tel que fourni ML suffit à écrire la plupart des programmes ; une insuffisance du typage statique se traduit en général par un morceau de programme (le plus souvent un cas de filtrage) que le programmeur sait inaccessible mais que le compilateur exige quand même.

Les cas où le typage statique est forcément insuffisant, et le typage dynamique est nécessaire, sont essentiellement ceux où des données sont présentées à un programme par une entité extérieure, et le programme doit vérifier que ces données ont le type attendu. Le paradigme de vérification dynamique de type est la **désérialisation**, qui consiste à traduire des données d'une chaîne de caractères vers une structure interne au programme ; ces données ayant été produites par l'opération inverse de **sérialisation** (*serialization*, *marshalling*, *pickling*, ...). Comme les données peuvent avoir été produites, ou au moins rendues accessibles, seulement après le début de l'exécution du programme,

[15]Une analyse statique peut être complète si elle est incorrecte, c'est-à-dire qu'elle laisse passer des programmes faux. Dans la communauté ML, on évite le terme « typage » pour désigner une telle analyse.

et que ces données sont codées sous une forme brute qui ne garantit pas leur validité, une vérification de typage est nécessaire durant la phase d'exécution.

La désérialisation d'une valeur commence par conversion de la chaîne de caractères lue en un objet en mémoire. Cette première étape est sans difficulté si la chaîne a été produite par la fonction de sérialisation correspondante ; en l'absence d'un moyen extérieur d'authentification de la chaîne lue, le mieux que l'on puisse faire est de vérifier que la chaîne est bien l'image d'un objet par l'opération de sérialisation. Une fois un objet obtenu, de deux choses l'une : soit le type de l'objet est précisé par le protocole de communication ou de stockage au même titre que le format de sérialisation, auquel cas la désérialisation est terminée ; soit le type de l'objet est *a priori* inconnu, ce qui est le cas le plus courant. Dans le deuxième cas, la valeur sérialisée doit contenir une indication de type, qui est comparée au type attendu par le programme : c'est alors que s'effectue une vérification de type dynamique.

Dans certains cas, le programme peut être amené à vérifier le type d'une expression, et pas seulement celui d'une valeur ; le paradigme de cela est la boucle d'exécution fournie par la plupart des implémentations de ML, qui infère le type d'une expression entrée par l'utilisateur avant de l'exécuter (si elle est bien typée). Dans ce cas, la séparation en phases statique et dynamique n'a plus guère de sens, puisqu'elle repose fondamentalement sur le fait que le code est écrit avant d'être exécuté ; chaque fragment de code successif est soumis a une phase statique avant d'être intégré à la phase dynamique commune.

I.3.2 Analyse et vérification de type à l'exécution

Dans cette section, nous examinons les principales propositions d'ajout à un langage de la famille ML de fonctionnalités d'analyse ou de vérification de type à l'exécution. Dans la présente thèse, nous nous intéressons essentiellement au second aspect ; nous mentionnons ici également le premier parce que la vérification de typage est souvent vue comme un cas particulier d'analyse (si le type est le bon, utiliser la valeur, sinon signaler une erreur). Nous appellerons **dynamique** une valeur munie d'une indication de type utilisable à l'exécution.

I.3.2.1 Type dynamique

Le langage ML permet de définir un type comme la réunion disjointe d'un nombre fini de types :

```
type s = Cons1 of t1 | ... | Consn of tn
```

Une valeur de type `s` contient une marque qui indique à partir de quel `ti` elle est construite : le constructeur `Consi`. Une valeur typée dynamiquement peut avoir n'importe quel type ; le type des dynamiques, que nous noterons ici `dynamic`, est donc la réunion disjointe de tous les types du langage. Comme ces types sont en nombre infini, l'ajout au langage du type `dynamic` constitue une fonctionnalité nouvelle.

La première proposition de type `dynamic` en ML qui a été formalisée est due à M. Abadi, L. Cardelli, B. Pierce et G. Plotkin [ACPP91]. La construction d'une valeur de type `dynamic` a la forme `dyn e at T` où `e` est une expression de type `T` ; le dynamique résultante combine la valeur de `e` avec une indication permettant de recouvrer le type `T`. Nous nous autoriserons ici à omettre `T` lorsqu'il est le type inféré pour `e`.

La destruction d'un dynamique se fait par un filtrage nommé traditionnellement `typecase` (réminiscent du filtrage « `match` »[16] qui sert à détruire les valeurs des types sommes usuels). Ce

[16] « `match` » en Objective Caml, « `case` » en Standard ML.

filtrage peut lier des variables d'expression comme des variables de types. À titre d'exemple, voici (dans une syntaxe proche de celle d'Objective Caml) une ébauche de fonction qui prend un argument de type `dynamic` et fabrique une chaîne de caractères le décrivant.

```
let rec descriptions (d : dynamic) =
  typecase d of
  | s : string ->            ("\"" ^ s "\",      "string")
  | n : int ->               (string_of_int n,   "int")
  | ∃β. ∃α. (x, y) : (α * β) ->
    let (xs, xt) = descriptions (dyn x at α) in
    let (ys, yt) = descriptions (dyn u at β) in
                 ("(" ^ xs ^ ", " ^ ys ^ ")", xt " * " yt)
  | _ : _ ->                 ("<inconnu>",       "<inconnu>")
let décrit d =
  let (v, t) = descriptions d in ("dyn " ^ v ^ " at " ^ t)
```

Dans le cas des paires, quatre identificateurs sont liés : deux variables d'expression `x` et `y` et deux variables de type α et β. Le motif `(x, y) : (α * β)` filtre toutes les paires, quels que soient les types des composantes ; comme le type effectif des composantes est inconnu, la fonction `descriptions` fabrique de nouvelles valeurs de type `dynamic` et s'appelle récursivement dessus. Nous avons omis dans cet exemple de nombreux cas : n-uplets généraux, fonctions, types sommes et produits nommés, etc. Nous verrons comment traiter certains de ces cas au cours des sections suivantes de ce chapitre.

Puisque notre intérêt porte avant tout sur la vérification dynamique de type, décrivons-là en termes de la construction `typecase`. Le but est de définir une construction `coerce` e' `at` T' telle que `coerce dyn` v `at` T `at` T s'évalue en v, et `coerce dyn` v `at` T `at` T' déclenche une erreur lorsque T et T' sont incompatibles. Nous pouvons voir `coerce` e' `at` T' comme du sucre syntaxique pour le fragment de code suivant :

```
typecase e′ of x : T′ -> x | _ : _ -> failwith "Erreur de typage dynamique"
```

Au niveau théorique, l'introduction de `typecase` n'est pas anodine : elle permet de définir un combinateur de point fixe. Nous renvoyons à l'article de M. Abadi, L. Cardelli, B. Pierce et G. Plotkin [ACPP91] sur ce point ; l'idée générale est que si `f` a le type `dynamic -> dynamic`, alors on peut l'appliquer à `dyn f at dynamic->dynamic` : à une étape d'encodage près, ceci est une auto-application.

De nombreuses théories intègrent une construction `typecase` dans le but d'ajouter à ML du polymorphisme non paramétrique : on parle de **polymorphisme *ad hoc***, de **programmation générique** ou de **surcharge** (*overloading*). Suivant les approches, le choix de la branche du `typecase` peut être effectué à la compilation ou à l'exécution. Ce n'est que dans le deuxième cas que les dynamiques apparaissent comme un cas particulier de programmation générique. Une revue de la programmation générique dépasserait le cadre de la présente thèse ; nous nous contentons ici de mentionner les travaux intéressants vis-à-vis du typage dynamique au sens d'une vérification de typage à l'exécution (par opposition à une analyse de typage qui pourrait même bénéficier d'une garantie statique de succès).

I.3.2.2 Dynamiques polymorphes

L'interaction du typage dynamique avec le polymorphisme se révèle délicate. Aussi bien la dynamisation d'une valeur polymorphe que la construction de code polymorphe utilisant `dyn` nécessitent une étude plus approfondie que celle de la section précédente.

Le système CAML (ancêtre d'Objective Caml) [W+89] possède une construction de typage dynamique similaire dans les grandes lignes à celle que nous avons décrite à la section I.3.2.1, mais qui gère partiellement le polymorphisme. La théorie correspondante est décrite par X. Leroy et M. Mauny [LM93]. Elle permet d'associer à une valeur dynamiquement typée un schéma de types, et non un simple type ; par exemple, on peut écrire `dyn [] at` α `list` (enrobage de la liste vide polymorphe dans un dynamique ; on pourrait écrire plus explicitement `dyn (fun x->x) at` $\forall\alpha, \alpha$ `list`). Si le polymorphisme est autorisé à l'intérieur du dynamique, il est en revanche interdit à l'extérieur : le type de la valeur enrobée doit être clos. Il est ainsi impossible d'écrire `fun (x:`α`) -> dyn x at` α. La motivation opérationnelle de cette restriction est qu'il faudrait alors fournir à cette fonction l'instance effective du type α à chaque appel, ce qui n'est pas normalement fait en ML. D'un point de vue théorique, cela signifierait passer d'un polymorphisme par valeur à un polymorphisme par nom [Ler93], ce qui est une modification significative du langage. X. Leroy et M. Mauny conjecturent en outre que la condition de clôture assure que les propriétés paramétriques du langage sont conservées (voir la section I.3.1.2).

La destruction de tels dynamiques fait apparaître une nouveauté : les variables de types dans les motifs n'ont plus une signification existentielle mais universelle. En d'autres termes, un motif de types n'est plus un type avec des variables à instancier mais un schéma de types clos. Parmi les vérifications de type dynamiques suivantes, la première et la dernière réussissent clairement puisque le schéma de types exigé est le même que l'original ; la deuxième réussit également, puisque le schéma de types exigé est plus précis que l'original ; ce n'est pas le cas de la troisième qui échoue : le dynamique a le type sous-jacent `int list` alors qu'une liste polymorphe est exigée.

```
coerce dyn [] at   α list at   α list    (*succès*)
coerce dyn [] at   α list at int list    (*succès*)
coerce dyn [] at int list at   α list    (*échec*)
coerce dyn [] at int list at int list    (*succès*)
```

X. Leroy et M. Mauny [LM93] proposent une extension de la théorie ci-dessus qui permet de combiner les variables universelles et existentielles dans un motif de dynamique. Cette extension recouvre la possibilité que nous avions dans le système de la section I.3.2.1 de filtrer un dynamique sur une information partielle de typage et de faire usage des portions non spécifiées du type, tout en conservant la possibilité d'associer un schéma de types à une valeur enrobée. Reprenons l'exemple de la fonction de description d'une valeur, et ajoutons-lui quelque cas convernant des fonctions (l'exemple est dû à X. Leroy et M. Mauny).

```
let rec descriptions (d : dynamic) =
  typecase d of
  | s : string ->          ("\"" ^ s "\",      "string")
  | n : int ->             (string_of_int n,   "int")
  | ∃β. ∃α. (x, y) : (α * β) ->
    let (xs, xt) = descriptions (dyn x at α) in
    let (ys, yt) = descriptions (dyn u at β) in
                 ("(" ^ xs ^ ", " ^ ys ^ ")", xt " * " yt)
  | ∀α. f : α → α ->        ("fun x -> x",      "forall 'a. 'a -> 'a")
  | ∃α. ∀β. f : α → β ->    ("fun x -> ⊥",      "forall 'b. 'a -> 'b")
  | ∀α. ∃β. f : α → β ->
    let (s, t) = descriptions (dyn f () at β) in
                           ("fun _ -> " ^ s,   "forall 'a. 'a -> " ^ t)
```

```
  | _ : _ ->                  ("<inconnu>",       "<inconnu>")
```

Le motif $\forall\alpha.$ `f : ` $\alpha \to \alpha$ reconnaît les fonctions polymorphes dont le type de retour est le même que l'argument ; par paramétricité[17] une telle fonction est forcément l'identité. Le motif $\exists\alpha.\ \forall\beta.\ f : \alpha \to \beta$ reconnaît les fonctions qui retournent une valeur de type universel ; comme il n'existe pas de telle valeur en ML, les fonctions reconnues ne peuvent pas terminer. Le motif $\forall\alpha.\ \exists\beta.\ f : \alpha \to \beta$ reconnaît quant à lui les fonctions dont l'argument est polymorphe, et le type de retour est fixé mais non spécifié : par paramétricité toujours, ce sont des fonctions constantes.

Une autre proposition pour gérer le polymorphisme dans les dynamiques est due à M. Abadi, L. Cardelli, B. Pierce et D. Rémy [ACPR94]. Elle diffère de celle de X. Leroy et M. Mauny en ce que les motifs complexes sont exprimés au moyen de variables d'ordre supérieur plutôt qu'en termes d'alternances de quantificateurs ; par exemple on écrira $\forall\alpha.\alpha \to T(\alpha)$ plutôt que $\forall\alpha.\exists\beta.\alpha \to \beta$. Le système résultant est un peu plus expressif, mais également plus compliqué.

Aucun des systèmes que nous avons présenté jusqu'ici ne permet d'écrire une fonction comme `fun x -> dyn x`. Nous avons vu que le problème vient de ce qu'à différents niveaux le type effectif de `x` doit être accessible au moment de l'exécution de la fonction, or les types sont effacés à la compilation. Une proposition qui vainc cette limitation est le système G'Caml de J. Furuse [Fur02, FW00]. Il introduit une nouvelle forme de variable de types, des variables dynamiques (dites aussi génériques) notées δ. Ces variables ont une manifestation à la fois au niveau des types et au niveau des expressions. Le système fournit alors deux primitives `dyn :` $\forall\delta.\delta \to$ `dynamic` et `coerce :` $\forall\delta.$`dyn` $\to \delta$. La fonction `fun x -> dyn x` a naturellement le type $\forall\delta.\delta \to$ `dynamic` comme `dyn` ; à l'exécution, elle reçoit un argument supplémentaire qui encode le type de son argument explicite. Lors de l'utilisation d'une variable de type $\forall\delta.T$, la variable de type δ est instanciée (les variables restant libres après l'inférence de type sont généralisées) et le type concret est passé aux primitives qui utilisent δ.

I.3.2.3 Dynamiques et existentiels

Un dynamique d est une valeur dont le type est inconnu. Une manière bien connue [CM88, MP88] de voir cela, est de dire qu'il existe un type tel que d ait ce type : d a le type $\exists t.t$. Ce point de vue est cohérent avec celui exprimé au début de la section I.3.2.1 qui fait du type `dynamic` la somme disjointe de tous les types : un quantificateur existentiel est en quelques sortes une somme sur son domaine.

L'idée a notamment été exploitée par A. Baars et D. Swierstra [BS02] pour coder des dynamiques dans des dialectes courants de Haskell. Un dynamique est décrit comme une paire dont la première composante a le type α et la seconde a le type $R(\alpha)$ où $R(T)$ est une réification du type T. La représentation d'un type est construite de manière systématique grâce à une classe de types judicieuse ; le succès d'une vérification de type repose sur la preuve d'égalité du type exigé avec le type original, et le programmeur doit fournir une preuve d'égalité en s'aidant d'une bibliothèque de combinateurs.

Dans le monde ML, nous avons rapproché les types existentiels des modules ayant des types abstraits (voir la section I.2.2.1). Ceci suggère de voir un dynamique comme un module plutôt qu'une expression du noyau ; la vérification ou l'analyse dynamique de typage opère alors sur des signatures. À notre connaissance, aucune étude théorique de cette idée n'a été effectuée. L'avantage de cette piste est qu'elle permet d'ajouter à très peu de frais à un compilateur ML existant des dynamiques supportant une forme limitée mais déjà très utile de polymorphisme ; par exemple, la

[17] Notons que la présence de typage dynamique n'invalide pas en soi la paramétricité du langage ; la paramétricité est en revanche mise à mal si l'on peut typer `fun x -> dyn x` de façon polymorphe (voir la suite de la discussion).

signature `sig type t val f : t -> t end` décrit une fonction monomorphe agissant sur un type non spécifié, tandis que la signature `sig val f : 'a -> 'a end` décrit une fonction polymorphe.

I.3.2.4 Typage dynamique et abstraction

L'objectif le plus courant de la programmation générique est de définir des fonctions génériques, c'est-à-dire des fonctions inductivement sur la structure du type de leur argument — par exemple, une fonction d'impression (que la fonction `décrit` que nous avons écrite à la section I.3.2.1 préfigure), ou une fonction qui calcule la taille d'une structure de données (longueur d'une liste ou d'un tableau, nombre de nœuds d'un arbre...), ou encore une fonction qui recopie une structure de données modifiable quelle que soit son type. Dans cette optique, les types abstraits apparaissent comme contradictoires avec la programmation générique. Si l'on respecte l'abstraction, un objet de type abstrait se présente comme une boîte noire à une fonction générique, dont l'intérêt est alors limité. À l'inverse, casser l'abstraction pour les fonctions génériques oblige à se passer des garanties que l'abstraction peut apporter.

Dans le cas particulier de la vérification dynamique de typage, la structure du type n'importe pas ; il suffit, pour exprimer `coerce` en termes de `typecase`, que ce dernier autorise une variable de type existentiellement quantifiée (voir la section I.3.2.1). Le caractère abstrait éventuel du type effectif de la valeur n'intervient pas.

La présence dans le langage d'existentiels — ce qui, comme nous l'avons signalé à la section I.3.2.3, inclut le cas de modules de première classe — fait apparaître une possibilité supplémentaire lors du typage dynamique du déballage d'un paquet existentiel (voir la section I.2.2.1), déjà signalée par M. Abadi, L. Cardelli, B. Pierce et D. Rémy [ACPR94]. Considérons en effet la fonction suivante, qui détruit et reconstruit aussitôt un dynamique :

```
let f d = let module M = coerce d at sig type t end in dyn M at sig type t end
```

Le déballage qui sert à calculer `M` peut avoir deux interprétations : soit `M.t` est considéré comme égal au champ type `t` du module qui a servi a former l'argument `d`, soit `M.t` est considéré comme un nouveau type abstrait. À la compilation, si l'on voit `coerce d at sig type t end` comme le déballage d'un paquet existentiel, c'est la seconde sémantique qui est utilisée — la première étant impossible puisqu'elle demanderait de garder trace des champs types de `d`, ce qui est contraire à la dynamicité du typage. La seconde interprétation permet donc de voir plus facilement le typage dynamique comme un pis-aller qui pourrait être éliminé si le typage statique était suffisamment puissant ; elle permet de raisonner sur le typage sans se préoccuper de la phase durant laquelle il est considéré. En revanche, la première interprétation est le plus souvent désirée, car elle n'introduit pas d'incompatibilité inutile.

Chapitre II

Empreintes pour les types abstraits distribués

II.1 Introduction

Dans cette partie, nous étudions comment étendre un langage de la famille ML pour l'adapter aux systèmes répartis[1]. Plus précisément, nous nous intéressons aux demandes que le caractère distribué de l'environnement impose au système de types — nous ne nous préoccuperons pas des aspects liés à l'exécution concurrente.

Considérons deux machines A et B qui exécutent chacun un programme. À un moment dans leur exécution, A et B se mettent à échanger des données. Le problème qui est au cœur de la présente thèse peut se résumer ainsi : comment s'assurer que A et B se comprennent ?

Une connexion réseau permet de transférer des suites de bits[2]. Lorsque A envoie une donnée à B, cette donnée doit donc être transformée en une suite de bits ; cette opération est appelée **sérialisation** (*serialization*, *pickling*, *marshaling*). B doit réaliser l'opération inverse, appelée **désérialisation**. De nombreux langages offrent une représentation standard des données sous forme de chaîne de caractères : s-expressions de Lisp, bibliothèque `Marshal` de Objective Caml [L$^+$] ou `Pickle` de Standard ML [PSL], interface `Serializable` en Java [Sun]... Il existe également des normes de description des données pour la communication indépendants d'un langage, comme ASN.1 et XML. Qu'il s'agisse d'une bibliothèque de sérialisation d'un langage ou d'une norme de représentation de données, il s'agit au minimum de spécifier comment représenter des nombres (entiers sur n bits, gros ou petit-boutiens, ou bien écriture décimale), des chaînes de caractères (jeux de caractères et encodages : ASCII, Unicode, ...), des séquences, etc.

Pour sérialiser, il faut savoir transformer une donnée en une suite de bits non ambiguë. La désérialisation pose quand à elle deux problèmes : il faut non seulement transformer la suite de bits en une donnée, mais aussi être capable de vérifier que la donnée en question correspond bien au type attendu. Par exemple, si le programme sur la machine B attend un nombre, et que le programme sur la machine A envoie la chaîne de caractères `"toto"`, une erreur doit être détectée. L'approche habituelle dans les langages de la famille ML est de détecter de telles erreurs le plus tôt possible, c'est-à-dire dès l'écriture du programme (dans la phase de typage lors de la compilation). Il paraît naturel d'exprimer en ML l'exigence du programme sur B sous forme d'une contrainte de type ; mais comment imposer cette contrainte ?

Suivant la démarche habituelle de ML, c'est lors de la compilation de A ou de B que l'incom-

[1] Nous emploierons indifféremment les expressions « système réparti » et « système distribué ».

[2] la plupart du temps groupés en octets.

patibilité doit être détectée. Ainsi, dans le programme tournant sur A, on définira un canal de type `string` (sur lequel il est correct d'envoyer `"toto"`), et sur la machine B, on définira un canal de type `int` (sur lequel on est assuré de ne recevoir que des nombres). Mais ceci ne fait que repousser le problème : ce n'est que lors de l'établissement de la communication entre A et B que l'on peut détecter qu'on est en train de mettre en relation un canal à entiers et un canal à chaînes de caractères.

Cette constatation nous pousse à souhaiter une vérification de types durant l'exécution des programmes, spécifiquement *lors de l'établissement d'un canal de communication entre deux programmes qui n'ont pas encore communiqué.* (Si les programmes ont déjà communiqué, la vérification n'est pas forcément nécessaire, puisqu'ils ont pu se mettre d'accord sur le type des canaux de communications qui seront établis dans le futur. Ainsi JoCaml [MM01] dispose d'un système de types statique [FLMR97] ; cependant, si deux programmes JoCaml veulent communiquer, ils doivent au début passer par un « serveur de noms », qui n'est pas correctement typé.)

Bien que ML soit conçu pour être typé statiquement, et que les compilateurs effacent en général les types de sorte qu'ils ne sont plus présents lors de l'exécution, il existe des systèmes permettant de vérifier à l'exécution qu'une valeur a un certain type (voir la section I.3). Cependant ces systèmes ne savent pas gérer les types abstraits : chacun permet de déclarer qu'une valeur a un certain type prédéfini, ou un certain type construit par des moyens reproductibles à partir des types prédéfinis (liste d'entiers, arbre n-aire contenant des chaînes de caractères aux feuilles, ...), mais les types abstraits ne disposent pas d'une telle représentation compatible sur les différentes machines d'un système réparti.

Une solution est d'interdire la sérialisation des valeurs de type abstrait. Une autre est d'obliger l'auteur du type abstrait à fournir des fonctions de sérialisation et de désérialisation ; mais ceci ne résoud en fait pas notre problème : un format de sérialisation peut en général être obtenu automatiquement en utilisant la représentation interne du type abstrait, mais le véritable problème est de déterminer si le type d'envoi et le type de réception sont les mêmes, lorsque tous deux sont abstraits. C'est bien là le cœur problème : quand deux types abstraits sont-ils les mêmes ?

Il y a deux intuitions principales quant à la nature d'un type abstrait. D'une part, un type abstrait est *caché* : il possède une *implémentation*, qui est un type « concret » (l'implémentation peut faire intervenir d'autres types abstraits, mais en remontant la chaîne on finit par tomber sur des types prédéfinis) ; un type abstrait est donc un type concret, on ne sait juste pas lequel. D'autre part, un type abstrait est un *nouveau* type, distinct de tous les autres types (en particulier il est différent de tous les types concrets, et il est différent de son implémentation, au sens où l'on ne peut pas convertir librement de l'un à l'autre).

Quand deux types cachés sont-ils les mêmes ? Un pré-requis qui vient immédiatement à l'esprit est que les implémentations (les parties cachées) soient les mêmes. Cette condition n'est pourtant ni nécessaire ni suffisante. On peut souhaiter considérer deux types cachés comme les mêmes dès lors que leurs implémentations ont des comportements identiques, même si elles n'ont pas exactement le même code. Inversement, ce n'est pas parce que l'on cache deux fois le même type que l'on veut que les deux types cachés soient compatibles — par exemple un type `Euro` ne doit pas être confondu avec un type `Dollar`, même si leurs implémentations se trouvent être identiques. Nous avons présenté à la section I.1.2 différents usages des types abstraits ; pour chaque usage, le degré idéal de compatibilité est différent.

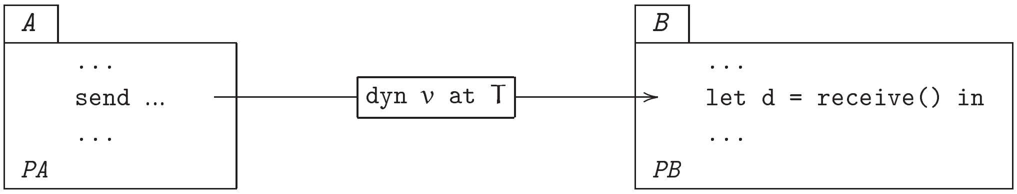
Quand deux types nouveaux sont-ils les mêmes ? Le modèle le plus simple consiste à répondre « quand ils ont été créés en même temps ». Cette approche a souvent été raffinée, en proposant des constructions du langage qui suivant les circonstances créent ou non de nouveaux types. En ML, le contrôle de la génération de nouveaux types s'effectue via le langage des modules, que nous avons examiné à la section I.2.2.

Dans le présent chapitre, nous allons chercher à dégager une notion fine de compatibilité de types abstraits. Pour ce faire, nous allons examiner différents exemples de programmes distribués mettant en jeu des types abstraits, en interrogeant pour chaque exemple sur le bien-fondé de considérer le programme comme valide. Nous dégagerons progressivement une notion d' qui transcrit les équivalences désirées entre types abstraits.

II.2 Étude de cas

Le but de cette section est de présenter des exemples de vérification de typage sur des valeurs sérialisées. Ces exemples seront présentés de façon informelle, dans un langage basé sur ML (les exemples seront donnés avec la syntaxe d'Objective Caml, mais les constructions sortant du noyau de ML seront expliquées au fur et à mesure). Pour chaque exemple, nous examinerons s'il est judicieux ou non que la valeur sérialisée soit acceptée, et nous étudierons l'impact de l'exemple sur la conception de l'algorithme de vérification de type lors de la désérialisation. Dans cette section, nous nous limitons à un sous-ensemble de ML qui *exclut les effets de bord à l'intérieur d'un module.*

Nous nous plaçons dans le scénario suivant. Deux machines *A* et *B* exécutent chacun un programme, respectivement *PA* et *PB*. Chacun de ces deux programmes a été typé statiquement de son côté. À un moment de l'exécution de *PA*, celui-ci émet une valeur sérialisée ; à un moment de l'exécution de *PB*, celui-ci lit et désérialise la valeur qu'a émise *PA*.



La communication s'effectue typiquement via un canal de communication réseau, par exemple une connexion TCP ; elle peut aussi passer par un stockage persistant, si *PA* écrit la valeur sérialisée sur le disque et que *PB* la relit, peut-être longtemps après. Si ces deux modes de communication peuvent souvent être traités de la même manière, le premier offre une possibilité dont le deuxième ne peut bénéficier, à savoir une négociation entre *A* et *B* durant laquelle les machines s'échangent plusieurs messages.

Nous supposons que *PA* a accès à une primitive `send : string->unit` et que *PB* a accès à une primitive `receive : unit->string`; dans nos exemples, *PA* appellera une fois `send`, et *PB* appellera une fois `receive` et recevra la valeur écrite par *PA*. Nous supposons de plus le langage étendu par deux constructions

`(dyn` *e* `at` *T*`)`	`: string`	où *e* : *T*
`(coerce` *s* `at` *T'*`)`	: *T'*	où *s* : `string`

La construction `dyn` *e* `at` *T* est le cœur de l'opération de sérialisation : elle transforme une valeur arbitraire du langage *e* en une chaîne de caractères qui la représente ; cette chaîne de caractères contient également une annotation représentant le type *T*, et il est vérifié statiquement que *e* a le type *T*. La construction `coerce` *s* `at` *T'* est duale : la chaîne de caractères *s* doit contenir une valeur *e* sérialisée à un type *T* ; le résultat est la valeur *e* si les types *T* et *T'* sont compatibles, et la désérialisation lève une exception si les types sont incompatibles. On a donc

`coerce (dyn` *e* `at` *T*`) at` *T*	$\longrightarrow e$	
`coerce (dyn` *e* `at` *T*`) at` *T'*	échoue	si *T* et *T'* sont incompatibles

Nous supposerons acquis la transformation d'une valeur arbitraire en une chaîne de caractères, pour nous concentrer sur le propos de la présente thèse, à savoir la vérification de compatibilité entre les types `T` et `T'`.

II.2.1 Types concrets

Pour fixer les idées, nous présentons d'abord une situation sans type abstrait. La machine `A` envoie à la machine `B` un entier ; la machine `B` attend un entier, et accepte la désérialisation.

P1A=

```
send (dyn 5 at int)
```

√ *P1B*=

```
print_int (coerce receive() at int)
```

En revanche, si `A` envoie un entier tandis que `B` attend une chaîne de caractères, la désérialisation doit échouer. (Nous ne cherchons pas ici à faire de conversion automatique.)

P2A=

```
send (dyn 5 at int)
```

× *P2B*=

```
print_int (coerce receive() at string)
```

II.2.2 Types abstraits : respect des invariants

II.2.2.1 Modules et types abstraits

Considérons maintenant un premier exemple de type abstrait. En Objective Caml, les types abstraits proviennent de déclarations de modules *scellés* à une signature. Une **structure** — la valeur de base dans la syntaxe des modules — est une collection de définitions de types et de termes, encadrée par les mots clés `struct` ... `end`. Voici par exemple une structure définissant quatre champs : un type appelé `t` et trois termes appelés `début`, `suivant` et `courant` :

```
module M = struct
  type t = int
  let début = 0
  let suivant x = x + 2
  let courant x = x
end
```

La **signature** est au module ce que le type est au terme. Voici deux signatures possibles pour la structure ci-dessus :

```
module type Sc = sig
  type t = int
  val début : t
  val suivant : t -> t
  val courant : t -> int
end
```

```
module type Sa = sig
  type t
  val début : t
  val suivant : t -> t
  val courant : t -> int
end
```

La signature S_c rappelle le champ type du module, et liste des déclarations de type pour les champs termes. La signature S_a est presque identique, à ceci près qu'elle cite la présence du type `t` sans donner sa définition. On dit que S_c est **transparente**, ou que le type `t` y est **concret** ; S_a est **opaque**, le type `t` y étant **abstrait**. (En plus de rendre certains types abstraits, une signature peut cacher certains champs ; cette fonctionnalité ne nous importe pas ici.)

Considérons le fragment de code suivant, qui déclare deux modules appelés respectivement `A` et `C`.

```
module C = (M : Sc)
module A = (M : Sa)
```

Ces deux modules ont la même implémentation M, mais deux signatures différentes : le type `t` du module `C` (auquel on accède par `C.t`) est concret, si bien qu'on peut utiliser indifféremment `int` ou `C.t` ; le type `t` du module `A` est abstrait, c'est un nouveau type, incompatible notamment avec `int`. L'opération M `:` S est appelée **scellage** (on scelle un module à une signature). Nous relâcherons la syntaxe par rapport à Objective Caml en nous autorisant à omettre les parenthèses autour du scellage, écrivant donc par exemple `module A =` M `:` S_a.

On appelle **représentation** d'un type abstrait le type défini dans le module et qui est caché dans la signature. Ainsi la représentation de `A.t` est `int`.

Le type `A.t` étant abstrait, les seules manières de produire des valeurs de ce type sont les champs du module M que S_a laisse paraître. Nous pouvons en déduire quelques propriétés, qui sont des **invariants** du module `A` (voir la section I.1.2.1) :

1. La valeur retournée par `A.courant` est un entier positif.
2. La valeur retournée par `A.courant` est un entier pair.
3. La valeur de `A.courant (A.suivant (... (A.suivant A.début)...))` est le double du nombre d'appels à `A.suivant` dans la chaîne.

Bien que le code présenté ici ressemble à celui d'un compteur qui renverrait une valeur différente à chaque invocation, il n'en est pas un : ce code est dans le fragment pur de ML, et il n'y a donc aucune restriction à l'existence de compteurs multiples ni à la duplication d'un compteur. Nous verrons plus loin (II.6.1) comment imposer de telles restrictions, et l'influence que cela a sur le typage et la sérialisation.

II.2.2.2 Compatibilité inter-machines

Examinons maintenant notre exemple de module définissant un type abstrait dans un contexte distribué. Définissons ce module exactement de la même manière sur les machines *A* et *B*.

P3A =

```
module CompteurPair =
  struct                        sig
    type t = int                  type t
    let début = 0           :     val début : t
    let suivant x = x + 2         val suivant : t -> t
    let courant x = x             val courant : t -> int
  end                           end
send (dyn CompteurPair.début at CompteurPair.t)
```

P3B = ✓

```
module CompteurPair =
  struct                          sig
    type t = int                    type t
    let début = 0             :     val début : t
    let suivant x = x + 2           val suivant : t -> t
    let courant x = x               val courant : t -> int
  end                             end
print_int (CompteurPair.courant (coerce receive() at CompteurPair.t))
```

Puisque le code de `CompteurPair` est le même des deux côtés, tous les invariants sont conservés. Il est donc parfaitement sûr d'envoyer une valeur créée par le `CompteurPair` de la machine *A* et de l'utiliser ensuite via le `CompteurPair` de *B*.

II.2.2.3 Concret n'est pas abstrait

Il serait en revanche néfaste d'autoriser la sérialisation à transformer un entier quelconque en une valeur de type `CompteurPair.t` : ceci introduirait un moyen de casser les invariants qui font l'intérêt du type abstrait.

P4A =

```
send (dyn 3 at int)
```

✗ *P4B* =

```
module CompteurPair =
  struct                          sig
    type t = int                    type t
    let début = 0             :     val début : t
    let suivant x = x + 2           val suivant : t -> t
    let courant x = x               val courant : t -> int
  end                             end
print_int (CompteurPair.courant (coerce receive() at CompteurPair.t))
```

On pourrait envisager d'autoriser la désérialisation à condition d'y vérifier les invariants. Si le code de *A* était `send (dyn 2 at int)`, l'invariant de parité serait bien respecté. Ceci demanderait au programmeur d'expliciter les invariants qu'il veut assurer.

Notons `t` le type abstrait étudié et *T* sa représentation ; le programmeur devrait fournir une fonction `acceptable` de type *T* `-> bool`, ou de manière équivalente une fonction `injecte` de type *T* `-> t` qui lève une exception si l'invariant n'est pas vérifié (on pourrait définir `injecte = fun x -> coerce dyn x at` *T* `at t`). Ceci serait équivalent à munir le module définissant `t` d'une fonction `injecte = fun x -> x` avec comme signature *T* `-> t`. Le langage fournit donc déjà une manière simple pour permettre la conversion voulue, il n'y a donc pas lieu d'introduire une fonctionnalité supplémentaire.

II.2.2.4 Abstrait n'est pas concret

Considérons le cas inverse, où une valeur d'un type abstrait est convertie vers sa représentation. S'agissant de `CompteurPair`, l'opération est sûre au sens ou elle ne permet pas d'écrire des programmes que l'auteur du module a voulu exclure : de façon générale, il n'y a pas de risque de briser un invariant. Pourtant nous ne proposons pas d'accepter la désérialisation dans ce cas. En effet, les types abstraits peuvent être utilisés pour assurer la confidentialité de certaines données (voir la section I.1.2.2).

P5A =

```
module Prive =
  struct                          sig
    type t = string * int           type t
    let encrypte p x = p, x         val encrypte : string -> int -> t
    let decrypte p (q, x) =         val decrypte : string -> t -> int
      if p = q then y else failwith "decrypte"
  end                             end
send (dyn encrypte at CompteurPair.t)
```

P5B = ×

```
print_int (coerce receive() at int)
```

Dans cet exemple de coffre-fort simplifié au maximum, convertir de `string * int` en `Prive.t` est anodin, mais le contraire casserait l'intérêt du module.

Ici aussi, si le programmeur souhaite que la conversion soit possible, il peut la prévoir dans l'implémentation du module, en fournissant une fonction `ejecte : t ->` *T* `= fun x -> x`.

II.2.2.5 Abstrait n'est pas abstrait

Une caractéristique importante des types abstraits est que de l'information est cachée ; alors que deux types concrets sont compatibles dès lors que leurs signatures sont les mêmes, comparer deux types abstraits nécessite de comparer aussi l'information cachée.

Il faut comparer les représentations :

P6A =

```
module CompteurPair =
  struct                                  sig
    type t = float                          type t
    let début = 0.0                 :       val début : t
    let suivant x = x +. 2.                 val suivant : t -> t
    let courant x = int_of_float x          val courant : t -> int
  end                                     end
send (dyn CompteurPair.début at CompteurPair.t)
```

P6B = ×

```
module CompteurPair =
  struct                          sig
    type t = int                    type t
    let début = 0           :       val début : t
    let suivant x = x + 2           val suivant : t -> t
    let courant x = x               val courant : t -> int
  end                             end
print_int (CompteurPair.courant (coerce receive() at CompteurPair.t))
```

Cet exemple est clairement incorrect : une valeur de type flottant se retrouve interprétée comme un entier.

Il faut comparer les invariants :

P7A =
```
  module CompteurPair =
    struct                          sig
      type t = int                    type t
      let début = 0           :       val début : t
      let suivant x = x + 2           val suivant : t -> t
      let courant x = x               val courant : t -> int
    end                             end
  send (dyn CompteurPair.suivant (CompteurPair.début) at CompteurPair.t)
```

P7B =
```
  module CompteurTriple =
    struct                          sig
      type t = int                    type t
      let début = 0           :       val début : t
      let suivant x = x + 3           val suivant : t -> t
      let courant x = x               val courant : t -> int
    end                             end
  print_int (CompteurPair.courant
               (CompteurTriple.suivant (coerce receive() at CompteurTriple.t)))
```

Les types `CompteurPair.t` et `CompteurTriple.t` n'ont pas les mêmes invariants ; le code ci-dessus affiche 5 sur *B*. L'abstration est cassée.

Il faut comparer les implémentations :

P8A =
```
  module CompteurPair =
    struct                          sig
      type t = int                    type t
      let début = 0           :       val début : t
      let suivant x = x + 1           val suivant : t -> t
      let courant x = 2 * x           val courant : t -> int
    end                             end
  send (dyn CompteurPair.suivant CompteurPair.début at CompteurPair.t)
```

P8B =
```
  module CompteurPair =
    struct                          sig
      type t = int                    type t
      let début = 0           :       val début : t
      let suivant x = x + 2           val suivant : t -> t
      let courant x = x               val courant : t -> int
    end                             end
  print_int (CompteurPair.courant (coerce receive() at CompteurPair.t))
```

Les implémentations de `CompteurPair` sur *A* et *B* ont la même signature ; elles utilisent le même type représentation du type abstrait, et sont indistinguables opérationnellement, au sens où toute succession d'opérations qui produit au final un entier renvoie le même entier que l'on ait choisi l'une ou l'autre des deux implémentations. Pourtant autoriser la désérialisation dans cet exemple conduirait *B* à afficher 1 comme valeur d'un compteur pair... Cet exemple illustre le fait qu'il ne suffit pas que les représentations aient le même type, il faut aussi qu'elles aient la même sémantique. Notons que cet exemple illustre également le danger qu'aurait une conversion automatique dans l'exemple *P4* : on n'obtiendrait pas le « même » compteur selon l'implémentation de `CompteurPair` sur *B*.

II.2.2.6 Un problème indécidable

Varions encore l'exemple : cette fois-ci, les deux implémentations de `CompteurPair`, si elles ne sont pas identiques, sont équivalentes en ce que l'on peut les mélanger au sein d'un même programme.

P9A=

```
module CompteurPair =
  struct                        sig
    type t = int                  type t
    let début = 0           :     val début : t
    let suivant x = x + 2         val suivant : t -> t
    let courant x = x             val courant : t -> int
  end                           end
send (dyn CompteurPair.début at CompteurPair.t)
```

P9B=

```
module CompteurPair =
  struct                        sig
    type t = int                  type t
    let début = 0           :     val début : t
    let suivant x = 2 + x         val suivant : t -> t
    let courant x = x             val courant : t -> int
  end                           end
print_int (CompteurPair.courant (coerce receive() at CompteurPair.t))
```

Rien ne s'oppose donc à autoriser la désérialisation dans ce cas. Cependant, dans le cas général, il faudrait tester l'équivalence de deux programmes, ce qui est bien entendu indécidable. Cet exemple sera donc accepté si et seulement si la commmutativité de l'addition est détectée par le système.

II.2.3 De l'importance, ou non, des noms

II.2.3.1 Noms des variables locales

Si l'on n'a changé que les noms des variables locales, autrement dit si les modules sont identiques sur les deux machines à alpha-conversion près, nous les considérerons encore comme équivalents.

P10A =

```
module CompteurPair =
  struct                        sig
    type t = int                  type t
    let début = 0            :    val début : t
    let suivant x = x + 2         val suivant : t -> t
    let courant x = x             val courant : t -> int
  end                           end
send (dyn CompteurPair.début at CompteurPair.t)
```

√ *P10B* =

```
module Compteur_pair =
  struct                        sig
    type t = int                  type t
    let début = 0            :    val début : t
    let suivant c = c + 2         val suivant : t -> t
    let courant c = c             val courant : t -> int
  end                           end
print_int (Compteur_pair.courant (coerce receive() at Compteur_pair.t))
```

II.2.3.2 Noms des points d'entrée

En revanche, si les noms des points d'entrée du module changent, nous ne les considérerons plus comme équivalents. Nous exigerons en effet que les deux modules aient la même signature.

P11A =

```
module CompteurPair =
  struct                        sig
    type t = int                  type t
    let début = 0            :    val début : t
    let suivant x = x + 2         val suivant : t -> t
    let courant x = x             val courant : t -> int
  end                           end
send (dyn CompteurPair.début at CompteurPair.t)
```

× *P11B* =

```
module EvenCouter =
  struct                        sig
    type t = int                  type t
    let start = 0            :    val start : t
    let next x = x + 2            val next : t -> t
    let current x = x             val current : t -> int
  end                           end
print_int (Compteur_pair.current (coerce receive() at Compteur_pair.t))
```

II.2.3.3 Nom(s) du module

Dans les exemples que nous avons présenté jusqu'à présent, lorsque deux types abstraits étaient compatibles, ils avaient le même nom. Pourtant notre argumention sur la compatibilité n'était pas basée sur les noms des modules ; si le même code et la même signature sont utilisée sur *A* et *B*, peu importe que l'une utilise le nom `CompteurPair` et l'autre `Compteur_pair`.

P12A=

```
module CompteurPair =
  struct                        sig
    type t = int                  type t
    let début = 0           :     val début : t
    let suivant x = x + 2         val suivant : t -> t
    let courant x = x             val courant : t -> int
  end                           end
send (dyn CompteurPair.début at CompteurPair.t)
```

P12B= ✓

```
module Compteur_pair =
  struct                        sig
    type t = int                  type t
    let début = 0           :     val début : t
    let suivant x = x + 2         val suivant : t -> t
    let courant x = x             val courant : t -> int
  end                           end
print_int (Compteur_pair.courant (coerce receive() at Compteur_pair.t))
```

Dans certains cas, le nom du module importe. Par exemple, voici un (squelette de) module manipulant des montants dans une monnaie.

```
module Euros =
  struct                       sig
    type t = int                 type t
    let fabrique x = x      :    val fabrique : int -> t
    let ajoute x y = x + y       val ajoute : t -> t -> t
    let valeur x = x             val valeur : t -> int
  end                          end
```

Il paraîtrait mal venu de mélanger des `Euros` et des `Dollars`, ce que l'exemple *P12* tendrait à permettre. D'un point de vue théorique, en revanche, manipuler des noms est peu attractif. En ML, on considère habituellement les constructions `let`, `type`, `module`, etc., comme liantes, et les noms liés peuvent être modifiés. Nous considérons cette propriété comme trop fondamentale pour être brisée. Les noms de champs de module échappent à cette règle ; c'est plutôt que les champs de modules servent au fur et à mesure de leur définition de noms liés.

Les noms de modules peuvent également être significatifs en Objective Caml lorsque ces modules sont des unités de compilation. Or le nom d'une unité de compilation sert à fabriquer des noms de fichiers, et des contraintes techniques peuvent conduire à renommer ces fichiers et donc à modifier le nom du module[3]. Donner une importance sémantique à ce nom semble à cet égard dangereux. Cela empécherait en outre de renommer un module lorsque l'on veut combiner dans un même programme deux bibliothèques pour lesquelles les auteurs ont choisi le même nom.

Avantage théorique et avantage pratique, donc, à ne pas considérer le nom du module comme significatif. Qu'en est-il alors de notre exemple monétaire ? Le programmeur peut facilement exprimer l'importance du nom en l'incluant au sein du module, par exemple :

```
module Euros =
  struct                        sig
    type t = int                  type t
    let fabrique x = x      :     val fabrique : int -> t
    let ajoute x y = x + y        val ajoute : t -> t -> t
    let valeur x = x              val valeur : t -> int
    let nom = "EUR"               val nom : t -> string
  end                           end
```

Le module `Dollar` n'a pas la même implémentation, puisque son champ `nom` vaut `"USD"`. (La section I.1.2.3 présente d'autres solutions.)

II.3 Empreintes

Dans cette section, nous présentons une notion d'*empreinte* de module, qui définit une notion de compatibilité entre modules de programmes compilés séparément de telle sorte à obtenir le comportement désiré dans les exemples présentés à la section II.2. Nous nous plaçons toujours dans le cadre d'un langage restreint, où tous les programmes ont la forme suivante : une suite de déclarations de modules dont tous les champs sont des valeurs immédiates (fonctions ou constantes), puis un programme principal.

II.3.1 Empreintes de modules et désérialisation

II.3.1.1 Concept d'empreinte

À la lumière des exemples qui précèdent, nous allons présenter une notion d'**empreinte** de module, de telle sorte que :

1. si deux modules ont la même empreinte, alors ils peuvent être utilisés de façon interchangeable, y compris mélangés au sein d'un même programme ;
2. il est possible de définir indépendamment (par exemple, sur deux machines différentes) deux modules ayant la même empreinte.

Nous considérerons alors deux types comme compatibles lorsqu'ils sont les champs de même nom de modules de mêmes empreintes.

Au niveau théorique, nous définirons une fonction `hash` dont le domaine est les scellages (c'est-à-dire que cette fonction prend en argument un module et une signature pour ce module) et l'image est l'espace des empreintes (qui reste à préciser). Ainsi `hash(`M `:` S`)` est l'empreinte du module M scellé à la signature S. Avec une telle définition, la propriété 2 est automatiquement vérifiée.

[3]En Objective Caml, l'espace des noms d'unités de compilation est plat, ce qui oblige quelquefois à renommer ou empaqueter certains modules lorsque l'on met ensemble plusieurs bibliothèques. Le problème est plus anecdotique dans un langage comme Java où l'espace des noms de paquets est hiérarchisé.

Avant de discuter de la définition précise de `hash`, examinons son utilisation. La notion d'empreinte est motivée par la vérification de type lors de la désérialisation. Nous définissons donc une nouvelle forme de types, les **types d'empreinte**. Un type d'empreinte a la forme ***h*.t** où ***h*** est une empreinte. Les types d'empreinte ne sont pas écrits directement par le programmeur ; ils proviennent de la compilation d'annotations de type dans la sérialisation et la désérialisation. Ainsi,

```
module A = M : S
... send(dyn ... at A.t) ...
... coerce receive() at A.t ...
```

est transformé en

```
module A = M : S
... send(dyn ... at hash(M : S).t) ...
... coerce receive() at hash(M : S).t ...
```

et la vérification de type à la désérialisation compare des empreintes et non plus des noms de modules.

Nous pouvons évaluer des candidats `hash` suivant plusieurs critères :
- correction : il est impératif que la propriété 1 soit vérifiée ;
- expressivité : autant que possible, la réciproque de la dite propriété 1 doit être vérifiée ;
- compacité : les empreintes sont stockées ou transmises dans les valeurs sérialisées, elles ne doivent donc pas être trop volumineuses ;
- efficacité : le calcul d'une empreinte, et la comparaison de deux empreintes doivent demander une quantité raisonnable de ressources ;
- simplicité : la notion doit être compréhensible par le programmeur ; nous souhaiterons aussi en prouver la correction.

II.3.1.2 Définition

Nous avons vu que l'intérêt des types abstraits est qu'ils permettent de caractériser les modules au delà de leur interface, promettant des invariants que le langage ne sait exprimer ; l'exemple ***P8*** montre qu'il est en outre nécessaire que les types aient la même représentation avec la même sémantique. Une manière sûre d'assurer ce dernier point est d'exiger que les modules aient le même code. Nous proposons donc comme définition : *l'empreinte d'un module est son code*.

Évaluons brièvement cette définition par rapport aux critères énoncés en II.3.1.1. La correction est évidente, puisque deux modules incompatibles le sont forcément parce que leur code est différent. L'expressivité est réduite à sa plus simple expression — l'indécidabilité de l'équivalence de code laisse toutefois dès le départ peu d'espoir. Pour ce qui est de la compacité, remarquons que la seule opération que nous effectuons sur des empreintes est un test d'égalité ; aussi pouvons-nous en pratique nous contenter d'une empreinte cryptographique du code[4]. Nous discuterons de ce point et de l'efficacité en II.3.3 (plus particulièrement II.3.3.3).

Précisons maintenant ce que veut dire pour deux modules que d'avoir le même code. Une première idée serait de considérer comme « code » d'un module le texte source d'une unité de compilation. Cette définition a l'avantage de la simplicité, et l'empreinte peut être définie de façon extrinsèque au langage de programmation utilisée. En revanche, cette définition est très fragile, ne résistant pas à la réindentation d'un fichier ou simplement à son transfert sur un système utilisant

[4] C'est bien sûr de là que vient notre terminologie.

un encodage différent des sauts de lignes. Un remède immédiat est de rendre tous les blancs canoniques (et de supprimer les commentaires). De manière quasiment équivalente, nous considérerons comme le « code » d'un module son *arbre de syntaxe abstraite* ; en fait, nous allons un peu plus loin, en accord avec l'exemple *P10*, et définissons : *le code d'un module est son arbre de syntaxe abstraite à alpha-équivalence près.*

Ainsi, d'un point de vue théorique, nous pouvons simplement définir `hash(`*M* `:` *S*`)` comme une nouvelle forme de module ; l'équivalence de deux empreintes est modélisée simplement par l'égalité des termes, c'est-à-dire que `hash(`*M* `:` *S*`)` = `hash(`*M'* `:` *S'*`)` si et seulement si *M* = *M'* et *S* = *S'*, sachant que nous raisonnerons systématiquement sur des termes à alpha-équivalence près.

Notre notion d'empreinte est ici à la frontière entre la syntaxe et la sémantique. Le côté syntaxique permet de comparer des empreintes facilement ; le côté sémantique nous permettra de raisonner sur des empreintes par des méthodes habituelles. (Par la suite, nous étendrons le langage avec des constructions de modules plus puissantes ; ceci introduira des égalités non syntaxiques sur les empreintes.)

II.3.1.3 Récapitulation des exemples

Récapitulons le comportement de la désérialisation pour les exemples de la section II.2. Dans l'exemple *P3*, les machines *A* et *B* ont des modules `CompteurPair` qui ont le même code, donc la même empreinte *h* ; la désérialisation d'une valeur de type *h*`.t` au type *h*`.t` est acceptée. Dans les exemples *P4* et *P5*, un type d'empreinte est comparé à un type concret, la désérialisation est donc refusée. Dans les exemples *P6*, *P7*, *P8* et *P9*, les modules définis de part et d'autre n'ont pas le même code, la désérialisation compare deux types d'empreintes différentes *h*`.t` et *h'*`.t` et échoue. Dans l'exemple *P10*, le code est le même à alpha-équivalence près, donc la désérialisation est acceptée ; dans l'exemple *P11*, il n'y a pas alpha-équivalence et la désérialisation est refusée ; enfin dans l'exemple *P12* les modules `CompteurPair` et `Compteur_pair` ont le même code donc la même empreinte et la désérialisation réussit.

II.3.1.4 Empreinte et signature

Bien que nous parlions habituellement d'empreinte d'un module, nous avons défini l'empreinte d'un scellage, `hash(`*M* `:` *S*`)`, c'est-à-dire que la signature est prise en compte dans l'empreinte. Ce n'est pas critique pour notre usage de vérification de type dynamique : tout ce qui nous importe est le comportement du code, qui n'est pas influencé par la signature. Plusieurs arguments se rejoignent pour motiver ce choix :

- le choix d'une signature pour un module dont le code est fixé est assez limité : on ne peut que cacher certains champs, ou cacher certaines de leurs propriétés (rendre un type abstrait, pour rester dans le cadre des fonctionnalités que nous avons déjà mentionnées) ;
- dès lors, si deux programmeurs se sont mis d'accord pour utiliser le même code, il leur est facile de s'accorder aussi sur la signature ;
- il en résulte que donner deux signatures différentes au même code a selon toute vraisemblance un caractère délibéré, que le système de types devrait respecter ;
- sur le plan technique, ceci permet de confiner la création d'empreinte à une construction du langage, le scellage, plutôt que de devoir traiter systématiquement toutes les constructions présentes dans le langage de programmation précis étudié.

Remarquons que si un type devient concret, l'empreinte n'est plus utilisée : ce n'est plus un nouveau type mais une abréviation de type, utilisable de façon interchangeable avec l'original.

II.3.2 Dépendances

Jusqu'à présent, les exemples que nous avons présentés définissent un unique module. Étudions maintenant ce qui se passe lorsqu'un programme définit plusieurs modules, et en particulier lorsque le module dont on prend l'empreinte dépend d'autres modules. Sur le plan théorique, ceci signifie que l'on s'interroge sur la définition de l'empreinte d'un module ayant des variables libres.

II.3.2.1 Un exemple

Le programme de la figure II.1 suivant définit utilise le module `CompteurPair` dont nous avons maintant l'habitude pour implémenter une table de symboles. D'inspiration lispienne, notre table de symbole associe à un nom deux données, une valeur et une fonction. Le passage du nom aux données passe par une étape intermédiaire, un entier appelé symbole[5]. L'implémentation de la table de symboles utilise un compteur pair pour assurer que deux noms différents sont toujours associés à deux entiers différents ; la parité du compteur permet de stocker la valeur associée dans la case `x` et la fonction associée dans la case `x`+1. Le module `SM` (censé préexister) définit une table d'association des chaînes de caractères vers un type arbitraire (ici `int`), il peut être défini en Objective Caml comme `Map.Make(String)` ; le module `IM` est analogue avec comme domaine `int`.

La signature du module `TableSymboles` cache le fait que les symboles sont des entiers ; un symbole doit donc être créé via la fonction `entre`. L'implémentation de la table est aussi cachée sous le type abstrait `t`.[6]

Pour que deux tables soient compatibles, il faut qu'elles utilisent le même code pour le module `TableSymboles` ; il faut aussi que les implémentations de `CP` soient compatibles. L'empreinte du module `TableSymboles` doit donc dépendre de l'empreinte de `CP`. Ainsi, si la désérialisation était acceptée dans le programme distribué suivant, `"toto"` utiliserait les entrées numéro 0 et 1 de la table, et `"tutu"` utiliserait les entrées numéros 1 et 2 au lieu de 2 et 3 comme prévu, puisque la table transmise aurait la forme `(1, ..., ...)`.

P13A=

```
module CP = ⟨comme sur A dans l'exemple P8⟩
module TableSymboles = ⟨comme ci-dessus⟩
send (dyn fst (TableSymboles.entre TableSymboles.vide "toto") at TableSymboles.t)
```

P13B=

```
module CP = ⟨comme sur B dans l'exemple P8⟩
module TableSymboles = ⟨comme ci-dessus⟩
TableSymboles.entre (coerce receive() at TableSymboles.t) "tutu"
```

Pour calculer l'empreinte de `TableSymboles`, nous proposons l'algorithme suivant : nous commençerons par calculer l'empreinte *h* de `CP`, puis remplacerons les occurences de `CP` dans le code de `TableSymboles` par *h*, avant de calculer l'empreinte du code obtenu. Ainsi, nous effectuons les substitutions, obtenant un terme clos (contenant des empreintes) dont nous savons prendre l'empreinte.

[5] l'intérêt étant que les opérations sur les symboles, notamment le test d'égalité, sont plus efficaces que sur leurs noms.

[6] Avec le code présenté ici, le typage n'empêche pas un symbole d'être utilisé avec une autre table que celle pour laquelle il a été créé.

```
module CP =
  struct                          sig
    type t = int                    type t
    let début = 0                   val début : t
    let suivant x = x + 2     :     val suivant : t -> t
    let courant x = x               val courant : t -> int
  end                             end
type valeur = ...
module TableSymboles = struct
    type symbole = int
    type t = CP.t * int SM.t * valeur IM.t
    let vide = (CP.début, SM.empty, IM.empty)
    let entre (c, tn, tv as t) nom =
      try (t, SM.find nom tn) with Not_found ->
        let c' = CP.suivant c in ((c', SM.add nom c tn, tv), CP.courant c')
    let valeur (c,tn,tv) x = IM.find x tv
    let fonction (c,tn,tv) x = IM.find (x+1) tv
    let donne (c,tn,tv) x v = (c, n, IM.add x v tv)
    let donnef (c,tn,tv) x v = (c, n, IM.add (x+1) v tv)
  end : sig
    type symbole
    type t
    val vide : t
    val entre : t -> string -> t * symbole
    val valeur : t -> symbole -> valeur
    val fonction : t -> symbole -> valeur
    val donne : t -> symbole -> valeur -> t
    val donnef : t -> symbole -> valeur -> t
  end
```

FIG. II.1 – Module `CP` et `TableSymboles`

II.3.2.2 Définition

Généralisons l'algorithme que nous avons donné pour l'exemple ci-dessus. Considérons un programme donné sous la forme d'une suite de modules scellés, suivi d'une expression (le programme principal) :

module A_1 = struct ...M_1... end : sig ...S_1... end
...
module A_n = struct ...M_n... end : sig ...S_n... end
e

Notons h_i l'empreinte du module A_i. Nous calculons successivement les empreintes $h_1, ..., h_n$ de la manière suivante :

$$h_i = \texttt{hash}(\sigma_i(M_i) : \sigma_i(S_i)) \qquad \text{où } \sigma_i \text{ est la substitution qui à tout } A_j \text{ pour } j < i \text{ associe } h_j$$

Autrement dit, nous remplaçons successivement les noms de modules par leurs empreintes.

Enfin, e est compilée en $\sigma_{n+1}(e)$, c'est-à-dire e dans laquelle les noms de modules sont remplacés par leurs empreintes respectives. Ceci précise et généralise la transformation décrite à la section II.3.1.1.

II.3.2.3 Dépendance faible

Dans l'exemple de la section II.3.2.1, nous avons discuté de la compatilibité pour le type `TableSymboles.t`. La situation est différente pour le type `symbole`, dont la représentation ne fait pas appel à `CP.t` : seule la sémantique « externe » de `CP` importe, et non son implémentation. Cependant toute valeur de type *h*`.symbole` peut être utilisée comme argument, par exemple, de `valeur`, avec pour autre argument une valeur de type *h*`.t` ; donc le typage de `symbole` doit être aussi exigeant que celui de `t`.

De manière générale, considérons deux définitions de modules scellés successives.

```
module A = struct type t = T ... end : sig type t ... end
module B = struct type t = T' ... end : sig type t ... end
```

Dans le cas où *T'* n'utilise pas `A.t`, on pourrait faire que l'empreinte de `B` ne change pas si l'implémentation de `A` (y compris le type représentation de `A.t`) change, pourvu que la sémantique soit conservée. Cependant l'équivalence observationnelle est indécidable, et une de nos motivations pour choisir de tester la très contraignante alpha-équivalence du code était que celle-ci est décidable. Aussi ne proposerons-nous pas de traiter ce cas différemment.

II.3.3 Implémentation

Jusqu'à présent, nous avons traité les empreintes comme des objets théoriques. Dans cette section, nous allons discuter du choix d'une représentation pratique des empreintes ; ce choix doit permettre au moins deux opérations : le calcul de l'empreinte par le compilateur, et la comparaison des empreintes lors de l'évaluation de `coerce ... at ...`.

Comme nous raisonnons toujours à renommage des variables près, nous supposerons dans cette que tous les arbres de syntaxes que nous manipulons sont eux-mêmes définis à alpha-conversion près, soit qu'ils utilisent une représentation anonyme comme la notation de de Bruijn, soit qu'une conversion préalable vers des noms canoniques a déjà été faite. Nous supposerons également que chaque lieur du programme lie un nom différent (la « convention de Barendregt »).

II.3.3.1 Réification

Principe Une première manière de coder les empreintes est de définir un type algébrique capable de représenter toutes les constructions du langage, et d'exprimer les empreintes comme des valeurs de ce type. Un tel type est forcément présent dans le compilateur lui-même si le compilateur est auto-amorcé (*bootstrapped*) ; en Objective Caml, un tel type est accessible aux utilisateurs via Camlp4. La comparaison de deux empreintes peut alors être fournie comme une fonction de bibliothèque, qui compare simplement deux arbres de syntaxe.

Un même mécanisme permet d'implémenter toute la vérification de type dynamique à peu de frais : il suffit de représenter un type par une valeur qui représente son arbre de syntaxe ; on **réifie** le type. Notons que la réification du type pourrait se faire indépendamment de la production d'empreintes : cette dernière est invoquée lorsque la réification du type atteint un nom de module (construction *A*`.t`), et d'autres solutions seraient possibles, comme de conserver le nom (ce qui donnerait une représentation du type valide seulement dans le même programme).

Typage dynamique Esquissons ce mécanisme de compilation des constructions de typage dynamique que nous avons proposées. Nous supposons l'existence d'un type paramétré `'a reified_type`, tel que pour tout type (monomorphe) T, T `reified_type` contienne une unique valeur, la réification du type T que nous noterons $\mathrm{reify}(\mathrm{T})$ (le type T `reified_type` n'est pas définissable en ML seul). Donnons quelques cas de la définition de reify (cette opération est paramétrée par un environnement H qui associe à chaque nom de module l'empreinte dudit module) :

$$
\begin{aligned}
\mathrm{reify}_{\mathsf{H}}(\mathsf{T}_1 \texttt{ * ... * } \mathsf{T}_n) &= \texttt{Ttuple([}\mathrm{reify}_{\mathsf{H}}(\mathsf{T}_1)\texttt{, ... , }\mathrm{reify}_{\mathsf{H}}(\mathrm{T}_n)\texttt{])} \\
\mathrm{reify}_{\mathsf{H}}(\mathsf{T}_1 \texttt{ -> } \mathsf{T}_2) &= \texttt{Tarrow(}\mathrm{reify}_{\mathsf{H}}(\mathsf{T}_1)\texttt{, }\mathrm{reify}_{\mathsf{H}}(\mathrm{T}_2)\texttt{)} \\
\mathrm{reify}_{\mathsf{H}}(\mathit{A}\texttt{.}\mathit{name}) &= \texttt{Thash\_field(}\mathsf{H}(\mathit{A})\texttt{, "}\mathit{name}\texttt{")} \\
&\ \ \ldots
\end{aligned}
$$

Nous compilons alors les constructions `dyn` ... `at` ... et `coerce` ... `at` ... vers des appels de fonction de bibliothèque suivant le schéma suivant ($\Rightarrow_{\mathsf{H}}$ signifie « est compilé vers ... dans l'environnement H ») :

$$
\begin{aligned}
\texttt{dyn } e \texttt{ at } \mathsf{T} &\quad \Rightarrow_{\mathsf{H}} \quad \texttt{Dyntypes.dynamicf (}\mathrm{reify}_{\mathsf{H}}(\mathsf{T})\texttt{) (}e\texttt{)} \\
\texttt{coerce } e \texttt{ at } \mathsf{T} &\quad \Rightarrow_{\mathsf{H}} \quad \texttt{Dyntypes.coercef (}\mathrm{reify}_{\mathsf{H}}(\mathsf{T})\texttt{) (}e\texttt{)}
\end{aligned}
$$

Voici une esquisse d'implémentation de la bibliothèque `Dyntypes`, dans un pseudo-langage non typé ressemblant à ML.

```
module Dyntypes = struct
  type t = ⟨non implémentable en ML seul⟩
  let dynamicf rtt x = (rtt, x)
  let coercef expected_rtt (actual_rtt, x) =
    if expected_rtt = actual_rtt then x else failwith "Dynamic.coercef"
end : sig
  type t
  val dynamicf : 'a reified_type -> 'a -> t
  val coercef : 'a reified_type -> t -> 'a
end
```

Empreintes Passons au calcul des empreintes proprement dit, c'est-à-dire à la définition de H. Le compilateur maintient au départ (pas forcément exactement sous cette forme) un environnement Γ qui associe à chaque nom de module A son code M et sa signature S, soit $\Gamma(\mathit{A}) = (\mathsf{M}, \mathsf{S})$. Nous posons donc $\mathsf{H}(\mathit{A}) = \mathrm{Reify}(\Gamma(\mathit{A}))$, où l'opération Reify est l'analogue sur les expressions et signatures de modules de l'opération Reify. À partir de la représentation interne du source du module (M, S), l'opération Reify génère du code qui construit une valeur du type `imprint`, le type des empreintes de module. Les modules peuvent contenir des expressions et des types, il faut donc définir Reify sur ces sortes aussi. Donnons quelques exemples typiques :

$$
\begin{aligned}
\mathrm{Reify}\begin{pmatrix}\texttt{struct } \mathsf{c}_1 \ldots \mathsf{c}_n \texttt{ end} \\ \texttt{: sig } \mathsf{C}_1 \ldots \mathsf{C}_N \texttt{ end}\end{pmatrix} &= \texttt{Tmod\_seal}\begin{pmatrix}\texttt{Tmod\_structure [}\mathrm{Reify}(\mathsf{c}_1)\texttt{; ...; }\mathrm{Reify}(\mathsf{c}_n)\texttt{],} \\ \texttt{Tmty\_signature [}\mathrm{Reify}(\mathsf{C}_1)\texttt{; ...; }\mathrm{Reify}(\mathsf{C}_N)\texttt{]}\end{pmatrix} \\
\mathrm{Reify}(\texttt{(}e_1\texttt{, ..., }e_n\texttt{)}) &= \texttt{Texp\_tuple(}\mathrm{Reify}(e_1)\texttt{, ..., }\mathrm{Reify}(e_n)\texttt{)} \\
\mathrm{Reify}(\mathit{A}) &= \mathsf{H}(\mathit{A})
\end{aligned}
$$

II.3.3.2 Représentation compacte : compression

Un inconvénient de l'implémentation de la sérialisation typée décrite ci-dessus est que les types peuvent être volumineux, ce qui résulte en une grosse quantité de données à transmettre. Un objectif raisonnable serait que l'information de type soit plus petite que la valeur transmise proprement dite. Or, s'il est relativement rare en ML que le type d'une valeur soit significativement plus gros que la valeur elle-même, ce n'est plus le cas si chaque occurence d'un nom de module est remplacé par l'empreinte de celui-ci, c'est-à-dire par une représentation complète de son source.

Une première mesure évidente pour réduire la taille des données échangées est de partager autant que possible les sous-termes identiques du type envoyé, et notamment de n'envoyer qu'une fois chaque empreinte. La bibliothèque de sérialisation non typée d'Objective Caml conserve le partage présent dans la représentation mémoire de son argument ; si l'on utilise un autre système qui ne prévoit pas cela, on peut rendre le partage explicite, par exemple en conservant les noms des modules et en envoyant avec le type une table d'association des noms de modules vers les empreintes (en veillant soit à utiliser des noms canoniques, soit à ignorer ces noms lors du test d'égalité des types).

Plutôt que de chercher à obtenir un partage maximal dans la représentation du type, on peut utiliser une méthode de compression généraliste — la détection de sous-séquences répétées est une propriété courante des algorithmes de compression. L'opportunité de réaliser une part plus ou moins importante de la compression par le partage implicite, le partage explicite, et la compression généraliste dépend finement du détail de ces opérations ; nous ne rentrerons pas ici plus avant dans ces détails.

II.3.3.3 Représentation compacte : empreintes cryptographiques

Dans le pseudo-code des opérations de sérialisation et de désérialisation typées donné en II.3.3.1, la seule opération qui est effectuée sur les représentations de types est le test d'égalité dans `coercef`. Dès lors, plutôt que de comprimer et décomprimer la représentation de type transmise, on peut comprimer chaque représentation indépendamment, et comparer les représentations comprimées ; ceci est possible à condition que la compression soit déterministe (c'est-à-dire que comprimer deux objets égaux produise toujours la même suite de bits). L'avantage de cette modification est qu'il n'est plus nécessaire de disposer d'un algorithme de décompression.

En pratique, il existe des algorithmes de compression non inversibles : les fonctions de hachage cryptographiques. Il s'agit bien de fonctions mathématiques, donc déterministes. Le résultat a une taille fixe petite : 16 octets pour MD5, 20 octets pour SHA-1, pour prendre deux fonctions courantes. Bien sûr, il peut exister des collisions, c'est-à-dire que deux arguments soient hachés vers la même somme de contrôle ; mais le principe des fonctions de hachage est justement que la probabilité d'une collision est négligeable.

La somme de contrôle de la représentation du type peut être calculée par le compilateur. On obtient donc un mécanisme très efficace, dont le coût à l'exécution est une constante pour chaque sérialisation et chaque désérialisation. (Le coût à la compilation est en tout état de cause linéaire en la taille du programme.) Un inconvénient de ce mécanisme est qu'il bloque l'intégration de la désérialisation avec un mécanisme d'analyse du type (voir la section I.3.2.4).

Un compromis qui semble intéressant est de transmettre une représentation analysable du type, et de ne calculer une somme de contrôle que pour les empreintes de modules. Ainsi, l'empreinte d'un module est une empreinte cryptographique, de petite taille, efficace à transmettre et à comparer.

II.4 Autres utilisations des empreintes

II.4.1 Analyse dynamique de type

II.4.1.1 Introduction

La construction `coerce` x `at` T renvoie la valeur sous-jacente à x si celle-ci a le type T, et lève une exception `Coerce_failure` sinon. Son utilisation suppose que le code qui reçoit la valeur sérialisée sait à l'avance quel est son type. C'est le cas lorsque la sérialisation est utilisée au sein d'un unique programme réparti sur plusieurs machines, la vérification de type à la désérialisation servant juste à s'assurer que ce sont bien des composants du même programme qui communiquent. Mais de nombreux scénarios dépassent ce cadre ; nous allons en étudier quelques-uns, et proposer au fur et à mesure des généralisations de notre construction de désérialisation.

II.4.1.2 Du test au filtrage

Dans un environnement réparti, il peut être difficile de déployer instantanément la nouvelle version d'un logiciel. Il peut donc coexister à un instant donné plusieurs versions d'un même logiciel. Considérons par exemple comme d'habitude deux machines A et B, chacune exécutant une version donnée d'une composante d'un logiciel, désignées respectivement par PA_i et PB_j.

P14A=

```
module A = struct Mi end : sig type t  val x : t  val f : t -> int end
send (dyn A.x at A.t)
```

P14B=

```
module A = struct Mj end : sig type t  val x : t  val f : t -> int end
print_int (A.f (coerce receive() at A.t))
```

Si le code du module M a changé entre les versions i et j du logiciel, la désérialisation échoue. On peut facilement modifier le programme qui désérialise pour assurer une compatibilité arrière ; le programme *P14B* modifié a la forme suivante :

```
module A_1 = struct M1 end : sig type t  val x : t  val f : t -> int end
...
module A_j = struct Mj end : sig type t  val x : t  val f : t -> int end
let n =
  let v = receive() in
  try A_1.f (coerce v at A_1.t) with Coerce_failure ->
  ...
  try A_j.f (coerce v at A_j.t) with Coerce_failure ->
  failwith "Version de A inconnue"
in print_int n
```

Le logiciel ainsi modifié fonctionne dès lors que $i \leqslant j$.

Traiter le cas où $i > j$ demande une machinerie plus compliquée : soit A et B doivent négocier pour s'accorder sur une version supportée par les deux parties, soit A doit envoyer plusieurs valeurs sérialisées, à chacune des versions supportées (seule option lorsque A écrit sur un moyen de stockage persistent que B relit plus tard).

Même dans ce cas où i > j, la construction simple de désérialisation dont nous disposons suffit à permettre la compatibilité arrière. La chaîne de tests levant en cas d'échec une exception sitôt rattrapée manque cependant d'élégance ; elle suggère qu'une construction proposant plusieurs éventualités serait plus appropriée. De même qu'en ML une suite de tests `if` est souvent exprimée à l'aide d'un filtrage `match`, on peut définir un filtrage sur les types. C'est ce que nous allons faire dans la suite de cette section.

Dans la syntaxe présentée à la section I.3.2.1, le code de désérialisation ci-dessus s'écrit plus lisiblement :

```
let n =
  typecase receive() of
    v : A_1.t -> A_1.f v
  | ...
  | v : A_j.t -> A_j.f v
  | _ : _ -> failwith "Version de A inconnue"
in print_int n
```

II.4.1.3 Analyse de type

Analysons l'interaction du filtrage de type dynamique (présenté en I.3.2.1) avec les types abstraits. Ce qui fait l'intérêt du filtrage par rapport à une succession de tests est la présence de variables ; il s'agit donc d'étudier quelles variables peuvent apparaître avec la présence de types abstraits. Deux pistes s'offrent à nous :

- autoriser certaines variables à être *bornées*, c'est-à-dire à ne correspondre qu'à certains types, la restriction sur les types faisant intervenir des types abstraits ;
- ajouter des motifs de module, $\mathfrak{m}$`.nom_de_champ` étant un motif de type lorsque $\mathfrak{m}$ est un motif de module.

Étant donné que les types abstraits sont de toute manière identifiés par les modules dont ils proviennent, nous allons introduire des motifs (et en particulier des variables) de module, et envisager des contraintes dessus.

Les modules, contrairement aux types, sont répartis suivant leurs signatures ; aussi la forme de base de variable de module que nous proposons est-elle contrainte à une signature donnée : nous noterons `exists A :` *S* la quantification d'une variable de module `A` (dont la signature *S* est imposée) là où nous notions `exists 'a` la quantification d'une variable de type. Notons que « il existe un type `'a` tel que ... » peut aussi s'écrire « il existe un module `A` de signature `sig type t end` tel qu'en posant `'a = A.t` on ait ... » ; l'existence de variables de type n'est donc plus théoriquement nécessaire en présence de variables de module, et nous considérerons par la suite `'a` comme une abréviation de (informellement parlant) `(A : sig type t end).t`.

Examinons un exemple de filtrage sur un type de module. Dans la section II.4.1.2, nous envisagions la cohabitation de composantes de plusieurs versions successives d'un même logiciel. Une autre situation courante est la cohabitation de plusieurs logiciels différents, typiquement un système client-serveur ou pair à pair pour lequel plusieurs implémentations coexistent, partageant un protocole commun. Dans le cadre de notre exemple, la signature du module `A` représente le protocole commun, tandis que le code de ce module reflète les choix d'implémentations spécifiques à chaque intervenant. Le fait que le type `t` soit abstrait dans la signature signifie que le protocole n'impose pas une représentation précise des données, seulement leur sémantique. Dans cette situation, l'auteur du programme qui tourne sur *B* ne peut plus énumérer les versions possibles du module `A`, l'implémenteur sur la machine *A* étant libre d'en inventer une nouvelle. La désérialisation sur *B* prend alors la forme suivante :

```
let n =
  typecase receive() of
    exists A : sig type t  val x : t  val f : t -> int end. v : A.t -> A.f v
  | _ -> failwith "Protocole non respecté"
in print_int n
```

Cet exemple soulève une difficulté que nous n'avons pas encore abordée : dans le membre de droite `A.f v`, nous utilisons la variable de module `A` pour le code qu'elle fournit, et non seulement pour ses types. La difficulté est très nettement perceptible lorsqu'on l'exprime en termes d'empreintes. L'annotation de type dans la valeur sérialisée a la forme `(`*h*`.t)`, où *h* est l'empreinte du module nommé `A` sur la machine *A* ; or suivant la section II.3.3 l'empreinte *h* ne laisse visible que des informations sur les types contenus dans le module : compiler `A.f` en « `h..f` » n'aurait pas de sens.

Deux voies s'offrent alors à nous, pour écrire un programme de cette forme. L'une consiste à revenir sur notre décision de ne laisser paraître que les types des empreintes, soit que l'empreinte laisse accessibles toutes les composantes du module, soit qu'elle permette d'accéder à ces composantes. Cette voie sera abordée plus en détail à la section II.4.2. Une deuxième voie est de faire explicitement transmettre à la machine *A* le module dans lequel *B* a besoin à la fois de types et de code ; c'est de cette deuxième voie que nous allons maintenant discuter.

II.4.1.4 Modules de première classe

Dans les langages de la famille ML, les expressions du langage de base sont évaluées lors de l'exécution du programme, tandis que leurs types sont calculés lors de la compilation ; cette distinction est nommée **séparation des phases** par Harper, Mitchell et Moggi [HMM90] (voir la section I.3.1.1). Elle est possible du fait qu'en ML les calculs sur les expressions et les calculs sur les types sont indépendants.

Les modules forment un langage propre, nettement séparé du langage de base ; ce langage a ses propres variables, ses propres fonctions (les foncteurs), ses marques syntaxiques propres (`struct ... end`). On dit que les modules sont de *seconde classe*. Des extensions de ML avec des modules de *première classe*, qui se fondent dans le langage de base et peuvent notamment être affectés à des variables ordinaires, ont été proposées [HL94, Rus98] ; des restrictions sur la manière dont un type peut dépendre d'un calcul au niveau des expressions sont nécessaires pour maintenir la séparation des phases.

Or la vérification de type dynamique casse justement la séparation des phases (dans un seul sens : des types apparaissent à l'exécution, mais la compilation n'a heureusement pas besoin de calculer sur les termes). Nous pouvons donc proposer des constructions de *sérialisation et désérialisation de module* :

`(dyn` *M* `at` *S*`)`	`: string`	où *M* `:` *S*
`(coerce` *e* `at` *S'*`)`	`:` *S'*	où *e* `: string`

Si *M* est un module ayant la signature *S*, `dyn` *M* `at` *S* est bien une expression du langage de base ; les modules, et par là les types qu'ils contiennent, s'immiscent ainsi dans les calculs à l'exécution. La construction `coerce` *e* `at` *S'* effectue le passage inverse ; comme son pendant dans le monde des expressions, celle-ci peut échouer (levant alors une exception).

Notre exemple de programmes partageant une signature mais pas son implémentation peut alors s'écrire de la manière suivante (avec *S* désignant `sig type t val x : t val f : t -> int end`) :

P15A =

```
module A = struct M end : S
send (dyn A at S)
```

P15B=

```
module A = coerce receive() at S
print_int (A.f A.x)
```

II.4.1.5 Exemple : affichage du type

Nous allons ici étudier un exemple d'analyse dynamique de type : une fonction `show_type` qui affiche une description du type annotant une valeur sérialisée. Une telle fonction est évidemment très utile pour déboguer certains programmes répartis. Nous avions donné un tel exemple (tiré de [ACPP91]) dans la section I.3.2.1 ; notre but ici est de traiter les valeurs d'un type abstrait.

Voici une première proposition, qui n'utilise pas de module sérialisé.

```
let rec show_type = typecase v of
  | int -> "int"
  | string -> "string"
  | exists 'a, 'b. x : 'a * 'b ->
     (show_type (dyn fst x at 'a)) ^ " * " ^ (show_type (dyn snd x at 'b))
  ...
  | exists A : sig val t_name : string end. A.t -> A.t_name
  | exists 'a. 'a -> "<unknown abstract type>"
```

Confrontés à un type abstrait nommé `t`, nous cherchons ici si le module définissant ce type possède un champ `t_name` de type `string`, auquel cas nous considérons que sa valeur est le nom du type `t`. Ce code a un défaut majeur : tous les types abstraits ne s'appellent pas `t` ! Il faudrait pouvoir filtrer sur les noms des champs fournis par le module, or ceux-ci sont contenus dans la signature.

II.4.1.6 Analyse de signature

Nous avions précédemment généralisé l'analyse dynamique de type aux modules en introduisant un filtrage sur les modules. Nous avons ensuite défini une vérification de type dynamique sur les modules. Nous pouvons combiner ces deux idées en effectuant un filtrage sur les signatures. Une construction `signaturecase` permettrait de tester la présence d'un champ dans une signature, et d'analyser le cas échéant son type. Nous laissons sa définition à des travaux futurs.

II.4.2 Négociation de code

II.4.2.1 Problématique

Nous avons introduit les empreintes de modules pour désigner des types. Or l'empreinte d'un module capture non seulement les types qu'il fournit, mais aussi ses valeurs. Ceci suggère que les empreintes peuvent servir, en plus des vérifications de types, à des négociations de présence et d'envoi de code.

Nous avons déjà motivé une telle utilisation à la section II.4.1.3. Le scénario était le suivant : une machine *A* définit un module `A` d'implémentation *M* et de signature *S*, et sérialise une valeur de type `A.t` (abstrait) ; une machine *B* reçoit cette valeur sérialisée en connaissant *a priori* *S* mais pas *M*. Rappelons la structure du code, la désérialisation utilisant un filtrage sur le type (l'exemple *P15* exprime le même scénario à l'aide d'un module de première classe typé dynamiquement). Nous utiliserons comme avant *S* = `sig type t val x : t val f : t -> int end`.

P16A =

```
module A = struct M end : S
send (dyn A.x at A.t)
```

P16B =

```
typecase receive() of exists A : S. v : A.t -> print_int (A.f v)
```

Notons *h* l'empreinte `hash(M : S)`.

Au départ (section II.3.1), nous avions défini l'empreinte d'un module comme étant son code. Le programme *PP16B* est compilé en le programme suivant (donné sous forme de pseudo-code) :

```
typecase receive() of exists h. v : h.t -> print_int (h.f v)
```

Si *h* contient le code du module, écrire `h.f` ne pose aucune difficulté.

II.4.2.2 Stratégies de propagation de code

Dans la section II.3.3, nous avons présenté plusieurs manières d'implémenter les empreintes de manière plus efficace que l'envoi systématique du code. Or si l'empreinte est par exemple réduite à une empreinte cryptographique, une expression comme `h.f` n'a plus de sens.

Une solution simple dans ce cas est de maintenir sur chaque machine une table d'association des empreintes vers le code associé. Ainsi la définition du module `A` sur la machine *A* donne lieu à l'ajout à la table d'empreintes de `A` d'une entrée *h*, valant *M*. Lorsque *A* envoie un message à *B* mettant en jeu *h*, plusieurs possibilités sont envisageables ; listons-en les plus importantes.

Copie systématique : *A* peut joindre au message une copie partielle de sa table d'empreintes. Cette copie doit inclure toutes les entrées indexées par une empreinte contenue dans le message, ainsi que les empreintes dont celles-ci dépendent, récursivement. Lorsque *B* reçoit le message, elle ajoute à sa table les entrées correspondant aux empreintes qu'elle ne connaissait pas.

Envoi sur demande : *A* peut n'envoyer au départ que les empreintes. Dans ce cas, lorsque *B* rencontre une empreinte inconnue, elle envoie une requête à *A* lui demandant le code de cette empreinte.

Accord préalable : Lorsque *A* doit envoyer une empreinte à *B*, elle commence par envoyer un message pour le signaler. La machine *B* répond en indiquant quelles empreintes y sont manquantes, et *A* en envoie la définition.

Notons que quelle que soit la représentation choisie pour les empreintes, celles-ci doivent bien identifier de manière non ambiguë le module sous-jacent. Les différentes méthodes sont donc équivalentes et cohérentes (en particulier, si *B* reçoit le code associé à une empreinte qu'elle a déjà, le code reçu est forcément interopérable avec le code déjà présent dans la table).

La copie systématique a l'inconvénient d'être inefficace dans le cas où *A* et *B* échangent de nombreux messages mettant en jeu une même empreinte *h* : celle-ci est renvoyée à chaque communication. Cette méthode revient en fait en termes de débit de données sur le réseau à considérer l'empreinte comme une donnée structurée dont on effectue un partage maximal au sein de chaque message (*cf.* la section II.3.3.2 ; le découplage en empreinte cryptographique et table de correspondance rend cependant la comparaison des empreintes effectuée par l'opération `coerce` ou `typecase` plus facile).

En revanche, la copie systématique est la seule qui puisse s'appliquer en toute généralité. Une limitation immédiate de toute stratégie nécessitant une négociation entre l'émetteur et le destinataire est qu'elle ne s'applique que dans un scénario distribué, pas dans un scénario persistent où l'« émetteur » n'est en général plus disponible lorsque la valeur est relue depuis le moyen de stockage où elle était consignée.

Même dans un système distribué, la négociation n'est pas forcément possible. La vérification de type et l'envoi sur le réseau sont en effet deux opérations de nature très différente, qui sont souvent découplées. Nous avons d'ailleurs distingué ici les opérations `send` et `receive`, qui transmettent des chaînes de caractères quelconques par le réseau, des opérations `dyn` et `coerce` qui gèrent le typage dynamique. Le code qui effectue la vérification de type peut donc ne pas avoir accès à l'identité de l'envoyeur (*A* dans notre exemple).

Une approche plus lourde, mais qui n'a pas ces limitations, est de constituer un dépôt de code : une base de données au choix centralisée ou répartie associant aux empreintes leur code. Nous ne donnerons pas ici de recommandation générale quant à l'approche à choisir, estimant que ceci dépend fortement de l'application.

II.5 Foncteurs

II.5.1 Problématique

II.5.1.1 Introduction

Nous avons vu à la section II.3.2 un exemple de module `TableSymboles` utilisant un autre module `CP` (compteur pair). La formulation que nous avons donnée permet d'écrire séparément `TableSymboles` et `CP` ; en revanche elle ne permet pas, par exemple, de construire des tables de symboles utilisant des compteurs pairs différents. Si ceci est désiré, on peut en ML définir `TableSymboles` de manière à rendre la paramétrisation par `CP` explicite, c'est-à-dire en faire un *foncteur* `TableSymbolesF` prenant `CP` en argument.

```
module TableSymbolesF = functor (CP : sig ⟨signature de CP⟩ end) -> struct
  ⟨code de TableSymboles⟩
end : sig
  ⟨signature de TableSymboles⟩
end
```

(Nous ne répétons pas le code et les signatures donnés à la section II.3.2.1 page 65.)

Cette définition de `TableSymboles` est typique d'un usage courant des foncteurs, les structures de données paramétriques. Un autre exemple, classique, est une bibliothèque pour des tables d'associations finies utilisant des arbres de recherche équilibrés. Une telle bibliothèque existe en Objective Caml sous le nom `Map`, et a la forme suivante :

```
module Map = struct
  module Make = functor (Ord : sig type t  val compare : t -> t -> int end) ->
    struct ⟨code omis⟩ end  :  sig ⟨signature omise⟩ end
end
```

L'argument `Ord` contient deux parties : le type `t` est le type des clés dans la table d'association ; la fonction `compare` fournit une relation d'ordre total sur les objets de type `t`[7]. Nous avons déjà rencontré `Map.Make` à la section II.3.2.1, où nous utilisions des modules `SM` et `IM` que l'on peut définir ainsi :

```
module SM = Map.Make (String)
module IM = Map.Make (struct type t = int  let compare x y = ... end)
```

Nous adoptons la terminologie habituelle : un **module** est n'importe quel terme du langage des modules. Le terme « **foncteur** » a deux sens (de même que le mot fonction) : il peut designer soit seulement les termes de la forme `functor(...)->...`, soit tous les termes dont la valeur est un foncteur dans le premier sens du mot (nous préciserons notre usage dans les rares cas où ce sera nécessaire). Une **structure** est un module « de base », une simple liste de champs, que l'on écrit sous la forme `struct ... end`.

II.5.1.2 Stratégies

Plaçons-nous à nouveau dans le scénario à deux machines présenté en II.2. Donnons un premier exemple de programme distribué utilisant le foncteur `TableSymbolesF`. Cet exemple présente deux scellages, la structure `CP` et le corps du foncteur `TableSymbolesF` ; le code est le même sur les deux machines, et le module `TS` est construit de la même manière. Nous souhaitons donc que la sérialisation d'une table de symboles soit acceptée.

P17A=

```
module CP = struct ⟨code de CP⟩ end : sig ⟨signature de CP⟩ end
module TableSymbolesF = functor (CP : sig ⟨signature de CP⟩ end) ->
  struct ⟨code de TableSymboles⟩ end   :
  sig ⟨signature de TableSymboles⟩ end
module TS = TableSymbolesF(CP)
let (table, s) = TS.entre TS.vide "toto" in
let table' = TS.donne "toto" (Valeur_entiere 42) in
send (dyn table' at TS.t)
```

P17B=

```
module CP = struct ⟨code de CP⟩ end : sig ⟨signature de CP⟩ end
module TableSymbolesF = functor (CP : sig ⟨signature de CP⟩ end) ->
  struct ⟨code de TableSymboles⟩ end   :
  sig ⟨signature de TableSymboles⟩ end
module TS = TableSymbolesF(CP)
let table = coerce receive() at TS.t in
let (_, s) = TS.entre table "toto" in
let (Valeur_entiere x) = TS.valeur table s in
print_int x
```

Nous pouvons comme auparavant calculer l'empreinte $\mathfrak{h}_{\texttt{CP}}$ du module `CP`. Mais pour désigner le type abstrait `TS.t`, il nous faut définir l'empreinte du module `TS`. Plusieurs approches pour généraliser le concept actuel s'offrent à nous :

[7] `compare` renvoie une valeur négative, 0 ou une valeur positive suivant si ses arguments sont en ordre décroissant, égaux ou en ordre croissant.

- considérer qu'un foncteur est d'un point de vue mathématique une fonction, et définir l'empreinte de `TableSymbolesF` comme la fonction f qui à une empreinte h associe une empreinte h' par un procédé à déterminer : l'empreinte de `TS` est alors $f(h)$;
- considérer que c'est d'une structure que l'on prend l'empreinte, et effectuer d'abord le calcul de la valeur de `TS` avant de calculer son empreinte ;
- considérer qu'un foncteur est un module, et prendre l'empreinte h_{F} de `TableSymbolesF`, ce qui nous amènera à définir l'application de h_{F} à h_{CP}.

Chacune de ces approche conserve un grand degré de flexibilité quant à ce qui nous intéresse au final, à savoir quand deux types sont compatibles. En fait, on peut considérer que la première approche consiste à définir une sémantique dénotationnelle ; la deuxième, une sémantique opérationnelle à grands pas ; la troisième, une sémantique opérationnelle à petits pas. Si une approche donnée peut rendre plus naturel tel ou tel choix sémantique, elle n'en exclut certainement aucun. Nous donnerons au chapitre IV une sémantique opérationnelle à petits pas ; dans la présente section, nous allons ébaucher plusieurs sémantiques, en utilisant à chaque fois une approche qui facilite l'exposition. Avant cela, nous allons examiner plusieurs exemples afin de cerner le comportement désiré.

II.5.2 Exemples

II.5.2.1 Empreinte d'un module non scellé

Considérons deux programmes qui appliquent un même foncteur (`Map.Make` de la bibliothèque standard d'Objective Caml) à deux arguments différents (`OrderedInt.compare` renvoie des résultats opposés sur *A* et sur *B*).

P18A=

```
module OrderedInt = struct
  type t = int
  let compare x y = if x = y then 0 else if x < y then 1 else -1
end
module IntMap = Map.Make(OrderedInt)
send (dyn IntMap.add 1 "I" (IntMap.add 2 "II" IntMap.empty) at string IntMap.t)
```

P18B=

```
module OrderedInt = struct
  type t = int
  let compare x y = if x = y then 0 else if x < y then -1 else 1
end
module IntMap = Map.Make(OrderedInt)
print_string (IntMap.find 1 coerce receive() at string IntMap.t)
```

Les modules `IntMap` sur les deux machines ne sont pas compatibles. De fait, `Map.Make` utilise des arbres de recherche ordonnés grâce à la fonction `compare`, et l'utilisation d'une relation d'ordre différente ferait que `1` ne serait pas trouvé dans l'arbre sur *B*.

Cet exemple fait apparaître un point crucial : le code (pas seulement le type) du module `OrderedInt` importe bien que celui-ci ne soit pas scellé, ne définisse pas de type abstrait. Lorsque le compilateur rencontre la définition de `OrderedInt`, rien ne permet de prédire si celui-ci sera scellé un jour ; et le scellage définissant un type abstrait est dans la définition de `Map.Make`, et non dans le code qui est en train d'être compilé.

Dans l'approche à grands pas, où l'on commence par calculer complètement les structures[8], le module `OrderedInt` n'interviendra pas : le code qui servira à calculer l'empreinte de `IntMap` sera intégré à la structure qui est la valeur de `IntMap`. Dans l'approche dénotationnelle, il faudra donner une sémantique à `OrderedInt` pour l'utiliser dans le calcul de la sémantique `Map.Make(OrderedInt)` ; il ne s'agira pas forcément d'une empreinte. C'est l'approche à petits pas qui est la plus affectée : il nous devrons calculer l'empreinte de `OrderedInt`, afin de pouvoir définir l'application d'un foncteur comme l'application d'une empreinte à une empreinte.

II.5.2.2 Paramétrisation

Considérons une variante de l'exemple *P17* dans laquelle le module `TableSymboles` est directement défini sur la machine *A* en utilisant `CP`, tandis que sur la machine *B* il est obtenu par application du foncteur `TableSymbolesF` au module `CP`.

P19A=

```
module CP = struct ⟨code de CP⟩ end : sig ⟨signature de CP⟩ end
module TableSymbolesF = functor (CP : sig ⟨signature de CP⟩ end) ->
  struct ⟨code de TableSymboles⟩ end  :
  sig ⟨signature de TableSymboles⟩ end
module TableSymboles = TableSymbolesF(CP)
let (table, s) = TableSymboles.entre TableSymboles.vide "toto" in
let table' = TableSymboles.donne "toto" (Valeur_entiere 42) in
send (dyn table' at TableSymboles.t)
```

? *P19B*=

```
module CP = struct code de CP end : sig signature de CP end
module TableSymboles =
  struct (*code de TableSymboles*) end  :
  sig ⟨signature de TableSymboles⟩ end
let table = coerce receive() at TableSymboles.t in
let (_, s) = TableSymboles.entre table "toto" in
let (Valeur_entiere x) = TableSymboles.valeur table s in
print_int x
```

Les deux manières de construire `TableSymboles` sont-elles compatibles ? Ce point est débatable.

Notons déjà qu'il n'y a aucun danger à rendre les types compatibles. Les données sont forcément compatibles, et les invariants sont les mêmes des deux côtés.

Du point de vue de l'ingénierie logicielle, le passage du code sur *A* au code sur *B* correspond à une évolution naturelle dans un programme, consistant à permettre de changer d'implémentation du compteur pair alors que ce n'était pas prévu au départ. Il serait utile qu'une telle évolution permette au programme évolué sur *B* de communiquer avec l'ancienne version sur *A*.

D'un point de vue théorique, le bilan est plus mitigé. D'un côté, il paraît raisonnable de considérer un module dépendant d'un autre module comme un cas particulier d'un foncteur, appliqué une seule fois. D'un autre côté, rendre les deux programmes équivalents peut compliquer la sémantique, ce qui est un inconvénient à la fois pour nous, auteur d'icelle, et pour le programmeur, qui doit la comprendre.

[8]Rappelons qu'une structure est un module de base, une liste de champs, par opposition ici à un foncteur.

II.5.2.3 Compositionnalité

Considérons l'extrait de programme Objective Caml suivant :

```
module TableSymbolesF =
    functor (A : sig ... end) -> struct ... end : sig type t ... end
module TableSymbolesF' = TableSymbolesF
module CompteurPair = struct ... end : sig type t ... end
module CP = CompteurPair
module TableSymboles = TableSymbolesF(CompteurPair)
module TS = TableSymbolesF(CP)
module TS1 = TableSymbolesF(CP)
module TS' = TableSymbolesF'(CP)
```

Après ces définitions, les types `CP.t` et `CompteurPair.t` sont compatibles, puisque `CP` est une simple abréviation de `CompteurPair`. Les types `TS.t` et `TS1.t` sont compatibles, parce que les foncteurs d'Objective Caml sont **applicatifs** : appeler deux fois le même foncteur sur le même argument produit deux fois les mêmes types. Les types `TS.t` et `TS'.t` sont également compatibles, puisque `TableSymbolesF'` est un alias de `TableSymbolesF`.

En revanche, les types `TableSymboles.t` et `TS.t` ne sont pas compatibles, bien que le premier puisse s'écrire `TableSymbolesF(CompteurPair)t`, le second `TableSymbolesF(CP).t`, et que `CP` soit défini comme un alias de `CompteurPair`. Ceci vient de ce qu'Objective Caml garde une trace des alias de types seulement, et pas des alias de modules. La définition « `module CP = CompteurPair` » fait de `CP.t` une abréviation de `CompteurPair.t`, ce qui ne suffit pas à assurer que `TableSymbolesF(CP)` et `TableSymbolesF(CompteurPair)` sont compatibles (*cf.* l'exemple *P18*).

Le principe général de notre système d'empreinte est de rendre indistinguables deux modules qui ont le même code. Dans l'exemple suivant, la désérialisation pourra réussir — en fait, nous considérerons comme un point positif le fait qu'elle réussisse, sans toutefois l'exiger.

P20A=

```
module TableSymbolesF = functor (CP : sig ⟨signature de CP⟩ end) ->
  struct ⟨code de TableSymboles⟩ end  :
  sig ⟨signature de TableSymboles⟩ end
module CompteurPair = struct ⟨code de CP⟩ end : sig ⟨signature de CP⟩ end
module TableSymboles = TableSymbolesF(CompteurPair)
let (table, s) = TableSymboles.entre TableSymboles.vide "toto" in
let table' = TableSymboles.donne "toto" (Valeur_entiere 42) in
send (dyn table' at TableSymboles.t)
```

P20B= ?

```
module TableSymbolesF = functor (CP : sig ⟨signature de CP⟩ end) ->
  struct ⟨code de TableSymboles⟩ end  :
  sig ⟨signature de TableSymboles⟩ end
module CompteurPair = struct ⟨code de CP⟩ end : sig ⟨signature de CP⟩ end
module CP = CompteurPair
module TableSymboles = TableSymbolesF(CP)
let table = coerce receive() at TableSymboles.t in
let (_, s) = TableSymboles.entre table "toto" in
let (Valeur_entiere x) = TableSymboles.valeur table s in
print_int x
```

II.5.2.4 Paramètre inutile

Considérons l'extrait de programme Objective Caml suivant :

```
module F = functor (U : sig end) ->
  struct              sig
    type t = int   :    type t
    let x = 3           val x : t
  end                 end
module U1 = struct end
module U2 = struct end
module M1 = F(U1)
module M2 = F(U2)
```

Ce programme définit un module `F` dont le paramètre n'est pas utilisé. Suivant la démarche expliquée dans la section II.5.2.3, les types `M1.t` et `M2.t` sont différents (ils sont les abréviations respectives de `F(U1).t` et `F(U2).t`, qui ne se simplifient pas plus).

Pourtant la vérification de type dynamique correspondante ne peut que réussir. La raison en est la même que dans l'exemple *P20* : les deux programmes ont exactement le même code, leurs types sont donc déclarés compatibles.

P21A=

```
module F = functor (U : sig end) ->
  struct type t = int  let x = 3 end  :  sig type t  val x : t  end
module U1 = struct end
module M1 = F(U1)
send (dyn M1.x at M1.t)
```

√ *P21B*=

```
module F = (*comme sur A*)
module U2 = struct end
module M2 = F(U2)
coerce receive() at M2.t
```

Nous ne considérons pas la réussite de la désérialisation dans ce cas comme réellement problématique, car le programmeur qui souhaite vraiment obtenir des types incompatibles peut le faire au prix de changements mineurs. Il faut au minimum qu'il include dans `U1` et `U2` des signes distinctifs ; par exemple un nom ou un numéro.

P22A=

```
module F = functor (U : sig end) ->
  struct type t = int  let x = 3 end  :  sig type t  val x : t  end
module U1 = struct let nom = "mètre" end
module M1 = F(U1)
send (dyn M1.x at M1.t)
```

? *P22B*=

```
module F = (*comme sur A*)
module U2 = struct let nom = "kilogramme" end
module M2 = F(U2)
coerce receive() at M2.t
```

Nous verrons cependant qu'il peut être difficile d'assurer que la désérialisation échoue même dans ce deuxième programme. Nous proposons une modification supplémentaire qui assurera l'échec de la désérialisation : inclure l'argument `U` dans le résultat de `F`, sous la forme d'un champ caché (ici `_U`).

P23A =
```
module F = functor (U : sig end) ->
  struct
    type t = int
    let x = 3
    module _U = U
  end  :  sig type t  val x : t  end
module U1 = struct let nom = "mètre" end
module M1 = F(U1)
send (dyn M1.x at M1.t)
```

P23B =
```
module F = (*comme sur A*)
module U2 = struct let nom = "kilogramme" end
module M2 = F(U2)
coerce receive() at M2.t
```

II.5.2.5 Foncteurs scellés

Lorsque nous scellions des foncteurs, nous avons à chaque fois écrit « `functor (...) -> struct ... end : sig ... end` », ce qui se parenthèse ainsi : « `functor (...) -> (struct ... end : sig ... end)` », c'est-à-dire que c'est la structure résultat du foncteur qui est scellée.

P24A =
```
module CP = struct ⟨code de CP⟩ end : sig ⟨signature de CP⟩ end
module TableSymbolesF = functor (CP : sig ⟨signature de CP⟩ end) ->
  struct ⟨code de TableSymboles⟩ end   :
  sig ⟨signature de TableSymboles⟩ end
module TS = TableSymbolesF(CP)
let (table, s) = TS.entre TS.vide "toto" in
let table' = TS.donne "toto" (Valeur_entiere 42) in
send (dyn table' at TS.t)
```

P24B =
```
module CP = struct ⟨code de CP⟩ end : sig ⟨signature de CP⟩ end
module TableSymbolesG =
  (functor (CP : sig ⟨signature de CP⟩ end) ->
           struct ⟨code de TableSymboles⟩ end) :
  (functor (CP : sig ⟨signature de CP⟩ end) ->
           sig ⟨signature de TableSymboles⟩ end)
module TS = TableSymbolesF(CP)
let table = coerce receive() at TS.t in
let (_, s) = TS.entre table "toto" in
let (Valeur_entiere x) = TS.valeur table s in
print_int x
```

Dans cet exemple, le code est le même sur les deux machines, seule la position du scellage change. Au final, `TS` a la même spécification sur les deux machines ; plus généralement `TableSymbolesG` et `TableSymbolesF` peuvent s'utiliser exactement de la même manière. Il n'est pas clair pour l'auteur si la différence entre les deux programmes représente une volonté du programmeur qu'il faut conserver, ou un détail d'écriture qu'il faut gommer.

Un cas analogue qui ne fait pas intervenir les foncteurs est celui du double scellage : dans un langage de modules plus évolué que celui de la section II.3, on peut sceller tout terme de sorte module, et on peut par exemple écrire `((struct ... end :` S_1`) :` S_2`)`. Ce module est-il équivalent à `(struct ... end :` S_2`)` ? Est-il équivalent à `((struct ... end :` S_1'`) :` S_2`)` ? Ici encore, nous n'avons pas d'objectif particulier quant à la compatibilité de deux programmes différant par la nature exacte du scellage.

II.5.2.6 Calculs sur les modules

Le langage de modules de ML permet de combiner des modules de façon complexe. Techniquement, il s'agit d'un lambda-calcul muni d'un système de types souvent subtil (et variable suivant les auteurs et suivant les implémentations). Notre calcul sur les empreintes doit gérer tous les modules bien typés ; nous souhaitons notamment que deux types compatibles statiquement le soient encore dynamiquement.

Nous donnons sous forme stylisée un exemple de construction de modules qui donne le même type à la compilation (si on définit `F`, `G1`, `M1`, `G2` et `M2` dans le même programme, `M1.t` et `M2.t` sont compatibles) ; il faut donc obtenir aussi le même type à l'exécution. Nous utilisons ici un foncteur `F` à deux arguments (le premier de signature *S*, le second de signature *T*).

P25A =

```
module F = functor (A : S) -> functor (B : T) -> struct ... end
module G1 = functor (A : S) -> F(A)(Bonga)
module M1 = G1(Aga)
send (dyn M1.x at M1.t)
```

P25B =

```
module F = functor (A : S) -> functor (B : T) -> struct ... end
module G2 = functor (B : S) -> F(Aga)(B)
module M2 = G2(Bonga)
coerce receive() at M2.t
```

De manière générale, le langage de module d'Objective Caml contient un lambda-calcul typé (avec des types dépendants), l'évaluation (la bêta-réduction). Nous devrons donc gérer la bêta-réduction des scellages.

II.5.3 Sémantiques

II.5.3.1 Empreintes de valeurs

Une manière simple d'éviter le recours à des empreintes d'ordre supérieur est de reporter le calcul des empreintes à l'exécution. Les empreintes servant à identifier des types qui sont des champs d'une structure, c'est lors de l'exécution de la construction de scellage que nous calculerons une empreinte. Le comportement de la construction *M* `:` *S* est donc de calculer *M* : par exemple, si *M* est un appel de foncteur `F(N)`, celui-ci est évalué ; si les champs de *M* font référence à des identificateurs externes à *M*, la valeur de ceux-ci a été substituée.

Une conséquence pratique de cette stratégie est que le module dont l'empreinte est calculée n'a plus aucune dépendance : c'est une structure, dont nous avons défini l'empreinte sans difficulté.

Un autre avantage de cette approche est qu'elle est largement indépendante du langage des modules. Formellement, seules importent les valeurs de ce langage et non les expressions, puisque c'est seulement d'une valeur que l'on prend l'empreinte ; qui plus est, seules les valeurs dont on peut projeter un champ, c'est-à-dire les structures, ont une empreinte.

Un inconvénient est en revanche que le comportement d'un programme devient difficile à prévoir : il est plus facile de comparer le code de deux programmes que de comparer les valeurs, il y a donc un risque de coïncidence imprévue (ou au contraire que le programmeur croie que deux constructions différentes mènent à la même valeur alors qu'il n'en est rien).

Regardons le comportement dans cette sémantique des exemples de la section II.5.2.

1. Le problème mentionné à la section II.5.2.1 ne se pose pas : l'empreinte est calculée après que tous les foncteurs ont été appliqués à leurs arguments. Ceux-ci apparaissent en ce qu'ils influencent la valeur du module qui est scellé.
2. Dans l'exemple *P17*, la désérialisation réussit, car on obtient la même valeur sur les deux machines.
3. L'exemple *P20* fonctionne comme attendu.
4. Pour ce qui est du paramètre inutile, la désérialisation réussit dans les exemples *P21* et *P22* et échoue dans l'exemple *P23*.
5. Nous n'avons pas encore précisé comment sceller un foncteur : nous ferons simplement comme si le scellé s'appliquait au résultat. Dans l'exemple *P24*, la désérialisation réussit donc.
6. Dans l'exemple *P25*, la désérialisation réussit, puisque les calculs mènent à la même valeur.

L'empreinte n'est plus calculée à la compilation mais à l'exécution. Ceci peut paraître poser un problème d'inefficacité. Lorsque les modules sont utilisés au niveau supérieur pour combiner une fois pour toutes des composants d'un programme, les calculs d'empreintes sont effectués au lancement du programme, durant ce que l'on peut considérer comme son édition de liens dynamique. Certains dialectes de ML, dont Objective Caml, permettent de définir des modules localement à un morceau de programme ; dans ce cas le calcul d'empreinte peut avoir lieu à tout moment. Ceci dit, la complexité du calcul d'empreinte est linéaire ou proche de linéaire en la taille du module (il s'agit essentiellement de parcourir une fois le module entier en tant que structure de données). Le coût supplémentaire n'est donc *a priori* pas rédhibitoire. On peut vouloir effectuer ce calcul paresseusement s'il est fréquent de construire des modules dont on n'utilisera jamais les types abstrait dans une sérialisation.

II.5.3.2 Empreintes systématiques

Au vu de la section II.5.2.1, il faut tenir compte dans les calculs d'empreintes des modules non scellés. Une approche radicale est d'attribuer systématiquement une empreinte à un module.

L'empreinte d'un module, scellé ou non, est son code ; l'empreinte d'un scellage indique qu'il s'agit d'un scellage, et contient comme d'habitude la signature apposée. L'empreinte d'un foncteur est le code du foncteur lui-même (`hash(functor(...)->struct...end)`, par opposition à `hash(struct...end)`).

Lorsqu'une structure a des variables libres, elle est transformée en foncteur afin d'en faire un terme clos dont on prend l'empreinte ; ce foncteur reste appliqué à des variables[9]. Les variables

[9] On fait du *lambda-lifting*.

restantes (qui sont les arguments du foncteur obtenu) sont remplacées par l'empreinte du module en question. L'application de foncteur est conservée sous forme symbolique.

Donnons par exemple les empreintes des modules définis dans le programme de l'exemple *P17* page 76 (h_A est l'empreinte du module `A`) :

h_{CP} = `"struct ...code de CP... end : sig ...signature de CP... end"`
$h_{TableSymbolesF}$ = `"functor(CP:...)-> struct...end : sig...end"`
h_{TS} = `"(functor(A1:...)->functor(A2:...)->A1(A2))` $h_{TableSymbolesF}$ h_{CP}`"`

Cette sémantique est très exigeante vis-à-vis des manières de définir un module : pour obtenir la même empreinte, deux modules doivent avoir été construits exactement de la même façon. Ce manque d'expressivité est à contrebalancer avec la simplicité de la sémantique.

Il peut paraître de prime abord que cette sémantique rend des modules distincts alors qu'ils sont considérés comme équivalents statiquement. Par exemple, une simple application d'un foncteur identité change l'empreinte. La clé réside dans l'utilisation que nous faisons des empreintes. Rappelons que celles-ci sont utilisées dans les types réifiés des constructions `dyn` ... `at` *T* et `coerce` ... `at` *T*. Suivant la terminologie d'Objective Caml, nous commençons par développer les abréviations dans l'annotation de type *T*, pour obtenir un chemin généralisé. C'est de ce chemin que nous prenons l'empreinte. Ainsi les équivalences statiques entre types sont automatiquement respectées.

Regardons le comportement dans cette sémantique des exemples de la section II.5.2.

1. Dans l'exemple *P18*, l'empreinte du module `IntMap` contient celle du module `OrderedInt`. La différence d'implémentation de ce dernier entraîne bien le refus de la désérialisation.
2. Dans l'exemple *P17*, la désérialisation échoue, puisque les modules ont été fabriqués de manières différentes.
3. Dans l'exemple *P20*, l'empreinte de `CP` n'est pas la même que celle de `CompteurPair`, donc la désérialisation échoue. L'empreinte de `CP` est `"(functor(A:...)->A)` $h_{CompteurPair}$`"` ; cette application superflue du foncteur identité n'est pas transparente. Nous reviendrons sur ce point dans la section II.5.3.3.
4. Pour ce qui est du paramètre inutile, la désérialisation est acceptée dans l'exemple *P21*, et échoue dans les exemples *P22* et *P23* où les arguments passés à `F` sont différents.
5. Dans l'exemple *P24*, le code n'est pas le même puisque les scellages sont à des endroits différents, donc la désérialisation échoue.
6. Dans l'exemple *P25*, l'annotation de type calculée sur la machine *A* est h_1`.t` où h_1 est l'empreinte de `F(Aga)(Bonga)` ; sur *B*, c'est h_2`.t` où h_2 est l'empreinte de `F(Aga)(Bonga)`. La désérialisation réussit donc.

L'essentiel du calcul des empreintes peut être fait dès la compilation. Le seul cas où le processus écrit ci-dessus n'y parvient pas est lorsqu'une variable libre dans un module ne correspond pas à un module préalablement défini : cette variable ne peut être que le paramètre formel présent dans le programme original. Ce paramètre est utilisé seulement lorsqu'il apparaît dans une annotation de type dynamique dans le corps du foncteur. Si ce cas est exclu, c'est-à-dire que la sérialisation et la désérialisation ne peuvent référencer que des types connus statiquement, toutes les empreintes utiles peuvent être calculées statiquement. Ceci participe d'une difficulté générale avec le typage dynamique dans les langages polymorphes, à savoir que si l'on peut utiliser des variables de types dans des annotations de types dynamiques, il devient impossible d'effacer systématiquement les types après la compilation, et difficile de cantonner le non-effacement à une petite partie du programme.

II.5.3.3 Une sémantique hybride

La sémantique que nous présenterons au chapitre IV est hybride entre les deux précédentes. Elle suit largement la sémantique par empreinte de valeurs décrite à la section II.5.3.1, avec une différence : pour calculer l'empreinte d'un scellage d'un module *M : S*, nous ne calculerons pas préalablement *M*. Il reste vrai que seule est prise l'empreinte d'un module clos : lorsque l'expression *M : S* est en position d'être évaluée, elle n'a pas de variable libre ; les valeurs des paramètres présents au départ ont été substituées.

Sur le plan théorique, la difficulté de définir l'une ou l'autre sémantique est comparable : elle consiste essentiellement à modifier la définition des contextes d'évaluation pour inclure ou non la construction de scellage.

Sur le plan pratique, seules les valeurs des variables libre d'un module sont utilisées, et celles-ci sont de toute façon présente à l'exécution. Le compilateur peut générer pour le module *M* qui est scellé une « empreinte à trous », et ces trous seront comblés par les valeurs des variables à l'exécution.

Sur le plan de l'expressivité, le caractère hybride de cette sémantique permet au programmeur de choisir au cas par cas s'il veut que le test de compatibilité à la désérialisation soit basé sur la valeur ou sur le code.

Regardons pour cette nouvelle sémantique le comportement des exemples dans lequel les sémantiques précédentes différaient. Dans l'exemple *P17*, le code des tables de symbole est le même, c'est seulement la variable libre `CP` qui n'a pas le même statut ; étant donné qu'elle a en revanche la même valeur, la sérialisation est acceptée. Dans l'exemple *P20*, seul change l'argument d'un foncteur dont le corps est scellé, la sémantique est la même qu'en empreintes par valeur. Il en est de même pour les exemples présentés à la section II.5.2.4. Dans l'exemple *P24*, la désérialisation est refusée parce que le code sous le scellé n'est pas le même (c'est un foncteur dans un cas et pas dans l'autre).

II.6 Compatibilité d'empreintes

II.6.1 Générativité

II.6.1.1 Introduction

Les exemples que nous avons présentés jusqu'ici, ainsi que les concepts que nous avons développés, n'utilisaient jamais d'effets de bords à l'intérieur des modules. Un module était donc « pur » (on dit aussi « référentiellement transparent »), au sens où évaluer deux fois le même module donnait le « même » résultat.

Or nous étudions des langages de programmation, qui peuvent faire des entrées-sorties (effets de bord externes) ; de plus nous nous proposons d'étendre ML, qui possède des effets de bord internes, les références.

Reprenons par exemple le compteur pair que nous avions introduit dans la section II.2.2. Si le code que nous avions présenté alors assurait qu'une valeur censée être un compteur pair était forcément un entier positif pair, il permettait en revanche des programmes comme le suivant où `m` et `n` ont la même valeur malgré l'appel apparent à `CompteurPair.suivant` :

```
let c = CompteurPair.début in
let m = CompteurPair.courant c in
let c' = CompteurPair.suivant c in
let n = CompteurPair.courant c in (*erreur : c devrait être c'*)
assert (m <> n)
```

Dans le système de types de ML, rien n'empêche de dupliquer une valeur, ici `c`. On peut en revanche définir un type abstrait qui empêche une duplication erronée telle que celle présentée ci-dessus, pourvu que l'on fasse confiance à l'implémentation du module définissant le type abstrait. Voici une implémentation impérative de `CompteurPair`.

```
module CompteurPair =
  struct                                sig
    type t = int                          type t
    let c = ref 0                   :     val c : t
    let suivant x = x := !x + 2           val suivant : t -> unit
    let courant x = !x                    val courant : t -> int
  end                                   end
```

M_g S_g

Cette implémentation assure qu'il n'y a dans chaque instance du programme qu'une seule valeur de type `CompteurPair.t`, à savoir la référence créé lors de l'initialisation du compteur. Une conséquence est que si deux appels à `CompteurPair.courant` sont entrecoupés d'au moins un appel à `CompteurPair.suivant`, les valeurs obtenues sont différentes : le compteur génère des numéros uniques, et le système de types nous assure que cette propriété visible sur la définition de `CompteurPair` restera vraie quoi que le reste du programme puisse contenir. Nous souhaitons bien entendu conserver cette propriété dans un contexte distribué.

P26A=

```
module CompteurPair = M_g : S_g
send (dyn CompteurPair.c at CompteurPair.t)
```

P26B=

```
module CompteurPair = M_g : S_g
let a = coerce receive() at CompteurPair.t in
CompteurPair.suivant a;
print_int (CompteurPair.courant ())
```

Le type `CompteurPair.t` sur la machine *A* est associé à la référence créée sur *A*, et le type `CompteurPair.t` sur la machine *B* est associé à la référence créée sur *B* : il ne faut pas que ces deux types soient compatibles, sous peine d'obtenir deux compteur qui peuvent évoluer indépendamment.

II.6.1.2 Empreintes singularisées

Les empreintes que nous avons définies à la section II.3.1 sont des empreintes *structurelles*, au sens ou l'empreinte d'un module ne dépend que du module lui-même. Or cette notion ne convient plus lorsque l'on introduit des effets de bord. Dans ce cas, au lieu de générer une empreinte structurelle, nous générerons une **empreinte singularisée**, qui n'est autre qu'un identifiant universellement unique associé au module.

Dans l'exemple *P26*, l'annotation de type `CompteurPair.t` sur *A* est compilée en k_A`.t`, où k_A est l'empreinte singularisée générée pour le module `CompteurPair` sur *A* ; sur *B*, l'annotation de type `CompteurPair.t` est compilée en k_B`.t` où k_B est l'empreinte singularisée générée pour `CompteurPair` sur *B*. Comme k_A et k_B sont deux empreintes différentes, les types k_A`.t` et k_B`.t` sont incompatibles, et la désérialisation est refusée.

Les empreintes singularisées sont une généralisation immédiate des « estampilles » ou « horodatages » (*timestamps*) utilisées traditionnellement pour modéliser la sémantique générative des modules de ML. Le concept est proche des *apax* (*nonces*) utilisés dans les analyses sécuritaires de protocoles, en ce que des noms uniques sont générés ; noter toutefois que c'est lors de la construction du module, et non lors de la sérialisation, que la génération a lieu : plusieurs valeurs sérialisées peuvent porter la même empreinte singularisée, pourvu que le type abstrait concerné soit le même.

II.6.1.3 Choix du type d'empreinte

Pour chaque module, le compilateur doit choisir s'il faut générer une empreinte structurelle ou singularisée. Nous considérons comme préférable l'utilisation d'une empreinte structurelle (la section II.2 constitue une argumentation à cet effet), mais nous devons utiliser une empreinte singularisée lorsque c'est nécessaire. La distinction est la suivante : si nous parvenons à prouver qu'un module est *pur*, au sens où son évaluation ne cause pas d'effet de bord, alors il sera doté d'une empreinte structurelle ; sinon une empreinte singularisée sera générée.

La méthode habituelle dans les théories des langages de la famille ML pour prouver qu'un module est pur est d'utiliser un système d'effets [HMN05]. Si les systèmes d'effets sont en général trop complexes pour être utilisés dans un langage de programmation généraliste, il en existe une approximation grossière mais qui a fait ses preuves : toute valeur est pure. La *restriction aux valeurs* (*value restriction*) [Wri95] est déjà utilisée pour le typage de ML pour une autre raison, le polymorphisme (en ML, il n'est pas sûr de généraliser le type d'une expression arbitraire, mais il est sûr de généraliser le type d'une valeur), et cette restriction a fait ses preuves en termes d'expressivité. Aussi proposons-nous de baser le choix de l'empreinte sur le même critère : nous qualifierons de purs, et attribuerons une empreinte structurelle, aux modules dont tous les champs sont des valeurs, et les autres modules, dits impurs, se verront attribuer des empreintes singularisées. Notons que dans tous les exemples présentés à la section II.2, les modules sont purs.

Nous avons évoqué à la section II.6.2 le besoin occasionnel pur le programmeur de forcer l'utilisation d'une empreinte singularisée. Un moyen détourné pour ce faire est de placer dans le module une référence par ailleurs inutilisée : le module est alors nécessairement impur. Un moyen plus direct serait d'avoir dans le langage deux opérations de scellage distinctes, un scellage applicatif qui ne peut s'appliquer qu'à un module pur et donne lieu à la création d'une empreinte structurelle, et un scellage génératif qui entraîne l'utilisation d'une empreinte singularisée. Nous présentons à la section IV.4.4 possède les deux types de scellages. Il n'est pas en revanche clair que l'ajout d'une construction supplémentaire soit justifié dans un langage de programmation.

II.6.1.4 Appel distant

Si le seul mécanisme de communication entre machines est le passage d'une valeur sérialisée par le mécanisme de base que nous avons présenté ici, une valeur dont le type fait intervenir une empreinte singularisée ne peut être manipulée que sur la machine où l'empreinte a été créée. Il est en effet par définition impossible de reproduire sur une autre machine l'empreinte singularisée.

Cette situation, où l'on autorise la sérialisation de données qui ne peuvent pas être désérialisées sur une autre machine, présente en elle-même un intérêt. Un serveur de stockage, par exemple, peut conserver des données sous forme sérialisée, quitte à ce que seul leur émetteur puisse plus tard les relire. Une empreinte singularisée peut aussi servir de mot de passe : la connaissance d'une telle empreinte créée sur une machine `A` par une machine `B` prouve que `B` a reçu cette empreinte de `A` par le passé (éventuellement par le biais de machines tierces).

Il est toutefois souvent utile de partager entre plusieurs machines un type abstrait, même génératif. Ceci nécessite des fonctionnalités supplémentaires dans le langage. Ils peuvent essentiellement être de deux natures : soit la ressource protégée par le type abstrait peut *migrer* d'une machine à l'autre, soit il peut y avoir un mécanisme d'*appel distant* permettant d'accéder sur une machine à une ressource d'une autre machine. La migration est en générale combinée avec des possibilités d'appel distant ; nous allons ici discuter de ce dernier.

Nous avons présenté à la section II.4.1.4 un mécanisme de sérialisation de module. L'implémentation d'un tel mécanisme en présence de ressources pose des problèmes spécifiques, dont l'analyse sort du cadre de la présente thèse ; le principe général est qu'une valeur référençant des ressources ne peut pas être copiée directement, les références doivent être transformées en une forme de pointeurs distants.

Sous réserve de l'existence d'une telle fonctionnalité, il est possible d'utiliser par exemple un compteur pair tel que présenté en II.6.1.1 sur une autre machine que celle sur laquelle il a été défini. Notons que la machine *B* n'a plus besoin de connaître le code de `CompteurPair` : elle le reçoit de *A*.

P27A=

```
module CompteurPair = Mg : Sg
print_int (CompteurPair.courant ())
send (dyn CompteurPair at Sg);
CompteurPair.suivant a;
print_int (CompteurPair.courant ())
```

P27B=

```
module CompteurPair = coerce receive() at Sg
CompteurPair.suivant a;
print_int (CompteurPair.courant ())
```

Le programme réparti ci-dessus affiche 0 sur *A*, puis selon l'ordonnancement de l'exécution 2 sur *A* et 4 sur *B* ou l'inverse (rappelons que `CompteurPair.suivant` incrémente le compteur de 2) ; la valeur finale du compteur est 4.

II.6.1.5 Foncteurs génératifs

Dans la section II.5, nous manipulions des foncteurs **applicatifs**, au sens où appliquer deux fois le même foncteur au même argument donnait deux fois le même résultat. C'était le cas aussi bien dans notre système d'empreintes que dans certains systèmes de typage statique des modules, notamment celui d'Objective Caml.

Lorsque le corps d'un foncteur est impur, le foncteur ne peut plus être applicatif : chaque application doit créer de nouveaux types. On dit alors que le foncteur est **génératif**.

Donnons un exemple de foncteur génératif. De même que nous avons écrit une implémentation à base de références de `CompteurPair`, nous donnons une implémentation à base de tables de hachage modifiables en place (la bibliothèque `Hashtbl` d'Objective Caml) de `TableSymboles` (*cf.* II.3.2.1). Nous avons ici choisi de cacher complètement l'unique table de symbole à l'intérieur du module, en n'exposant que les fonctions de manipulation. Nous désignons toujours par M_g et S_g le code et la signature du `CompteurPair` génératif présenté à la section II.6.1.1.

```
module TableSymboles = functor (CP : Sg) ->
  struct
    type symbole = int
    let tn = Hashtbl.create 42
    let tv = Hashtbl.create 42
```

```
    let entre nom =
      try Hashtbl.find tn nom with Not_found ->
        CP.suivant CP.c; let c' = CP.courant CP.c in Hashtbl.add tn nom c'; c'
    let valeur x = Hashtbl.find tv x
    let fonction x = Hashtbl.find tv (x+1)
    let donne x v = Hashtbl.add tv x v
    let donnef x v = Hashtbl.add tv (x+1) v
  end : sig
    type symbole
    val entre : string -> symbole
    val valeur : symbole -> valeur
    val fonction : symbole -> valeur
    val donne : symbole -> valeur -> unit
    val donnef : symbole -> valeur -> unit
end
```

Si l'on applique deux fois le foncteur `TableSymboles`, on obtient deux paires de tables différentes `(tn,tv)` ; la générativité du foncteur fait que les types `symbole` associés sont incompatibles, ce qui est heureux puisque chaque nom de symbole peut avoir des numéros associés différents dans les deux tables. Avec les définitions en vigueur de `CompteurPair` et `TableSymboles`, le fragment suivant définit deux types `S.symbole` et `T.symboles` incompatibles.

```
module S = TableSymboles(CompteurPair)
module T = TableSymboles(CompteurPair)
```

Notons que si `S` et `T` ont des tables de symboles et de valeurs différentes, le même compteur est partagé entre les deux ; ceci est sûr, puisque rien n'interdit à un compteur d'avoir plusieurs utilisateurs.

II.6.1.6 Empreintes de valeurs

Nous avons évoqué à la section II.5.3 deux manières de coder les empreintes de modules : l'une utilisant les valeurs des modules, l'autre gardant systématiquement trace du code. C'est cette deuxième manière qui correspond au plus près à notre présentation originale des empreintes à la section II.3.1.

Si l'on prend l'empreinte de code, il faut gérer deux types d'empreintes, structurelles ou singularisées. En revanche, si l'on prend l'empreinte de valeurs, il n'y a pas de distinction à opérer. On peut remarquer en effet que toute valeur est pure. D'un point de vue plus opératoire, la valeur d'une expression créant une référence contient un emplacement (*location*)[10]. C'est dès l'allocation de cet emplacement que l'identifiant unique nécessaire est créé : l'inclusion de l'emplacement dans l'empreinte assure sa singularisation.

Si l'utilisation des empreintes de valeurs accomode naturellement le typage génératif, il n'est pas sans soulever des difficultés pratiques. En particulier, la cohabitation avec un ramasse-miettes est contraignante. Considérons le programme (stylisé) suivant :

```
module A = struct let r = ref ()  type t = int  let x = 3 end :
           sig                    type t        val x : t end
```

[10] Intuitivement, l'emplacement est l'adresse mémoire où est stocké le contenu de la référence ; le point important étant que cette adresse est spécifique à cette référence.

```
let s = dyn A.x at A.t
module B = struct let r = ref ()  type t = int  let x = 3 end :
           sig                    type t        val x : t end
let z = coerce s at B.t (*doit échouer*)
```

Bien que A et B aient le même code, ils n'ont pas la même empreinte, à cause de la référence `r`. Cette référence a pour valeur un emplacement ℓ dans le module A et un emplacement ℓ' différent dans le module B. Notons que la valeur sérialisée en `s` ne fait pas référence à cet emplacement. Si la référence `r` dans A est considérée comme morte après la définition de `s`, le ramasse-miette peut libérer l'adresse correspondant à l'emplacement ℓ, ce qui permet d'utiliser la même adresse pour ℓ'. Or si c'est cette adresse qui est utilisée pour singulariser l'empreinte de B, celle-ci se retrouve alors égale à l'empreinte de A, ce qui est contraire à la sémantique attendue. L'une des deux hypothèses que nous avons formulée doit donc être fausse :

- on peut considérer que `A.t` pointe sur l'emplacement ℓ au sens du ramassage de miettes ; alors cet emplacement reste vivant tout au long du programme ci-dessus, puisqu'il est contenu dans la valeur de `s` via l'empreinte contenue dans son annotation de type ; ceci oblige à considérer l'empreinte comme une donnée structurée et non une simple somme de contrôle ;
- on peut exiger que les emplacements ne soient jamais réutilisés (contrairement aux adresses mémoires) ; ceci ne fait que repousser la contrainte de génération d'un identifiant unique non seulement dans l'espace mais aussi dans le temps du système d'empreintes au système d'allocation mémoire.

II.6.1.7 Empreintes de code

Examinons l'impact des foncteurs génératifs sur la sémantique d'empreintes de code décrite à la section II.5.3.2. Chaque appel au foncteur doit créer une empreinte singularisée. Il n'est pas nécessaire de tenir compte de l'argument : de toute façon, le résultat du foncteur est incompatible avec toute autre construction de types.

Dans le monde purement applicatif, nous traitions l'empreinte d'une application de foncteur comme symbolique, c'est-à-dire que l'empreinte de l'application de F à A est l'application de l'empreinte de F à l'empreinte de A : $\mathrm{hash}(\mathsf{F}(\mathsf{A})) = (\mathrm{hash}(\mathsf{F}))(\mathrm{hash}(\mathsf{A}))$. Lorsque le module A est génératif, il se voit attribuer une empreinte singularisée k ; si F est un foncteur applicatif, l'empreinte de l'application est encore construite de la même manière : $\mathrm{hash}(\mathsf{F}(\mathsf{A})) = (\mathrm{hash}(\mathsf{F}))\mathsf{k}$. Lorsque F est un foncteur génératif, l'empreinte de $\mathsf{F}(\mathsf{A})$ est une empreinte singularisée.

Par exemple, supposons définis un foncteur applicatif `F` et un foncteur applicatif `G`, ainsi qu'un module `A`. Notons h_A l'empreinte de `A` et h' l'empreinte de `F`. Le fragment ci-dessous entraîne la génération de deux empreintes singularisées k et k', et les empreintes calculées sont indiquées :

```
module M = G(A)              hash(M) = k
module N = G(A)              hash(N) = k′
module U = F(M)              hash(U) = h′ k
module V = F(N)              hash(V) = h′ k′
module W = F(N)              hash(W) = h′ k′
```

II.6.2 Respect de l'abstraction

II.6.2.1 Compatibilité d'empreintes et rupture d'abstraction

Nous avons introduit la notion d'empreinte (structurelle, au départ) comme une traduction de l'abstraction présente dans le système de type des langages de la famille ML au niveau du typage dynamique. Nous avons ensuite vu que la modélisation d'un trait important du langage, les effets de bord, imposait l'utilisation d'une notion plus contraignante, les empreintes singularisées. Nous avions motivé au fur et à mesure les concepts introduits par des exemples ; nous allons maintenant porter un regard plus général sur la qualité de la modélisation.

La question à laquelle nous allons nous intéresser peut être formulée ainsi : la vérification de types modélisant les types abstraits par des empreintes structurelles casse-t-elle l'abstraction ? Dans la section I.1, nous avons mis en lumière quatre usages différents de l'abstraction, que nous allons examiner tour à tour.

Invariants L'abstraction peut être utilisée pour assurer certains invariants d'une structure de données. La propriété fondamentale d'un type abstrait dans ce cadre est que seules peuvent agir sur lui les fonctions qui sont définies en même temps. Notre proposition de typage dynamique n'affecte en rien cet usage : si deux types abstraits sont égalisés parce qu'ils proviennent de la même empreinte, c'est toujours le même code qui agit sur eux. Les invariants restent les mêmes.

Intimité Dualement, un type abstrait peut servir à garder certaines données secrètes. Dans ce cadre, la sérialisation d'une donnée peut représenter une fuite d'information. En effet, tester une désérialisation permet de comparer deux empreintes, laissant passer une valeur booléenne qui sinon resterait secrète (bien sûr, une construction `typecase` plus puissante permettrait d'extraire de l'information plus facilement).

Une telle utilisation des types abstraits est rare en pratique (souvent, des moyens externes au langage, tels que l'inspection manuelle du code du programme ou l'inspection du flux de caractères qui représente la valeur sérialisée, permettent d'observer toute l'information concernée). Nous offrons néanmoins un moyen de rendre une empreinte différente de toute empreinte choisie par un attaquant bien typé : les empreintes singularisées.

Singularisation On peut en ML définir un type génératif dont la structure est exposée mais qui est distinct de tout autre type ayant la même définition. Cette distinction est gommée par notre sémantique — c'en est même le principe directeur. Nous avons vu à la section II.2.3.3 comment le programmeur peut faire intervenir dans les empreintes des noms choisis par l'utilisateur ; une autre solution est ici encore pour le programmeur de forcer l'utilisation d'une empreinte singularisée.

Ressources Lorsqu'un type abstrait sert à limiter l'accès à une ressource, c'est la notion d'empreinte singularisée qui est adaptée ; nous avons vu à la section II.6.1.3 comment le compilateur peut choisir automatiquement de générer une empreinte singularisée.

II.6.2.2 Paramétricité

Une propriété généralement attendue de l'abstraction est la **paramétricité** [Mit86], c'est-à-dire que le reste du programme est indépendant du contenu du module protégé par l'abstraction : ce module peut être passé comme paramètre au reste du programme, et la valeur du paramètre n'affectera pas le comportement du programme. En particulier, le choix de la représentation d'un type abstrait ne doit pas affecter le comportement du programme.

Cette propriété est apparemment vérifiée par nos empreintes, dès lors qu'elles sont opaques : il est impossible d'examiner le contenu d'une empreinte, la seule opération fournie étant le test d'égalité entre deux empreintes qui peut être effectué lors d'une vérification de typage dynamique.

En fait, seules les empreintes singularisées sont vraiment paramétriques. Les empreintes structurelles, de par leur caractère reproductible, laissent échapper un peu d'information : on peut écrire un programme dont le comportement dépend de la coïncidence des implémentations de deux modules, via le test d'égalité de leurs empreintes structurelles. Dans ce cas, changer la représentation d'un des modules sans changer l'autre peut changer la sémantique d'un programme. On recouvre la paramétricité à condition de faire l'hypothèse quelque peu inhabituelle que les changements de représentations ne doivent pas faire varier la structure en classes d'équivalence des modules.

II.6.3 Conversions entre empreintes

II.6.3.1 Problématique

Le présent exposé a traité des types abstraits dans un environnement distribué en se concentrant avant tout sur l'utilisation de l'abstraction pour assurer des invariants. Les empreintes singularisées (section II.6.1) nous permettent de « respecter plus » l'abstraction, au sens où des types abstraits sont moins souvent compatibles. Nous allons maintenant chercher à « respecter moins » l'abstraction, c'est-à-dire à rendre plus de types abstraits compatibles.

Un problème fréquent dans un environnement réparti est la mise à jour des logiciels tournant sur les différentes machines. Il n'est généralement pas praticable de mettre à jour en même temps un programme sur toutes les machines sur lesquelles il est déployé. Nous avons mentionné ce problème en II.4.1.2 ; la solution dont nous avons alors discuté est de prévoir que chaque version successive d'un programme sache gérer les données provenant des versions précédentes.

Une telle solution passe à l'échelle du point de vue du réseau : elle n'a d'impact que sur l'écriture du programme, pas sur son déploiement. En revanche, elle ne passe pas à l'échelle du point de vue de la taille et de l'âge du programme, puisque celui-ci doit contenir éternellement toutes les versions de tous les modules qui y ont été présents dans une version passée.

Certes, la simple utilisation d'empreintes (par opposition à la démarche simpliste de ne jamais autoriser la compatibilité de types abstraits entre programmes distincts) permet de ne mettre à jour que les bibliothèques qui ont effectivement changé. Si un programme comporte N bibliothèques et que chaque version en affecte une proportion α, la version ν du programme a une taille proportionnelle à $\alpha\nu N$ (alors que la taille serait N si l'on n'incluait que les dernières versions), à comparer avec une taille νN si l'on devait combiner toutes les versions du programme entier.

On peut néanmoins s'attendre à une amélioration considérable si l'on remarque que dans de nombreux cas, la nouvelle version d'une bibliothèque peut remplacer l'ancienne. C'est possible si la nouvelle version ne fait qu'améliorer l'efficacité, ou ajouter des fonctionnalités. Dans le premier cas, les types abstraits de l'ancienne et de la nouvelle versions sont compatibles : il serait sûr d'attribuer la même empreinte aux deux versions. Dans le deuxième cas, les données produites par l'ancienne version sont utilisables sur la nouvelle, mais pas l'inverse : nous dirons que l'ancienne version a une **sous-empreinte** de la nouvelle, car les types abstraits qu'elle définit sont des sous-types de ceux définis par la nouvelle version.

II.6.3.2 Sous-empreinte vérifiée

Considérons un module `M` qui implémente une structure de données (de type `M.t`) caractérisée par certains invariants de représentation, par exemple les listes d'entiers triées en ordre croissant. Considérons également un module `N` qui utilise la même représentation des données mais fournit un invariant plus fort, par exemple les listes d'entiers strictements croissantes. Il est alors sûr de convertir une valeur de type `N.t` vers le type `M.t`, au sens où toutes les propriétés désirables sont conservées. Autrement dit, `hash(N).t` peut être un sous-type de `hash(M).t`.

Malheureusement, il est peu vraisemblable qu'un compilateur vérifie que `hash(M)` est une sous-empreinte de `hash(N)`. Il faudrait en effet que le système de types utilisé soit suffisamment puissant pour exprimer les invariants mis en jeu, et que le compilateur sache vérifier leur respect. Une sous-empreinte vérifiée est donc probablement irréalisable en pratique.

Un problème supplémentaire est que nous n'avons considéré ici que l'aspect de l'abstraction consistant à protéger des invariants. Cet aspect nous conduit à considérer une empreinte `h` comme sous-empreinte de $\mathtt{h}'$ lorsque $\mathtt{h}'$ a des invariants plus forts, autrement dit si le passage de $\mathtt{h}'$ a `h` consiste uniquement en l'ajout de constructeurs. Or nous avons mentionné à la section I.1 d'autres usages de l'abstraction, en particulier l'usage dual d'intimité. Si l'abstraction est utilisée pour conserver une information privée, `h` ne peut être une sous-empreinte de $\mathtt{h}'$ que si le passage de $\mathtt{h}'$ à `h` consiste uniquement en l'ajout de destructeurs. Ainsi, en l'absence d'indication supplémentaire, une éventuelle relation de sous-empreinte apparaît comme symétrique : il peut y avoir une équivalence entre modules ayant le même comportement, et c'est tout.

II.6.3.3 Sous-empreinte affirmée

Afin de faciliter l'écriture et le déploiement de programme répartis, il apparaît utile de fournir au programmeur un moyen de relâcher la vision très stricte de l'abstraction exposée dans la théorie. L'outil de base est une déclaration de compatibilité de deux empreintes. Lorsque par exemple deux versions d'une bibliothèque peuvent être utilisées de façon interchangeable, parce que seules des considérations d'efficacité les distingue (elles ont donc le même comportement observable), le programmeur pourrait déclarer ces deux versions comme compatibles. P. Sewell a proposé un langage de modules [Sew01] adapté à la cohabitation de versions multiples d'une même abstraction, dans lequel une construction « **`with!`** » permet de déclarer qu'une nouvelle abstraction est compatible avec une abstraction existante. Son descendant, le langage Acute [SLW$^+$04], est muni d'un langage de versions plus flexible, qui permet de déclarer des compatibilités entre ensembles de versions. Dans les deux cas, les types représentations doivent être les mêmes, et le programmeur assume la responsabilité de la compatibilité du code.

Une nouvelle version d'un module n'est pas toujours interchangeable à l'ancienne : il est souvent possible de remplacer l'ancienne version par la nouvelle sans que la réciproque ne soit systématiquement vraie. Il faut alors utiliser une relation entre empreintes qui n'est pas une relation d'équivalence. Un modèle dans lequel le typage dynamique est paramétrisé par une relation d'ordre partielle entre empreintes (déclarée par le programmeur) a été étudié par P.-M. Déniélou et J. Leifer [DL06]. L'exigence d'équivalence des types représentations y est remplacée par une exigence de sous-typage.

Chapitre III

HAT : Un premier calcul d'empreintes

Dans le chapitre II, nous avons expliqué comment l'on peut donner un nom à un type abstrait, en utilisant l'empreinte du module qui le définit. Il reste à voir comment intégrer les empreintes dans un langage de programmation : comment intégrer leur calcul, et quel usage en faire. Le présent chapitre décrit un premier langage formel capable de décrire des modules et leurs empreintes. Ce langage est volontairement restreint afin de pouvoir focaliser l'attention sur les aspects nouveaux (nous étudierons au chapitre IV comment intégrer lesdits aspects à un langage de programmation de la famille ML).

Les contributions de ce chapitre ont fait l'objet d'une présentation en congrès [LPSW03a], complétée par un rapport plus détaillé [LPSW03b].

III.1 Concrétiser l'abstraction : les crochets colorés

III.1.1 Introduction

III.1.1.1 Valeurs de types abstraits

Considérons un module simple définissant un type abstrait :

```
module M = struct          sig
  type t = int               type t
  let x = 3                  val x : t
  let y = 4             :    val y : t
  let f z = z + 1            val f : t -> int
  let g c = c                val g : int * t -> t * int
end                        end
```

Conformément à la définition donnée à la section II.3.1, la représentation du type `M.t` à l'exécution est `h.t` où `h` = `hash(struct...end:sig...end)` est l'empreinte du module `M`. Quelle est la valeur de l'expression `M.x` ?

Une première idée est d'utiliser le même mécanisme pour les expressions que pour les types : la valeur de `M.x` — sa représentation une fois la définition de `M` évaluée — serait `h.x`. Cette piste se heurte immédiatement à un écueil de taille : comment alors évaluer l'expression `(M.f M.x)`, ce qui nécessiterait l'évaluation de `h.f h.x` ? Alors qu'un type abstrait est déterminé par son nom (et éventuellement, dans des langages suffisamment complexes, par sa place dans la structure des types), une expression fournie par un module, eût-elle un type abstrait, a une structuration interne qu'il est impératif de conserver. Nous pouvons oublier que `M.t` est en interne `int`, mais pas que `M.x` vaut `3`.

Inversement, nous pourrions faire de telle sorte que `M.x` ait la valeur `3`, et c'est ce qui se fait dans certaines présentations des types abstraits. Mais cela ne nous convient pas : `M.x` a le type `M.t`, aussi vu sous le nom `h.t`, qui n'est pas égal au type `int`. Attribuer à `M.x` la valeur `3` consiste à oublier l'abstraction, ce qui est au mieux discutable, et impossible si l'on veut pouvoir effectuer des vérifications de typage dynamiques — ce qui est justement notre cas.

La valeur de `M.x` doit donc intégrer au moins deux informations : sa structure interne `3`, et son type externe `M.t` (soit à l'exécution `h.t`). La manière la plus simple est de réunir les deux indications au sein une conversion de type : la valeur de `M.x` est alors $\texttt{coerce}_{\texttt{h}}$ `3 to h.t`, celle de `M.f` est $\texttt{coerce}_{\texttt{h}}$ `(fun z -> z+1) to (h.t->int)`, et ainsi de suite. Cette méthode est suffisamment expressive, si l'on autorise la conversion $\texttt{coerce}_{\texttt{h}}$ `E to T` dès que le type de `E` est équivalent à `T` modulo l'égalité entre `h.t` et son type représentation (ici `int`). On remarque que les conversions sont indexées par l'empreinte `h` qui les autorise.

L'utilisation de telles conversions souffre de deux difficultés. La première est qu'elles sont par trop expressives ; il faut empêcher d'écrire quelque chose comme $\texttt{coerce}_{\texttt{h}}$ `(fun x -> x) to h.t->int`, sous peine de perdre toute abstraction dans le langage. En particulier, ces conversions doivent être interdites au programmeur ; ceci se fait sans peine en lui cachant les empreintes. Il reste néanmoins difficile, étant donnée une expression comportant des conversions, de déterminer si celles-ci sont légitimes ; le bon typage d'une telle expression n'implique pas que celle-ci puisse résulter d'un calcul considéré comme valide par l'auteur du type abstrait.

La deuxième difficulté est le comportement des conversions vis-à-vis des destructeurs. Considérons par exemple l'expression `M.f M.x`. Elle s'évalue vers

$$\texttt{(coerce}_{\texttt{h}}\texttt{ (fun z -> z+1) to (h.t->int)) (coerce}_{\texttt{h}}\texttt{ 3 to h.t)},$$

suite à quoi il faut reconnaître le sous-terme de gauche comme une fonction. Il est tentant de réduire cette expression en une simple conversion du résultat, soit $\texttt{coerce}_{\texttt{h}}$ `((fun z -> z+1) 3) to int` ; ceci nécessite de spécifier l'interaction entre les conversions et les destructeurs du langage. L'enchaînement de deux conversions pose également un problème ; soit il faut savoir combiner les conversions (par exemple en annotant chaque conversion par une liste ou un ensemble d'empreintes), soit il faut considérer l'interaction des chaînes de conversions avec les destructeurs.

III.1.1.2 Scellage

Historiquement, la première description d'un langage fournissant des types abstraits [Mor73b, Mor73a] utilisait la métaphore du **scellage** : la valeur de `M.x` est $\texttt{seal}_{\texttt{h.t}}\texttt{(3)}$, ce qui se lit « **scellage** de `3` par `h.t` ». Mais cette approche ne capture qu'une partie du concept de type abstrait tel qu'il est incarné dans les modules à la ML. En effet, dans les systèmes proposés par J. Morris, le scellage est indiqué explicitement, et la corrélation entre scellage et module est de la responsabilité du programmeur. Nous voulons au contraire que le langage maintienne la correspondance de lui-même. En particulier, l'utilisation d'une valeur de type abstrait ne peut être un simple déscellage $\texttt{unseal}_{\texttt{h.t}}\texttt{(...)}$ — ou alors il faut indiquer comment un compilateur peut insérer systématiquement les instructions de déscellage là où elles sont nécessaires.

Examinons dans l'exemple du module `M` ci-dessus quels sont les points où un compilateur devrait ajouter des instructions de scellage ou de déscellage. Pour les membres `M.x` et `M.y`, la situation est simple : `M.x` devient $\texttt{seal}_{\texttt{h.t}}\texttt{(3)}$ et `M.y` devient $\texttt{seal}_{\texttt{h.t}}\texttt{(4)}$. Pour ce qui est de la fonction `M.f`, son type d'entrée est abstrait, donc elle reçoit une valeur déjà scellée ; comme cette valeur fait l'objet d'un calcul arithmétique, il faut la désceller, si bien que la fonction `M.f` est compilée en `fun z ->`

$\texttt{unseal}_{\texttt{h.t}}\texttt{(z) + 1}$. La compilation de la fonction `M.g` est en revanche problématique. En effet, pour fabriquer une valeur de type `M.t * int` à partir d'une valeur de type `int * M.t`, il faut sceller la première composante et désceller la seconde. On pourrait compiler `M.g` en `fun (z1, z2) ->` $\texttt{(seal}_{\texttt{h.t}}\texttt{(z1), unseal}_{\texttt{h.t}}\texttt{(z2))}$, mais cela introduit un calcul supplémentaire (la destruction et la construction d'une paire), ce qui est quelque peu inhabituel au cours d'une compilation.

Nous voyons ainsi que la compilation d'un langage avec types abstraits à la ML vers un langage avec scellage et déscellage est délicate. Elle présente cependant une intuition intéressante : le scellage s'effectue à chaque sortie d'une valeur du module, et le déscellage à chaque entrée d'une valeur dans le module. Le scellage et le déscellage marquent les frontières du module.

III.1.1.3 Crochets colorés

Nous allons réunir les différents éléments satisfaisants dans les solutions envisagées précédemment. De notre étude se dégage qu'il y a trois informations pertinentes : la structure interne de l'expression, son type externe, et l'empreinte désignant le type abstrait. De la conversion, nous retenons la qualification des changements de type autorisés, à savoir que la conversion est possible entre types équivalents modulo l'égalité des types abstraits avec leurs représentations respectives. Du scellage, nous retenons l'idée que les conversions s'effectuent aux frontières du module : ce n'est pas un champ du module qui est converti, mais une valeur qui en entre ou qui en sort.

Nous adoptons les **crochets colorés** proposés par S. Zdancewic, D. Grossman et G. Morrisett [ZGM99, GMZ00]. Si $\mathtt{E}$ est une expression produite par un module d'empreinte $\mathtt{h}$ et dont le type externe est $\mathtt{T}$, nous produisons une expression de type $\mathtt{T}$ en protégeant $\mathtt{E}$ par des crochets colorés, ce qui s'écrit $[\mathtt{E}]_{\mathtt{h}}^{\mathtt{T}}$. Considérant toujours l'exemple introduit à la section III.1.1.1, l'expression source `M.x` est compilée en $[3]_{\mathtt{h}}^{\mathtt{int}}$, tandis que `M.f` est compilée en $[\texttt{fun z -> z+1}]_{\mathtt{h}}^{\texttt{h.t->int}}$, et ainsi de suite. De manière générale, un champ défini par `let x = E` et spécifié dans la signature par `val x : T` est compilé en $[\mathtt{E}]_{\mathtt{h}}^{\mathtt{T}'}$ où $\mathtt{h}$ est l'empreinte du module et $\mathtt{T}'$ est $\mathtt{T}$ dans lequel les références aux champs types du module sont qualifiées comme provenant de l'empreinte (`t` devient `h.t`).

Au niveau syntaxique, les crochets colorés sont des coercitions. Mais au niveau sémantique, l'idée du scellage apporte une intuition importante : une expression de la forme $[\mathtt{E}]_{\mathtt{h}}^{\mathtt{T}}$ signifie que $\mathtt{E}$ provient du module dont l'empreinte est $\mathtt{h}$, et que son type, vu de l'extérieur, n'est pas forcément $\mathtt{T}$; l'expression $[\mathtt{E}]_{\mathtt{h}}^{\mathtt{T}}$ est en revanche utilisable partout avec le type $\mathtt{T}$.

En fait, les crochets colorés diffèrent des coercitions sur un point important : dans l'expression $[\mathtt{E}]_{\mathtt{h}}^{\mathtt{T}}$, l'égalité entre les types abstraits de $\mathtt{h}$ et leurs représentations est utilisable tout au long du typage de $\mathtt{E}$, et non seulement au point frontière. Par exemple, on peut écrire

$$[\texttt{fun (z : h.t) -> (z : int)}]_{\mathtt{h}}^{\texttt{h.t->int}}$$

parce que l'équation $\texttt{h.t} = \texttt{int}$ est visible durant le typage de la fonction. Cette possibilité supplémentaire facilite les manipulations d'expressions contenant des crochets colorés, particulièrement en présence d'annotations de type. Nous tendrons quelquefois à confondre l'empreinte $\mathtt{h}$ avec les équations de type qu'elle représente (à savoir les égalités entre les types abstraits et leurs représentations).

On peut voir les crochets colorés comme une version statique du scellage. Le scellage entre en jeu lorsqu'une valeur traverse la frontière du module ; les crochets incarnent la frontière du module dans la syntaxe. Le code qui provient du module est à l'intérieur des crochets, et le reste du code du programme est à l'extérieur.

L'annotation d'empreinte $\mathtt{h}$ dans le crochet[1] coloré $[\mathtt{E}]_{\mathtt{h}}^{\mathtt{T}}$ est la **couleur** du crochet. Ce nom vient

[1] Nous emploierons souvent le mot « crochet » au singulier pour désigner un nœud de syntaxe qui est matérialisé par une paire de crochets, ou par métonymie à une expression dont un tel nœud de syntaxe est la racine.

de la métaphore suivante : le module est éclairé par une lumière particulière, symbolisée par $\mathtt{h}$, et seul cet éclairage permet de voir les équations de typage utilisables à l'intérieur du module. Les crochets sont une paroi opaque, et une lampe à la fréquence $\mathtt{h}$ en éclaire l'intérieur. L'annotation $\mathtt{h}$ est en fait un cas particulier, signalant une lumière intérieure monochromatique ; en général, on peut spécifier une couleur complexe en indiquant non pas une unique empreinte mais un ensemble d'empreintes ; on notera ainsi $[\mathtt{E}]^{\mathtt{T}}_{\mathtt{h}_1,\ldots,\mathtt{h}_k}$ un crochet à l'intérieur duquel les empreintes $\mathtt{h}_1,\ldots,\mathtt{h}_k$ sont transparentes. Nous noterons (comme nous l'avons fait depuis le début) simplement $\mathtt{h}$ la couleur monochromatique $\{\mathtt{h}\}$. Un crochet peut aussi être d'intérieur sombre, en indiquant que l'ensemble d'équations de typage visibles à l'intérieur est vide ; nous noterons $[\mathtt{E}]^{\mathtt{T}}_{\bullet}$ un tel crochet dans lequel l'expression $\mathtt{E}$ n'« appartient » à aucun module — nous notons $\bullet$ la couleur vide (noire), qui est l'ensemble vide d'empreintes.

III.1.2 Crochets et évaluation

Les exemples de cette section font référence à l'empreinte $\mathtt{h}$ d'un module $\mathtt{M}$ défini par

```
module M = struct           sig
  type t = int                type t
  type u = int * bool   :     type u
  ...                         ...
end                         end
```

III.1.2.1 Types de base

Quand une expression de la forme $[\mathtt{E}]^{\mathtt{T}}_{\mathtt{h}}$ est-elle une valeur ? Une , en principe, est la forme complètement calculée d'une expression ; si nous n'avons pas de raison de distinguer deux expressions, nous voulons qu'elles s'évaluent vers une valeur commune. De plus, les crochets colorés viennent s'ajouter au comportement calculatoire habituel d'un langage, et il convient de minimiser leur influence afin que le langage résultant soit aussi proche que possible d'un langage habituel.

Dans un programme source, les modules forment des unités d'abstraction, et leurs signatures indiquent quels aspects du module sont privés en cachant l'implémentation de certains types. Les crochets jouent ce rôle de barrière d'abstraction durant l'exécution : l'annotation de type $\mathtt{T}$ détermine si (ou plutôt où) le crochet $[\mathtt{E}]^{\mathtt{T}}_{\mathtt{h}}$ est une barrière d'abstraction.

Considérons l'expression $[\mathtt{3}]^{\mathtt{h.t}}_{\mathtt{h}}$. Le type de cette expression est $\mathtt{h.t}$, et la valeur sous-jacente est $\mathtt{3}$. Nous ne pouvons pas nous passer de crochets dans une expression de type $\mathtt{h.t}$, puisqu'une expression sans crochet a forcément un type concret. L'expression $[\mathtt{3}]^{\mathtt{h.t}}_{\mathtt{h}}$ est sous sa forme la plus simple : c'est une valeur.

Considérons maintenant l'expression $[\mathtt{3}]^{\mathtt{int}}_{\mathtt{h}}$. La valeur sous-jacente est encore $\mathtt{3}$, mais le type est $\mathtt{int}$. Les crochets n'apportent ici rien, et il n'y a aucune raison de distinguer cette expression de $\mathtt{3}$. Il est donc logique d'évaluer $[\mathtt{3}]^{\mathtt{int}}_{\mathtt{h}}$ en effaçant les crochets pour former $\mathtt{3}$. Plus généralement, tout crochet coloré dont l'annotation de type est concrète peut être effacé. Par exemple, $[\mathtt{(3,true)}]^{\mathtt{int*bool}}_{\mathtt{h}}$ a pour valeur `(3,true)`.

Nous voyons que c'est l'annotation de type sur un crochet qui détermine si celui-ci est une valeur. Notre règle provisoire est qu'un crochet est une valeur si le type est abstrait, sinon il est concret. Provisoire, parce que de nombreux cas restent à préciser.

III.1.2.2 Poussée de crochets

Considérons l'expression $[\texttt{(3,4)}]_{\mathsf{h}}^{\texttt{int*h.t}}$. La première composante de la paire a au final un type concret, et la seconde un type abstrait. Le crochet est indispensable au bon typage de la seconde composante, mais cela n'implique pas que l'expression doive être une valeur. En effet, une autre expression ayant la même valeur sous-jacente et le même type est $\texttt{(3, }[4]_{\mathsf{h}}^{\texttt{h.t}}\texttt{)}$. Dans cette seconde expression, le crochet est autour de la partie abstraite de l'expression, et la partie concrète est libre de tout ajout. Dans l'expression originale $[\texttt{(3,4)}]_{\mathsf{h}}^{\texttt{int*h.t}}$, nous pouvons pousser le crochet vers l'intérieur de l'expression, pour obtenir $\texttt{(3, }[4]_{\mathsf{h}}^{\texttt{h.t}}\texttt{)}$ qui est une valeur.

Ce phénomène est général : lorsqu'un crochet entoure un constructeur, il peut être poussé vers l'intérieur. Ici encore, c'est l'annotation de type qui détermine si la poussée est possible : toute expression de la forme $[(\mathsf{V}_1, \mathsf{V}_2)]_{\mathsf{h}}^{\mathsf{T}_1 * \mathsf{T}_2}$ s'évalue en $([\mathsf{V}_1]_{\mathsf{h}}^{\mathsf{T}_1}, [\mathsf{V}_2]_{\mathsf{h}}^{\mathsf{T}_2})$ (si le type de l'expression à l'intérieur du crochet est un produit, l'expression est forcément un produit manifeste, sous réserve qu'elle soit une valeur). La poussée peut éventuellement continuer à l'intérieur de V_1 et V_2. En revanche, une expression comme $[\texttt{(3,true)}]_{\mathsf{h}}^{\texttt{h.u}}$ est une valeur, puisque le type abstrait `h.u` interdit de pousser plus avant.

Pousser un crochet à l'intérieur d'un constructeur n-aire remplace un crochet par n crochets, mais chacun d'eux entoure une expression plus petite. Le cas d'un crochet portant une annotation comme `int` peut être vu comme un cas particulier : on pousse le crochet à l'intérieur d'un crochet 0-aire, et il reste zéro crochet, autrement dit le crochet disparaît.

Ce que nous avons affirmé pour le type `int` ne dépend pas de si celui-ci est un type prédéfini ou un type abstrait fourni par une bibliothèque. Il semble donc permettre d'effacer tout crochet dont l'annotation de type est un type abstrait qui n'est pas fourni par le module qui a donné lieu au crochet : si h et h' sont deux empreintes distinctes, $[\mathsf{E}]_{\mathsf{h}}^{\mathsf{h}'\texttt{.t}}$ devrait s'évaluer en E. Ce crochet n'a effectivement pas de rôle de conversion de type : la connaissance de h ne peut aider à fabriquer une valeur de type $\mathsf{h}'\texttt{.t}$, c'est forcément un autre crochet, plus profond dans E, qui a permis la création d'une valeur de ce type. Nous ne pouvons cependant pas réduire systématiquement $[\mathsf{E}]_{\mathsf{h}}^{\mathsf{h}'\texttt{.t}}$ en E, car rien n'empêche E d'utiliser la connaissance de h. Dans le système HAT, ous énoncerons pour traiter ce cas une règle spécialisée ($\mathcal{H}$/ered.col.col), dont la suffisance demandera une démonstration.

III.1.2.3 Couleur ambiante

Nous avons vu à la section III.1.1.3 que l'empreinte annotant un crochet coloré influence le typage de l'expression à l'intérieur du crochet, puisqu'elle permet d'utiliser les équations de typage correspondantes. Pour typer une expression, il faut donc fixer un environnement indiquant non seulement les types des variables libres mais aussi la **couleur ambiante**. De même, l'évaluation d'une expression est définie par rapport à une couleur ambiante ; nous notons $\mathsf{E} \longrightarrow_{\mathsf{h}} \mathsf{E}'$ lorsque E s'évalue en E' dans la couleur h. La couleur ambiante interviendra bien sûr pour le typage dynamique, mais aussi pour évaluer les crochets. Dans les sections précédentes, nous avons implicitement supposé que la couleur ambiante était vide ; ce n'est plus le cas dès que l'on observe les calculs effectués à l'intérieur d'un module, ce qui se traduit à l'exécution par l'évaluation sous les crochets.

Un cas particulier important de constructeur, que nous n'avons pas encore abordé, est celui d'un type fonctionnel $\mathsf{T}_0 \to \mathsf{T}_1$. Une valeur de ce type est une fonction immédiate `fun x -> E` ; que signifie pousser un crochet dans une fonction ? Une expression de la forme $[\texttt{fun x -> E}]_{\mathsf{h}}^{\mathsf{T}_0 \to \mathsf{T}_1}$ désigne une fonction à l'intérieur du module dont l'empreinte est h. La révéler comme une fonction consiste à la faire apparaître comme un mécanisme qui prend une valeur, la fait rentrer dans le module, calcule, et enfin fait ressortir le résultat du module. Le résultat appartient au monde du module, il doit donc être protégé par un crochet annoté par h. Inversement, l'argument appartient au monde extérieur :

il doit être protégé par la couleur ambiante. Nous en déduisons la règle de réduction suivante :

$$[\texttt{fun (x : }\mathsf{T}_2\texttt{) -> E}]_{\mathsf{h}}^{\mathsf{T}_0\to\mathsf{T}_1} \quad \longrightarrow_{\mathsf{c}} \quad \texttt{fun (y : }\mathsf{T}_0\texttt{) -> }[\{\mathsf{x}\leftarrow[\mathsf{y}]_{\mathsf{h}}^{\mathsf{T}_2}\}\mathsf{E}]_{\mathsf{h}}^{\mathsf{T}_1}$$

(La notation $\{\mathsf{x}\leftarrow[\mathsf{y}]_{\mathsf{h}}^{\mathsf{T}_2}\}\mathsf{E}$ désigne E dans laquelle $[\mathsf{y}]_{\mathsf{c}}^{\mathsf{T}_0}$ est substituée à x.) On note que les types T_2 et T_0 doivent être équivalents dans la couleur interne h pour que le rédex soit bien typé, mais ils peuvent ne pas l'être dans la couleur ambiante. Ceci force l'annotation de type sur l'argument de la fonction dans le réduit à être T_0, son type extérieur. De plus, un passage de l'argument par la couleur c est nécessaire, puisqu'il a le type T_0 *a priori* alors que T_2 est attendu dans E.

On remarque qu'il apparaît ici un crochet encadrant une expression $\{\mathsf{x}\leftarrow[\mathsf{y}]_{\mathsf{h}}^{\mathsf{T}_2}\}\mathsf{E}$ (le corps d'une fonction) qui n'est pas une valeur. La poussée des crochets ne peut donc être décrite comme une opération autonome sur les valeurs ; elle doit être intégrée à la relation de réduction qui décrit la sémantique opérationnelle du langage.

Une autre manière dont la couleur ambiante intervient est qu'elle rend certains crochets inutiles. Si la couleur ambiante contient h, un crochet de la forme $[\mathsf{E}]_{\mathsf{h}}^{\mathsf{T}}$ est inutile, puisqu'il n'apporte aucune connaissance supplémentaire à l'expression E. Plus généralement, un crochet $[\mathsf{E}]_{\mathsf{c}'}^{\mathsf{T}}$ peut être effacé dès que c' est incluse dans la couleur ambiante. En particulier, un crochet dont l'annotation est la couleur vide $\bullet$ apparaît comme inutile ; nous en verrons l'intérêt lorsque nous aborderons le typage dynamique.

Nous avons désormais assez d'informations pour décider quand un crochet est une valeur. Pour que l'expression $[\mathsf{E}]_{\mathsf{c}'}^{\mathsf{T}}$ soit une valeur, il faut que E soit une valeur (les crochets ne bloquent pas l'évaluation). Il faut également que T soit de la forme $\mathtt{h.t}$ où $\mathtt{t}$ est un type abstrait dans le module dont h est l'empreinte. Mais même cela ne suffit pas : si le type abstrait n'est pas transparent à l'intérieur, c'est-à-dire si h n'appartient pas à c', le crochet ne contribue pas à l'abstraction ; d'autre part, si le type abstrait est transparent à l'extérieur, c'est-à-dire si h appartient à la couleur ambiante c, la couleur ambiante suffit à typer l'intérieur et l'effet protecteur du crochet n'est pas utile. Finalement, $[\mathsf{E}]_{\mathsf{c}'}^{\mathsf{T}}$ est une valeur dans la couleur c si et seulement si E est-une valeur (dans la couleur c') et $\mathsf{h} \in \mathsf{c}' \setminus \mathsf{c}$.

III.2 Le langage HAT

III.2.1 Introduction

Dans cette section, nous décrivons le langage HAT[2], un langage restreint équipé de crochets colorés. Il intègre les fonctionnalités suivantes :
- un noyau de lambda-calcul simplement typé ;
- un système de modules simplifié ;
- un calcul d'empreintes, et des crochets colorés ;
- une possibilité de typage dynamique ;
- un canal de communication inter-machines.

Le système de modules de HAT est réduit au minimum permettant de définir des types abstraits : un module de HAT consiste en exactement un champ type et un champ terme. Nous nous restreignons au cas où le champ terme est une valeur, ce qui revient à supposer que le module n'a pas de code d'initialisation. Une signature qualifie chacun des champs ; le champ type reçoit une indication de **sorte** (*kind*) : TYPE indique un type abstrait, tandis que EQ(T) indique un type concret (on parle de **sorte singleton**). La table suivante donne la correspondance entre la syntaxe d'Objective Caml et celle de HAT :

[2] *Hashed Abstract Types* (empreintes de types abstraits).

`struct type t = `$\mathsf{T_0}$` let y = `V` end`	$\leftrightarrow$	$[\mathsf{T_0}, \mathsf{V}]$
`sig type t val y : T end`	$\leftrightarrow$	$[\mathsf{X}{:}\textsc{Type}, \mathsf{T}]$
`sig type t = `$\mathsf{T_1}$` val y : T end`	$\leftrightarrow$	$[\mathsf{X}{:}\textsc{Eq}(\mathsf{T_1}), \mathsf{T}]$

Comme le langage HAT est destiné à illustrer la notion d'empreinte, nous omettons toute possibilité de composer des modules (modules imbriqués, foncteurs, ...). Un programme du langage HAT consiste en une suite de définitions de modules suivies d'une expression (le programme principal) :

$$\mathsf{let}\, U_1 = M_1 : S_1 \ \mathsf{in} \quad \ldots \quad \mathsf{let}\, U_k = M_k : S_k \ \mathsf{in}\ E$$

Un module peut être référencé par son nom U_i aussi bien dans les définitions de modules suivantes que dans le programme principal.

Le langage HAT peut décrire des réseaux de machines communicantes. Un programme de la forme indiquée ci-dessus est compilé et exécuté sur chaque machine. La compilation (typage et liaison des modules) agit sur chaque machine indépendamment, mais les différents programmes peuvent s'échanger des messages typés dynamiquement durant leur exécution.

Dans la section III.2.2, nous décrivons la syntaxe complète du langage HAT, ainsi que de la métathéorie subséquente. La section III.2.3 décrit le système de types. La section III.2.4 présente le mécanisme de liaison des modules qui a lieu après le typage sur chaque machine. La section III.2.5 décrit l'évaluation d'un réseau de programmes HAT. La section III.2.6 illustre les principales fonctionnalités du langage par un exemple de programme. La section III.2.7 énonce des propriétés mathématiques de cohérence et d'adéquation du langage.

III.2.2 Syntaxe

III.2.2.1 Syntaxe du langage

La figure III.1 décrit la syntaxe du langage HAT. Le langage des expressions est basé sur le lambda-calcul typé. Nous incluons les paires comme exemple de structure de données. Les notations $U.\mathsf{term}$ et $U.\mathsf{type}$ désignent respectivement les champs terme et type du module de nom U. Après la compilation, les références à des types abstraits sont transformées en empreintes h ; nous pouvons traiter l'empreinte d'un module comme un type parce que chaque module définit exactement un type.

Les crochets colorés ont été décrits à la section III.1.2. L'expression $\mathsf{dyn}\, E\, \mathsf{at}\, T$ fabrique un **dynamique** annoté par le type T (voir la section I.3.2) ; le type des dynamiques est DYN ; la variante $\mathsf{dynned}\, E\, \mathsf{at}\, T$ sera motivée à la section III.2.5. L'expression $\mathsf{undyn}\, E'\, \mathsf{at}\, T$ vérifie que E' est un dynamique de type T et renvoie l'expression sous-jacente ; en cas d'échec, elle lève l'exception Fail_T. Les primitives de communication sont notées ? et $!E$ suivant la tradition des calculs concurrents.

Les modules de HAT sont des structures composées d'un type et d'un terme. Les signatures sont en conséquences composées d'une sorte (qualifiant le type) et d'un type (qualifiant le terme). Dans une signature $[X{:}K, T]$, le nom X peut être utilisé dans le type T pour désigner le type contenu dans le premier champ ; à noter que ce type est abstrait (c'est-à-dire qu'il est distinct du champ type du module) lorsque la signature est opaque (c'est-à-dire que $K = \textsc{Type}$).

Les empreintes de HAT sont toujours structurelles. Pour décrire HAT, nous n'avons pas besoin de couleurs contenant plus d'une empreinte ; une couleur de HAT est donc soit la couleur vide $\bullet$, soit une couleur singleton notée simplement h. Si $h = \mathsf{hash}([T, V] : [X{:}\textsc{Type}, T'])$, le **type implémentation** (ou **type représentation**) de h, noté $\mathbf{impl}\, h$, est T.

Nous décrivons dès à présent les valeurs du langage HAT. La figure III.2 définit les **quasi-valeurs**, notées V. Une quasi-valeur est une valeur aux couleurs près : n'importe quel crochet coloré y est autorisé. La notion de valeur est paramétrée par la couleur ambiante : une valeur dans la couleur $\mathbf{c}$

$\mathsf{E} ::=$	**expression**
$x \mid y \mid z$	variables
$()$	valeur unité
$(\mathsf{E}_1, \mathsf{E}_2)$	paire
$\pi_i\, \mathsf{E}$	$i^{\text{ème}}$ projection
$\lambda x : \mathsf{T}.\ \mathsf{E}$	lambda-abstraction
$\mathsf{E}_1\, \mathsf{E}_2$	application de fonction
$\mathsf{U}.\mathsf{term}$	champ terme d'un module
$[\mathsf{E}]_c^{\mathsf{T}}$	† crochet coloré
$\mathsf{dyn}\ \mathsf{E}\ \mathsf{at}\ \mathsf{T}$	dynamique
$\mathsf{dynned}\ \mathsf{E}\ \mathsf{at}\ \mathsf{T}$	† dynamique universel
$\mathsf{undyn}\ \mathsf{E}\ \mathsf{at}\ \mathsf{T}$	vérification de typage dynamique
$\mathsf{Fail}_{\mathsf{T}}$	† exception (levée par $\mathsf{undyn}\ \_\ \mathsf{at}\ \mathsf{T}$)
$?$	réception de message
$!\mathsf{E}$	envoi de message

$h ::=$	† **empreinte**
$\mathsf{hash}(M : [X{:}\textsc{Type},\ \mathsf{T}])$	† empreinte structurelle

$c ::=$	† **couleur**
$\bullet$	† couleur vide
h	† empreinte

$\mathsf{L} ::=$	**machine**
E	programme principal
$\mathsf{let}\ \mathsf{U} = \mathsf{M} : \mathsf{S}\ \mathsf{in}\ \mathsf{L}$	déclaration de module

$\mathsf{T} ::=$	**type**
$X \mid Y \mid Z$	variables
$\textsc{unit}$	type unité
$\mathsf{T}_1 * \mathsf{T}_2$	type produit
$\mathsf{T}_1 \to \mathsf{T}_2$	type fonctionnel
$\mathsf{U}.\mathsf{type}$	champ type d'un module
h	† empreinte
$\textsc{dyn}$	type dynamique

$\mathsf{K} ::=$	**sorte**
$\textsc{Type}$	sorte opaque (abstraite)
$\textsc{Eq}(\mathsf{T})$	sorte singleton (concrète)

$\mathsf{U} ::=$	**variable de module**

$\mathsf{M} ::=$	**structure**
$[\mathsf{T},\ \mathsf{E}]$	champ type, champ terme

$\mathsf{S} ::=$	**signature**
$[X{:}\mathsf{K},\ \mathsf{T}]$	champ type (nommé X) et champ terme

$\mathsf{N} ::=$	**réseau**
0	réseau vide
E	expression sur une machine
$\mathsf{N} \parallel \mathsf{N}$	composition parallèle

La note † indique les éléments qui ne sont en principe pas présents dans les programmes sources, mais qui apparaissent lors de la compilation ou de l'exécution.

FIG. III.1 – Syntaxe du langage HAT

$\mathsf{V} ::=$	**quasi-valeur**
$()$	valeur unité
$(\mathsf{V}_1, \mathsf{V}_2)$	paire
$\lambda x : \mathsf{T}.\ \mathsf{E}$	lambda-abstraction
$[\mathsf{V}]_{c'}^{\mathsf{T}}$	crochet coloré
$\mathsf{dynned}\ \mathsf{V}\ \mathsf{at}\ \mathsf{T}$	dynamique universel

$\mathsf{V}^c ::=$	**valeur dans c**
$()$	valeur unité
$(\mathsf{V}_1^c, \mathsf{V}_2^c)$	paire
$\lambda x : \mathsf{T}.\ \mathsf{E}$	lambda-abstraction
$[\mathsf{V}^h]_h^h$	crochet coloré, si $h \neq c$
$\mathsf{dynned}\ \mathsf{V}^c\ \mathsf{at}\ \mathsf{T}$	dynamique universel

FIG. III.2 – Quasi-valeurs et valeurs

est notée V^c. Pour qu'une quasi-valeur soit une valeur, il faut que les seuls crochets colorés qu'elle contienne (sauf sous un lambda) soient de la forme $[\mathsf{V}]_h^h$ où h n'est pas dans la couleur ambiante ; nous verrons le pourquoi de cette restriction à la section III.2.5.

III.2.2.2 Métathéorie

La figure III.3 présente des notations utilisées dans la métathéorie. Le langage HAT comporte trois espèces de variables : les variables d'expression, les variables de type et les variables de module ; une variable d'espèce indifférente pourra être notée ζ.

$\zeta ::=$	**variable**
$x \mid y \mid z$	variable d'expression
$X \mid Y \mid Z$	variable de type
U	variable de module
$\ddot{x} ::=$	**substituable d'expression**
$x \mid y \mid z$	variable d'expression
$U.\mathsf{term}$	partie expression d'un module
$\ddot{X} ::=$	**substituable de type**
$X \mid Y \mid Z$	variable de type
$U.\mathsf{type}$	partie type d'un module
$\chi ::=$	**entité substituable**
$x \mid y \mid z$	variable d'expression
$U.\mathsf{term}$	partie expression d'un module
$X \mid Y \mid Z$	variable de type
$U.\mathsf{type}$	partie type d'un module
$\sigma ::=$	**substitution**
id	identité
$\{\ddot{x}\leftarrow E\}$	remplacement de $\ddot{x}$ par E
$\{\ddot{X}\leftarrow T\}$	remplacement de $\ddot{X}$ par T
$\sigma_2\sigma_1$	composition (σ_1 puis σ_2)

FIG. III.3 – Notations de la métathéorie

$J ::=$	**membre droit de jugement coloré**
ok	correction de l'environnement
$K\,\mathsf{ok}$	correction d'une sorte
$K \equiv K'$	équivalence de sortes
$K <: K'$	sous-sortage
$T : K$	sorte d'un type
$T \equiv T'$	équivalence de types
$S\,\mathsf{ok}$	correction d'une signature
$S <: S'$	sous-signaturage
$E : T$	type d'une expression
$M : S$	signature d'une structure
$U{:}S$	signature d'un nom de module
$L : T$	correction d'une machine
$\Gamma ::=$	**environnement**
nil	environnement vide
$x{:}T$	liaison de variable d'expression
$X{:}K$	liaison de variable de type
$U{:}S$	liaison de variable de module
Γ_1, Γ_2	concaténation
$\mathscr{M} ::=$	**jugement monochrome**
$\vdash c\,\mathsf{ok}$	correction de couleur
$\vdash N\,\mathsf{ok}$	correction de réseau
$\mathscr{J} ::=$	**jugement de typage**
$\zeta \notin \mathbf{dom}\,\Gamma$	absence de conflit de variables
$\Gamma \vdash_c J$	jugement coloré
$\mathscr{M}$	jugement monochrome

FIG. III.4 – Syntaxe des jugements de typage

Les substitutions sont notées sous forme préfixe, comme le sont habituellement les fonctions en Mathématiques : le remplacement de x par y dans E est noté $\{x\leftarrow y\}E$. Les modules ne peuvent pas être substitués directement : la syntaxe ne permet pas de placer une structure là où un nom de module est autorisé. En revanche, une projection d'un champ de module peut être substituée ; par exemple nous aurons $\{U.\mathsf{term}\leftarrow x\}(U.\mathsf{term}, x) = (x, x)$. Ceci motive notre définition des **entités substituables** χ : les variables d'expression et de type, et les champs de noms de module.

Nous travaillons systématiquement à alpha-conversion près. L'égalité est l'égalité syntaxique : la notation $\aleph = \beth$ signifie que $\aleph$ et $\beth$ sont égaux à alpha-conversion près.

Les substitutions sont définies de la manière usuelle sur tous les éléments de la syntaxe du langage et de la métathéorie, en tenant compte du fait qu'elles agissent sur des entités substituables. Nous définissons également de la manière usuelle les variables libres $\mathbf{fv}\,\aleph$ et les entités substituables libres $\mathbf{fse}\,\aleph$ d'une entité syntaxique.

III.2.3 Typage

Le système de types de HAT comporte des jugements de diverses sortes dont la figure III.4 décrit la syntaxe. Nous présentons tour à tour les règles permettant de dériver chaque forme de jugement.

III.2.3.1 $\boxed{\zeta \notin \mathbf{dom}\,\Gamma}$ Absence de conflit de variables

Le jugement $\zeta \notin \mathbf{dom}\,\Gamma$ a exactement sa signification intuitive, à savoir que ζ n'est pas dans le domaine de Γ. Il nous est commode dans certaines démonstrations de traiter formellement ces jugements. Nous supposons disposer de la famille d'axiomes $\zeta \neq \zeta'$ pour toutes les paires de variables distinctes (ζ, ζ').

$$\frac{}{\zeta \notin \mathbf{dom}\,\mathsf{nil}}\ (\mathcal{H}/\textsf{clash.nil}) \qquad \frac{\zeta \notin \mathbf{dom}\,\Gamma \qquad \text{si } \zeta \neq \zeta'}{\zeta \notin \mathbf{dom}\,(\Gamma, \zeta' : \tau)}\ (\mathcal{H}/\textsf{clash.cons})$$

III.2.3.2 $\boxed{\vdash \mathsf{c\,ok}}$ Correction d'une couleur

Une empreinte est correcte lorsque son module a la signature annoncée. Le module est typé dans une couleur vide. Seuls les définitions de modules dont le type est abstrait peuvent donner lieu à une empreinte.

$$\frac{}{\vdash \bullet\,\mathsf{ok}}\ (\mathcal{H}/\textsf{hmok.o}) \qquad \frac{\mathsf{nil} \vdash_\bullet \mathsf{M} : [\mathsf{X}{:}\textsc{Type},\ \mathsf{T}]}{\vdash \mathsf{hash}(\mathsf{M} : [\mathsf{X}{:}\textsc{Type},\ \mathsf{T}])\,\mathsf{ok}}\ (\mathcal{H}/\textsf{hmok.hash})$$

III.2.3.3 $\boxed{\Gamma \vdash_\mathsf{c}\ \mathsf{ok}}$ Correction de l'environnement

Les environnement sont dépendants : l'annotation portant sur une variable peut mentionner les variables précédentes. La construction d'un environnement s'effectue par ajouts successifs d'une variable à droite ; la variable ajoutée ne doit pas cacher une variable précédente. La couleur est constante au cours de la construction, elle est vérifiée une fois pour toute par $(\mathcal{H}/\textsf{envok.nil})$.

$$\frac{\vdash \mathsf{c\,ok}}{\mathsf{nil} \vdash_\mathsf{c}\ \mathsf{ok}}\ (\mathcal{H}/\textsf{envok.nil}) \qquad \frac{\Gamma \vdash_\mathsf{c} \mathsf{T} : \textsc{Type} \qquad \mathsf{x} \notin \mathbf{dom}\,\Gamma}{\Gamma, \mathsf{x}{:}\mathsf{T} \vdash_\mathsf{c}\ \mathsf{ok}}\ (\mathcal{H}/\textsf{envok.x})$$

$$\frac{\Gamma \vdash_\mathsf{c} \mathsf{K\,ok} \qquad \mathsf{X} \notin \mathbf{dom}\,\Gamma}{\Gamma, \mathsf{X}{:}\mathsf{K} \vdash_\mathsf{c}\ \mathsf{ok}}\ (\mathcal{H}/\textsf{envok.X}) \qquad \frac{\Gamma \vdash_\mathsf{c} \mathsf{S\,ok} \qquad \mathsf{U} \notin \mathbf{dom}\,\Gamma}{\Gamma, \mathsf{U}{:}\mathsf{S} \vdash_\mathsf{c}\ \mathsf{ok}}\ (\mathcal{H}/\textsf{envok.U})$$

III.2.3.4 $\boxed{\Gamma \vdash_\mathsf{c} \mathsf{K\,ok}}$ Correction d'une sorte

$$\frac{\Gamma \vdash_\mathsf{c}\ \mathsf{ok}}{\Gamma \vdash_\mathsf{c} \textsc{Type}\,\mathsf{ok}}\ (\mathcal{H}/\textsf{Kok.Type}) \qquad \frac{\Gamma \vdash_\mathsf{c} \mathsf{T} : \textsc{Type}}{\Gamma \vdash_\mathsf{c} \textsc{Eq}(\mathsf{T})\,\mathsf{ok}}\ (\mathcal{H}/\textsf{Kok.Eq})$$

III.2.3.5 $\boxed{\Gamma \vdash_\mathsf{c} \mathsf{K} \equiv \mathsf{K}'}$ Équivalence de sortes

L'équivalence des sortes est essentiellement celle des types imbriqués.

$$\frac{\Gamma \vdash_\mathsf{c}\ \mathsf{ok}}{\Gamma \vdash_\mathsf{c} \textsc{Type} \equiv \textsc{Type}}\ (\mathcal{H}/\textsf{Keq.Type}) \qquad \frac{\Gamma \vdash_\mathsf{c} \mathsf{T} \equiv \mathsf{T}'}{\Gamma \vdash_\mathsf{c} \textsc{Eq}(\mathsf{T}) \equiv \textsc{Eq}(\mathsf{T}')}\ (\mathcal{H}/\textsf{Keq.Eq})$$

III.2.3.6 $\boxed{\Gamma \vdash_\mathsf{c} \mathsf{K} <: \mathsf{K}'}$ Sous-sortage

Le but du sous-sortage est de déclarer la sorte $\textsc{Type}$ comme plus grande que les sortes singletons, ce qui signifie que $\textsc{Type}$ est une information moins précise concernant un type que $\textsc{Eq}(\mathsf{T})$.

$$\frac{\Gamma \vdash_c \mathsf{T} : \textsc{Type}}{\Gamma \vdash_c \textsc{Eq}(\mathsf{T}) <: \textsc{Type}}\ (\mathcal{H}/\textsf{\textbf{Ksub.Eq}})$$

$$\frac{\Gamma \vdash_c \mathsf{K} \equiv \mathsf{K}'}{\Gamma \vdash_c \mathsf{K} <: \mathsf{K}'}\ (\mathcal{H}/\textsf{\textbf{Ksub.refl}}) \qquad \frac{\Gamma \vdash_c \mathsf{K} <: \mathsf{K}' \qquad \Gamma \vdash_c \mathsf{K}' <: \mathsf{K}''}{\Gamma \vdash_c \mathsf{K} <: \mathsf{K}''}\ (\mathcal{H}/\textsf{\textbf{Ksub.tran}})$$

III.2.3.7 $\boxed{\Gamma \vdash_c \mathsf{T} : \mathsf{K}}$ Correction d'un type

Chaque production de la grammaire des types donne lieu à une règle de correction correspondante. Les règles structurelles attribuent aux types la sorte $\textsc{Type}$; à charge de $(\mathcal{H}/\textsf{TK.Eq})$ de permettre une sorte singleton. La règle $(\mathcal{H}/\textsf{TK.var})$ dévie de ce principe car elle reprend simplement la sorte issue de l'environnement ; $(\mathcal{H}/\textsf{TK.sub})$ peut être utilisée si une autre sorte est désirée. Dans la règle $(\mathcal{H}/\textsf{TK.hash})$, on remarque que la correction de l'empreinte h ne fait pas appel à l'environnement ni à la couleur ambiante (d'où la prémisse supplémentaire qui assure leur correction).

$$\frac{\Gamma, \mathsf{X}{:}\mathsf{K}, \Gamma' \vdash_c \mathsf{ok}}{\Gamma, \mathsf{X}{:}\mathsf{K}, \Gamma' \vdash_c \mathsf{X} : \mathsf{K}}\ (\mathcal{H}/\textsf{\textbf{TK.var}}) \qquad \frac{\Gamma \vdash_c \mathsf{ok}}{\Gamma \vdash_c \textsc{unit} : \textsc{Type}}\ (\mathcal{H}/\textsf{\textbf{TK.Unit}})$$

$$\frac{\Gamma \vdash_c \mathsf{T}_1 : \textsc{Type} \qquad \Gamma \vdash_c \mathsf{T}_2 : \textsc{Type}}{\Gamma \vdash_c \mathsf{T}_1 * \mathsf{T}_2 : \textsc{Type}}\ (\mathcal{H}/\textsf{\textbf{TK.pair}}) \qquad \frac{\Gamma \vdash_c \mathsf{T}_1 : \textsc{Type} \qquad \Gamma \vdash_c \mathsf{T}_2 : \textsc{Type}}{\Gamma \vdash_c \mathsf{T}_1 \to \mathsf{T}_2 : \textsc{Type}}\ (\mathcal{H}/\textsf{\textbf{TK.fun}})$$

$$\frac{\Gamma \vdash_c \mathsf{U} : [\mathsf{X}{:}\mathsf{K},\ \mathsf{T}]}{\Gamma \vdash_c \mathsf{U}.\mathsf{type} : \mathsf{K}}\ (\mathcal{H}/\textsf{\textbf{TK.mod}}) \qquad \frac{\vdash \mathsf{h}\,\mathsf{ok} \qquad \Gamma \vdash_c \mathsf{ok}}{\Gamma \vdash_c \mathsf{h} : \textsc{Type}}\ (\mathcal{H}/\textsf{\textbf{TK.hash}}) \qquad \frac{\Gamma \vdash_c \mathsf{ok}}{\Gamma \vdash_c \textsc{dyn} : \textsc{Type}}\ (\mathcal{H}/\textsf{\textbf{TK.Dyn}})$$

Nous disposons également de deux règles non structurelles. La règle $(\mathcal{H}/\textsf{TK.sub})$ est une règle de sous-sortage ; elle permet en particulier d'attribuer la sorte $\textsc{Type}$ à tout type ayant une sorte singleton. La règle $(\mathcal{H}/\textsf{TK.Eq})$ énonce que si deux types sont équivalents alors l'un est dans la sorte de l'autre ; grâce à $(\mathcal{H}/\textsf{Teq.refl})$, si un type T a la sorte $\textsc{Type}$ alors il a aussi la sorte $\textsc{Eq}(\mathsf{T})$.

$$\frac{\Gamma \vdash_c \mathsf{T} : \mathsf{K} \qquad \Gamma \vdash_c \mathsf{K} <: \mathsf{K}'}{\Gamma \vdash_c \mathsf{T} : \mathsf{K}'}\ (\mathcal{H}/\textsf{\textbf{TK.sub}}) \qquad \frac{\Gamma \vdash_c \mathsf{T} \equiv \mathsf{T}'}{\Gamma \vdash_c \mathsf{T} : \textsc{Eq}(\mathsf{T}')}\ (\mathcal{H}/\textsf{\textbf{TK.Eq}})$$

III.2.3.8 $\boxed{\Gamma \vdash_c \mathsf{T} \equiv \mathsf{T}'}$ Équivalence de types

Nous explicitons le fait que $\mathsf{T} \equiv \mathsf{T}'$ est une relation d'équivalence.

$$\frac{\Gamma \vdash_c \mathsf{T} : \textsc{Type}}{\Gamma \vdash_c \mathsf{T} \equiv \mathsf{T}}\ (\mathcal{H}/\textsf{\textbf{Teq.refl}}) \qquad \frac{\Gamma \vdash_c \mathsf{T} \equiv \mathsf{T}'}{\Gamma \vdash_c \mathsf{T}' \equiv \mathsf{T}}\ (\mathcal{H}/\textsf{\textbf{Teq.sym}}) \qquad \frac{\Gamma \vdash_c \mathsf{T} \equiv \mathsf{T}' \qquad \Gamma \vdash_c \mathsf{T}' \equiv \mathsf{T}''}{\Gamma \vdash_c \mathsf{T} \equiv \mathsf{T}''}\ (\mathcal{H}/\textsf{\textbf{Teq.tran}})$$

La règle $(\mathcal{H}/\textsf{Teq.Eq})$ a l'effet inverse de $(\mathcal{H}/\textsf{TK.Eq})$; la combinaison de ces deux règles énonce que les affirmations « T a la sorte singleton de T' » et « T et T' sont équivalents » ont la même signification.

$$\frac{\Gamma \vdash_c \mathsf{T} : \textsc{Eq}(\mathsf{T}')}{\Gamma \vdash_c \mathsf{T} \equiv \mathsf{T}'}\ (\mathcal{H}/\textsf{\textbf{Teq.Eq}})$$

L'équivalence des types est une congruence par rapport aux manières de construire ceux-ci.

$$\frac{\Gamma \vdash_c \mathsf{T}_0 \equiv \mathsf{T}_0' \qquad \Gamma \vdash_c \mathsf{T}_1 \equiv \mathsf{T}_1'}{\Gamma \vdash_c \mathsf{T}_0 \to \mathsf{T}_1 \equiv \mathsf{T}_0' \to \mathsf{T}_1'}\ (\mathcal{H}/\textsf{\textbf{Teq.cong.fun}}) \qquad \frac{\Gamma \vdash_c \mathsf{T}_1 \equiv \mathsf{T}_1' \qquad \Gamma \vdash_c \mathsf{T}_2 \equiv \mathsf{T}_2'}{\Gamma \vdash_c \mathsf{T}_1 * \mathsf{T}_2 \equiv \mathsf{T}_1' * \mathsf{T}_2'}\ (\mathcal{H}/\textsf{\textbf{Teq.cong.pair}})$$

La règle qui donne son contenu essentiel à l'équivalence des types est $(\mathcal{H}/\textsf{Teq.hash})$. Elle indique que si la couleur ambiante est une empreinte h, alors h est transparente : en tant que type, h est équivalente au type implémentation contenu dans l'empreinte (noté $\mathbf{impl}\,\mathsf{h}$ — rappelons que $\mathbf{impl}(\mathsf{hash}([\mathsf{T},\ \mathsf{V}] : [\mathsf{X}{:}\textsc{Type},\ \mathsf{T}'])) = \mathsf{T}$).

$$\frac{\Gamma \vdash_h \mathsf{ok}}{\Gamma \vdash_h h \equiv \mathbf{impl}\, h}\ (\mathcal{H}/\textsf{\textbf{Teq.hash}})$$

III.2.3.9 $\boxed{\Gamma \vdash_c \mathsf{S}\,\mathsf{ok}}$ Correction d'une signature

Une signature [X:K, T] est dépendante : la variable X lie dans T.

$$\frac{\Gamma, \mathsf{X}{:}\mathsf{K} \vdash_c \mathsf{T} : \textsc{Type}}{\Gamma \vdash_c [\mathsf{X}{:}\mathsf{K},\ \mathsf{T}]\,\mathsf{ok}}\ (\mathcal{H}/\textsf{\textbf{Sok}})$$

III.2.3.10 $\boxed{\Gamma \vdash_c \mathsf{S} <: \mathsf{S}'}$ Sous-signaturage

Nous spécifions que le sous-signaturage est un préordre (même si cela est en fait une conséquence de ce que le sous-sortage et l'équivalence des types le sont).

$$\frac{\Gamma \vdash_c \mathsf{S}\,\mathsf{ok}}{\Gamma \vdash_c \mathsf{S} <: \mathsf{S}}\ (\mathcal{H}/\textsf{\textbf{Ssub.refl}}) \qquad \frac{\Gamma \vdash_c \mathsf{S} <: \mathsf{S}' \qquad \Gamma \vdash_c \mathsf{S}' <: \mathsf{S}''}{\Gamma \vdash_c \mathsf{S} <: \mathsf{S}''}\ (\mathcal{H}/\textsf{\textbf{Ssub.tran}})$$

L'origine du sous-signaturage est le sous-sortage : une signature est plus précise qu'une autre si l'information donnée sur le champ type y est plus précise. Comme notre langage ne comporte pas de sous-typage, deux types ne sont commensurables que lorsqu'ils sont équivalents, donc [X:K, T] ne peut être une sous-signature de [X:K′, T′] que si $\mathsf{T} \equiv \mathsf{T}'$ (en plus de $\mathsf{K} <: \mathsf{K}'$). Les types doivent être équivalents quelle que soit l'hypothèse faite sur X (sorte K ou K′) ; comme K est plus précise que K′, il suffit de vérifier $\mathsf{T} \equiv \mathsf{T}'$ sous l'hypothèse plus forte X:K.

$$\frac{\Gamma \vdash_c \mathsf{K} <: \mathsf{K}' \qquad \Gamma, \mathsf{X}{:}\mathsf{K} \vdash_c \mathsf{T} \equiv \mathsf{T}'}{\Gamma \vdash_c [\mathsf{X}{:}\mathsf{K},\ \mathsf{T}] <: [\mathsf{X}{:}\mathsf{K}',\ \mathsf{T}']}\ (\mathcal{H}/\textsf{\textbf{Ssub.struct}})$$

III.2.3.11 $\boxed{\Gamma \vdash_c \mathsf{E} : \mathsf{T}}$ Type d'une expression

Chaque production de la grammaire des expressions donne lieu à une règle de correction correspondante.

$$\frac{\Gamma, x{:}\mathsf{T}, \Gamma' \vdash_c \mathsf{ok}}{\Gamma, x{:}\mathsf{T}, \Gamma' \vdash_c x : \mathsf{T}}\ (\mathcal{H}/\textsf{\textbf{ET.var}}) \qquad \frac{\Gamma \vdash_c \mathsf{ok}}{\Gamma \vdash_c () : \textsc{unit}}\ (\mathcal{H}/\textsf{\textbf{ET.unit}})$$

$$\frac{\Gamma \vdash_c \mathsf{E}_1 : \mathsf{T}_1 \qquad \Gamma \vdash_c \mathsf{E}_2 : \mathsf{T}_2}{\Gamma \vdash_c (\mathsf{E}_1, \mathsf{E}_2) : \mathsf{T}_1 * \mathsf{T}_2}\ (\mathcal{H}/\textsf{\textbf{ET.pair}}) \qquad \frac{\Gamma \vdash_c \mathsf{E} : \mathsf{T}_1 * \mathsf{T}_2 \qquad \text{si } i \in \{1,2\}}{\Gamma \vdash_c \pi_i\, \mathsf{E} : \mathsf{T}_i}\ (\mathcal{H}/\textsf{\textbf{ET.proj}})$$

$$\frac{\Gamma, x{:}\mathsf{T} \vdash_c \mathsf{E} : \mathsf{T}'}{\Gamma \vdash_c \lambda x : \mathsf{T}.\ \mathsf{E} : \mathsf{T} \to \mathsf{T}'}\ (\mathcal{H}/\textsf{\textbf{ET.fun}}) \qquad \frac{\Gamma \vdash_c \mathsf{E}' : \mathsf{T} \to \mathsf{T}' \qquad \Gamma \vdash_c \mathsf{E} : \mathsf{T}}{\Gamma \vdash_c \mathsf{E}'\,\mathsf{E} : \mathsf{T}'}\ (\mathcal{H}/\textsf{\textbf{ET.ap}})$$

Pour extraire un champ terme d'un module, il faut disposer d'une signature non dépendante (c'est à cela que sert la deuxième prémisse de la règle ($\mathcal{H}$/ET.mod)). Cela nécessite la plupart du temps l'usage des règles ($\mathcal{H}$/US.sub) et ($\mathcal{H}$/US.self).

$$\frac{\Gamma \vdash_c \mathsf{U} : [\mathsf{X}{:}\mathsf{K},\ \mathsf{T}] \qquad \Gamma \vdash_c \mathsf{T} : \textsc{Type}}{\Gamma \vdash_c \mathsf{U}.\mathsf{term} : \mathsf{T}}\ (\mathcal{H}/\textsf{\textbf{ET.mod}})$$

Un crochet coloré indique un changement de couleur : l'annotation sur le crochet est une couleur qui remplace la couleur extérieure. L'expression intérieure doit avoir un type T (dans la couleur intérieure c'), et le tout a également le type T, mais dans la couleur T′. L'hypothèse de correction de T est en fait superflue car que la correction d'un type est indépendante de la couleur : la prémisse $\vdash \mathsf{c}\,\mathsf{ok}$ suffirait, mais la forme que nous donnons ici est quelquefois plus maniable.

$$\frac{\Gamma \vdash_{c} \mathsf{T} : \textsc{Type} \qquad \Gamma \vdash_{c'} \mathsf{E} : \mathsf{T}}{\Gamma \vdash_{c} [\mathsf{E}]^{\mathsf{T}}_{c'} : \mathsf{T}} \; (\mathcal{H}/\textbf{ET.col})$$

La fabrication d'un dynamique se fait par la règle ($\mathcal{H}$/ET.dyn). La règle ($\mathcal{H}$/ET.dynned) permet la formation d'une variante qui requiert que l'expression soit typable dans la couleur vide ; l'utilité de cette variante sera démontrée à la section III.2.5. Dans la règle ($\mathcal{H}$/ET.undyn), une prémisse doit préciser que le type T est correct, puisque ce n'est pas forcément le type de E.

$$\frac{\Gamma \vdash_{c} \mathsf{E} : \mathsf{T}}{\Gamma \vdash_{c} \mathsf{dyn}\,\mathsf{E}\,\mathsf{at}\,\mathsf{T} : \textsc{dyn}} \; (\mathcal{H}/\textbf{ET.dyn}) \qquad \frac{\Gamma \vdash_{c} \mathsf{ok} \qquad \mathsf{nil} \vdash_{\bullet} \mathsf{E} : \mathsf{T}}{\Gamma \vdash_{c} \mathsf{dynned}\,\mathsf{E}\,\mathsf{at}\,\mathsf{T} : \textsc{dyn}} \; (\mathcal{H}/\textbf{ET.dynned})$$

$$\frac{\Gamma \vdash_{c} \mathsf{T} : \textsc{Type} \qquad \Gamma \vdash_{c} \mathsf{E} : \textsc{dyn}}{\Gamma \vdash_{c} \mathsf{undyn}\,\mathsf{E}\,\mathsf{at}\,\mathsf{T} : \mathsf{T}} \; (\mathcal{H}/\textbf{ET.undyn}) \qquad \frac{\Gamma \vdash_{c} \mathsf{T} : \textsc{Type}}{\Gamma \vdash_{c} \mathsf{Fail}_{\mathsf{T}} : \mathsf{T}} \; (\mathcal{H}/\textbf{ET.Undynfailure})$$

Un unique canal transporte des valeurs de type DYN.

$$\frac{\Gamma \vdash_{c} \mathsf{E} : \textsc{dyn}}{\Gamma \vdash_{c} \,!\mathsf{E} : \textsc{unit}} \; (\mathcal{H}/\textbf{ET.send}) \qquad \frac{\Gamma \vdash_{c} \mathsf{ok}}{\Gamma \vdash_{c} \,? : \textsc{dyn}} \; (\mathcal{H}/\textbf{ET.recv})$$

Une règle non structurelle permet la réécriture du type d'une expression en un type équivalent.

$$\frac{\Gamma \vdash_{c} \mathsf{E} : \mathsf{T} \qquad \Gamma \vdash_{c} \mathsf{T} \equiv \mathsf{T}'}{\Gamma \vdash_{c} \mathsf{E} : \mathsf{T}'} \; (\mathcal{H}/\textbf{ET.eq})$$

III.2.3.12 $\boxed{\Gamma \vdash_{c} \mathsf{M} : \mathsf{S}}$ Signature d'une structure

Pour qu'une structure [T, V] soit correcte, il faut que ses composantes le soient. Le type T' attribué à V dans la signature n'est pas forcément le type T'' donné *a priori* à V : T' peut contenir des occurences de X là où T'' contient des sous-termes équivalents à T (la variable X n'est pas accessible dans la structure). Les prémisses de ($\mathcal{H}$/MS.struct) expriment respectivement la correction du champ type de la structure, la correction de la signature, l'équivalence entre le type interne et le type externe du champ expression, et la correction du champ expression de la structure.

$$\frac{\Gamma \vdash_{c} \mathsf{T} : \mathsf{K} \qquad \Gamma, \mathsf{X}{:}\mathsf{K} \vdash_{c} \mathsf{T}' : \textsc{Type} \qquad \Gamma, \mathsf{X}{:}\textsc{Eq}(\mathsf{T}) \vdash_{c} \mathsf{T}'' \equiv \mathsf{T}' \qquad \Gamma \vdash_{c} \mathsf{V} : \mathsf{T}''}{\Gamma \vdash_{c} [\mathsf{T},\ \mathsf{V}] : [\mathsf{X}{:}\mathsf{K},\ \mathsf{T}']} \; (\mathcal{H}/\textbf{MS.struct})$$

Nous exigeons que le champ expression du module soit une valeur dans la couleur c. En pratique, si une expression arbitraire E est désirée, celle-ci doit être protégée par une lambda-abstraction (soit $\lambda x : \textsc{unit}.\ \mathsf{E}$), et les utilisations du module doivent référencer U.term () au lieu de U.term.

Nous ne fournissons pas de règle de sous-signaturage pour les structures. En fait, une telle règle peut être vue comme contenue dans ($\mathcal{H}$/MS.struct), qui veille à autoriser une sorte K pas forcément singleton et un type extérieur T' à équivalence près.

III.2.3.13 $\boxed{\Gamma \vdash_{c} \mathsf{U} : \mathsf{S}}$ Signature d'un nom de module

Un nom de module peut être extrait de l'environnement, et est sujet à un sous-signaturage.

$$\frac{\Gamma, \mathsf{U}{:}\mathsf{S}, \Gamma' \vdash_{c} \mathsf{ok}}{\Gamma, \mathsf{U}{:}\mathsf{S}, \Gamma' \vdash_{c} \mathsf{U} : \mathsf{S}} \; (\mathcal{H}/\textbf{US.var}) \qquad \frac{\Gamma \vdash_{c} \mathsf{U} : \mathsf{S} \qquad \Gamma \vdash_{c} \mathsf{S} <: \mathsf{S}'}{\Gamma \vdash_{c} \mathsf{U} : \mathsf{S}'} \; (\mathcal{H}/\textbf{US.sub})$$

La signature d'un nom de module est le plus souvent dépendante. Si c'est le cas et que le module définit un type abstrait, soit une signature [X:TYPE, T] où X apparaît dans T, il n'y a pas de signature équivalente qui ne soit pas dépendante, ce qui au vu de la règle ($\mathcal{H}$/ET.mod) interdit d'utiliser le champ terme du module. La règle ($\mathcal{H}$/US.self) résoud ce problème : c'est une règle d'**auto-typage** (voir la section I.2.2.2). Elle affirme que le champ type du module U est U.type, rendant ainsi la signature non abstraite. Une fois sous cette forme, la règle ($\mathcal{H}$/US.sub) peut retirer la dépendance.

$$\frac{\Gamma \vdash_c \mathsf{U} : [\mathsf{X}{:}\mathsf{K},\ \mathsf{T}]}{\Gamma \vdash_c \mathsf{U} : [\mathsf{X}{:}\mathrm{E{\scriptsize Q}}(\mathsf{U.type}),\ \mathsf{T}]}\ (\mathcal{H}/\textbf{US.self})$$

III.2.3.14 $\boxed{\Gamma \vdash_c \mathsf{L} : \mathsf{T}}$ Correction d'une machine

Les machines contiennent des définitions de modules autour d'une expression. Dans la règle ($\mathcal{H}$/mT.expr), la prémisse juge le type de l'expression E tandis que la conclusion concerne E en tant que machine. Dans la règle ($\mathcal{H}$/mT.let), la première prémisse interdit à la variable liée U d'apparaître dans le type de la machine.

$$\frac{\Gamma \vdash_\bullet \mathsf{E} : \mathsf{T}}{\Gamma \vdash_\bullet \mathsf{E} : \mathsf{T}}\ (\mathcal{H}/\textbf{mT.expr}) \qquad \frac{\Gamma \vdash_\bullet \mathsf{T} : \mathrm{T{\scriptsize YPE}} \qquad \Gamma \vdash_\bullet \mathsf{M} : \mathsf{S} \qquad \Gamma, \mathsf{U}{:}\mathsf{S} \vdash_\bullet \mathsf{L} : \mathsf{T}}{\Gamma \vdash_\bullet (\mathsf{let}\ \mathsf{U} = \mathsf{M} : \mathsf{S}\ \mathsf{in}\ \mathsf{L}) : \mathsf{T}}\ (\mathcal{H}/\textbf{mT.let})$$

III.2.3.15 $\boxed{\vdash \mathsf{N}\ \mathsf{ok}}$ Correction d'un réseau

Un réseau est une simple juxtaposition de machines. Nous supposons le code sur chaque machine compilé en une expression (voir la section III.2.4).

$$\frac{}{\vdash 0\ \mathsf{ok}}\ (\mathcal{H}/\textbf{nok.zero}) \qquad \frac{\vdash \mathsf{N}_1\ \mathsf{ok} \qquad \vdash \mathsf{N}_2\ \mathsf{ok}}{\vdash \mathsf{N}_1 \parallel \mathsf{N}_2\ \mathsf{ok}}\ (\mathcal{H}/\textbf{nok.par}) \qquad \frac{\mathsf{nil} \vdash_\bullet \mathsf{E} : \mathrm{UNIT}}{\vdash \mathsf{E}\ \mathsf{ok}}\ (\mathcal{H}/\textbf{nok.expr})$$

III.2.4 Compilation

La compilation transcrit les définitions de modules en du code contenant des annotations de provenance. Ces annotations prennent la forme de crochets colorés entourant le code du module. Les références au module sont remplacées par le contenu du module formaté convenablement : U.type est remplacé par une représentation externe du champ type, et U.term par une représentation externe du champ valeur.

III.2.4.1 $\boxed{\mathsf{L} \Longrightarrow \mathsf{L}'}$ Compilation sur une machine

Nous définissons la compilation par une réduction à petits pas sur les machines, notée $\mathsf{L} \Longrightarrow \mathsf{L}'$. Le résultat final de la compilation est une expression.

Si la signature du module est concrète, les représentations externes sont simplement respectivement le type mentionné par la sorte singleton et la valeur contenue dans le module. (Nous pourrions utiliser le champ type du module puisque celui-ci est équivalent.)

$$\mathsf{let}\ \mathsf{U} = [\mathsf{T},\ \mathsf{V}] : [\mathsf{X}{:}\mathrm{E{\scriptsize Q}}(\mathsf{T}_0),\ \mathsf{T}']\ \mathsf{in}\ \mathsf{L} \Longrightarrow \{\mathsf{U.type}{\leftarrow}\mathsf{T}_0, \mathsf{U.term}{\leftarrow}\mathsf{V}\}\mathsf{L} \qquad (\mathcal{H}/\mathsf{mred.Eq})$$

Si le module définit un type abstrait, il y a plus de travail. Le seul moyen de faire référence au type abstrait depuis l'extérieur est d'utiliser U.type, or le nom U va disparaître. C'est à cela que sert l'empreinte que nous fabriquons ici : elle constitue un nom valable universellement pour ce type, y compris sur d'autres machines. L'empreinte h remplace donc U.type ; la valeur V est entourée par un crochet coloré par h sur lequel l'annotation de type est le type T′ dans lequel la variable X est remplacée par le nouveau nom donné au type abstrait, à savoir h.

$$\mathsf{let}\ \mathsf{U} = [\mathsf{T},\ \mathsf{V}] : [\mathsf{X}{:}\mathrm{T{\scriptsize YPE}},\ \mathsf{T}']\ \mathsf{in}\ \mathsf{L} \Longrightarrow \{\mathsf{U.type}{\leftarrow}\mathsf{h}, \mathsf{U.term}{\leftarrow}[\mathsf{V}]_{\mathsf{h}}^{\{\mathsf{X}\leftarrow\mathsf{h}\}\mathsf{T}'}\}\mathsf{L} \qquad (\mathcal{H}/\mathsf{mred.Type})$$
$$\text{où } \mathsf{h} = \mathsf{hash}([\mathsf{T},\ \mathsf{V}] : [\mathsf{X}{:}\mathrm{T{\scriptsize YPE}},\ \mathsf{T}'])$$

III.2.5 Exécution

Nous définissons le comportement dynamique d'une expression et celui d'un réseau par des règles de réduction à petits pas. La réduction des expressions est annotée par la couleur ambiante.

III.2.5.1 $\boxed{\mathsf{E} \longrightarrow_{\mathsf{c}} \mathsf{E}'}$ Réduction des expressions

Les valeurs ont été définies à la section III.2.2.1 (voir notamment la figure III.2).

Contextes Comme les réductions et les valeurs, les contextes d'évaluation sont paramétrés par la couleur ambiante ; ils sont également paramétrés par la couleur intérieure. Les couleurs paramètres influent sur les couleurs des valeurs. Nous notons $\mathsf{C}^{c_2}_{c_1}$ un contexte d'évaluation de profondeur 1 de couleur intérieure c_1 et de couleur extérieure c_2, et $\mathsf{CC}^{c_2}_{c_1}$ un contexte de profondeur quelconque.

$\mathsf{C}^{c_2}_{c_1}$ ::=	**contexte d'intérieur c_1 et d'extérieur c_2**
$(\_\_, \mathsf{E}_2)$	première composante d'une paire, si $c_1 = c_2$
$(\mathsf{V}^{c_1}_1, \_\_)$	seconde composante d'une paire, si $c_1 = c_2$
$\pi_i \_\_$	projection ($i \in \{1, 2\}$), si $c_1 = c_2$
$\mathsf{E}_1 \_\_$	argument de fonction, si $c_1 = c_2$
$\_\_ \mathsf{V}^{c_1}_2$	fonction appliquée, si $c_1 = c_2$
$[\_\_]^{\mathsf{T}}_{c_1}$	crochet coloré
$\mathsf{dyn}\,\_\_\,\mathsf{at}\,\mathsf{T}$	dynamique, si $c_1 = c_2$
$\mathsf{dynned}\,\_\_\,\mathsf{at}\,\mathsf{T}$	dynamique universel, si $c_1 = \bullet$
$\mathsf{undyn}\,\_\_\,\mathsf{at}\,\mathsf{T}$	vérification de typage dynamique, si $c_1 = c_2$
$!\_\_$	envoi de message, si $c_1 = c_2$
$\mathsf{CC}^{c_2}_{c_1}$ ::=	**contexte de profondeur quelconque**
$\_\_$	identité, si $c_1 = c_2$
$\mathsf{CC}^{c_2}_{c'} \cdot \mathsf{C}^{c'}_{c_1}$	niveau supplémentaire

La réduction dans un contexte peut entraîner un changement de couleur (si le contexte est $[\_\_]^{\mathsf{T}}_{c'}$ ou $\mathsf{dynned}\,\_\_\,\mathsf{at}\,\mathsf{T}$).

$$\frac{\mathsf{E} \longrightarrow_{c'} \mathsf{E}'}{\mathsf{C}^{c}_{c'} \cdot \mathsf{E} \longrightarrow_{c} \mathsf{C}^{c}_{c'} \cdot \mathsf{E}'} \;(\mathcal{H}/\mathsf{ered.cong})$$

Réductions calculatoires Lors de la bêta-réduction, la ($\mathcal{H}$/ered.ap) force un crochet coloré autour de l'argument. Ceci est dû au fait que les occurences de x dans E peuvent survenir n'importe où, pas forcément dans la couleur c, or l'argument n'a à coup sûr le type T que dans c. Nous suivons sur ce point S. Zdancewic, D. Grossman et G. Morrisett [GMZ00].

$$(\lambda x : \mathsf{T}.\ \mathsf{E})\, \mathsf{V}^{c} \longrightarrow_{c} \{x \leftarrow [\mathsf{V}^{c}]^{\mathsf{T}}_{c}\}\mathsf{E} \qquad (\mathcal{H}/\mathsf{ered.ap})$$

$$\pi_i\, (\mathsf{V}^{c}_1, \mathsf{V}^{c}_2) \longrightarrow_{c} \mathsf{V}_i \qquad (\mathcal{H}/\mathsf{ered.proj})$$

$$\text{où } i \in \{1, 2\}$$

Lors de la fabrication d'un dynamique, la valeur a le type indiqué dans la couleur c, mais pas forcément dans une autre couleur. Aussi la règle ($\mathcal{H}$/ered.dyn) introduit-elle un nouveau constructeur : $\mathsf{dynned}\,\mathsf{E}\,\mathsf{at}\,\mathsf{T}$ a le même sens que $\mathsf{dyn}\,\mathsf{E}\,\mathsf{at}\,\mathsf{T}$, mais force E à avoir le type dans n'importe quelle couleur[3] et non seulement dans la couleur ambiante. La méthode utilisée pour forcer la valeur dynamisée à être valide partout est de la protéger par un crochet qui porte toute l'information de couleur nécessaire. Notons que $[\mathsf{V}^{c}]^{\mathsf{T}}_{c}$ n'est pas forcément une valeur, il peut rester des crochets à pousser.

$$\mathsf{dyn}\,\mathsf{V}^{c}\,\mathsf{at}\,\mathsf{T} \longrightarrow_{c} \mathsf{dynned}\,[\mathsf{V}^{c}]^{\mathsf{T}}_{c}\,\mathsf{at}\,\mathsf{T} \qquad (\mathcal{H}/\mathsf{ered.dyn})$$

[3] Comme les couleurs ne font qu'ajouter des equivalences entre types, si E a le type T dans la couleur vide, elle a ce même type dans n'importe quel couleur.

La destruction d'un dynamique est toujours possible, mais elle a deux résultats différents suivant si le dynamique a ou non le type attendu. Si le type est le bon, la valeur est renvoyée. Sinon, l'exception Fail_T est levée. Dans ce document, nous ne décrivons pas de mécanisme de propagation des exceptions : un programme qui a levé une exception est bloqué. Notons que c'est l'égalité syntaxique qui est employée ici (dans HAT, deux types clos ne sont équivalents quelle que soit la couleur que s'ils sont égaux) ; une conséquence pratique est qu'il suffit dans une implémentation d'annoter le dynamique par une empreinte cryptographique (voir la section II.3.3).

$$\mathsf{undyn}\,(\mathsf{dynned}\,\mathsf{V}^\bullet\;\mathsf{at}\,\mathsf{T})\;\mathsf{at}\,\mathsf{T}' \longrightarrow_\mathsf{c} \begin{cases} \mathsf{V}^\bullet & \text{si } \mathsf{T} = \mathsf{T}' \\ \mathsf{Fail}_\mathsf{T} & \text{sinon} \end{cases} \qquad (\mathcal{H}/\mathsf{ered.undyn})$$

Poussée de crochets Nous poussons les crochets colorés vers l'intérieur, comme annoncé à la section III.1.2.2. Le choix de la règle est basé sur le constructeur de tête de l'annotation de type. Dans le cas de $[\mathsf{dynned}\,\mathsf{V}^\bullet\;\mathsf{at}\,\mathsf{T}]_{\mathsf{c}'}^{\mathrm{DYN}}$, aucun crochet n'est nécessaire puisque la valeur est valable sous n'importe quelle couleur.

$$\begin{aligned}
[()]_{\mathsf{c}'}^{\mathrm{UNIT}} &\longrightarrow_\mathsf{c} () && (\mathcal{H}/\mathsf{ered.col.unit})\\
[(\mathsf{V}_1^{\mathsf{c}'},\mathsf{V}_2^{\mathsf{c}'})]_{\mathsf{c}'}^{\mathsf{T}_1*\mathsf{T}_2} &\longrightarrow_\mathsf{c} ([\mathsf{V}_1^{\mathsf{c}'}]_{\mathsf{c}'}^{\mathsf{T}_1},[\mathsf{V}_2^{\mathsf{c}'}]_{\mathsf{c}'}^{\mathsf{T}_2}) && (\mathcal{H}/\mathsf{ered.col.pair})\\
[\lambda \mathsf{x}' : \mathsf{T}_2.\ \mathsf{E}]_{\mathsf{c}'}^{\mathsf{T}_0\to\mathsf{T}_1} &\longrightarrow_\mathsf{c} \lambda \mathsf{x} : \mathsf{T}_0.\ [\{\mathsf{x}'\leftarrow[\mathsf{x}]_{\mathsf{c}'}^{\mathsf{T}_2}\}\mathsf{E}]_{\mathsf{c}'}^{\mathsf{T}_1} && (\mathcal{H}/\mathsf{ered.col.fun})\\
[\mathsf{dynned}\,\mathsf{V}^\bullet\;\mathsf{at}\,\mathsf{T}]_{\mathsf{c}'}^{\mathrm{DYN}} &\longrightarrow_\mathsf{c} \mathsf{dynned}\,\mathsf{V}^\bullet\;\mathsf{at}\,\mathsf{T} && (\mathcal{H}/\mathsf{ered.col.dynned})
\end{aligned}$$

Si deux crochets consécutifs portent la même annotation de type, la duplication est inutile ; comme la valeur $\mathsf{V}^{\mathsf{h}_0}$ doit *a priori* rester protégée par un crochet annoté par h_0, la règle $(\mathcal{H}/\mathsf{ered.col.col})$ supprime le crochet extérieur. Les conditions de bord de $(\mathcal{H}/\mathsf{ered.col.col})$ sont inutiles pour la correction du système, mais assurent l'absence de paire critique triviale avec $(\mathcal{H}/\mathsf{ered.col.le})$.

$$[[\mathsf{V}^{\mathsf{h}_0}]_{\mathsf{h}_0}^{\mathsf{h}_0}]_{\mathsf{h}_1}^{\mathsf{h}_0} \longrightarrow_\mathsf{c} [\mathsf{V}^{\mathsf{h}_0}]_{\mathsf{h}_0}^{\mathsf{h}_0} \qquad (\mathcal{H}/\mathsf{ered.col.col})$$
$$\text{si } \mathsf{h}_0 \neq \mathsf{h}_1 \text{ et } \mathsf{h}_1 \neq \mathsf{c}$$

Un autre cas d'inutilité de crochet est celui où la couleur intérieure est contenue dans la couleur extérieure : le crochet n'apporte alors pas d'équation de typage supplémentaire. Nous nous limitons au cas où le type indiqué sur le crochet est une empreinte : sinon, une autre règle de poussée s'applique.

$$[\mathsf{V}^{\mathsf{c}'}]_{\mathsf{c}'}^{\mathsf{h}''} \longrightarrow_\mathsf{c} \mathsf{V}^{\mathsf{c}'} \qquad (\mathcal{H}/\mathsf{ered.col.le})$$
$$\text{si } \mathsf{c}' \subseteq \mathsf{c}$$

III.2.5.2 $\boxed{\mathsf{N} \longrightarrow \mathsf{N}'\ ;\ \mathsf{N} \equiv \mathsf{N}'}$ Réduction des réseaux

Nous définissons sur les réseaux une relation de réduction $\mathsf{N} \longrightarrow \mathsf{N}'$ et une équivalence $\mathsf{N} \equiv \mathsf{N}'$.

L'équivalence est la congruence structurelle ; elle a pour but de considérer les réseaux comme de simples ensembles fini de machines.

$$\frac{}{\mathsf{N}_1 \equiv \mathsf{N}_2}\ (\mathcal{H}/\textbf{nsc.refl}) \qquad \frac{\mathsf{N}_1 \equiv \mathsf{N}_2}{\mathsf{N}_2 \equiv \mathsf{N}_1}\ (\mathcal{H}/\textbf{nsc.sym}) \qquad \frac{\mathsf{N}_1 \equiv \mathsf{N}_2 \equiv \mathsf{N}_3}{\mathsf{N}_1 \equiv \mathsf{N}_3}\ (\mathcal{H}/\textbf{nsc.trans})$$

$$\frac{}{0 \parallel \mathsf{N} \equiv \mathsf{N}}\ (\mathcal{H}/\textbf{nsc.id}) \qquad \frac{}{\mathsf{N}_1 \parallel \mathsf{N}_2 \equiv \mathsf{N}_2 \parallel \mathsf{N}_1}\ (\mathcal{H}/\textbf{nsc.commut})$$

$$\frac{}{\mathsf{N}_1 \parallel (\mathsf{N}_2 \parallel \mathsf{N}_3) \equiv (\mathsf{N}_1 \parallel \mathsf{N}_2) \parallel \mathsf{N}_3}\ (\mathcal{H}/\textbf{nsc.assoc})$$

La réduction des réseaux contient la réduction des expressions sur chaque machine. Elle est définie à la congruence structurelle près.

$$\frac{\mathsf{E} \longrightarrow_\bullet \mathsf{E}'}{\mathsf{N} \longrightarrow \mathsf{N}'}\ (\mathcal{H}/\textbf{nred.expr}) \qquad \frac{\mathsf{N} \longrightarrow \mathsf{N}'}{\mathsf{N} \parallel \mathsf{N}'' \longrightarrow \mathsf{N}' \parallel \mathsf{N}''}\ (\mathcal{H}/\textbf{nred.par}) \qquad \frac{\mathsf{N} \equiv \mathsf{N}_0 \longrightarrow \mathsf{N}_0' \equiv \mathsf{N}'}{\mathsf{N} \longrightarrow \mathsf{N}'}\ (\mathcal{H}/\textbf{nred.sc})$$

Nous supposons disposer d'un canal de communication synchrone partagé par toutes les machines (nous ne distinguons pas de noms de canaux pour simplifier la présentation). Le canal transporte des dynamiques. On note que la valeur peut changer de couleur ambiante ; c'est pour cela que nous nous assurons que les valeurs dynamiques $\mathsf{dynned}\ \mathsf{V}^\bullet\ \mathsf{at}\ \mathsf{T}$ sont valables dans toute couleur.

$$\frac{}{\mathsf{CC}^\bullet_{c_1} \cdot !\mathsf{V}^{c_1} \parallel \mathsf{CC}^\bullet_{c_2} \cdot ? \longrightarrow \mathsf{CC}^\bullet_{c_1} \cdot () \parallel \mathsf{CC}^\bullet_{c_2} \cdot \mathsf{V}^{c_1}}\ (\mathcal{H}/\textbf{nred.comm})$$

III.2.6 Un exemple

Considérons un programme distribué sur deux machines. Chaque machine définit un même module qui fournit un type abstrait. La machine A fabrique une valeur de ce type abstrait et l'envoie sur le réseau ; la machine B lit et utilise cette valeur.

Nous utilisons dans cet exemple des valeurs entières ; nous supposons le système étendu par des règles ($\mathcal{H}$/TK.Int), ($\mathcal{H}$/ET.int) et ($\mathcal{H}$/ered.col.int) sur le modèle des règles correspondantes pour le type UNIT.

machine A :

$$\begin{array}{l} \mathsf{let}\ \mathsf{U} = [\textsc{int},\quad ((\lambda x:\textsc{int}.\ x), (\lambda x:\textsc{int}.\ x))] \\ \qquad : [\mathsf{X}{:}\textsc{Type}, (\textsc{int} \to \mathsf{X}) * (\mathsf{X} \to \textsc{int})] \\ \mathsf{in}\ !(\mathsf{dyn}\,(\pi_1\,\mathsf{U.term})\,3\ \mathsf{at}\ \mathsf{U.type}) \end{array}$$

machine B :

$$\begin{array}{l} \mathsf{let}\ \mathsf{U} = [\textsc{int},\quad ((\lambda x:\textsc{int}.\ x), (\lambda x:\textsc{int}.\ x))] \\ \qquad : [\mathsf{X}{:}\textsc{Type}, (\textsc{int} \to \mathsf{X}) * (\mathsf{X} \to \textsc{int})] \\ \mathsf{in}\ \mathtt{print\_int}\,((\pi_2\,\mathsf{U.term})\,(\mathsf{undyn}\,?\ \mathsf{at}\ \mathsf{U.type})) \end{array}$$

Commençons par regarder le typage du module. Nous devons appliquer la règle ($\mathcal{H}$/MS.struct) en choisissant un type interne pour le champ terme ; nous choisissons le type évident $(\textsc{int} \to \textsc{int}) * (\textsc{int} \to \textsc{int})$.

$$\frac{\begin{array}{c} \mathsf{nil} \vdash_\bullet \textsc{int} : \textsc{Type} \\ \mathsf{X}{:}\textsc{Type} \vdash_\bullet (\textsc{int} \to \mathsf{X}) * (\mathsf{X} \to \textsc{int}) : \textsc{Type} \\ \mathsf{X}{:}\textsc{Eq}(\textsc{int}) \vdash_\bullet (\textsc{int} \to \textsc{int}) * (\textsc{int} \to \textsc{int}) \equiv (\textsc{int} \to \mathsf{X}) * (\mathsf{X} \to \textsc{int}) \\ \mathsf{nil} \vdash_\bullet ((\lambda x:\textsc{int}.\ x), (\lambda x:\textsc{int}.\ x)) : (\textsc{int} \to \textsc{int}) * (\textsc{int} \to \textsc{int}) \end{array}}{\mathsf{nil} \vdash_\bullet [\textsc{int},\ ((\lambda x:\textsc{int}.\ x), (\lambda x:\textsc{int}.\ x))] : [\mathsf{X}{:}\textsc{Type},\ (\textsc{int} \to \mathsf{X}) * (\mathsf{X} \to \textsc{int})]}\ (\mathcal{H}/\mathsf{MS.struct})$$

Montrons comment dériver la troisième prémisse (les autres ne présentent pas d'intérêt particulier). Après application de règles de congruence, le principal problème est de prouver $\mathsf{X}{:}\textsc{Eq}(\textsc{int}) \vdash_\bullet \textsc{int} \equiv \mathsf{X}$. Nous passons par le jugement $\mathsf{X}{:}\textsc{Eq}(\textsc{int})$ qui provient de l'environnement.

$$\dfrac{\dfrac{\begin{array}{c}\vdots \\ \mathsf{X}{:}\textsc{Eq}(\textsc{int}) \vdash_\bullet \textsc{int} \equiv \textsc{int}\end{array} \qquad \dfrac{\dfrac{\dfrac{\begin{array}{c}\vdots \\ \mathsf{X}{:}\textsc{Eq}(\textsc{int}) \vdash_\bullet \mathsf{ok}\end{array}}{\mathsf{X}{:}\textsc{Eq}(\textsc{int}) \vdash_\bullet \mathsf{X} : \textsc{Eq}(\textsc{int})}\ (\mathcal{H}/\mathsf{TK.var})}{\mathsf{X}{:}\textsc{Eq}(\textsc{int}) \vdash_\bullet \mathsf{X} \equiv \textsc{int}}\ (\mathcal{H}/\mathsf{Teq.Eq})}{\mathsf{X}{:}\textsc{Eq}(\textsc{int}) \vdash_\bullet \textsc{int} \equiv \mathsf{X}}\ (\mathcal{H}/\mathsf{Teq.sym})}{\mathsf{X}{:}\textsc{Eq}(\textsc{int}) \vdash_\bullet \textsc{int} \to \textsc{int} \equiv \textsc{int} \to \mathsf{X}}\ (\mathcal{H}/\mathsf{Teq.cong.fun}) \qquad \vdots}{\mathsf{X}{:}\textsc{Eq}(\textsc{int}) \vdash_\bullet (\textsc{int} \to \textsc{int}) * (\textsc{int} \to \textsc{int}) \equiv (\textsc{int} \to \mathsf{X}) * (\mathsf{X} \to \textsc{int})}\ (\mathcal{H}/\mathsf{Teq.cong.pair})$$

Le typage du programme principal s'effectue dans l'environnement $\Gamma = \mathsf{U}{:}[\mathsf{X}{:}\textsc{Type}, (\textsc{int} \to \mathsf{X}) * (\mathsf{X} \to \textsc{int})]$. Le principal problème est de typer $\mathsf{U.term}$. Comme nous l'avons remarqué à l'occasion de la présentation de la règle ($\mathcal{H}$/US.self), le typage de $\mathsf{U}.\mathsf{term}$ s'effectue par la règle ($\mathcal{H}$/ET.mod), qui nécessite une signature non dépendante obtenue par ($\mathcal{H}$/US.sub) précédée de ($\mathcal{H}$/US.self).

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\Gamma \vdash_\bullet \mathsf{ok}}}{\Gamma \vdash_\bullet \mathsf{U} : [X{:}\textsc{Type},\ (\textsc{int}{\to}X) * (X{\to}\textsc{int})]}\,(\mathcal{H}/\mathsf{US.var})}{\Gamma \vdash_\bullet \mathsf{U} : [X{:}\textsc{Eq}(\mathsf{U.type}),\ (\textsc{int}{\to}X) * (X{\to}\textsc{int})]}\,(\mathcal{H}/\mathsf{US.self}) \quad \cdots}{\Gamma \vdash_\bullet \mathsf{U} : [X{:}\textsc{Eq}(\mathsf{U.type}),\ (\textsc{int}{\to}\mathsf{U.type}) * (\mathsf{U.type}{\to}\textsc{int})]}\,(\mathcal{H}/\mathsf{US.sub}) \quad \cdots}{\Gamma \vdash_\bullet \mathsf{U.term} : (\textsc{int}{\to}\mathsf{U.type}) * (\mathsf{U.type}{\to}\textsc{int})}\,(\mathcal{H}/\mathsf{ET.mod})$$

La compilation du programme séparément sur chaque machine introduit une même empreinte $h = \mathsf{hash}([\textsc{Int},\ ((\lambda x : \textsc{int}.\ x), (\lambda x : \textsc{int}.\ x))] : [X{:}\textsc{Type},\ (\textsc{int} \to X) * (X \to \textsc{int})])$. Chaque machine nécessite une application de la règle ($\mathcal{H}$/mred.Type) ; les programmes compilés sont les suivants (nous posons $V = ((\lambda x : \textsc{int}.\ x), (\lambda x : \textsc{int}.\ x))$) :

machine A :
$E_A = !(\mathsf{dyn}\,(\pi_1\,[V]_h^{(\textsc{int}\to h)*(h\to\textsc{int})})\ 3\ \mathsf{at}\ h)$

machine B :
$E_B = \texttt{print\_int}\,((\pi_2\,[V]_h^{(\textsc{int}\to h)*(h\to\textsc{int})})\ (\mathsf{undyn}\ ?\ \mathsf{at}\ h))$

Examinons l'exécution du programme sur la machine A. Les premières étapes consistent à pousser des crochets :

$$\begin{aligned}
[((\lambda x : \textsc{int}.\ x), (\lambda x : \textsc{int}.\ x))]_h^{(\textsc{int}\to h)*(h\to\textsc{int})} &\longrightarrow_\bullet\ ([\lambda x : \textsc{int}.\ x]_h^{\textsc{int}\to h}, [\lambda x : \textsc{int}.\ x]_h^{h\to\textsc{int}}) && (\mathcal{H}/\mathsf{ered.col.pair})\\
&\longrightarrow_\bullet^2\ (\lambda x : \textsc{int}.\ [[x]_h^{\textsc{int}}]_h^h, \lambda x : \textsc{int}.\ [[x]_h^h]_h^{\textsc{int}}) && (\mathcal{H}/\mathsf{ered.col.fun}) \times 2
\end{aligned}$$

Une fois ces étapes et la projection effectuées, nous pouvons passer à l'application de la fonction, suivie d'étapes techniques (poussée de crochets et ($\mathcal{H}$/ered.dyn)) :

$$\begin{aligned}
E_A &\longrightarrow_\bullet^*\ !(\mathsf{dyn}\,(\lambda x : \textsc{int}.\ [[x]_h^{\textsc{int}}]_h^h)\ 3\ \mathsf{at}\ h) &&\\
&\longrightarrow_\bullet\ !(\mathsf{dyn}\ [[[3]_\bullet^{\textsc{int}}]_h^{\textsc{int}}]_h^h\ \mathsf{at}\ h) && (\mathcal{H}/\mathsf{ered.ap})\\
&\longrightarrow_\bullet\ !(\mathsf{dyn}\ [[3]_h^{\textsc{int}}]_h^h\ \mathsf{at}\ h) && (\mathcal{H}/\mathsf{ered.col.int})\\
&\longrightarrow_\bullet\ !(\mathsf{dyn}\ [3]_h^h\ \mathsf{at}\ h) && (\mathcal{H}/\mathsf{ered.col.int})\\
&\longrightarrow_\bullet\ !(\mathsf{dynned}\ [[3]_h^h]_\bullet^h\ \mathsf{at}\ h) && (\mathcal{H}/\mathsf{ered.dyn})\\
&\longrightarrow_\bullet\ !(\mathsf{dynned}\ [3]_h^h\ \mathsf{at}\ h) && (\mathcal{H}/\mathsf{ered.col.le})
\end{aligned}$$

La machine A est désormais prête à communiquer.

Le programme B contient une application de fonction dans un contexte d'évaluation. Nous avons choisi arbitrairement en définissant les contextes d'évaluation de réduire la fonction avant l'argument ; après quelques poussées de crochets (identiques à celles ayant eu lieu sur A) et une projection, la machine B est prête à communiquer.

L'étape de communication est la suivante :

$$\begin{aligned}
&!(\mathsf{dynned}\ [3]_h^h\ \mathsf{at}\ h) \parallel \texttt{print_int}\,((\lambda x : h.\ [[x]_h^h]_h^{\textsc{int}})\ (\mathsf{undyn}\ ?\ \mathsf{at}\ h)) \longrightarrow\\
&\qquad () \parallel \texttt{print_int}\,((\lambda x : h.\ [[x]_h^h]_h^{\textsc{int}})\ (\mathsf{undyn}\ (\mathsf{dynned}\ [3]_h^h\ \mathsf{at}\ h)\ \mathsf{at}\ h)) \quad (\mathcal{H}/\mathsf{nred.comm})
\end{aligned}$$

Attachons-nous maintenant au devenir de l'argument de `print_int` sur B. La première étape est une vérification de type dynamique qui réussit ; ensuite l'expression est simplifiée en une valeur.

$$
\begin{array}{llr}
(\lambda x : h.\ [[x]^h_h]^{\mathrm{INT}}_h)\,(\mathsf{undyn}\,(\mathsf{dynned}\,[3]^h_h\ \mathsf{at}\ h)\ \mathsf{at}\ h) & \longrightarrow_\bullet (\lambda x : h.\ [[x]^h_h]^{\mathrm{INT}}_h)\,([3]^h_h) & (\mathcal{H}/\mathsf{ered.undyn}) \\
& \longrightarrow_\bullet [[[[3]^h_h]^h_\bullet]^h_h]^{\mathrm{INT}}_h & (\mathcal{H}/\mathsf{ered.ap}) \\
& \longrightarrow_\bullet [[[3]^h_h]^h_h]^{\mathrm{INT}}_h & (\mathcal{H}/\mathsf{ered.col.le}) \\
& \longrightarrow_\bullet [[3]^h_h]^{\mathrm{INT}}_h & (\mathcal{H}/\mathsf{ered.col.le}) \\
& \longrightarrow_\bullet [3]^{\mathrm{INT}}_h & (\mathcal{H}/\mathsf{ered.col.le}) \\
& \longrightarrow_\bullet 3 & (\mathcal{H}/\mathsf{ered.col.int})
\end{array}
$$

III.2.7 Résultats

Nous énonçons dans cette section les principaux résultats relatifs à HAT. Nous ne donnons pas ici les preuves ; nous renvoyons pour cela le lecteur au rapport technique [LPSW03b]. Les références des théorèmes correspondant aux énoncés donnés ici sont indiquées entre crochets.

III.2.7.1 Correction de l'exécution

Theorème III.2.1 (préservation du typage) [G.15, G.17] Si $E \longrightarrow_c E'$ et $\mathsf{nil} \vdash_c E : T$ alors $\mathsf{nil} \vdash_c E' : T$. Si $N \longrightarrow N'$ et $\vdash N\,\mathsf{ok}$ alors $\vdash N'\,\mathsf{ok}$.

Un lemme-clé est la préservation du typage par substitution. Celui-ci nécessite une précaution quant aux couleurs d'utilisation de la variable substituée : sous l'hypothèse $\Gamma_0 \vdash_c E : T$, on ne peut substituer E à x dans un jugement que si x est toujours utilisée dans une couleur qui inclut c.

Theorème III.2.2 (progrès des expressions) [H.6] Si $\mathsf{nil} \vdash_c E : T$ alors l'une des conditions suivantes est vérifiée :

– E est une valeur dans c ;
– E est réductible, c'est-à-dire qu'il existe E' telle que $E \longrightarrow_c E'$;
– E a levé une exception, c'est-à-dire qu'il existe $CC^c_{c'}$ et T' tels que $E = CC^c_{c'} \cdot \mathsf{Fail}_{T'}$;
– E est en attente de communication, c'est-à-dire qu'il existe $CC^c_{c'}$ et E' tels que $E = CC^c_{c'} \cdot !E'$ ou $E = CC^c_{c'} \cdot ?$.

Theorème III.2.3 (déterminisme des expressions) [H.10, H.11] Si $E \longrightarrow_c E'$ et $E \longrightarrow_c E''$ alors $E' = E''$ et les deux réductions appliquent la même règle au même rédex dans le même contexte. De plus, si $E \longrightarrow_c E'$ alors E n'est pas une valeur dans c.

La démonstration suit le schéma classique consistant à prouver que le constructeur de tête d'une valeur dont on connaît le type est bien celui que l'on attend. Une valeur de type abstrait a pour « constructeur » de tête un crochet coloré dont l'annotation de type est un type abstrait.

Dans une couleur ambiante c, le comportement de l'expression $[V^{c'}]^h_{c'}$ dépend des propriétés relatives de c, c' et h :

<table>
<tr><td>$c' \subseteq c$, c.à.d. $c' = \bullet$ ou $c' = c$:</td><td colspan="2">$[V^c]^h_c \longrightarrow_c V^c$ par $(\mathcal{H}/\mathsf{ered.col.le})$</td></tr>
<tr><td rowspan="2">$c' \notin \{c, \bullet\}$</td><td>$h = c'$:</td><td>$[V^h]^h_h$ est une valeur</td></tr>
<tr><td>$h \neq c'$:</td><td>le typage assure que $V^{c'} = [V^h]^h_h$,
et $[[V^h]^h_h]^h_{c'} \longrightarrow_c [V^h]^h_h$ par $(\mathcal{H}/\mathsf{ered.col.col})$</td></tr>
</table>

III.2.7.2 Correction de la compilation

Theorème III.2.4 (préservation du typage par compilation) [G.18] Si $\mathsf{L} \Longrightarrow \mathsf{L}'$ et $\mathsf{nil} \vdash_\bullet \mathsf{L} : \mathsf{T}$ alors $\mathsf{nil} \vdash_\bullet \mathsf{L}' : \mathsf{T}$.

Un programme sur une machine a la forme $\mathsf{L} = \mathsf{let}\, \mathsf{U}_1 = \mathsf{M}_1 : \mathsf{S}_1 \;\mathsf{in}\; \mathsf{dddLlet}\, \mathsf{U}_n = \mathsf{M}_n : \mathsf{S}_n \;\mathsf{in}\; \mathsf{E}$. Nous pouvons lui associer l'**environnement de compilation** $\Gamma = \mathsf{U}_1{:}\mathsf{S}_1, \ldots, \mathsf{U}_n{:}\mathsf{S}_n$ et la **substitution de compilation** $\sigma = \sigma_1 \ldots \sigma_n$ où $\sigma_i = \{\mathsf{U}_i.\mathsf{type} \leftarrow \mathsf{T}_i''\}$ si $\mathsf{S}_i = [\mathsf{X}{:}\mathrm{E}\textsc{q}(\mathsf{T}_i''),\ \mathsf{T}_i']$ et $\sigma_i = \textbf{id}$ si $\mathsf{S}_i = [\mathsf{X}{:}\mathrm{T}\textsc{ype},\ \mathsf{T}_i']$. Le résultat de la compilation de la machine L est $\sigma\mathsf{E}$.

Theorème III.2.5 (normalisation de la compilation) [H.8, H.9] Si $\mathsf{nil} \vdash_\bullet \mathsf{L} : \textsc{unit}$ alors il existe une unique expression E telle que $\mathsf{L} \Longrightarrow^* \mathsf{E}$.

En fait, l'hypothèse de bon typage de la machine n'est pas nécessaire : toute machine se réduit en une expression en autant d'étapes qu'elle ne compte de définitions de modules. L'unicité de E est une conséquence du déterminisme de la compilation.

III.2.7.3 Décidabilité du typage

Theorème III.2.6 (décidabilité du typage) [I.14] La vérification d'un jugement de typage est décidable.

Le point-clé est la décision de l'équivalence de deux types, qui est simple : deux types sont équivalents si et seulement si après substitution des variables ayant une sorte singleton et de l'éventuelle empreinte transparente les termes obtenus sont identiques.

Proposition III.2.7 (typage des programmes sources) [I.16] Le typage d'un programme source (sans empreinte ni crochet coloré) ne fait pas intervenir d'empreinte ni de crochet coloré.

III.2.7.4 Effacement des crochets colorés

On peut scinder les règles d'exécution en deux blocs : les règles d'élimination des crochets et les règles calculatoires. L'élimination des crochets est la partie de la sémantique opérationnelle engendrée par les règles de poussée des crochets ($\mathcal{H}$/ered.col.*) ainsi que la règle de contexte ($\mathcal{H}$/ered.cong). La sémantique calculatoire consiste en les autres axiomes ainsi que la règle de contexte.

On appelle **effacement des crochets** l'opération sur les expressions et autres entités syntaxiques consistant à éliminer tous les crochets colorés qui ne se trouvent pas à l'intérieur d'une empreinte. Notons $\Box\mathsf{E}$ le résultat de l'effacement des crochets sur l'expression E.

Theorème III.2.8 (terminaison de l'élimination des crochets) [I.19] L'élimination des crochets est déterministe et normalisante.

Ce résultat se démontre en remarquant que les règles de poussée des crochets diminuent la taille des expressions sous les crochets.

Theorème III.2.9 (compatibilité d'effacement des crochets) [I.26, I.27] L'effacement des crochets préserve la réduction.

En particulier, si une expression E se réduit en une valeur V, alors $\Box\mathsf{E}$ se réduit calculatoirement en $\Box\mathsf{V}$. Ceci autorise une implémentation du langage HAT à effacer les crochets colorés à la fin de la compilation (après le calcul des empreintes), afin d'obtenir une exécution plus efficace.

III.2.7.5 Correspondance entre le typage statique et le typage dynamique

Nous disposons de deux relations d'équivalence sur les types : l'équivalence statique, pour laquelle T et T' sont équivalents (dans un environnement Γ et une couleur c donnés) si et seulement si le jugement $\Gamma \vdash_{\mathsf{c}} \mathsf{T} \equiv \mathsf{T}'$ est dérivable ; et l'équivalence dynamique, pour laquelle T et T' sont équivalents (dans une couleur c et pour une expression E de type T données) si et seulement si $\mathsf{undyn}\,(\mathsf{dyn}\,\mathsf{E}\,\mathsf{at}\,\mathsf{T})\,\mathsf{at}\,\mathsf{T}'$ se réduit sans lever d'exception (ce qui *a priori* dépend du contexte d'exécution).

Ces deux équivalences n'agissent pas dans le même contexte. En particulier, l'équivalence dynamique est relative à une compilation et une trace d'exécution donnée du programme — cependant, puisque notre sémantique est déterministe, la seule variable est là le choix entre les copies de l'expression initiale qui sont finalement évaluées. Nous pouvons néanmoins cadrer une comparaison entre les deux formes d'équivalence.

Soit une machine $\mathsf{L} = \mathsf{let}\,\mathsf{U}_1 = \mathsf{M}_1 : \mathsf{S}_1\ \mathsf{in}\ \mathsf{dddLlet}\,\mathsf{U}_n = \mathsf{M}_n : \mathsf{S}_n\ \mathsf{in}\ \mathsf{E}$, Γ l'environnement de compilation associé et σ la substitution de compilation. Supposons que E contient une sous-expression de la forme $\mathsf{dyn}\,\mathsf{E}_1\,\mathsf{at}\,\mathsf{T}_1$ et une sous-expression de la forme $\mathsf{undyn}\,\mathsf{E}_2\,\mathsf{at}\,\mathsf{T}_2$. Ces sous-expression apparaissent dans la forme compilée de E comme $\mathsf{dyn}\,\sigma\mathsf{E}_1\,\mathsf{at}\,\sigma\mathsf{T}_1$ et $\mathsf{undyn}\,\sigma\mathsf{E}_2\,\mathsf{at}\,\sigma\mathsf{T}_2$. Notons que le contexte dans E de ces expressions ne peut lier que des variables d'expression, qui n'apparaissent pas dans les types.

Si ces deux opérations sont mises en correspondance lors de l'exécution, nous pouvons dire qu'il y a équivalence dynamique entre T_1 et T_2 si et seulement si $\mathsf{undyn}\,(\mathsf{dynned}\,\mathsf{V}\,\mathsf{at}\,\sigma\mathsf{T}_1)\,\mathsf{at}\,\sigma\mathsf{T}_2$ ne lève pas d'exception, ce qui signifie que $\sigma\mathsf{T}_1 = \sigma\mathsf{T}_2$ (condition indépendante de V). D'autre part, il y a équivalence statique entre T_1 et T_2 si et seulement si $\Gamma \vdash_{\bullet} \mathsf{T}_1 \equiv \mathsf{T}_2$ (la couleur ambiante est vide puisque le code considéré provient du programme principal et non du module). La correspondance entre équivalences statique et dynamique revient donc au lien entre les deux conditions $\sigma\mathsf{T}_1 = \sigma\mathsf{T}_2$ et $\Gamma \vdash_{\bullet} \mathsf{T}_1 \equiv \mathsf{T}_2$.

Theorème III.2.10 (correspondance entre typage statique et typage dynamique) [J.16] Si la machine L est bien formée, que son code source ne contient pas d'empreinte, et si les définitions de modules sont sans répétition, alors $\sigma\mathsf{T}_1 = \sigma\mathsf{T}_2$ si et seulement si $\Gamma \vdash_{\bullet} \mathsf{T}_1 \equiv \mathsf{T}_2$.

Autrement dit, les équivalences statique et dynamique de typage coïncident, sous réserve d'une condition d'absence de répétition qu'il nous reste à expliciter. Chaque définition de module sur une machine donne lieu à la production d'une empreinte $h_1, \ldots, h_n$. Il se peut que par coïncidence deux empreintes soient identiques ; cela arrive notamment si deux modules ont exactement le même code ($\mathsf{M}_i = \mathsf{M}_j$ et $\mathsf{S}_i = \mathsf{S}_j$), mais peut également se produire si deux modules ont le même code à une substitution sur les dépendances près, et que les dépendances ont à leur tour des empreintes identiques. Hors de ce cas pathologique — qui peut être évité en attribuant à chaque module un nom distinct — il y a bien coïncidence entre typage statique et typage dynamique pour un programme constitué d'une seule machine.

Notons que si l'on considère un programme réparti, il est attendu que des définitions de modules situées sur des machines différentes donnent lieu à des empreintes identiques. Dans ce cas, le typage dynamique est en quelque sorte plus permissif que le typage statique, mais seulement dans le cas précis où une même implémentation d'un type abstrait se retrouve en plusieurs exemplaires, cas que nous souhaitions justement permettre.

Chapitre IV

TOPHAT : un calcul de modules adapté aux environnements répartis

IV.1 Introduction

Objectifs

Ce chapitre est consacré à la présentation d'un système de modules mariant une gestion flexible des types abstraits dans les programmes répartis (comme dans le système HAT) avec les fonctionnalités usuelles des systèmes de modules pour ML.

Notre système prend la forme d'un lambda-calcul typé, pour lequel nous donnons des règles de typage et une sémantique opérationnelle à petits pas préservant le typage. Le système décrit ici vise à être une modélisation théorique et non directement un langage de programmation, même si nos choix de conception seront motivés par des besoins pratiques. Nous omettrons donc certains aspects pouvant être considérés comme du sucre syntaxique. Nous discuterons des possibilités d'implémentation à la section V.3.3.

L'espace des systèmes de modules existants est vaste, aussi bien en termes d'expressivité qu'en termes de style (nous en avons touché quelques mots à la section I.2). En termes d'expressivitié, notre but est d'intégrer les fonctionnalités que nous jugeons fondamentales à celles qui constituent notre objectif particulier (types abstraits pour la programmation répartie). Pour ce qui est du style, notre approche est résolument compositionnelle, c'est-à-dire que nous chercherons à capturer chaque facette du langage dans une construction adaptée, compréhensible par elle-même.

Vocabulaire et notations

Nous distinguerons dans notre exposition les mots « expression », qui désignera une catégorie syntaxique particulière (usuellement notée E), et « terme », mot générique désignant un élément de n'importe quelle catégorie syntaxique (expression, type, module, environnement, etc.). Nous utiliserons habituellement les notations $\aleph$ et $\beth$ pour désigner un terme sans présupposer de sa catégorie syntaxique.

Si $\aleph$ est un terme quelconque, x une variable, et E une expression du langage correspondant à x, nous noterons $\{\mathsf{x}{\leftarrow}\mathsf{E}\}\aleph$ la substitution de x par E dans $\aleph$. De même, si X est une variable de module et M un module, $\{\mathsf{X}{\leftarrow}\mathsf{M}\}\aleph$ désignera la susbtitution de X par M dans $\aleph$.

Plan

Nous allons présenter notre langage de manière incrémentale, en cinq étapes. Chaque étape raffine ou augmente le langage précédent.

```
module M0 = struct
    type t = int
    let début = 0
    let courant x = x
    let suivant x = courant x + 2
  end
```

$$\begin{array}{l} \mathsf{M}_0 = \\ \quad \langle \mathsf{X}_{\mathtt{t}} = \langle \textsc{int} \rangle, \\ \quad \langle \mathsf{X}_{\mathtt{d\acute{e}but}} = \langle 0 \rangle, \\ \quad \langle \mathsf{X}_{\mathtt{courant}} = \langle \lambda x : \mathsf{Typ}\,\mathsf{X}_{\mathtt{t}}.\ x \rangle, \\ \quad \langle \mathsf{X}_{\mathtt{suivant}} = \langle \lambda x : \mathsf{Typ}\,\mathsf{X}_{\mathtt{t}}.\ (\mathsf{Val}\,\mathsf{X}_{\mathtt{courant}})x + 2 \rangle, \\ \quad \langle\rangle \rangle\rangle\rangle\rangle \end{array}$$

FIG. IV.1 – Un exemple de structure

$\mathcal{B}$ Dans un premier temps, nous présenterons un système de modules de base, présentant les manières de construire des modules qui se suffiraient en l'absence de typage. En présence de typage, ce système est simple mais trop peu expressif pour nos besoins.

$\mathcal{S}$ Nous ajouterons au système de base des types singleton, qui permettent d'exprimer des égalités entre types. Nous obtiendrons un langage décrivant adéquatement un système de modules dépourvu d'abstraction.

$\mathcal{E}$ Nous serons alors mûr pour ajouter au système la construction de scellage, qui permet de créer des types abstraits. Nous verrons que le scellage a un impact sur la pureté du langage, et présenterons alors un système d'effets adapté.

$\mathcal{C}$ Le système précédent permet de construire des abstractions dans un programme source, mais elles ne sont pas visibles lors de l'exécution du programme. Nous munirons alors le langage d'empreintes et de couleurs qui permettent de préserver les abstractions lors de l'exécution.

$\mathcal{D}$ Nous pourrons enfin définir des constructions destinées au typage dynamique, et montrer que celles-ci sont utilisables y compris dans des programmes répartis.

Le système $\mathcal{D}$ forme le langage TOPHAT[1].

À chaque étape, nous motiverons par des exemples les fonctionnalités introduites, et les étudierons du point de vue du programmeur. Nous discuterons ensuite de la forme que peut prendre une théorie incluant les constructions désirées. Après cela, nous donnerons des règles précises définissant les sémantiques statique (typage) et dynamique (réduction) d'un calcul typé.

IV.2 Un calcul de modules $\boxed{\mathcal{B}}$

Dans cette section, nous présentons le noyau d'un langage de description de modules. Ce noyau, appelé $\mathcal{B}$, s'articule autour de deux fonctionnalités essentielles :
- l'agrégation de valeurs et de types, au sein de *structures* ;
- la possibilité d'exprimer des modules paramétriques, ou *foncteurs*.

IV.2.1 Fondements

IV.2.1.1 Structures

Le terme **structure** est habituellement employé pour désigner la brique de base dans un langage de modules que constitue l'agrégat de définitions de types et de valeurs. Suivant D. Dreyer, K. Crary et R. Harper [DCH03], nous allons exprimer les structures en termes d'opérations plus fondamentales.

[1] *Total or Partial Hashed Abstract Types* (empreintes de types abstraits totaux ou partiels) — les termes « totaux » et « partiels » étant inspirés de la cohabitation de foncteurs applicatifs et génératifs, quelquefois dits totaux et partiels.

La première colonne de la figure IV.1 donne un exemple de structure dans le langage Objective Caml. Cet exemple nous inspire quelques remarques.

- La structure est composée de plusieurs (quatre) champs.
- L'un des champs est un type ; les autres sont des valeurs.
- Chaque champ est nommé.
- La structure limite la portée des noms de champs.
- Les champs sont présentés dans un certain ordre.

Nous allons ici faire une simplification : nous ne prendrons pas en compte les noms de champs. Nous repérerons donc par exemple le champ `courant` au fait qu'il s'agit du troisième champ, et non par son nom. Ceci a des conséquences sur l'expressivité du langage .

Ceci étant fait, nous pouvons séparer la construction d'une structure en deux familles d'opérations.

- Une **structure élémentaire** est composée d'un unique champ, qui peut être un type ou la valeur d'une expression. Nous noterons $\langle \mathsf{T} \rangle$ une structure élémentaire composée d'un unique champ qui est un type T, et $\langle \mathsf{E} \rangle$ une structure élémentaire composée d'un unique champ qui est la valeur d'une expression E.
- Deux structures (plus généralement, deux modules) M_1 et M_2 peuvent être assemblées en une paire ordonnée $\langle \mathsf{X} = \mathsf{M}_1,\ \mathsf{M}_2 \rangle$. Il s'agit d'une **somme dépendante** ou **paire dépendante** (nous emploierons les deux termes indifféremment[2]) : l'expression de module M_2 peut utiliser la variable X, qui est liée à la valeur de l'expression de module M_1. Notons que X n'est pas un nom de champ : sa portée est limitée à M_2, alors qu'un nom de champ serait accessible depuis l'extérieur de la structure.

Les destructeurs correspondant à la paire sont les **projections** que nous noterons π_1 et π_2 : $\pi_1 \langle \mathsf{X} = \mathsf{M}_1,\ \mathsf{M}_2 \rangle$ s'évalue en la valeur de M_1 et $\pi_2 \langle \mathsf{X} = \mathsf{M}_1,\ \mathsf{M}_2 \rangle$ s'évalue en la valeur de M_2 dans lequel la valeur de M_1 est substituée à X. Nous complétons ces opérations par la **structure vide** $\langle \rangle$. Nous traduirons par convention une structure en une liste dont les éléments sont des structures élémentaires.

La conversion d'une structure élémentaire en l'objet sous-jacent sera notée $\mathsf{Typ}\,\mathsf{M}$ si l'objet en question est un type, et $\mathsf{Val}\,\mathsf{M}$ s'il s'agit de la valeur d'une expression. La seconde colonne de la figure IV.1 décrit la structure dans les notations que nous venons d'introduire.

IV.2.1.2 Signatures de structures

La **signature** d'une structure peut soit spécifier entièrement les champs types qu'elle contient ainsi que les types des champs valeurs (on parle alors de **signature transparente**), soit omettre la définition des champs types en mentionnant seulement leur existence. Dans le second cas, on dit que les types dont seule l'existence est déclarée sont **abstraits** (un type dont la définition est donnée dans la signature est dit **concret**). Une signature est dite **opaque** lorsqu'elle contient un type abstrait.

La figure IV.2 donne deux signatures possibles pour la structure de la figure IV.1, l'une transparente, l'autre opaque.

Nous aurons dans notre système des constructions pour les signatures analogues à celles que nous avons présentées pour les structures, à ceci près qu'il y a deux formes possibles pour la signature d'un champ type.

Nous noterons classiquement $\Sigma \mathsf{X}{:}\mathsf{S}_1.\mathsf{S}_2$ la signature (dite signature **somme dépendante**) d'une paire dont la première composante a la signature S_1 et la seconde a la signature S_2 ; dans S_2, la

[2] Il conviendra de ne pas confondre les sommes dépendantes $\Sigma x : \mathsf{T1}.\mathsf{T2}$, qui sont des signatures de paires, avec les produits dépendants, qui sont des signatures de fonctions.

```
module type Concrete = sig
  type t = int
  début : t
  courant : t->int
  suivant : t->t
end
```

$$\begin{array}{l} S_c = \\ \quad \Sigma X_t : \mathsf{S}(\langle \text{INT} \rangle). \\ \quad \Sigma X_{\texttt{début}} : [\![\mathsf{Typ}\, X_t]\!]. \\ \quad \Sigma X_{\texttt{courant}} : [\![\mathsf{Typ}\, X_t \to \text{INT}]\!]. \\ \quad \Sigma X_{\texttt{suivant}} : [\![\mathsf{Typ}\, X_t \to \mathsf{Typ}\, X_t]\!]. \\ \quad \mathbb{1} \end{array}$$

(a) transparente (`t` est concret)

```
module type Abstraite = sig
  type t
  début : t
  courant : t->int
  suivant : t->t
end
```

$$\begin{array}{l} S_a = \\ \quad \Sigma X_t : \text{TYPE}. \\ \quad \Sigma X_{\texttt{début}} : [\![\mathsf{Typ}\, X_t]\!]. \\ \quad \Sigma X_{\texttt{courant}} : [\![\mathsf{Typ}\, X_t \to \text{INT}]\!]. \\ \quad \Sigma X_{\texttt{suivant}} : [\![\mathsf{Typ}\, X_t \to \mathsf{Typ}\, X_t]\!]. \\ \quad \mathbb{1} \end{array}$$

(b) opaque (avec `t` abstrait)

FIG. IV.2 – Un exemple de signatures

variable X désigne la première composante (dont la signature est S_1). Nous noterons $\mathbb{1}$ la signature de la structure vide $\langle\rangle$.

Nous noterons $[\![T]\!]$ la signature d'une structure élémentaire $\langle E \rangle$ où E est une expression de type T. Une structure contenant un champ type T a deux signatures possibles : la signature TYPE laisse le champ abstrait, tandis que la signature $\mathsf{S}(\langle T \rangle)$ spécifie que la composante est le champ T. La signature $\mathsf{S}(\langle T \rangle)$ se lit « **singleton** (de) $\langle T \rangle$ » ; elle est ainsi nommée car seul le module $\langle T \rangle$ (ou un module équivalent) la possède. À l'inverse, toute structure élémentaire bien formée qui contient un champ type a la signature TYPE ; on peut voir $\Sigma X{:}\text{TYPE}.S$ comme « il existe un type T tel que $\{X \leftarrow \langle T \rangle\}S$ », à ceci près que l'opération π_1 appliquée à un module ayant une telle signature « existentielle » permet d'accéder au type ainsi quantifié.

Au lieu de distinguer TYPE et $\mathsf{S}(\langle T \rangle)$ au niveau des signatures, nous pourrions utiliser une forme commune de signature de champ type $[\![K]\!]$ où K est une **sorte** (*kind*), et distinguer deux formes de sortes $K = \top$ et $K = \mathsf{S}(T)$. Cette présentation (plus courante, et plus ancienne [Asp95]) aurait l'inconvénient *a priori* de nécessiter une catégorie syntaxique supplémentaire, mais l'avantage d'être conforme à notre principe de structuration du langage. L'utilisation de singletons de module sera justifiée *a posteriori* par le fait que les équivalences entre types résultent de calculs sur les modules, les types eux-mêmes étant des objets statiques.

Notons que lorsque l'on parle d'un type abstrait, il s'agit d'un champ type d'une signature, et non d'un terme de la grammaire des types. Cela n'a pas de sens, dans l'absolu, de se demander si INT est un type abstrait ou non. Nous le traitons comme un type prédéfini ; en pratique, il pourrait être défini comme un type abstrait de la bibliothèque standard. (Nous serons cependant amené à discuter du caractère abstrait de types de la forme $\mathsf{Typ}\, M$, c'est-à-dire d'un champ type d'un module ; ce sera en relation avec des équations entre types, à partir du système $\mathcal{C}$.)

Notamment, une structure a « naturellement » une signature transparente (qui n'abstrait aucun de ses types) ; elle en admet en général d'autres, opaques, par sous-signaturage (voir la section IV.3.1.1).

IV.2.1.3 Foncteurs

Fidèle à notre principe directeur d'assemblage de constructions maîtrisées, nous équipons notre système de familles de modules paramétriques sous la forme de lambda-abstractions, appelées **foncteurs**. Nous noterons $(\Lambda X{:}S.M)$ un foncteur, et $F\,M$ l'application du foncteur F au module M. La signature correspondant au foncteur $(\Lambda X{:}S.M)$ est $(\Pi X{:}S.S')$, dans laquelle S' est la signature de M sachant que X a la signature S ; une telle signature est appelée **produit dépendant** ou **fonction dépendante**[3].

Voici un exemple de signature de foncteur, en syntaxe Objective Caml et dans notre système. Il s'agit d'un foncteur qui implémente des ensembles finis sur un type muni d'une relation d'ordre total.

```
functor (A : sig
    type elt
    val inférieur : elt->elt->bool
end) -> sig
  type ens
  val vide : ens
  val ajout : A.elt->ens->ens
...
end
```

$$
\begin{array}{l}
\Pi A : (\\
\quad \Sigma X_{\mathtt{elt}} : \mathrm{TYPE}. \\
\quad \Sigma X_{\mathtt{inférieur}} : [\![\mathsf{Typ}\,X_{\mathtt{elt}} \to \mathsf{Typ}\,X_{\mathtt{elt}} \to \mathrm{BOOL}]\!]. \\
\quad \mathbb{1}). \\
\Sigma X_{\mathtt{ens}} : \mathrm{TYPE}. \\
\Sigma X_{\mathtt{vide}} : [\![\mathsf{Typ}\,X_{\mathtt{ens}}]\!]. \\
\Sigma X_{\mathtt{ajout}} : [\![\mathsf{Typ}\,\pi_1 A \to \mathsf{Typ}\,X_{\mathtt{ens}} \to \mathsf{Typ}\,X_{\mathtt{ens}}]\!]. \\
\dots \\
\mathbb{1}
\end{array}
$$

Le type `ens` du résultat est abstrait : si on sait que le foncteur F a la signature ci-dessus, cela ne donne aucune information quant au contenu du champ correspondant dans $F\,M$ pour un argument M donné.

Le type `elt` de l'argument est également non spécifié[4]. Cela signifie que l'argument effectif passé au foncteur peut être n'importe quel type ; il est d'ailleurs courant au sein d'un même programme d'utiliser des ensembles finis d'entiers, de chaînes de caractères, etc., construits avec le même foncteur. La signature TYPE dans l'argument d'un foncteur a ainsi une interprétation universelle, qui va de paire avec l'interprétation existentielle qu'elle a dans le résultat du foncteur.

IV.2.1.4 Liaisons locales

Nous disposons pour l'instant dans le langage des modules de deux constructions liant une variable : la paire dépendante $\langle X{=}M_1, M_2\rangle$ et la lambda-abstraction $(\Lambda X : S_0.M_1)$. Nous serons amené à en utiliser une troisième, la très classique **liaison locale** $\mathsf{let}\,X = M_0\,\mathsf{in}\,M$. Cette profusion est-elle justifiée ?

La lambda-abstraction est évidemment nécessaire, puisque c'est la seule manière d'instancier potentiellement la variable par plusieurs termes différents. Dans un monde non typé, la liaision locale est un cas particulier de la lambda-abstraction et de l'application : $(\mathsf{let}\,X = M_0\,\mathsf{in}\,M) = (\Lambda X.M)M_0$ — toutefois en présence de types nous devons trouver une signature suffisamment générale S_0 à M_0). La liaison locale devient plus que du sucre syntaxique lorsque l'unicité de l'instanciation permet un typage plus général, ce qui est notre cas — nous verrons dans le cadre du système $\mathcal{E}$ que la construction « let » autorise plus d'effets[5].

[3]Voir la note 2 page 119.

[4]On note au passage qu'en Objective Caml, le type de des éléments est désigné dans la signature du résultat du foncteur par son nom qualifié `A.elt` ; dans notre système sans noms de champs, l'accès est fait au premier champ de l'argument, le contenu de la structure élémentaire $\pi_1 A$.

[5]En ML, le typage du « let » est plus général pour une raison différente, à savoir que généraliser les variables de type y est décidable alors que ce n'est pas le cas pour la lambda-abstraction en général.

Pour ce qui est des paires, si elles sont le seul moyen de mettre côte à côte deux sous-termes, l'aspect dépendant n'est en revanche pas nécessaire : nous pouvons nous contenter de la **paire ordinaire** que nous noterons $(\mathsf{M}_1, \mathsf{M}_2)$. La paire dépendante peut alors être codée sous l'une des formes

$$\langle \mathsf{X}{=}\mathsf{M}_1, \mathsf{M}_2\rangle \;=\; \mathsf{let}\,\mathsf{X} = \mathsf{M}_1\,\mathsf{in}\,(\mathsf{X}, \mathsf{M}_2) \quad \text{ou} \quad \langle \mathsf{X}{=}\mathsf{M}_1, \mathsf{M}_2\rangle \;=\; (\Lambda \mathsf{X} : \mathsf{S}_1.(\mathsf{X}, \mathsf{M}_2))\mathsf{M}_1$$

Les règles de calcul sont les mêmes dans tous les cas (calcul de M_1 si l'on est en appel par valeur, puis substitution de M_1 à X dans M_2, puis calcul de M_2) ; en présence d'effets, les règles de typage différeront, et il conviendra de choisir la traduction souhaitée. En tout état de cause, nous nous limiterons dans la présentation formelle aux paires ordinaires.

Notons que même des paires ordinaires peuvent bénéficier de types sommes dépendants. Par exemple, le module $(\langle \textsc{int}\rangle, 3)$ a entre autres signatures $\Sigma \mathsf{t}{:}\textsc{type}.[\![\mathsf{Typ}\,\mathsf{t}]\!]$ (« un type et une valeur de ce type »), dans laquelle la dépendance de la deuxième composante envers la première est fondamentale.

IV.2.2 Du langage de base

IV.2.2.1 Exigences

Le langage des modules, qui décrit en principe la structuration d'un programme, s'articule autour d'un langage de base, le langage des expressions et des types, qui décrit l'exécution du programme. Il est courant lors de la présentation d'un langage de modules de laisser peu spécifié le langage de base [Ler00, AZ02], supposant seulement qu'il est muni d'une relation de réduction et d'une relation de typage vérifiant des propriétés raisonnables. Ceci a le double avantage de donner un large champ d'application à la théorie et de ne pas encombrer la présentation avec des détails inutiles.

Il y a cependant une condition, qui est que les détails en question soient effectivement inutiles. Or ne pas spécifier le langage de base limite les interactions que les deux niveaux peuvent avoir, en rejetant *a priori* certaines constructions. Par exemple, on peut vouloir lier localement un module dans une expression de base, ou lier une expression de base dans une définition de module, ou plus généralement paramétrer un module par une fonction ou vice versa.

Nous serons conduit à décrire plusieurs constructions du langage de base. D'une part, nous souhaitons analyser certains aspects (notamment la liaison de variable et l'application de fonction) plus finement que ne le permettrait une approche complètement opaque du langage de base, ce qui nous conduit à expliciter les constructions en question. D'autre part, notre travail ne se limite pas aux modules, il a, rappelons-le, pour but de dégager une notion de type abstrait valable dans un environnement réparti, ce qui fera intervenir plusieurs nouvelles fonctionnalités à la fois au niveau des modules et à la base. Nous utiliserons également quelques valeurs et types de base (booléens, entiers, chaînes de caractères) dans des exemples.

IV.2.2.2 Un langage unique

Nous voulons un langage de base typé, et contenant notamment le lambda-calcul (variables, lambda-abstractions et applications). Nous aurons de plus besoin de le munir de certaines constructions, notamment les crochets colorés du système $\mathcal{C}$, qui existent de manière similaire dans le langage des modules. Ceci suggère de considérer les langages des modules et de base comme un unique langage, quitte à ce que certaines constructions soient vues comme appartenant au domaine cognitif de la structuration et d'autres au domaine cognitif de l'exécution.

Ainsi allons-nous définir et utiliser une syntaxe unique couvrant les deux aspects structuration et exécution du programme. Nous ne prétendons pas couvrir entièrement l'aspect exécution, qui sera couvert par l'utilisation de constantes prédéfinies supplémentaires[6] (nous référons le lecteur à la section V.3.2.2).

Une autre manière de voir notre approche est de considérer que nous avons fusionné le langage des modules et le langage de base. Nous ne marquerons pas syntaxiquement le passage de l'un à l'autre, notant simplement E pour $[\![\mathsf{E}]\!]$ et M pour $\mathsf{Val}\,\mathsf{M}$.

Nous ne prétendons néanmoins pas que la définition d'un langage unique soit une panacée. En effet, si elle simplifie la présentation de notre théorie, elle s'accomode plus mal de certaines extensions. Nous évaluerons les mérites comparés du langage unique et des deux langages à la section V.3.1.2.

IV.2.3 Présentation formelle du noyau

IV.2.3.1 Syntaxe

Nous sommes maintenant en mesure de donner une syntaxe formelle à notre langage, ainsi que des règles de typage et d'exécution. Nous nous limitons ici aux fonctionnalités que nous avons déjà mentionnées ; nous ajouterons d'autres constructions et annotations au fur et à mesure des besoins. Nous excluons pour l'instant les singletons, auxquels nous consacrerons la section IV.3.

Puisque nous définissons un seul langage, les objets que nous avons jusqu'ici noté E et ceux que nous avons noté M appartiennent désormais au même monde ; nous les noterons E et les appellerons des **expressions**. De même, nous noterons T plutôt que S et parlerons de **types**. Certaines expressions resteront néanmoins vues intuitivement comme des modules, et leurs types comme des signatures.

$\mathsf{E} ::=$	**expression ou module**	$\mathsf{T} ::=$	**type ou signature**
$x \mid y \mid t \mid \ldots$	variables		
$()$	valeur unité	UNIT	unité
$\mathsf{false} \mid \mathsf{true}$	booléen (génériquement $b\nu$)	BOOL	booléens
$0 \mid 1 \mid \ldots$	entier (génériquement $\underline{n}$)	INT	entiers
$\langle \mathsf{T} \rangle$	champ type	TYPE	champ type abstrait
		$\mathsf{Typ}\,\mathsf{E}$	projection d'un champ type
$(\mathsf{E}_1, \mathsf{E}_2)$	paire	$\Sigma x : \mathsf{T}_1.\, \mathsf{T}_2$	somme dépendante
$\pi_i\, \mathsf{E}$	projection ($i \in \{1, 2\}$)		
$\lambda x : \mathsf{T}.\ \mathsf{E}$	lambda-abstraction	$\Pi x : \mathsf{T}_0.\, \mathsf{T}_1$	produit dépendant
$\mathsf{E}_1\, \mathsf{E}_2$	application		
$\mathsf{let}\, x = \mathsf{E}_0\, \mathsf{in}\, \mathsf{E} : \mathsf{T}$	liaison locale		

Introduisons d'ores et déjà quelques abréviations.

$\mathsf{T}_1 * \mathsf{T}_2 := \Sigma x : \mathsf{T}_1.\, \mathsf{T}_2$ type produit

$\mathsf{T}_1 \to \mathsf{T}_2 := \Pi x : \mathsf{T}_1.\, \mathsf{T}_2$ type « flèche » (fonction)

Dans les définitions de $\mathsf{T}_1 * \mathsf{T}_2$ et $\mathsf{T}_1 \to \mathsf{T}_2$, x est une variable fraîche (c'est-à-dire non libre dans T_2).

IV.2.3.2 Variables

Nous définissons de manière standard les notions de **variables libres**, **occurrences liées**, **alpha-conversion** et **substitutions**. Un terme **clos** est un terme sans variable libre.

Nous noterons **fv** $\aleph$ l'ensemble des variables libres de $\aleph$. La substitution de x par E dans le terme $\aleph$ sera notée $\{x \leftarrow \mathsf{E}\}\aleph$.

[6] Par exemple, notre langage ne traite pas des définitions récursives, mais l'on pourra utiliser un combinateur de point fixe prédéfini pour construire des fonctions récursives.

Nous travaillerons systématiquement *à alpha-conversion près*, c'est-à-dire que chacun des termes que nous écrirons désignera formellement sa classe d'équivalence modulo alpha-conversion. Chaque étape de typage et de réduction pourra s'accompagner du renommage de variables. Par exemple, si une règle de typage ou de réduction nécessite d'instancier plusieurs occurrences d'une même méta-variable, il sera toujours possible d'employer des représentants différents d'une classe d'équivalence pour les différentes instanciations. Ce choix d'alpha-conversion implicite a été motivé par l'usage, ainsi que sa supériorité en termes de clarté d'exposition. Nous ne rappellerons pas en général la possibilité permanente d'alpha-conversion, mais signalerons au fur et à mesure les constructions liantes qu'il conviendrait de traiter adéquatement si l'on souhaitait par exemple modéliser notre système dans un outil formel en utilisant des indices de de Bruijn ou une logique d'ordre supérieur.

IV.2.3.3 Environnements

Un **environnement** est une liste finie ordonnée de couples (variable, type). Nous noterons nil l'environnement vide, $x : T$ un environnement de longueur 1, et « , » l'opération (associative) de concaténation. Ainsi un environnement liant trois variables pourra être noté $x : T, y : T', z : T''$; d'autres manières de noter ce même environnement sont $((x : T, y : T'), z : T'')$ ou $(x : T, y : T'), z : T''$ ou encore $(((\mathsf{nil}, x : T), y : T'), z : T'')$; nous omettrons en général les parenthèses (puisqu'elles sont inutiles) et n'utiliserons nil que pour traiter de l'environnement vide isolément.

Nous pouvons engendrer les environnements par la grammaire suivante :

Γ ::=	**environnement**
nil	vide
$\Gamma, x : T$	liaison d'une variable x

Nous pouvons également décrire les environnements comme les objets de la forme $x_1 : T_1, \ldots, x_k : T_k$ avec $k \in \mathbb{N}$.

Le **domaine** d'un environnement $\Gamma = x_1 : T_1, \ldots, x_k : T_k$, noté $\mathbf{dom}\,\Gamma$, est l'ensemble des variables $\{x_1, \ldots, x_k\}$.

Un environnement lie les variables de son domaine, qui sont comme partout sujettes à alpha-conversion. L'écriture (Γ, Γ') suppose que les domaines de Γ et Γ' sont disjoints, et il conviendra d'appliquer une alpha-conversion si nécessaire. Dans une concaténation d'environnements (Γ, Γ'), les variables du domaine de Γ lient dans Γ'.

Notons que le caractère ordonné de nos environnements vient du fait que nous manipulons des types dépendants. Ainsi $(x : \text{INT}, y : \mathsf{Typ}\, x)$ est un environnement correct, alors que $(y : \mathsf{Typ}\, x, x : \text{INT})$ ne l'est pas (il peut s'écrire aussi $(y : \mathsf{Typ}\, x, z : \text{INT})$ après renommage de la variable liée en z ; l'occurence restante de x est libre).

IV.2.4 Typage

IV.2.4.1 Introduction

Nous considérons comme (fragment de) programme correct une expression E munie d'un type T tels que E ait le type T. Lorsque E contient des variables libres, il convient de munir celles-ci d'un type à l'aide d'un environnement.

Nous manipulerons plusieurs formes de jugements de typages ; la table suivante donne leur syntaxe. La seule forme de jugements que nous manipulerons est le **jugement local**.

$\mathscr{J}$ ::=	**jugement de typage**
$\Gamma \vdash J$	jugement local

J ::=	**membre droit de jugement local**
ok	correction de l'environnement
$\mathsf{T}\,\mathsf{ok}$	correction du type T
$\mathsf{E} : \mathsf{T}$	typage d'une expression

Nous présentons les règles de typage sous la forme classique d'un système de déduction.

IV.2.4.2 $\boxed{\Gamma \vdash \mathsf{ok}}$ Correction de l'environnement

Les environnements sont construits de gauche à droite, liaison par liaison. Chaque type donné à une variable dans un environnement doit être validé dans l'environnement préfixe à la liaison considérée. Notons que les variables liées par un environnement sont automatiquement distinctes de par notre convention d'alpha-conversion.

$$\frac{}{\mathsf{nil} \vdash \mathsf{ok}}\ (\mathcal{B}/\textbf{envok.nil}) \qquad \frac{\Gamma \vdash \mathsf{T}\,\mathsf{ok}}{\Gamma, x : \mathsf{T} \vdash \mathsf{ok}}\ (\mathcal{B}/\textbf{envok.x})$$

IV.2.4.3 $\boxed{\Gamma \vdash \mathsf{T}\,\mathsf{ok}}$ Correction des types

Les règles de correction des types sont essentiellement standards : on demande pour les types de base la correction de l'environnement, et les types construits sont corrects lorsque leurs parties le sont (avec une gestion des dépendances).

$$\frac{\Gamma \vdash \mathsf{ok}}{\Gamma \vdash \textsc{unit}\,\mathsf{ok}}\ (\mathcal{B}/\textbf{tok.base.unit}) \qquad \frac{\Gamma \vdash \mathsf{ok}}{\Gamma \vdash \textsc{bool}\,\mathsf{ok}}\ (\mathcal{B}/\textbf{tok.base.bool}) \qquad \frac{\Gamma \vdash \mathsf{ok}}{\Gamma \vdash \textsc{int}\,\mathsf{ok}}\ (\mathcal{B}/\textbf{tok.base.int})$$

$$\frac{\Gamma \vdash \mathsf{ok}}{\Gamma \vdash \textsc{type}\,\mathsf{ok}}\ (\mathcal{B}/\textbf{tok.type})$$

$$\frac{\Gamma \vdash \mathsf{T}'\,\mathsf{ok} \qquad \Gamma, x : \mathsf{T}' \vdash \mathsf{T}''\,\mathsf{ok}}{\Gamma \vdash \Sigma x : \mathsf{T}'.\, \mathsf{T}''\,\mathsf{ok}}\ (\mathcal{B}/\textbf{tok.pair}) \qquad \frac{\Gamma \vdash \mathsf{T}'\,\mathsf{ok} \qquad \Gamma, x : \mathsf{T}' \vdash \mathsf{T}''\,\mathsf{ok}}{\Gamma \vdash \Pi x : \mathsf{T}'.\, \mathsf{T}''\,\mathsf{ok}}\ (\mathcal{B}/\textbf{tok.fun})$$

IV.2.4.4 $\boxed{\Gamma \vdash \mathsf{E} : \mathsf{T}}$ Typage des expressions

Constantes Les constantes élémentaires reçoivent leur type.

$$\frac{\Gamma \vdash \mathsf{ok}}{\Gamma \vdash () : \textsc{unit}}\ (\mathcal{B}/\textbf{et.base.unit}) \qquad \frac{\Gamma \vdash \mathsf{ok}}{\Gamma \vdash bv : \textsc{bool}}\ (\mathcal{B}/\textbf{et.base.bool}) \qquad \frac{\Gamma \vdash \mathsf{ok}}{\Gamma \vdash \underline{n} : \textsc{int}}\ (\mathcal{B}/\textbf{et.base.int})$$

Variables Les variables ont le type que leur donne l'environnement.

$$\frac{\Gamma \vdash \mathsf{ok} \qquad \text{si } x : \mathsf{T} \in \Gamma}{\Gamma \vdash x : \mathsf{T}}\ (\mathcal{B}/\textbf{et.x})$$

Paires Bien que notre syntaxe comporte des sommes dépendantes, le système $\mathcal{B}$ est trop restreint pour les exploiter (il faudra attendre le système $\mathcal{S}$) : il n'est capable d'attribuer à une paire qu'un type produit ordinaire, pour lequel nous énonçons les règles classiques.

$$\frac{\Gamma \vdash \mathsf{E}_1 : \mathsf{T}_1 \qquad \Gamma \vdash \mathsf{E}_2 : \mathsf{T}_2}{\Gamma \vdash (\mathsf{E}_1, \mathsf{E}_2) : \mathsf{T}_1 * \mathsf{T}_2}\ (\mathcal{B}/\textbf{et.pair}) \qquad \frac{\Gamma \vdash \mathsf{E} : \mathsf{T}_1 * \mathsf{T}_2}{\Gamma \vdash \pi_1\, \mathsf{E} : \mathsf{T}_1}\ (\mathcal{B}/\textbf{et.proj.1}) \qquad \frac{\Gamma \vdash \mathsf{E} : \mathsf{T}_1 * \mathsf{T}_2}{\Gamma \vdash \pi_2\, \mathsf{E} : \mathsf{T}_2}\ (\mathcal{B}/\textbf{et.proj.2})$$

Fonctions Le typage des fonctions (ou foncteurs) et de l'application ne pose pas de difficulté particulière, nous énonçons les règles usuelles en présence de produits dépendants. On note que le pour typer l'application d'une fonction ayant un type dépendant, il faut substituer les occurences du paramètre x dans le type du résultat T. Ceci peut faire apparaître dans le type une expression quelconque là où il y avait avant une simple variable : c'est là une illustration de la difficulté qu'il y aurait à restreindre à certaines catégories syntaxiques la présence d'expressions dans les types.

$$\frac{\Gamma, x : T_0 \vdash E : T_1}{\Gamma \vdash \lambda x : T_0.\ E : \Pi x : T_0.\, T_1}\ (\mathcal{B}/\textbf{et.fun}) \qquad \frac{\Gamma \vdash E_1 : \Pi x : T_0.\, T \qquad \Gamma \vdash E_0 : T_0}{\Gamma \vdash E_1\ E_0 : \{x \leftarrow E_0\} T}\ (\mathcal{B}/\textbf{et.app})$$

Liaison locale Pour typer la liaison locale d'une variable à une valeur, nous demandons au programmeur de spécifier le type du résultat de l'expression toute entière. De plus, ce type résultat ne doit pas mentionner la variable liée localement. Le second point se comprend par le fait la variable x pourrait être liée dans le type de E, mais pas dans le type de $(\mathsf{let}\, x = E_0\, \mathsf{in}\, E : T)$ toute entière ; si E_0 crée des types abstraits, il n'y a donc aucun moyen de les référencer à l'extérieur de la liaison. La nécessité pour le programmeur de spécifier le type est due au problème de l'évitement que nous avons décrit à la section I.2.2.6, qui fait que l'inférence de T serait indécidable.

$$\frac{\Gamma \vdash E_0 : T_0 \qquad \Gamma, x : T_0 \vdash E : T \qquad \Gamma \vdash T\ \mathsf{ok}}{\Gamma \vdash (\mathsf{let}\, x = E_0\, \mathsf{in}\, E : T) : T}\ (\mathcal{B}/\textbf{et.let})$$

IV.2.4.5 $\boxed{\langle T \rangle,\ \mathsf{Typ}\, E}$ Champs types

Nous pouvons voir $\langle \_\_ \rangle$ comme un constructeur pour le type TYPE et $\mathsf{Typ}\, \_\_$ comme le destructeur correspondant, ce qui nous donne immédiatement les règles de typage idoines.

$$\frac{\Gamma \vdash T\ \mathsf{ok}}{\Gamma \vdash \langle T \rangle : \textsc{type}}\ (\mathcal{B}/\textbf{et.type}) \qquad \frac{\Gamma \vdash E : \textsc{type}}{\Gamma \vdash \mathsf{Typ}\, E\ \mathsf{ok}}\ (\mathcal{B}/\textbf{tok.field})$$

IV.2.5 Exécution

Valeurs La classe des **valeurs** (notées génériquement V) est une sous-classe des expressions définie par la grammaire suivante.

$V ::=$	**valeur**
$() \mid bv \mid \underline{n}$	constante
$\langle T \rangle$	champ type
(V_1, V_2)	paire
$\lambda x : T.\ E$	lambda-abstraction

IV.2.5.1 $\boxed{E \longrightarrow E'}$ Réduction des expressions

Nous définissons le comportement dynamique d'une expression par des règles de réduction à petits pas.

Règles de réduction en tête Dans le langage que nous avons défini jusqu'à présent, la réduction en tête confronte chacun des deux destructeurs produisant une expression au constructeur correspondant, et effectue les liaisons locales. Nous imposons dans les règle $(\mathcal{B}/\mathsf{ered.app})$ et $(\mathcal{B}/\mathsf{ered.let})$ la stratégie d'*appel par valeur*. Pour l'instant, nous pourrions autoriser la β réduction dans toute sa généralité, et le système obtenu serait confluant ; mais nous introduirons plus tard des effets de bord, ce qui suggère fortement de requérir l'appel par valeur.

$$(\lambda x : \mathsf{T}.\ \mathsf{E})\, \mathsf{V} \longrightarrow \{x \leftarrow \mathsf{V}\}\mathsf{E} \qquad (\mathcal{B}/\mathsf{ered.app})$$

$$\pi_i\,(\mathsf{V}_1, \mathsf{V}_2) \longrightarrow \mathsf{V}_i \qquad (\mathcal{B}/\mathsf{ered.proj})$$

$$\mathsf{let}\ x = \mathsf{V}\ \mathsf{in}\ \mathsf{E} : \mathsf{T} \longrightarrow \{x \leftarrow \mathsf{V}\}\mathsf{E} \qquad (\mathcal{B}/\mathsf{ered.let})$$

Pas de réduction dans les types Nous ne définissons pas de relation de réduction sur les types. De même, nous n'imposons pas de contrainte particulière pour que $\langle \mathsf{T} \rangle$ soit une valeur ; notamment, si $\langle \mathsf{T} \rangle$ contient des expressions imbriquées (par exemple $\langle \mathsf{Typ}\,((\lambda x : \textsc{type}.\ x)\,\langle \textsc{int} \rangle)\rangle$), celles-ci n'ont pas besoin d'être des valeurs. La raison à cela est que les calculs sur les types appartiennent traditionnellement au monde de la compilation, donc aux règles de typage (et le cas échéant aux algorithmes de typage) et non en général au monde de l'exécution que nous décrivons actuellement. Nous ajouterons plus loin une possibilité de typage dynamique, donc du calcul sur les types à l'exécution (section IV.6.1) ; la manipulation de types à l'exécution est également utile pour la programmation générique (voir la section V.3.2.3).

Contextes d'évaluation Nous notons génériquement C un **contexte d'évaluation** de profondeur 1. Les contextes d'évaluation sont définis par la grammaire suivante.

$\mathsf{C} ::=$	**contexte d'évaluation (de profondeur** 1**)**
$\quad \mathsf{E}_1\,\_\_$	argument de fonction
$\quad \_\_\,\mathsf{V}_2$	fonction appliquée
$\quad (\_\_, \mathsf{E}_2)$	première composante d'une paire
$\quad (\mathsf{V}_1, \_\_)$	seconde composante d'une paire
$\quad \pi_i\,\_\_$	projection ($i \in \{1, 2\}$)
$\quad \mathsf{let}\ x = \_\_\ \mathsf{in}\ \mathsf{E} : \mathsf{T}$	lié local

Nous avons fixé arbitrairement un ordre d'évaluation pour l'application de fonction (l'argument est évalué avant la fonction) et la paire (la première composante est évaluée avant la seconde). Ceci simplifie l'étude théorique en n'introduisant pas de non-déterminisme local inutile. Nous pourrions relâcher ces contraintes en autorisant les contextes $\_\_\,\mathsf{E}_2$ et $(\mathsf{E}_1, \_\_)$; il est classique que nous obtiendrions alors un système de réduction confluent.

La règle suivante permet l'évaluation à l'intérieur d'un contexte. La notation $\mathsf{C} \cdot \mathsf{E}$ désigne l'expression obtenue en plaçant E dans le contexte C.

$$\frac{\mathsf{E} \longrightarrow \mathsf{E}'}{\mathsf{C} \cdot \mathsf{E} \longrightarrow \mathsf{C} \cdot \mathsf{E}'}\ (\mathcal{B}/\mathsf{ered.context})$$

IV.3 Singletons $\boxed{\mathcal{S}}$

IV.3.1 Motivation

IV.3.1.1 Types abstraits, types concrets

Nous cherchons dans notre système de module à capturer la notion de type abstrait. La notion de type abstrait, type dont la nature exacte est cachée, s'oppose à celle de type concret, dont la nature est connue. Dans le système $\mathcal{B}$, les signatures ne font pas de différence entre les champs types des modules : ils ont tous le type TYPE. Nous avons exprimé à la section IV.2.1.2 les champs types concrets à l'aide de signatures singletons. Nous allons former le système $\mathcal{S}$ à partir du système $\mathcal{B}$ en lui ajoutant des **types singletons**.

Au niveau syntaxique, le système $\mathcal{S}$ a par rapport au système $\mathcal{B}$ une seule construction supplémentaire, une nouvelle manière de former un type.

T ::=		**type**
	$\dots$	
	$\mathsf{S}(\mathsf{E})$	singleton

Exemple Considérons le module $(\langle\textsc{int}\rangle, 3)$, qui contient un champ type et un champ valeur. Nous souhaitons pouvoir donner à ce module trois signatures :

- la signature $\Sigma\mathsf{x} : \textsc{type}.\ \textsc{int} = \textsc{type} * \textsc{int}$ décrit le champ type comme abstrait, et le champ valeur comme ayant un type sans rapport avec le champ type ;
- la signature $\Sigma\mathsf{x} : \textsc{type}.\ \mathsf{Typ}\,\mathsf{x}$ décrit le champ type comme abstrait, et le champ valeur comme une valeur du type abstrait en question ;
- la signature $\Sigma\mathsf{x} : \mathsf{S}(\langle\textsc{int}\rangle).\ \textsc{int} = \mathsf{S}(\langle\textsc{int}\rangle) * \textsc{int}$ décrit le champ type comme concret, et la valeur comme ayant ce même type ; la signature $\Sigma\mathsf{x} : \mathsf{S}(\langle\textsc{int}\rangle).\ \mathsf{Typ}\,\mathsf{x}$ serait aussi possible, mais peu intéressante car équivalente.

Nous disposons maintenant de deux manières de typer la première composante $\langle\textsc{int}\rangle$: elle possède le type $\mathsf{S}(\langle\textsc{int}\rangle)$ et le type $\textsc{type}$ (et nous verrons que le premier est un sous-type du second — en fait c'est par sous-typage que le second est obtenu). En utilisant le premier type, la règle de base de typage de la paire ($\mathcal{B}$/et.pair) donne désormais à $(\langle\textsc{int}\rangle, 3)$ le type $\mathsf{S}(\langle\textsc{int}\rangle) * \textsc{int}$. En utilisant le second type pour la première composante, nous obtenons la signature $\textsc{type} * \textsc{int}$ pour le module entier. L'obtention de la signature $\Sigma\mathsf{x} : \textsc{type}.\ \mathsf{Typ}\,\mathsf{x}$ est plus délicate : la connaissance de x limitée au type $\textsc{type}$ interdit de donner à 3 le type $\mathsf{Typ}\,\mathsf{x}$. Pour que $\mathsf{Typ}\,\mathsf{x}$ soit un type correct de la seconde composante, il est nécessaire de conserver dans un premier temps le type plus précis $\mathsf{S}(\langle\textsc{int}\rangle)$ pour la première composante, ce qui donne $\Sigma\mathsf{x} : \mathsf{S}(\langle\textsc{int}\rangle).\ \mathsf{Typ}\,\mathsf{x}$ comme signature pour la paire (une signature qui en elle-même n'apporte rien par rapport à $\mathsf{S}(\langle\textsc{int}\rangle) * \textsc{int}$). Ensuite nous pouvons utiliser un sous-typage, cette fois-ci au niveau de la paire : $\Sigma\mathsf{x} : \mathsf{S}(\langle\textsc{int}\rangle).\ \mathsf{Typ}\,\mathsf{x}$ est une sous-signature de $\Sigma\mathsf{x} : \textsc{type}.\ \mathsf{Typ}\,\mathsf{x}$.

Cas général Un module définissant un type et des fonctions de manipulation sur ce type et autres valeurs a souvent la forme $(\langle\mathsf{T}\rangle, \mathsf{E})$ où T est le type représentation et E rassemble les valeurs fournies par le module. La signature « naturelle » de ce module est $\mathsf{S}(\langle\mathsf{T}\rangle) * \mathsf{T}'$ où T' est le type « naturel » de E. Par sous-typage, l'on obtient des signatures dans lequel le type est abstrait ; elles ont la forme $\Sigma\mathsf{x} : \textsc{type}.\mathsf{T}''$ où $\{\mathsf{x}\leftarrow\langle\mathsf{T}\rangle\}\mathsf{T}'' = \mathsf{T}'$, le choix de la substitution reflétant les endroits où le type abstrait apparaît dans le type des valeurs fournies par le module.

Sous-typage La relation de **sous-typage** (ou **sous-signaturage**) exprime le fait que certaines signatures sont plus précises que d'autres. Nous noterons classiquement $\mathsf{T} <: \mathsf{T}'$ lorsque T est plus précis que T'. Le cas de base du sous-typage est le fait qu'une signature concrète est plus précise qu'une signature abstraite : $\mathsf{S}(\langle\mathsf{T}\rangle) <: \textsc{type}$. Le passage de cette relation entre signatures d'un champ à la signature d'une structure repose sur la règle de sous-typage des paires ($\mathcal{S}$/tsub.cong.pair) permettant de remplacer le type de chaque composante en parallèle par un sur-type — ainsi de $\mathsf{S}(\langle\textsc{int}\rangle) <: \textsc{type}$, combiné avec $\textsc{int} <: \mathsf{Typ}\,\mathsf{x}$ sous l'hypothèse $\mathsf{x} : \mathsf{S}(\langle\textsc{int}\rangle)$ (qui rend $\mathsf{Typ}\,\mathsf{x}$ équivalent à $\textsc{int}$), on déduit $\Sigma\mathsf{x} : \mathsf{S}(\langle\textsc{int}\rangle).\ \textsc{int} <: \Sigma\mathsf{x} : \textsc{type}.\ \mathsf{Typ}\,\mathsf{x}$.

Signatures principales Notons que grâce à la possibilité de spécifier exactement la première composante, toutes les signatures que nous avons mentionné pour le module $(\langle \text{INT} \rangle, 3)$ sont des sous-signatures de $\mathsf{S}(\langle \text{INT} \rangle) * \text{INT}$ (que nous qualifiions de signature « naturelle »). En fait, nous voyons là que les types singletons assurent aux modules l'existence de signatures principales, c'est-à-dire que tout module a une signature plus générale (plus petite) que toutes les autres[7]. En l'absence de foncteurs, cette signature principale est celle où tous les types sont concrets.

Remarquons que c'est l'aspect dépendant des paires qui fait apparaître le besoin du singleton pour disposer de signatures principales. En effet, dans le système $\mathcal{B}$, le champ $\langle \text{INT} \rangle$ a pour signature principale (et unique (à équivalence près)) TYPE. En revanche la paire $(\langle \text{INT} \rangle, 3)$ admet des signatures incomparables $\Sigma \mathsf{x} : \text{TYPE}. \text{INT}$ et $\Sigma \mathsf{x} : \text{TYPE}. \mathsf{Typ}\, \mathsf{x}$ à cause de la possibilité de dépendre du x inconnu.

IV.3.1.2 Partage de types

Motivation Notre exemple de module $(\langle \text{INT} \rangle, 3)$ fait apparaître un champ type particulièrement simple, puisqu'il s'agit d'un type prédéfini. Or il n'y a fondamentalement pas de différence entre un type prédéfini et un type fourni par une bibliothèque que l'on désigne par son nom : là où INT peut aller, nous voulons pouvoir utiliser $\mathsf{Typ}\, \pi_1\, \mathsf{x}_{\mathtt{BigInt}}$ où $\mathsf{x}_{\mathtt{BigInt}}$ est un module de bibliothèque implémentant des entiers arbitraires. La signature principale d'un module dont le premier champ est le type des entiers arbitraires a ainsi la forme $\mathsf{S}(\langle \mathsf{Typ}\, \pi_1\, \mathsf{x}_{\mathtt{BigInt}} \rangle) * \mathsf{T}_2$. Nous observons ici le besoin de singletons de la forme $\mathsf{S}(\langle \mathsf{T} \rangle)$ où T est une projection d'une variable — ce qui, au gré des réductions, peut conduire à syntaxiquement parlant n'importe quel terme T.

De manière générale, les signatures singletons permettent d'exprimer le **partage** de types, c'est-à-dire le fait que plusieurs modules peuvent contenir des champs types égaux. Ces égalités doivent être observables pour que certaines expressions puissent être utilisées indifféremment dans des contextes requérant l'une ou l'autre manière d'exprimer le type.

Exemple Un exemple typique de besoin d'égalités apparentes entre types est le problème de la cohérence des imports indirects[8]. Considérons un logiciel constitué de quatre composants `A`, `B`, `C` et `D` ; le module `A` fournit un type et des opérations sur ce type, les composants `B` et `C` étendent les opérations fournies sur ce type, et le programme principal `D` utilise à la fois `B` et `C`.

```
module A = struct  type t = ...  let f = ...       end
         : sig     type t        val f : int -> t  end
module B = struct  type t = A.t  let f = A.f       let g b = ...      end
         : sig     type t = A.t  val f : int -> t  val g : bool -> t  end
module C = struct  type t = A.t  let f x = x       let print = ...         end
         : sig     type t = A.t  val f : int -> t  val print : t -> unit  end
module D = struct  let h b = C.print (B.g b)  end
         : sig     val h : bool -> unit       end
```

Le module `D` utilise le module `A` à travers les modules `B` et `C` ; sa correction exige que la vérification de type tienne compte du fait que les types `B.t` et `C.t` sont tous les deux égaux à `A.t`. Autrement dit, il faut que `B` et `C` aient le même ancêtre `A` (et non deux ancêtres `A1` et `A2` ayant chacun une signature convenable). Dans le système $\mathcal{S}$, nous exprimons le fait que les champs types de `B` et `C` sont égaux à celui de `A` grâce à une signature singleton. Nous donnons ci-dessous le squelette du code correspondant à cet exemple dans $\mathcal{S}$, en indiquant pour chaque module une signature convenable

[7] Il peut y avoir plusieurs signatures principales en raison des équivalences entre signatures ($\mathsf{T}_1 <: \mathsf{T}_2$ et $\mathsf{T}_2 <: \mathsf{T}_1$ n'implique pas l'égalité syntaxique de T_1 et T_2) ; dans les exemples nous choisissons en principe la plus simple.

[8] *diamond import problem*, décrit notamment par D. MacQueen [Mac84].

(notre langage ne requiert pas d'annotation de signature à ce point — en pratique, il une annotation de signature est nécessaire si l'on souhaite compiler séparément les modules `A`, `B`, `C` et `D`).

$$\begin{array}{ll}
\mathsf{let}\ x_{\mathtt{A}} = (\langle\ldots\rangle, \ldots)\ \mathsf{in} & x_{\mathtt{A}} : \Sigma \mathsf{t} : \text{TYPE}.\ (\text{INT} \to \mathsf{Typ}\,\mathsf{t}) \\
\mathsf{let}\ x_{\mathtt{B}} = (\pi_1\, x_{\mathtt{A}}, \ldots)\ \mathsf{in} & x_{\mathtt{B}} : \Sigma \mathsf{t} : \mathsf{S}(\pi_1\, x_{\mathtt{A}}).\ (\text{INT} \to \mathsf{Typ}\,\mathsf{t}) * (\text{BOOL} \to \mathsf{Typ}\,\mathsf{t}) \\
\mathsf{let}\ x_{\mathtt{C}} = (\pi_1\, x_{\mathtt{A}}, \ldots)\ \mathsf{in} & x_{\mathtt{C}} : \Sigma \mathsf{t} : \mathsf{S}(\pi_1\, x_{\mathtt{A}}).\ (\text{INT} \to \mathsf{Typ}\,\mathsf{t}) * (\mathsf{Typ}\,\mathsf{t} \to \text{UNIT}) \\
\mathsf{let}\ x_{\mathtt{D}} = \ldots\ \mathsf{in}\ \ldots & x_{\mathtt{D}} : (\text{BOOL} \to \text{UNIT})
\end{array}$$

Foncteurs Le paradigme de la propagation de types est le foncteur application $f_{\mathtt{Apply}}$, qui prend comme arguments un foncteur g et un argument x et renvoie $g\,x$. Nous supposons ici fixés les signatures de g et x, soit $g : T_0 \to T_1$ et $x : T_0$. Le problème est d'exprimer la signature de $f_{\mathtt{Apply}}$ de manière à ne pas perdre d'information, afin que $f_{\mathtt{Apply}}\, g\, x$ soit interchangeable avec $g\,x$, et ce quel que soit le degré d'abstraction fourni par g. Dans un premier temps, supposons que le résultat de g soit un module contenant un champ type et un champ terme, bref T_1 est de la forme $\Sigma \mathsf{t} : \text{TYPE}.\ T_2$. Grâce aux singletons, la signature de $f_{\mathtt{Apply}}$ s'exprime simplement :

$$f_{\mathtt{Apply}} : \Pi g : T_0 \to T_1.\ \Pi x : T_0.\ \mathsf{S}(\pi_1\,(g\,x)) * \{\mathsf{t} \leftarrow \pi_1\,(g\,x)\} T_2$$

Le type de retour de $f_{\mathtt{Apply}}$ peut se décrire ainsi : « un paire formée d'un type, qui est celui de la première composante de $g\,x$, et d'une expression, dont le type est T_2 dans lequel on a remplacé t par sa valeur actuelle, à savoir $\pi_1\,(g\,x)$ ».

Plus simplement, le foncteur identité, qui prend comme argument un module comportant uniquement un type, et renvoie un module comportant le même type, peut s'écrire et s'utiliser ainsi :

```
module Id0 = functor (Arg : sig type t end) -> struct type t = Arg.t end
    (* Id0 : functor (Arg : sig type t end) -> sig type t = Arg.t end *)
module A0 = struct type t = int end
module B0 = Id0(A0)           (*A0.t et B0.t sont compatibles*)
```

Dans notre système, un tel foncteur identité peut s'écrire $\lambda x : \text{TYPE}.\ x$ et a la signature $\Pi x : \text{TYPE}.\ \mathsf{Typ}\, x$.

Isomorphismes explicites Notons qu'en l'absence de signatures singletons, on pourrait émuler le passage de types en fournissant des isomorphismes explicites entre types. Ainsi un foncteur identité s'écrirait

```
module IdIsom = functor (Arg : sig type t end) -> struct
  type t
  val i1 : Arg.t -> t
  val i2 : t -> Arg.t
end
```

Un tel style a deux défauts majeurs. L'un est sa lourdeur, d'autant plus importante que les chaînes de types partagés sont longues — il faut soit multiplier les isomorphismes en progression quadratique, soit composer explicitement les isomorphismes successifs. L'autre défaut est que rien n'indique plus au niveau du typage que `i1` et `i2` sont des isomorphismes réciproques, d'où une perte considérable d'expressivité.

IV.3.1.3 Singletons de valeurs

Nous avons jusqu'à présent utilisé des types singletons pour exprimer des égalités de types ; c'est-à-dire que les singletons ont la forme $\mathsf{S}(\langle\mathsf{T}\rangle)$ pour un certain T. La nécessité de ces singletons émane du besoin que x ait le type T' lorsque x a le type T et $\langle\mathsf{T}\rangle$ et $\langle\mathsf{T}'\rangle$ sont équivalents ; en d'autres termes le jugement $\mathsf{t} : \mathsf{S}(\langle\mathsf{T}\rangle), x : \mathsf{T} \vdash x : \mathsf{Typ}\,\mathsf{t}$ doit être valide ($\langle\mathsf{T}'\rangle$ pouvant alors être substitué à t).

Or considérons un foncteur f qui crée un type abstrait à partir d'un type et d'une valeur, plus précisément ayant une signature de la forme $\Pi x : (\Sigma\mathsf{t} : \textsc{type}.\, \mathsf{T}_0).\, (\Sigma\mathsf{t} : \textsc{type}.\, \mathsf{T}_1)$. Conformément à ce que nous avons vu aux sections I.2.1.3 et II.5.1.1, $\mathsf{Typ}\,\pi_1\,(f\,x)$ et $\mathsf{Typ}\,\pi_1\,(f\,y)$ ne sont les mêmes types que lorsque x et y ont le même comportement — il ne suffit pas que x et y fournissent les mêmes types.

Considérons un exemple d'argument potentiel pour f (nous prenons ici $\mathsf{T}_0 = \mathsf{Typ}\,\mathsf{t} * (\mathsf{Typ}\,\mathsf{t} \to \textsc{unit})$).

```
module A = struct  type t = int  let x = (... : t * (t->unit))  end
```

La signature principale du module `A` en Objective Caml est

```
module type S = sig  type t = int  val x : t * (t->unit)  end
```

Si nous souhaitons exprimer qu'un module `B` est compatible avec le module `A`, le mieux que nous puissions faire (que ce soit en Objective Caml ou dans un autre dialecte de ML, ou dans le language que nous avons défini jusqu'à présent) est de spécifier que `B` a la signature `S`. Or cette spécification est incomplète : elle ne distingue pas deux modules qui ont des champs `x` de même type mais des valeurs différentes.

Une manie d'exhiber cette limitation est de considérer un foncteur identité `Id1` capable de prendre `A` comme argument. La signature principale d'un tel foncteur en Objective Caml (qui est basé sur la théorie des types manifestes [Ler94] avec foncteurs applicatifs [Ler95] de X. Leroy) est la suivante :

```
functor (A : sig  type t        val x : t * (t->unit)  end) ->
             sig  type t = A.t  val x : t * (t->unit)  end
```

On constate que la signature de `Id1(A)` est la signature `S` définie ci-dessus ; elle indique bien que `Id1(A).t` est égal à `A.t` mais reste muette sur le fait que `Id1(A).x` est égal à `A.x`. La théorie des modules d'ordre supérieur de D. Dreyer, K. Crary et R. Harper [DCH03] a ici le même pouvoir d'expression qu'Objective Caml. Avec nos notations, la signature du foncteur `Id1` est $\Pi x : (\Sigma\mathsf{t} : \textsc{type}.\, \mathsf{Typ}\,\mathsf{t} * (\mathsf{Typ}\,\mathsf{t} \to \textsc{unit})).\, (\Sigma\mathsf{t}' : \mathsf{S}(\pi_1 x).\, \mathsf{Typ}\,\mathsf{t}' * (\mathsf{Typ}\,\mathsf{t}' \to \textsc{unit}))$.

Dans le cas d'un foncteur identité $\mathsf{E}_{\mathtt{Id0}}$ agissant uniquement sur un type, c'est-à-dire prenant un argument de signature $\textsc{type}$, la signature obtenue est plus précise : $\Pi x : \textsc{type}.\, \mathsf{S}(x)$ indique clairement que l'application de ce foncteur identité à un argument est interchangeable avec l'argument lui-même — $\mathsf{Typ}\,\mathsf{E}_{\mathtt{Id0}}\,(\mathsf{E}_{\mathtt{A0}})$ a la signature $\mathsf{S}(\mathsf{E}_{\mathtt{Id0}}\,(\mathsf{E}_{\mathtt{A0}}))$. Nous allons étendre notre langage pour que cette propriété soit vérifiée également pour les champs valeurs.

Nous munissons notre langage de types singletons de la forme $\mathsf{S}(\mathsf{E})$ où E est une expression. Par exemple, le type principal de l'expression 3 est désormais $\mathsf{S}(3)$, type des valeurs égales à 3. Nous pouvons donner au foncteur `Id1` une signature plus précise, en donnant à la deuxième composante un type singleton : $\Pi x : (\Sigma\mathsf{t} : \textsc{type}.\, \mathsf{Typ}\,\mathsf{t} * (\mathsf{Typ}\,\mathsf{t} \to \textsc{unit})).\, (\Sigma\mathsf{t}' : \mathsf{S}(\mathsf{t}).\, \mathsf{S}(\pi_1 x) * \mathsf{S}(\pi_2 x))$, soit dans une notation basée sur celle d'Objective Caml

```
functor (A : sig  type t        val x : t * (t->unit)  end) ->
             sig  type t = A.t  val x = A.x            end
```

Grâce à cette signature, nous pouvons attribuer à `Id1(A)` la signature `sig type t = A.t val x = A.x end`, et tout module ayant cette signature est utilisable de manière interchangeable avec `A`.

IV.3.1.4 Singletons d'ordre supérieur

Nous avons ci-dessus donné un exemple de singleton de valeur du noyau. Mais puisque nous n'avons pas fait la différence entre langage du noyau et langage des modules, nous pouvons en fait écrire un singleton d'un module arbitraire : tout module `E` a la signature `S(E)`. Notre langage dispose ainsi de singletons d'ordre supérieur ; en particulier, de singletons de foncteurs.

Nous insistons néanmoins sur le fait que le point clé est la possibilité de manipuler des singletons de valeurs du noyau. Nous pourrions en fait nous dispenser de définir des singletons pour autre chose que des valeurs de base telles que () ou 3 et des champs types ⟨`T`⟩. Il y a deux avantages à définir des singletons à des signatures arbitraires. L'un est que l'on peut alors écrire directement `S(E)` sans devoir examiner la signature de `E` — sinon nous serions amené à définir `S(E)` comme une forme dérivée basée sur la signature de `E` (comme le font D. Dreyer, K. Crary et R. Harper [DCH03] — notons qu'ils ne disposent pas de signatures de valeurs de base, et leurs signatures singletons dérivées ignorent les champs valeurs des modules). Un autre avantage est lié à notre choix d'unifier les strates du langage : il est plus naturel d'écrire systématiquement `S(E)` lorsque `E` est une expression du noyau, et l'ordre supérieur en découle.

IV.3.1.5 Un exemple pratique

Illustrons l'usage des singletons d'ordre supérieur sur un exemple tiré de l'expérience de l'auteur. La bibliothèque standard d'Objective Caml fournit une implémentation d'ensembles finis sous la forme d'un foncteur appelé `Set.Make`. Ce foncteur prend un argument ayant la signature `Set.OrderedType` définie ainsi :

```
module type Set.OrderedType = sig
  type t
  val compare : t -> t -> int
end
```

Un module de signature `Set.OrderedType` fournit un type ainsi qu'une fonction de comparaison qui doit définir un ordre total (un ensemble est représenté par un arbre de recherche). Un exemple de module ayant la signature `Set.OrderedType` est le module `String` des chaînes de caractères ; `Set.Make(String).t` est donc un type d'ensembles de chaînes de caractères. Quant au résultat retourné par `Set.Make`, il a une signature appelée `Set.S` dont nous donnons un extrait :

```
module type Set.S = sig
  type elt                    (*type des éléments*)
  type t                      (*type des ensembles*)
  val add : elt -> t -> t     (*fonction d'ajout*)
  ...
end
```

Une annotation dans la définition de `Set.Make` précise que `Set.Make(M).elt = M.t`.

Le programme auquel nous nous intéressons manipule des symboles, qui sont en fait des chaînes de caractères. Cependant seule une chaîne de caractères « approuvée » peut être un symbole ; le type des symboles est donc un type abstrait fourni par un module, appelé `Syntaxe`.

```
module Syntaxe : sig
  type symbole
  val nom : symbole -> String.t
  ...
end
```

Comme plusieurs autres modules du programme manipulent des ensembles de symboles, nous souhaitons fournir également ce type. Mais comment exprimer cela dans la signature du module `Syntaxe` ? Il faut spécifier un module ayant la signature `Set.S`, en indiquant que le type des éléments de l'ensemble est le type des symboles. Ce dernier point oblige la définition d'un module des symboles.

```
module Syntaxe : sig
  module Symbole : Set.OrderedType
  module Ensemble : Set.S with type elt = Symbole.t
                          and type t = Set.Make(Symbole).t
  ...
end
```

On pourrait également écrire ceci :

```
module Syntaxe : sig
  module Symbole : Set.OrderedType
  module Ensemble : Set.S
  ...
end with module Ensemble = Set.Make(Symbole)
```

Les deux signatures ci-dessus sont équivalentes en Objective Caml.

Comment écrire le module `Syntaxe` ? C'est là que le bât blesse. En effet, on peut écrire

```
module Syntaxe = struct
  module Symbole = struct type t = String.t let compare = String.compare end
  module Ensemble = Set.Make(Symbole)
  ...
end
```

ou même carrément

```
module Syntaxe = struct
  module Symbole = String
  module Ensemble = Set.Make(Symbole)
  ...
end
```

Malheureusement, le module `Ensemble` ainsi défini n'est pas compatible avec `Set.Make(String)`. En effet, puisque Objective Caml ne compare jamais que des champs types de modules, son analyse de types retient que `Symbole.t` = `String.t` mais pas que `Symbole` = `String`, et il n'est donc pas sûr de rendre compatibles `Set.Make(String).t` et `Set.Make(Symbole).t`.

Comme le module `Syntaxe` fait appel à des modules de plus bas niveau manipulant des ensembles de chaînes de caractères, l'incompatibilité de `Set.Make(String).t` et `Set.Make(Symbole).t` est un problème majeur. La solution retenue fut de n'exposer que le type des symboles, et non sa fonction de comparaison :

```
module Syntaxe = struct
  type symbole = String.t
  module Ensemble = Set.Make(String)
```

```
    ...
  end : sig
    type symbole
    module Ensemble : Set.S with type t = symbole
    ...
  end
```

L'inconvénient de cette signature est qu'elle cache le choix d'implémentation des ensembles : le fait que `Ensemble` est le résultat d'une application du foncteur `Set.Make` n'apparaît pas. Or parmi les modules utilisant le module `Syntaxe`, certains manipulent des structures de données telles que des ensembles d'ensembles basées sur l'implémentation `Set.Make`. Il fallut donc fournir dans le module `Syntaxe` ces structures de données composées, alors que celles-ci n'étaient pas du tout utilisées dans le module `Syntaxe` et n'auraient pas dû y être mentionnées d'un point de vue d'organisation du code.

Dans le cas que nous venons d'étudier, la simple possibilité de poser dans le module `Syntaxe` la définition `module Symbole = String` de manière à ce que les types `Set.Make(String).t` et `Set.Make(Symbole).t` soient interchangeables aurait permis l'organisation optimale du code, notamment vis-à-vis de l'abstraction. Dans le système $\mathcal{S}$, la déclaration de `Symbole` dans la définition du module `Syntaxe` a naturellement la signature $\mathsf{S}(x_{\mathtt{String}})$ qui rend `Set.Make(Symbole)` compatible avec `Set.Make(String)`.

IV.3.2 Propriétés

IV.3.2.1 Nature syntaxique

Dans la section précédente, nous avons défini des types (ou signatures) singletons de la forme $\mathsf{S}(\mathsf{E})$, E étant une expression dont le type peut être quelconque. Nous allons maintenant nous pencher sur un problème orthogonal : y a-t-il lieu de restreindre les types singletons à certaines formes syntaxiques d'expressions ?

S'agissant de champs types, nous avons vu l'utilité de singletons de valeurs $\mathsf{S}(\langle \mathsf{T} \rangle)$, mais aussi de variables $\mathsf{S}(x)$. Les premiers expriment des types concrets, tandis que les seconds expriment des contraintes de partage. En fait, ces dernières peuvent avoir une forme plus générale : il est fréquent de référer à une composante particulière d'un module, ce qui donne par exemple $\mathsf{S}(\pi_1 \pi_2 x)$ pour exprimer l'égalité avec la deuxième composante du module désigné par x (en ML on écrirait `type t = X.u` où `u` est le deuxième champ de `X`). De manière générale, une contrainte de partage s'exprime par un singleton d'une projection d'une variable, le mot « projection » incluant ici non seulement les projections de paires mais aussi les applications de foncteurs telles que $\mathsf{S}(\pi_1(\mathsf{f}(\langle \textsc{int} \rangle, 3)))$.

Or au cours de l'exécution, les variables sont substituées tour à tour par des valeurs. Il faut donc pouvoir former des singletons de projections de valeurs. Ceci peut donner lieu à des appels de fonction. Aussi ne peut-on pas imposer de contrainte syntaxique sur l'expression E pour former $\mathsf{S}(\mathsf{E})$. La classe des expressions E telles que $\mathsf{S}(\mathsf{E})$ soit bien formé contient au moins les variables, et est close par application de constructeur (pour les valeurs) ou de projection (destructeurs de paires et de fonctions).

IV.3.2.2 Sémantique de l'appartenance au singleton

L'interprétation intuitive du type $\mathsf{S}(\mathsf{E})$ est que c'est un type qui ne contient que la valeur E. Mais en fait rien ne contraint E à être une valeur, et le type $\mathsf{S}(\mathsf{E})$ contient également les expressions interconvertibles avec E. Plus précisément, si E a le type $\mathsf{S}(\mathsf{E}_0)$ et que E se réduit en E', alors E'

doit aussi avoir le type $\mathsf{S(E_0)}$ pour la préservation du typage. En particulier, si E a le type $\mathsf{S(E)}$ et que E se réduit vers une valeur V alors V a aussi le type $\mathsf{S(E)}$.

Considérons la relation sur les expressions définie par $\mathsf{E_1} : \mathsf{S(E_2)}$ ($\mathsf{E_1}$ a le type $\mathsf{S(E_2)}$). Nous appellerons cette relation l'**appartenance au singleton**. La préservation du typage demande que cette relation soit transitive. Il est de plus raisonnable de demander que E ait le type $\mathsf{S(E)}$ dès que celui-ci est bien formé (en revanche, il se peut que $\mathsf{S(E)}$ ne soit pas bien formé même si E est bien typée). L'appartenance au singleton est donc un préordre sur la classe des expressions dont le singleton est bien formé. Cette relation contient la relation de réduction (si E se réduit en $\mathsf{E'}$ alors $\mathsf{E} : \mathsf{S(E')}$).

Il n'y a en revanche pas de raison essentielle pour laquelle la réciproque devrait être vraie : si E se réduit en $\mathsf{E'}$, $\mathsf{E'}$ peut avoir un type singleton sans que E ne l'ait également. En particulier, si E se réduit vers une valeur V, on a $\mathsf{V} : \mathsf{S(V)}$, mais cela n'impose pas $\mathsf{E} : \mathsf{S(V)}$ ni $\mathsf{E} : \mathsf{S(E)}$.

Si nous posons comme intuition qu'un type $\mathsf{S(E)}$ caractérise les expressions qui ont la même valeur que E, alors la classe des expressions de type $\mathsf{S(E)}$ est la classe des expressions interconvertibles avec E (à condition du moins que E ait une valeur). Dans cette interprétation, la relation définie par $\mathsf{E_1} : \mathsf{S(E_2)}$ est symétrique, c'est la relation d'équivalence partielle d'interconvertibilité (de domaine contenant au moins les expressions normalisantes).

En fait, le type $\mathsf{S(E)}$ caractérise plutôt les expressions ayant le même *comportement* que E, ou plus précisément une expression de type $\mathsf{S(E)}$ a au plus les comportements de E. Cette nuance peut jouer dans les deux sens.

- D'une part, elle autorise deux singletons $\mathsf{S(V)}$ et $\mathsf{S(V')}$ a être équivalents (au sens ou toute expression ayant l'un des types a aussi l'autre) même si V et $\mathsf{V'}$ sont des valeurs différentes, à condition que V et $\mathsf{V'}$ soient « suffisamment » interchangeables. Un cas plausible est celui où V et $\mathsf{V'}$ sont deux fonctions extensionnellement égales. Cet aspect engendre des cas d'équivalence dans l'appartenance au singleton, qui n'est alors pas une relation d'ordre.
- D'autre part, dans un langage muni de non terminaison, d'effets de bord ou de non-déterminisme, le comportement d'une expression n'est pas déterminé uniquement par son éventuelle valeur. Cet aspect peut introduire des dissymétries dans l'appartenance au singleton.

Dans le présent chapitre, nous nous limiterons à attribuer des types singletons à des expressions fortement normalisantes, des raffinements supplémentaires nous paraissant superflus. L'appartenance au singleton est alors une relation d'équivalence. Cette équivalence est contenue dans l'équivalence observationnelle des expressions, et contient l'interconvertibilité à l'exécution. Comme l'équivalence observationnelle n'est pas décidable, nous serons naturellement amené à en choisir une approximation.

IV.3.2.3 Correction d'un singleton

Si $\mathsf{S(E)}$ est bien formé, c'est le type le plus précis possible de E. Nous avons vu que l'on ne peut pas restreindre syntaxiquement les expressions E telles que $\mathsf{S(E)}$ soit bien formé. En revanche, il peut exister des conditions nécessaires sémantiques.

Dans le système $\mathcal{S}$, nous autorisons toujours la formation de $\mathsf{S(E)}$ (exigeant seulement, bien sûr, que E soit bien typée). En revanche, nous ajouterons dans le système $\mathcal{E}$ des effets de bord ; seule une expression **pure**, c'est-à-dire sans effet de bord, et fortement normalisante, admettra un type singleton. En effet, le jugement $\mathsf{E} : \mathsf{S(E)}$ signifie intuitivement que E est complètement connue statiquement ; ceci ne peut être le cas si E peut avoir un effet de bord ; et admettre la formation de $\mathsf{S(E)}$ lorsque E peut ne pas terminer conduirait à rendre le typage statique indécidable.

De manière générale, la présence d'un singleton dans le type d'une expression signifie que cet aspect de l'expression est entièrement connu statiquement. Si c'est toujours possible dans le système

$\mathcal{S}$, ce ne le sera plus dans les systèmes ultérieurs.

Un système de modules fournissant essentiellement les équivalences décrites ici a été proposé par J. Courant [Cou97a, Cou98]. En l'état, ce système n'est pas applicable à un langage de programmation dans lequel un module peut avoir des effets de bords lorsqu'il est évalué, ce qui le rend en quelque sorte non équivalent à lui-même. Il est en revanche à la base du système de modules de Coq [Cou97b, Coq].

IV.3.3 Règles de typage

Nous donnons ici les règles de typage pour le système $\mathcal{S}$. La sémantique opérationnelle (règles de réduction (($\mathcal{S}$/ered.app), ($\mathcal{S}$/ered.proj), ($\mathcal{S}$/ered.let), ($\mathcal{S}$/ered.context)), contextes de réduction, valeurs) est inchangée par rapport à $\mathcal{B}$.

Le système $\mathcal{S}$ comprend de nouvelles forme de jugement de typage, pour le sous-typage et les équivalences par conversion.

J ::=	**membre droit de jugement local**
...	
$\mathsf{T} \longrightarrow \mathsf{T}'$	conversion des types
$\mathsf{T} \equiv \mathsf{T}'$	équivalence par conversion des types
$\mathsf{E} \longrightarrow \mathsf{E}'$	conversion des expressions
$\mathsf{E} \equiv \mathsf{E}'$	équivalence par conversion des expressions
$\mathsf{T}_1 <: \mathsf{T}_2$	sous-typage

Le système de types de $\mathcal{S}$ inclut la plupart des règles de $\mathcal{B}$.

Les règles suivantes sont reprises telles quelles du système $\mathcal{B}$:
- correction de l'environnement : toutes — ($\mathcal{S}$/envok.nil), ($\mathcal{S}$/envok.x) ;
- correction des types : toutes — ($\mathcal{S}$/tok.base.unit), ($\mathcal{S}$/tok.base.bool), ($\mathcal{S}$/tok.base.int), ($\mathcal{S}$/tok.type), ($\mathcal{S}$/tok.pair), ($\mathcal{S}$/tok.fun), ($\mathcal{S}$/tok.field) ;
- typage des expressions : toutes sauf les projections et la liaison locale — ($\mathcal{S}$/et.base.unit), ($\mathcal{S}$/et.base.bool), ($\mathcal{S}$/et.base.int), ($\mathcal{S}$/et.x), ($\mathcal{S}$/et.pair), ($\mathcal{S}$/et.fun), ($\mathcal{S}$/et.app), ($\mathcal{S}$/et.type).

Nous omettons la liaison locale du système $\mathcal{S}$ parce qu'elle y est superflue (voir la section IV.2.1.4) et nous ne voulons pas devoir la gérer dans les singletons.

IV.3.3.1 $\boxed{\Gamma \vdash \mathsf{T} <: \mathsf{T}' \;;\; \ldots}$ Sous-typage

La relation de sous-typage sert à exprimer le fait qu'une expression peut avoir plusieurs types, dont certains sont plus précis que d'autres. Intuitivement, T est un sous-type de T' lorsque toute expression ayant le type T a aussi le type T'. Nous érigeons en règle l'implication dans un sens : si T est un sous-type de T' alors toute expression de type T a aussi le type T'. Notre système de typage admet ainsi un *sous-typage implicite*.

$$\frac{\Gamma \vdash \mathsf{E} : \mathsf{T} \qquad \Gamma \vdash \mathsf{T} <: \mathsf{T}'}{\Gamma \vdash \mathsf{E} : \mathsf{T}'} \;(\mathcal{S}/\mathbf{et.sub})$$

Notre relation de sous-typage est definie syntaxiquement (par des règles que nous présenterons au fur et à mesure), et non sémantiquement, en ce que nous ne demandons pas l'implication réciproque : il se peut que le jugement $\Gamma \vdash \mathsf{E} : \mathsf{T}'$ soit dérivable dès lors que $\Gamma \vdash \mathsf{E} : \mathsf{T}$ l'est, sans pour autant que le jugement $\Gamma \vdash \mathsf{T} <: \mathsf{T}'$ ne soit dérivable. Le caractère sémantique d'une relation de sous-typage est à double tranchant. D'un côté, il permettrait une interprétation ensembliste des types comme ensemble des expressions de ce type. D'un autre côté, il obligerait à ajouter des règles quelque peu fragiles, en ce sens qu'elles devraient être retirées en cas d'extension du système. En particulier, si V est une valeur de type T, avec un sous-typage sémantique, l'on devrait avoir $\mathsf{T} <: \mathsf{S}(\mathsf{V})$ si et seulement si T ne contient qu'une valeur V, ce qui peut être le cas par coïncidence. Considérons par exemple le type $\Pi t : \text{TYPE}.\ \mathsf{Typ}\, t \to \mathsf{Typ}\, t$: il contient évidemment l'identité polymorphe ($\lambda t{:}\text{TYPE}.\ \lambda x{:}\mathsf{Typ}\, t.\ x$) et, même si le fragment du langage que nous avons présenté pour l'instant ne contient pas valeur de ce type ayant un comportement différent (en vertu d'un résultat de paramétricité [Wad89] que nous ne chercherons pas à démontrer dans notre cadre), l'on pourrait en fabriquer à l'aide d'un combinateur de point fixe, ou du typage dynamique tel que défini à la section IV.6.

Si deux types sont interconvertibles, ils sont sous-types l'un de l'autre. Le sous-typage inclut donc l'équivalence calculatoire sur les types.

$$\frac{\Gamma \vdash \mathsf{T} \equiv \mathsf{T}'}{\Gamma \vdash \mathsf{T} <: \mathsf{T}'}\ (\mathcal{S}/\textbf{tsub.eq})$$

Le sous-typage est un préordre. La réflexivité est assurée par ($\mathcal{S}$/tsub.eq) ; nous énonçons la transitivité.

$$\frac{\Gamma \vdash \mathsf{T} <: \mathsf{T}' \qquad \Gamma \vdash \mathsf{T}' <: \mathsf{T}''}{\Gamma \vdash \mathsf{T} <: \mathsf{T}''}\ (\mathcal{S}/\textbf{tsub.trans})$$

IV.3.3.2 $\boxed{\mathsf{S}(\mathsf{E})}$ Singletons

Les types singletons sont introduits dans le système par trois règles génériques, c'est-à-dire sans contrainte quant au type de l'expression dont on considère le type singleton. Dès lors qu'une expression E a un type T, le type singleton $\mathsf{S}(\mathsf{E})$ est bien formé ; l'expression E a le type $\mathsf{S}(\mathsf{E})$; et le type $\mathsf{S}(\mathsf{E})$ est un sous-type de T. On note en particulier que pour donner à une expression son type singleton, l'on commence par lui prouver un type plus large, puis l'on applique la règle ($\mathcal{S}$/et.sing).

$$\frac{\Gamma \vdash \mathsf{E} : \mathsf{T}}{\Gamma \vdash \mathsf{S}(\mathsf{E})\ \mathsf{ok}}\ (\mathcal{S}/\textbf{tok.sing}) \qquad \frac{\Gamma \vdash \mathsf{E} : \mathsf{T}}{\Gamma \vdash \mathsf{E} : \mathsf{S}(\mathsf{E})}\ (\mathcal{S}/\textbf{et.sing}) \qquad \frac{\Gamma \vdash \mathsf{E} : \mathsf{T}}{\Gamma \vdash \mathsf{S}(\mathsf{E}) <: \mathsf{T}}\ (\mathcal{S}/\textbf{tsub.sing})$$

La règle ($\mathcal{S}$/et.sing) (jointe aux autres règles de sous-typage) est une instance particulièrement puissante de règle d' (*selfification*) dans un système de modules avec types manifestes [HL94, Ler94] (voir la section I.2.2.2).

IV.3.3.3 $\boxed{\Gamma \vdash \mathsf{E} : \mathsf{T} \ ;\ \Gamma \vdash \mathsf{T}_1 <: \mathsf{T}_2}$ Typage des expressions

Comme dans le système $\mathcal{B}$, nous donnons au départ aux paires un type non dépendant. Les types sommes dépendants s'obtiennent via la règle de sous-typage ($\mathcal{S}$/tsub.cong.pair) (rappelons que $\mathsf{T}_1 * \mathsf{T}_2$ est une notation abrégée pour $\Sigma x : \mathsf{T}_1.\ \mathsf{T}_2$). Dans cette dernière, on note que la deuxième prémisse contient l'hypothèse la plus forte sur x, à savoir $x : \mathsf{T}_1'$, ce qui est logique puisque l'hypothèse $x : \mathsf{T}_2'$ ne suffit pas en général à ce assurer que T_1'' soit correct. Une troisième prémisse est nécessaire pour assurer que $\Sigma x : \mathsf{T}_2'.\ \mathsf{T}_2''$ est bien formé, ce qui nécessite que T_2' soit bien formé sous l'hypothèse plus faible $x : \mathsf{T}_2'$.

$$\frac{\Gamma \vdash \mathsf{T}_1 <: \mathsf{T}_1' \qquad \Gamma, x : \mathsf{T}_1 \vdash \mathsf{T}_2 <: \mathsf{T}_2' \qquad \Gamma, x : \mathsf{T}_1' \vdash \mathsf{T}_2'\ \mathsf{ok}}{\Gamma \vdash \Sigma x : \mathsf{T}_1.\ \mathsf{T}_2 <: \Sigma x : \mathsf{T}_1'.\ \mathsf{T}_2'}\ (\mathcal{S}/\textbf{tsub.cong.pair})$$

Pour typer une projection, nous attribuons à l'expression argument un type de paire, à savoir de manière générale une somme dépendante. Le typage de la première projection est simple : elle a le premier type de la paire. Le typage de la seconde projection est plus compliqué. Si l'expression E a le type $\Sigma x : \mathsf{T}_1.\mathsf{T}_2$, alors $\pi_2\,\mathsf{E}$ n'a le type T_2 qu'à condition de disposer d'une hypothèse suffisamment forte sur la variable x. Par exemple, l'expression $(3,3)$ a le type $\Sigma x : \text{INT}.\ \mathsf{S}(x)$; on a bien $x : \mathsf{S}(3) \vdash \pi_2\,(3,3) : \mathsf{S}(x)$ mais pas $x : \text{INT} \vdash \pi_2\,(3,3) : \mathsf{S}(x)$.

$$\frac{\Gamma \vdash \mathsf{E} : \Sigma x : \mathsf{T}_1.\ \mathsf{T}_2}{\Gamma \vdash \pi_1\,\mathsf{E} : \mathsf{T}_1}\ (\mathcal{S}/\textbf{et.proj.1}) \qquad \frac{\Gamma \vdash \mathsf{E} : \Sigma x : \mathsf{T}_1.\ \mathsf{T}_2 \qquad \Gamma \vdash \mathsf{E}_1 : \mathsf{S}(\pi_1\,\mathsf{E})}{\Gamma \vdash \pi_2\,\mathsf{E} : \{x \leftarrow \mathsf{E}_1\}\mathsf{T}_2}\ (\mathcal{S}/\textbf{et.proj.2})$$

Les hypothèses de la règle ($\mathcal{S}$/et.proj.2) sont la plupart du temps indûment contraignantes, et nous utiliserons une variante admissible qui ne demande que l'hypothèse $\Gamma \vdash E : \Sigma x : T_1.\, T_2$. La règle d'emploi la plus courante consiste à remplacer la variable x dans T_2 par la première projection de E. Une autre version conserve la première composante sous la forme d'une variable x qui est contrainte au type $S(\pi_1 E)$.

$$\frac{\Gamma \vdash E : \Sigma x : T_1.\, T_2}{\Gamma \vdash \pi_2 E : \{x \leftarrow_c \pi_1 E\} T_2}\ (\text{et.proj.2s}) \qquad \frac{\Gamma \vdash E : \Sigma x : T_1.\, T_2}{\Gamma, x : S(\pi_1 E) \vdash \pi_2 E : T_2}\ (\text{et.proj.2x})$$

Nous énonçons pour les produits dépendants la règle classique de congruence pour le sous-typage. Cette règle est similaire à ($\mathcal{S}$/tsub.cong.pair), mais le type de l'argument est contravariant.

$$\frac{\Gamma \vdash T_0' <: T_0 \qquad \Gamma, x : T_0' \vdash T_1 <: T_1' \qquad \Gamma, x : T_0 \vdash T_1 \text{ ok}}{\Gamma \vdash \Pi x : T_0.\, T_1 <: \Pi x : T_0'.\, T_1'}\ (\mathcal{S}/\textbf{tsub.cong.fun})$$

IV.3.3.4 $\boxed{\Gamma \vdash T \equiv T' \text{ ; } \Gamma \vdash E \equiv E'}$ Équivalences de convertibilité

Nous définissons une relation d'équivalence sur les types et une sur les expressions[9]. Il s'agit de relations de convertibilité : nous les définissons chacune comme la plus petite relation d'équivalence contenant la relation de conversion correspondante.

$$\frac{\Gamma \vdash T \text{ ok}}{\Gamma \vdash T \equiv T}\ (\mathcal{S}/\textbf{teq.refl}) \qquad \frac{\Gamma \vdash T_2 \equiv T_1}{\Gamma \vdash T_1 \equiv T_2}\ (\mathcal{S}/\textbf{teq.sym})$$

$$\frac{\Gamma \vdash T_1 \equiv T_2 \qquad \Gamma \vdash T_2 \equiv T_3}{\Gamma \vdash T_1 \equiv T_3}\ (\mathcal{S}/\textbf{teq.trans}) \qquad \frac{\Gamma \vdash T_1 \longrightarrow T_2}{\Gamma \vdash T_1 \equiv T_2}\ (\mathcal{S}/\textbf{teq.conv})$$

$$\frac{\Gamma \vdash E : T}{\Gamma \vdash E \equiv E}\ (\mathcal{S}/\textbf{eeq.refl}) \qquad \frac{\Gamma \vdash E_2 \equiv E_1}{\Gamma \vdash E_1 \equiv E_2}\ (\mathcal{S}/\textbf{eeq.sym})$$

$$\frac{\Gamma \vdash E_1 \equiv E_2 \qquad \Gamma \vdash E_2 \equiv E_3}{\Gamma \vdash E_1 \equiv E_3}\ (\mathcal{S}/\textbf{eeq.trans}) \qquad \frac{\Gamma \vdash E_1 \longrightarrow E_2}{\Gamma \vdash E_1 \equiv E_2}\ (\mathcal{S}/\textbf{eeq.conv})$$

IV.3.3.5 $\boxed{\Gamma \vdash T \longrightarrow T'}$ Conversion des types

La **conversion des types** consiste essentiellement en la conversion des expressions imbriquées ; nous effectuons également quelques simplifications.

La conversion n'est définie que sur les types bien formés ; de nombreuses règles comportent des prémisses à cet effet, en plus des prémisses de conversion dans les règles contextuelles.

Contextes Au niveau des types, la conversion s'applique récursivement à tous les sous-termes. Nous donnons donc des règles contextuelles, permettant de convertir les parties de types composés ainsi que les expressions imbriquées.

$$\frac{\Gamma \vdash T_1 \longrightarrow T_1' \qquad \Gamma, x : T_1 \vdash T_2 \text{ ok}}{\Gamma \vdash \Sigma x : T_1.\, T_2 \longrightarrow \Sigma x : T_1'.\, T_2}\ (\mathcal{S}/\textbf{tconv.cong.pair.1})$$

$$\frac{\Gamma, x : T_1 \vdash T_2 \longrightarrow T_2' \qquad \Gamma \vdash T_1 \text{ ok}}{\Gamma \vdash \Sigma x : T_1.\, T_2 \longrightarrow \Sigma x : T_1.\, T_2'}\ (\mathcal{S}/\textbf{tconv.cong.pair.2})$$

$$\frac{\Gamma \vdash T_0 \longrightarrow T_0' \qquad \Gamma, x : T_0 \vdash T_1 \text{ ok}}{\Gamma \vdash \Pi x : T_0.\, T_1 \longrightarrow \Pi x : T_0'.\, T_1}\ (\mathcal{S}/\textbf{tconv.cong.fun.arg})$$

[9]Strictement parlant, il s'agit en fait de deux familles de relations d'équivalence indexées par des environnements.

$$\frac{\Gamma \vdash T_0 \; \mathsf{ok} \qquad \Gamma, x : T_0 \vdash T_1 \longrightarrow T_1'}{\Gamma \vdash \Pi x : T_0 .\, T_1 \longrightarrow \Pi x : T_0 .\, T_1'} \;(\mathcal{S}/\textbf{tconv.cong.fun.ret})$$

$$\frac{\Gamma \vdash E \longrightarrow E'}{\Gamma \vdash S(E) \longrightarrow S(E')} \;(\mathcal{S}/\textbf{tconv.cong.sing}) \qquad \frac{\Gamma \vdash E \longrightarrow E' \qquad \Gamma \vdash E : \textsc{type}}{\Gamma \vdash \mathsf{Typ}\, E \longrightarrow \mathsf{Typ}\, E'} \;(\mathcal{S}/\textbf{tconv.cong.field})$$

Simplifications Nous pouvons voir $\mathsf{Typ}\,\langle T\rangle$ comme un destructeur appliqué à un constructeur appliqué à T ; ce type est équivalent à T.

$$\frac{\Gamma \vdash T \; \mathsf{ok}}{\Gamma \vdash \mathsf{Typ}\,\langle T\rangle \longrightarrow T} \;(\mathcal{S}/\textbf{tconv.field})$$

Règles sémantiques Nous proposons de déclarer que le type UNIT ne contient qu'une valeur. Ce type est isomorphe un singleton, et la règle ($\mathcal{S}$/tconv.unit) le rend équivalent. Le choix d'orientation de cette règle est sans importance.

$$\frac{\Gamma \vdash \mathsf{ok}}{\Gamma \vdash S(()) \longrightarrow \textsc{unit}} \;(\mathcal{S}/\textbf{tconv.unit})$$

IV.3.3.6 $\boxed{\Gamma \vdash E \longrightarrow E'}$ Conversion des expressions

Si E s'évalue en E' et que E a le type $S(E)$, la correction de l'évaluation par rapport au typage requiert que E' ait le type $S(E)$, ce qui est assuré si E et E' sont convertibles. Les règles de conversion s'assurent que E se convertit en E' : la relation de conversion contient la relation de réduction à l'exécution[10].

La **conversion** n'est définie que sur les expressions bien typées ; de nombreuses règles comportent des prémisses à cet effet.

Nous omettons toute conversion d'une expression de la forme $\mathsf{let}\, x = E_0 \,\mathsf{in}\, E : T$, que ce soit en tête ou à l'intérieur. En effet, comme indiqué à la section IV.2.1.4, cette construction est pour l'instant seulement du sucre syntaxique ; dans le système $\mathcal{E}$, où elle devient véritablement utile, la liaison locale est jugée impure et donc non sujette à conversion.

Contextes Au contraire de la relation d'exécution, nous n'avons pas d'intérêt particulier à ce que la conversion soit déterministe, et au contraire la confluence des règles est un avantage important ; nous autorisons une stratégie d'évaluation arbitraire. Les règles qui suivent font de toutes les constructions qui peuvent survenir dans une expression pure des contextes d'évaluation. En préparation à l'ajout de constructions impures, et au fait qu'une fonction immédiate peut être pure même si son corps est impur, la règle correspondante ($\mathcal{S}$/econv.cong.fun.body) a une forme particulière qui autorise à convertir toute sous-expression convertible du corps d'une fonction.

$$\frac{\Gamma \vdash T_0 \longrightarrow T_0' \qquad \Gamma, x : T_0 \vdash E_1 : T_1}{\Gamma \vdash (\lambda x : T_0.\; E_1) \longrightarrow (\lambda x : T_0'.\; E_1)} \;(\mathcal{S}/\textbf{econv.cong.fun.arg})$$

$$\frac{\Gamma, x : T_0 \vdash E \longrightarrow E' \qquad \Gamma, x : T_0, y : S(E) \vdash E_1 : T_1}{\Gamma \vdash (\lambda x : T_0.\; \{y \leftarrow E\}E_1) \longrightarrow (\lambda x : T_0.\; \{y \leftarrow E'\}E_1)} \;(\mathcal{S}/\textbf{econv.cong.fun.body})$$

$$\frac{\Gamma \vdash E \longrightarrow E' \qquad \Gamma \vdash E : \Pi x : T_0 .\, T_1 \qquad \Gamma \vdash E_0 : T_0}{\Gamma \vdash E\, E_0 \longrightarrow E'\, E_0} \;(\mathcal{S}/\textbf{econv.cong.app.fun})$$

[10] Pour les expressions pures seulement, comme nous le verrons dans le système $\mathcal{E}$.

$$\frac{\Gamma \vdash E \longrightarrow E' \qquad \Gamma \vdash E : T_0 \qquad \Gamma \vdash E_1 : \Pi x : T_0.\, T_1}{\Gamma \vdash E_1\, E \longrightarrow E_1\, E'}\ (\mathcal{S}/\textbf{econv.cong.app.arg})$$

$$\frac{\Gamma \vdash E \longrightarrow E' \qquad \Gamma \vdash E_2 : T_2}{\Gamma \vdash (E, E_2) \longrightarrow (E', E_2)}\ (\mathcal{S}/\textbf{econv.cong.pair.1}) \qquad \frac{\Gamma \vdash E \longrightarrow E' \qquad \Gamma \vdash E_1 : T_1}{\Gamma \vdash (E_1, E) \longrightarrow (E_1, E')}\ (\mathcal{S}/\textbf{econv.cong.pair.2})$$

$$\frac{\Gamma \vdash E \longrightarrow E' \qquad \Gamma \vdash E : \Sigma x : T_1.\, T_2}{\Gamma \vdash \pi_i\, E \longrightarrow \pi_i\, E'}\ (\mathcal{S}/\textbf{econv.cong.proj}) \qquad \frac{\Gamma \vdash T \longrightarrow T'}{\Gamma \vdash \langle T \rangle \longrightarrow \langle T' \rangle}\ (\mathcal{S}/\textbf{econv.cong.field})$$

Réduction en tête Les règles qui suivent décrivent l'évaluation habituelle sur les lambda-termes typés avec paires.

$$\frac{\Gamma, x : T_0 \vdash E_1 : T_1 \qquad \Gamma \vdash E_0 : T_0}{\Gamma \vdash (\lambda x : T_0.\ E_1)\, E_0 \longrightarrow \{x \leftarrow E_0\} E_1}\ (\mathcal{S}/\textbf{econv.app}) \qquad \frac{\Gamma \vdash E_1 : T_1 \qquad \Gamma \vdash E_2 : T_2}{\Gamma \vdash \pi_i\, (E_1, E_2) \longrightarrow E_i}\ (\mathcal{S}/\textbf{econv.proj})$$

IV.3.3.7 Extensionnalité

Nous munissons le système $\mathcal{S}$ de règles d'extensionnalité. De telles règles peuvent revêtir différentes formes ; nous choisissons des règles de conversion, dans le sens d'êta-expansions. Une règle telle que ($\mathcal{S}$/econv.eta.pair) peut par exemple être lue comme « tout expression typable comme une paire peut être mise sous une forme faisant apparaître la structure de paire ».

Une fois le choix fait de passer par des règles de conversion, reste à décider entre expansions et contractions. D'un point de vue technique, les expansions ont l'intérêt de conserver la confluence du système, contrairement aux êta-contractions [Klo80]. Les expansions ont en revanche le défaut de briser évidemment la normalisation. En pratique, il semble préférable d'exprimer les règles dans le sens de l'expansion, quitte à restreindre ultérieurement leur usage à un domaine fini (dicté par la structure du type de l'expression convertie) [Gog05].

$$\frac{\Gamma \vdash E : \textsc{type}}{\Gamma \vdash E \longrightarrow \langle \mathsf{Typ}\, E \rangle}\ (\mathcal{S}/\textbf{econv.eta.field})$$

$$\frac{\Gamma \vdash E : \Pi x : T_0.\, T_1}{\Gamma \vdash E \longrightarrow (\lambda x : T_0.\ E\, x)}\ (\mathcal{S}/\textbf{econv.eta.fun}) \qquad \frac{\Gamma \vdash E : \Sigma x : T_1.\, T_2}{\Gamma \vdash E \longrightarrow (\pi_1\, E, \pi_2\, E)}\ (\mathcal{S}/\textbf{econv.eta.pair})$$

IV.4 Scellage $\boxed{\mathcal{E}}$

IV.4.1 Scellage

IV.4.1.1 Types concrets inconnus

Au vu des règles de typage du système $\mathcal{S}$, nous pouvons donner à n'importe quelle expression E le type $\mathsf{S}(E)$ (à condition que E n'ait pas d'effet de bord, ce qui est le cas dans le langage présenté jusqu'ici). Or tous les champs types fournis par $\mathsf{S}(E)$ sont concrets ; le type donné à E est le plus précis possible. Le système $\mathcal{E}$ décrit intuitivement donc un système de modules dans lequel tous les types sont concrets.

Cette intuition est en fait prise en défaut lorsque E n'est close. Si E est le corps d'un foncteur, le type $\mathsf{S}(E)$ peut mentionner les paramètres du foncteur, et désigner des types inconnus statiquement. Par exemple, si E est le corps x du foncteur identité $\lambda x : \textsc{type}.\ x$, le type du corps est $\mathsf{S}(x)$, qui peut désigner un type différent à chaque utilisation du foncteur. Nous disposons ainsi d'un moyen d'exprimer un type inconnu : un foncteur qui attend ce type (éventuellement équipé d'opérations) comme argument.

```
module A = (struct type t = int  let x = 3  let f y = y end :
            sig type t  val x : t  val f : t -> int end)
module B = struct let x = A.f A.x end
```

(a) manière habituelle en Objective Caml

```
module A' = struct type t = int  let x = 3  let f y = y end
module F = functor (Arg : sig type t  val x : t  val f : t -> int end) ->
           struct let x = Arg.f Arg.x end
module B' = F(A')
```

(b) argument de foncteur en Objective Caml

$\mathsf{let}\ x_{\mathtt{A'}} = (\langle\text{INT}\rangle, (3, \lambda x : \text{INT}.\ x))\ \mathsf{in}$
$\mathsf{let}\ x_{\mathtt{F}} = \lambda y : (\Sigma t : \text{TYPE}.\ \Sigma x : \mathsf{Typ}\, t.\ (\Pi y : \mathsf{Typ}\, t.\ \text{INT})).\ (\pi_2 \pi_2 y)\, (\pi_1 \pi_2 y)\ \mathsf{in}$
$\mathsf{let}\ x_{\mathtt{B'}} = x_{\mathtt{F}}\, x_{\mathtt{A'}}\ \mathsf{in}\ \ldots$

(c) argument de foncteur dans le système $\mathcal{S}$

Fig. IV.3 – Un module définissant un type abstrait et un module l'utilisant

La figure IV.3 présente la définition d'un module `A` fournissant un type abstrait et d'un autre module `B` l'utilisant, d'une part de la manière usuelle en Objective Caml, d'autre part de la manière que nous venons de suggérer en syntaxe Objective Caml et dans le système $\mathcal{S}$. La différence essentielle tient au déplacement de l'opération « rendre abstrait » du module original `A` vers le module utilisant `B'`. Ce déplacement a plusieurs conséquences.

- D'un point de vue de structuration des programmes, la responsabilité de la limitation une signature restrictive passe du fournisseur du type abstrait à son utilisateur.
- Si le programme complet comprend plusieurs modules $\mathtt{B'}_1, \mathtt{B'}_2, \ldots$ qui utilisent `A'`, chacun lui appose sa signature ; il n'y a pas moyen de contraindre ces signatures à être les mêmes, et le fait que les modules $\mathtt{B'}_i$ utilisent la même abstraction n'est pas apparent. Ceci restreint l'expressivité du langage de modules.
- Le passage par un foncteur, c'est-à-dire un module paramétrique, force une utilisation évidemment paramétrique du module `A'`. C'est *a priori* une qualité, puisque, idéalement, le module `B'` ne doit dépendre que de de l'interface publiée de `A'` et non de son implémentation. Toutefois la contrainte serait trop forte pour le système adapté aux programmes répartis que nous présenterons à la section IV.6.

Notons que, en outre, le codage proposé oblige l'emploi d'un foncteur `F` ; ceci n'est pas pour nous une source de difficulté, mais a pu l'être historiquement (les modules définissant des types abstraits prédatent les foncteurs).

Il existe deux codages classique de types abstraits dans un lambda-calcul typé : les types existentiels, et le scellage. Nous allons examiner successivement ces deux méthodes, la première correspondant plutôt à une vision théorique, la seconde étant plus opérationnelle.

IV.4.1.2 Types existentiels

Considérons le module $(\langle\text{INT}\rangle, 3)$: il s'agit d'une paire formée d'un champ type et d'un terme de ce type. Outre sa signature singleton $\mathsf{S}((\langle\text{INT}\rangle, 3))$, nous pouvons lui attribuer de nombreuses signatures de paires (peut-être dépendantes) : toute signature de la forme $\Sigma x : \mathsf{T}_1.\, \mathsf{T}_2$ convient où T_1 est $\mathsf{S}(\langle\text{INT}\rangle)$ ou TYPE, et T_2 est $\mathsf{S}(3)$, INT ou $\mathsf{Typ}\, x$. Lorsque T_1 est INT, le type est concret ; lorsque T_1 est TYPE, le type est abstrait, et c'est ce deuxième cas qui nous intéresse. Les signatures $\Sigma x : \text{TYPE}.\ \text{INT} = \text{TYPE} * \text{INT}$ et $\Sigma x : \text{TYPE}.\ \mathsf{Typ}\, x$ sont incomparables ; ce sont deux sur-signatures de

$\mathsf{S}(\langle \textsc{int} \rangle) * \textsc{int}$ (qui est équivalente à $\Sigma x : \mathsf{S}(\langle \textsc{int} \rangle).\, \mathsf{Typ}\, x$).

Une manière classique d'exprimer des types abstraits est d'utiliser des **paquets existentiels** [MP88, CL90] : intuitivement, une signature comme $\Sigma x : \textsc{type}.\, \mathsf{Typ}\, x$ peut se lire « il existe un type x tel que $\mathsf{Typ}\, x$ ». Cette observation conduit à ajouter au langage une nouvelle forme de types, les **types existentiels**, ainsi que deux nouvelles formes de termes, un constructeur et un destructeur. Dans ce formalisme, le type $\exists t.\, T$ correspond à ce que nous notons $\Sigma t : \textsc{type}.\, T$. La construction d'une valeur typée existentiellement se fait par la construction $\mathsf{pack}\ T_1\ \mathsf{with}\ E_2\ \mathsf{to}\ T$, qui produit un « paquet » rassemblant le type T_1 et l'expression E_2 en cachant T_1 et en rendant E_2 visible avec le type T ; T doit être un type existentiel : $T = \exists t.\, T_2$. On pourra ainsi écrire

$$\mathsf{pack}\ \textsc{int}\ \mathsf{with}\ 3\ \mathsf{to}\ \exists t.\, \mathsf{Typ}\, t \qquad \text{ou} \qquad \mathsf{pack}\ \textsc{int}\ \mathsf{with}\ 3\ \mathsf{to}\ \exists t.\, \textsc{int}$$

Un paquet existentiel ne peut être utilisé que via la construction $\mathsf{open}\ E\ \mathsf{as}\ t\ \mathsf{with}\ x\ \mathsf{in}\ E' : T'$, dans laquelle E est un paquet existentiel, dont les composantes type et terme sont nommées respectivement t et x dans E' (t peut apparaître dans T). On écrira ainsi

$$\begin{array}{l} \mathsf{let}\ x_M = \mathsf{pack}\ \textsc{int}\ \mathsf{with}\ 3\ \mathsf{to}\ \exists t.\, \mathsf{Typ}\, t\ \mathsf{in} \\ \mathsf{open}\ x_M\ \mathsf{as}\ t\ \mathsf{with}\ x\ \mathsf{in}\ \ldots \end{array}$$

Une limitation des paquets existentiels tels que nous venons de les présenter est qu'ils ne permettent de rendre qu'un type abstrait. Or un module ML peut avoir plusieurs champs types, et l'on peut en rendre abstraits un sous-ensemble arbitraire. Nous pouvons modifier les existentiels pour que la variable n'ait plus forcément la sorte (chez nous, le type) $\textsc{type}$. Les différentes composantes à abstraire doivent alors être regroupées en tête du module, ce qui correspond à extraire la **racine flexible** (*flexroot*) au sens de Z. Shao [Sha99]. La principale modification syntaxique nécessaire est d'écrire un type existentiel sous la forme $\exists t : T'.\, T$; la notation $\exists t.\, T$ apparaît alors comme une abrévitation pour $\exists t : \textsc{type}.\, T$. La fabrication d'un paquet existentiel devient $\mathsf{pack}\ E_1\ \mathsf{with}\ E_2\ \mathsf{to}\ T$ (où $T = \exists t : T_1.\, T_2$), tandis que sa destruction reste $\mathsf{open}\ E\ \mathsf{as}\ t\ \mathsf{with}\ x\ \mathsf{in}\ E' : T'$.

Les types existentiels peuvent en fait être encodés dans le langage dont nous disposons déjà ; le logicien reconnaîtra une non-non-traduction, en rapprochant le type produit $\Pi x : T_0.\, T_1$ d'une quantification universelle $\forall x : T_0.\, T_1$.

$$\begin{array}{rcl} \exists t : T'.\, T & = & \Pi x : \textsc{type}.\, (\Pi t : T'.\, T \to \mathsf{Typ}\, x) \to \mathsf{Typ}\, x \\ \mathsf{pack}\ E_1\ \mathsf{with}\ E_2\ \mathsf{to}\ \exists t : T_1.\, T_2 & = & \lambda x : \textsc{type}.\ \lambda f : (\Pi t : T_1.\, T_2 \to \mathsf{Typ}\, x).\ f\, E_1\, E_2 \\ \mathsf{open}\ E\ \mathsf{as}\ t\ \mathsf{with}\ x\ \mathsf{in}\ E' : T' & = & E\, \langle T' \rangle\, (\lambda t : T_1.\ \lambda x : T_2.\ E') \end{array}$$

(Dans la dernière équation, T_1 et T_2 désignent les types respectifs de t et x dans le membre de gauche ; E a le type $\exists t : T_1.\, T_2$.)

IV.4.1.3 Scellage

Dans le noyau du langage ML, il est possible d'annoter le type d'une expression : dans `(E : T)`, E doit avoir le type T. Par exemple, le type de la fonction `(fun x -> ((x, 3) : string * int))` est `string -> string * int`. En Objective Caml, le type principal de l'expression `(E : T)` n'est pas forcément T, il peut être plus précis : ainsi le type de `(fun x -> ((x, 3) : string * 'a))` est `string -> string * int`. (Standard ML est plus restrictif.) En tout état de cause, le type de `(E : T)` n'est jamais plus général que T ; dans les exemples de ce paragraphe, l'annotation de type force `x` à être de type `string`.

La construction analogue dans un système de modules s'appelle **scellage**. Elle permet de restreindre la signature d'un module, en lui donnant une signature potentiellement abstraite (tel que décrit à la section IV.2.1.2). Si E est un module ayant la signature T, nous notons $\mathsf{E} \mathbin{!!} \mathsf{T}$ (« E scellé avec T »[11]) un module dont le contenu est le même que E, à ceci près que T restreint l'accès de par les types qui y sont abstraits.

En ML, le scellage se note $\mathsf{M} \; : \; \mathsf{S}$, comme l'annotation de types. En fait, il existe plusieurs variétés de scellage, différant notamment dans leur interaction avec les foncteurs, un point auquel nous consacrerons la section IV.4.4 ; celle que nous notons[12] $\mathsf{E} \mathbin{!!} \mathsf{T}$, et que nous appellerons scellage !dynamique, n'est pas identique au $\mathsf{M} \; : \; \mathsf{S}$ de Standard ML ni à celui d'Objective Caml. Nous discuterons d'autres formes de scellage aux sections IV.4.4.2 et IV.4.4.4.

Par exemple, nous transcrivons ainsi le fragment de programme de la figure IV.3(a) :

$$\begin{aligned}&\mathsf{let}\, x_{\mathtt{A}} = (\langle \text{INT}\rangle, (3, \lambda z : \text{INT}.\ z)) \mathbin{!!} (\Sigma t : \text{TYPE}.\, \Sigma y : \mathsf{Typ}\, t.\, (\Pi z : \mathsf{Typ}\, t.\, \text{INT}))\, \mathsf{in}\\ &(\pi_2\, \pi_2\, x_{\mathtt{A}})\, (\pi_1\, \pi_2\, x_{\mathtt{A}})\end{aligned}$$

Le module scellé a la même forme que le module original, la différence essentielle étant que certains types peuvent être cachés dans le module scellé. Ceci est conditionné par la relation de sous-typage, qui telle que nous l'avons définie à la section IV.3.3 pour le système $\mathcal{S}$ ne permet essentiellement que d'abstraire un champ type pour former un sur-type. Le scellage s'accomoderait sans difficulté d'une relation quelconque de sous-typage (pour l'annotation de type en ML, le sous-typage mesure le degré de polymorphisme dans les schémas de types).

Dans le cas particulier où l'on scelle un module par sa signature naturelle, le module obtenu est interchangeable avec l'original — le scellage ne se manifeste qu'au travers des types abstraits ou des valeurs ayant ces types. Autrement dit, si T_0 est la signature naturelle d'un module E_0, la typabilité et le comportement du programme $\mathsf{let}\, x = \mathsf{E}_0\, \mathsf{in}\, \mathsf{let}\, y = (\mathsf{E}_0 \mathbin{!!} \mathsf{T}_0)\, \mathsf{in}\, \mathsf{E}$ sont inchangés si l'on remplace certaines occurences de x dans E par y ou vice versa.

Dès lors qu'un module est scellé à une sur-signature stricte de sa signature naturelle, le module scellé est distinguable de l'original au niveau du typage : certains programmes deviennent non typables. Si E_0 a le type T_0 et que T_1 est un sur-type de T_0, alors le programme $\mathsf{let}\, x = (\mathsf{E}_0 \mathbin{!!} \mathsf{T}_1)\, \mathsf{in}\, (\lambda y : \mathsf{T}_0.\ ())\, x$ est mal typé quand bien même le programme $\mathsf{let}\, x = \mathsf{E}_0\, \mathsf{in}\, (\lambda y : \mathsf{T}_0.\ ())\, x$ est correct. Plus concrètement, le choix de la sur-signature interdit l'accès direct à certains champs. Ainsi, en reprenant l'exemple de la figure IV.3, le programme suivant est mal typé puisque la fonction $\pi_2\, \pi_2\, x_{\mathtt{A}}$ attend un argument de type $\mathsf{Typ}\, \pi_1\, x_{\mathtt{A}}$ et non INT :

$$\begin{aligned}&\mathsf{let}\, x_{\mathtt{A}} = (\langle \text{INT}\rangle, (3, \lambda z : \text{INT}.\ z)) \mathbin{!!} (\Sigma t : \text{TYPE}.\, \Sigma y : \mathsf{Typ}\, t.\, (\Pi z : \mathsf{Typ}\, t.\, \text{INT}))\, \mathsf{in}\\ &(\pi_2\, \pi_2\, x_{\mathtt{A}})\; 3\end{aligned}$$

Le scellage rend le type `t` abstrait.

On remarque que la création d'un type abstrait passe par une opération sur le module qui le contient, et non directement sur le type lui-même. Ceci n'est guère surprenant : un type abstrait est la donnée non seulement d'un type, mais aussi des opérations sur ce types, qui sont les autres champs du module. C'est pour cela que l'on parle quelquefois plutôt d'abstraction que de type abstrait (nous employons le plus souvent le second terme, moins techniquement adéquat mais grammaticalement plus clair). Notons au passage qu'un module scellé peut définir plusieurs types abstraits, lorsque le module est une structure ayant plusieurs champs types ; par exemple, un module implémentant

[11] Ou « E scellé par T », ou (abusivement) « E scellé à T ».

[12] Nous avons choisi la notation $\mathsf{E} \mathbin{!!} \mathsf{T}$ pour évoquer le fait que cette construction a un effet de bord (voir la section IV.4.1.4). Il n'existe pas de notation standard pour les différentes formes de scellage ; $\mathsf{E}{:}\mathsf{T}$, $\mathsf{E}{::}\mathsf{T}$ et $\mathsf{E}{:>}\mathsf{T}$ sont fréquentes dans la littérature.

une table de symboles peut offrir un type abstrait des symboles et un type abstrait des tables de symboles. En fait, un module scellé peut définir une famille de types abstraits, lorsqu'il s'agit d'un foncteur ; nous reviendrons plus en détail sur les interactions entre foncteurs et scellage à la section IV.4.4.1.

IV.4.1.4 L'effet du scellage

Considérons le programme suivant, dans lequel on pourra pour fixer les idées considérer que $\mathsf{E}_0 = (\langle\textsc{int}\rangle, (3, \lambda z : \textsc{int}.\ z))$ et $\mathsf{T}_0 = \Sigma t : \textsc{type}.\ \Sigma y : \mathsf{Typ}\, t.\ (\Pi z : \mathsf{Typ}\, t.\ \textsc{int})$:

$$\mathsf{let}\, x = (\mathsf{E}_0 \mathbin{!!} \mathsf{T}_0)\, \mathsf{in}\, \mathsf{let}\, x' = (\mathsf{E}_0 \mathbin{!!} \mathsf{T}_0)\, \mathsf{in}\, \mathsf{E}$$

Bien que définies à l'identique, les variables x et x' ne sont pas interchangeables dans E. En effet, si $(\pi_2\pi_2 x)\,(\pi_1\pi_2 x)$ et $(\pi_2\pi_2 x')\,(\pi_1\pi_2 x')$ sont bien typées, ce n'est pas le cas de $(\pi_2\pi_2 x)\,(\pi_1\pi_2 x')$, dans laquelle la fonction attend un argument de type $\mathsf{Typ}\,\pi_1 x$ mais reçoit un argument de type $\mathsf{Typ}\,\pi_1 x'$. Ces deux types ne sont pas compatibles, conformément à l'intuition que chaque scellage crée un nouveau type, distinct de tous les types existants jusqu'alors — en particulier, $\mathsf{Typ}\,\pi_1 x'$ est un nouveau type, distinct du type préexistant $\mathsf{Typ}\,\pi_1 x$.

Cet exemple illustre le fait que l'on ne peut pas directement tester l'équivalence une expression de module de la forme $\mathsf{E} \mathbin{!!} \mathsf{T}$ avec une autre expression ; étant donné que les types singletons servent justement à cela, le type $\mathsf{S}(\mathsf{E} \mathbin{!!} \mathsf{T})$ n'a pas de sens. Deux voies nous sont ouvertes : nous pouvons donner un sens à $\mathsf{S}(\mathsf{E} \mathbin{!!} \mathsf{T})$, ce qui nécessite de relâcher les règles de typage du langage ; ou nous pouvons l'interdire, ce qui implique de raffiner le typage pour détecter les cas indésirables. Nous faisons choix d'interdire, car le langage résultant est plus expressif et plutôt plus facile à comprendre ; nous évoquerons le sens que pourrait avoir un singleton d'un scellage, mais d'un *scellage faible*, à la section IV.4.4.4.

On ne peut pas dupliquer librement une expression de la forme $\mathsf{E} \mathbin{!!} \mathsf{T}$: cette expression a donc un *effet de bord*. On peut rapprocher cet effet de celui de l'allocation d'une cellule mémoire par `ref` en ML ou `malloc` en C ; ici nous allouons un nom de type. Il y a trois moyens classiques de gérer les effets dans un système de types :

- ne pas avoir d'effets dans le langage, mais avoir des programmes qui calculent non plus le résultat mais la suite d'effets qui y mène, c'est l'approche des monades ;
- les systèmes d'effets, consistant à annoter les jugements de typage avec une indication des effets de bords autorisés à l'expression ;
- des types linéaires, qui empêchent la duplication d'une expression.

Les types existentiels se rapprochent du point de vue monadique ; nous allons étudier cette ressemblance. Nous verrons qu'en un sens, les types existentiels nécessitent un encodage qui ne fait que repousser le problème. La solution que nous retiendrons est un système d'effet approprié, qui sera l'objet de la section IV.4.2. Nous n'avons pas tenté d'appliquer aux types abstraits une vision linéaire.

IV.4.1.5 Comparaison entre types existentiels et scellage

Soit un module $\mathsf{E} = (\mathsf{E}_1, \mathsf{E}_2)$ que l'on souhaite sceller avec une signature $\mathsf{T} = \Sigma t : \mathsf{T}_1.\ \mathsf{T}_2$ (avec T_1 abstraite et T_2 concrète) pour l'utiliser dans un programme E'. Avec le scellage, on écrit simplement

$$\mathsf{let}\, x = \mathsf{E} \mathbin{!!} \mathsf{T}\, \mathsf{in}\, \mathsf{E}' : \mathsf{T}'$$

Avec les types existentiels (voir la section I.2.2.1), il faut deux étapes :

$$\begin{array}{l} \mathsf{let}\, y = \mathtt{pack}\ \mathsf{E}_1\ \mathtt{with}\ \mathsf{E}_2\ \mathtt{to}\ \exists t : \mathsf{T}_1.\ \mathsf{T}_2\, \mathsf{in} \\ \mathtt{open}\ y\ \mathtt{as}\ x_1\ \mathtt{with}\ x_2\ \mathtt{in}\ (\mathsf{let}\, x = (x_1, x_2)\, \mathsf{in}\, \mathsf{E}' : \mathsf{T}') : \mathsf{T}' \end{array}$$

Quel est le sens de la valeur de y ? Le module E a déjà été mis sous une forme abstraite, où l'on ne peut plus retrouver la signature initiale. On pourrait donc voir le scellage comme du sucre syntaxique pour un passage par un paquet existentiel, $\mathsf{let}\, x = E \mathbin{!!} T\, \mathsf{in} \ldots$ étant traduit en

$$\mathsf{open}\ (\mathsf{pack}\ E_1\ \mathsf{with}\ E_2\ \mathsf{to}\ \exists t : T_1.\, T_2)\ \mathsf{as}\ x_1\ \mathsf{with}\ x_2\ \mathsf{in}\ \ldots$$

(la traduction doit séparer la (E_1 de type T_1 nommée x_1) de la partie rigide (E_2 de type T_2 nommée x_2) du module et de sa signature, comme signalé dans la section IV.4.1.2). Ainsi peut-on se passer de la forme intermédiaire du paquet existentiel, qui devient un scellage plus un travail d'encodage.

L'intérêt du stade intermédiaire de paquet existentiel est qu'il peut être déballé plusieurs fois, ce qui permet d'avoir plusieurs instances incompatibles d'un type abstrait de même implémentation. Cette séparation entre élaboration d'une abstraction et instantiation d'icelle est présente dans la première proposition de système de modules pour ML [Mac84] ; mais elle a rapidement mué vers la notion de foncteur. En effet, on peut obtenir le même effet en faisant du stade intermédiaire un foncteur qui produit un nouveau type abstrait à chaque fois qu'il est appliqué ; le foncteur a l'intérêt supplémentaire de pouvoir passer un paramètre à chaque instance. Ainsi, on pourra traduire

$$\begin{array}{l} \mathsf{let}\, y = \mathsf{pack}\ E_1\ \mathsf{with}\ E_2\ \mathsf{to}\ \exists t : T_1.\, T_2\ \mathsf{in} \\ ((\mathsf{open}\ y\ \mathsf{as}\ x_1\ \mathsf{with}\ x_2\ \mathsf{in}\ \ldots), \\ \ (\mathsf{open}\ y\ \mathsf{as}\ x_1'\ \mathsf{with}\ x_2'\ \mathsf{in}\ \ldots)) \end{array}$$

en termes de foncteur et de scellage :

$$\begin{array}{l} \mathsf{let}\, y = \lambda z : \textsc{unit}.\ ((E_1, E_2) \mathbin{!!} \Sigma t : T_1.\, T_2)\ \mathsf{in} \\ ((\mathsf{let}\, x = y\,()\ \mathsf{in}\ \ldots), \\ \ (\mathsf{let}\, x = y\,()\ \mathsf{in}\ \ldots)) \end{array}$$

On remarque que le passage par un paquet existentiel force le nommage du module abstrait : le déballage (« unpack ») a une portée limitée. Le problème est que dans ($\mathsf{open}\ y\ \mathsf{as}\ x_1\ \mathsf{with}\ x_2\ \mathsf{in}\ E' : T'$), le nom x_1 est le seul moyen de désigner la partie abstraite du paquet. Comme T' est le type de la construction de déballage toute entière, x_1 ne peut y apparaître ; la manière naturelle d'envisager une construction de déballage à portée illimitée, à savoir d'être équivalente à $\mathsf{open}\ y\ \mathsf{as}\ x_1\ \mathsf{with}\ x_2\ \mathsf{in}\ (x_1, x_2) : T'$, est donc impraticable faute de pouvoir écrire T'. Le problème est en fait analogue à celui de typer $(E_1, E_2) \mathbin{!!} (\Sigma t : T_1.\, T_2)$: la construction crée un nouveau type, ce qui est un effet. Il faudrait donc nous équiper d'un système d'effets ; à ce compte, autant adopter la construction de scellage qui est finalement bien plus simple et toute aussi expressive.

IV.4.2 Un système d'effets

IV.4.2.1 Introduction

Au départ, dans le contexte des langages de programmation, un système de types classifie des valeurs — c'est-à-dire, en général, des objets représentables dans la mémoire d'un ordinateur — en les répartissant en types (3 est un entier, `true` est un booléen...) Les types des valeurs donnent des informations sur leur représentation en mémoire, à la fois en contraignant ses caractéristiques intrinsèques (par exemple, la taille d'une cellule mémoire pour une variable, ou le registre à utiliser pour passer un argument à une fonction) et en indiquant sa sémantique. Partant de là, l'on peut associer un type à d'autres éléments d'un langage de programmation ; en particulier, on dit qu'une variable a un type T lorsque les valeurs qu'elle stocke sont de ce type, et on dit qu'un programme a un type T lorsque la valeur calculée par ce programme a le type T. Un système de types, tel que ceux que nous avons présenté pour les langages $\mathcal{B}$ et $\mathcal{S}$, donne des règles pour attribuer un type à un programme ; les programmes qui admettent un type sont considérés comme corrects, et le (ou éventuellement les) type obtenu donne une indication quant à ce que calcule le programme.

Or il est d'autres informations intéressantes à propos d'un programme que la valeur qu'il calcule : s'il termine normalement ou non, le temps qu'il met à s'exécuter, les fichiers ou les périphériques auxquels il accède, les variables partagées qu'il lit ou modifie... Un **système d'effets** est une forme de système de types qui s'intéresse non pas aux valeurs calculées par un programme mais aux effets de bord qu'il a durant son exécution, c'est-à-dire typiquement à ce que l'on peut observer du programme durant son exécution. En particulier, les systèmes d'effet servent souvent à analyser les lectures et écritures en mémoire ou sur des périphériques.

Nous noterons génériquement γ une annotation d'effet[13] ; cette notation est analogue à T pour les types. Le membre droit d'un jugement de typage d'une expression aura désormais la forme $\mathsf{E} :^{\gamma} \mathsf{T}$, signifiant que le fragment de programme E calcule une valeur de type T, et que les effets de son exécution sont caractérisés par γ. Un jugement de typage d'une expression a donc la forme $\Gamma \vdash \mathsf{E} :^{\gamma} \mathsf{T}$; la définition des environnements Γ est inchangée, car notre système de types est destiné à un langage à appel par valeurs, dans lequel les variables reçoivent des valeurs et non des expressions arbitraires qu'il y aurait lieu d'évaluer en causant des effets.

Il est en général commode de munir le domaine des effets d'une relation d'ordre : nous noterons $\gamma_1 \sqsubseteq \gamma_2$ lorsque tout effet autorisé par γ_1 l'est également par γ_2. De même que nous avons muni notre système de types de sous-typage implicite (par la règle $(\mathcal{S}/\mathsf{et.sub})$), nous autoriserons la **sous-effectuation** implicite, c'est-à-dire que si $\gamma_1 \sqsubseteq \gamma_2$ et $\Gamma \vdash \mathsf{E} :^{\gamma_1} \mathsf{T}$ alors $\Gamma \vdash \mathsf{E} :^{\gamma_2} \mathsf{T}$ (« qui peut le moins peut le plus » : plus un effet γ est petit, moins il autorise de comportements ; l'ordre sur les effets est ainsi analogue à celui sur les types où un type plus petit est plus précis, autorisant moins de valeurs).

Une valeur est déjà toute calculée, donc si V est une valeur de type T, alors $\Gamma \vdash \mathsf{V} :^{\gamma} \mathsf{T}$ quelle que soit la contrainte γ sur les effets autorisés. En particulier, nous noterons $\mathtt{P}$ l'annotation qui n'autorise aucun effet, si bien que $\Gamma \vdash \mathsf{V} :^{\mathtt{P}} \mathsf{T}$; $\mathtt{P}$ est l'élément minimal pour la relation $\sqsubseteq$. Une expression bien typée avec l'annotation d'effet $\mathtt{P}$ est dite **pure**. En vertu du sous-typage implicite, une expression pure E de type T dans un environnement vérifie $\Gamma \vdash \mathsf{E} :^{\gamma} \mathsf{T}$ quel que soit γ.

L'interprétation ensembliste fondamentale des effets est de considérer une annotation γ comme un ensemble d'effets élémentaires (de même que l'interprétation ensembliste usuelle d'un type est de le voir comme un ensemble de valeurs). Un effet élémentaire peut être, par exemple, l'accès en lecture ou en écriture à une cellule mémoire donnée ou à un périphérique donné. De même qu'un système de types ne peut souvent pas caractériser un ensemble arbitraire de valeurs (par exemple, peu de systèmes possèdent un type correspondant à l'ensemble $\{\mathtt{true}, 3, 4\}$), un système d'effet peut ne caractériser que certains ensembles d'effets.

Il est souvent commode de disposer d'une annotation d'effet indiquant que l'on ne sait rien du comportement de l'expression. Nous noterons cette annotation $\mathtt{I}$; elle est maximale pour l'ordre $\sqsubseteq$; le jugement $\Gamma \vdash \mathsf{E} :^{\mathtt{I}} \mathsf{T}$ exprime donc que E est bien typée sans préjuger de ce qui peut advenir lorsqu'on l'exécute.

Nous avons jusqu'à présent caractérisé le type d'une fonction par le type de son argument et le type de la valeur qu'elle renvoie. Dans un système d'effets, il faut en plus indiquer quels sont les effets qui sont produits lorsque l'on appelle la fonction : un type de fonction a donc la forme $\Pi\mathsf{x} : \mathsf{T}_0.\ {}^{\gamma}\mathsf{T}$. En effet, une lambda-expression $\lambda\mathsf{x} : \mathsf{T}_0.\ \mathsf{E}$ est une valeur ; les effets éventuels de E sont suspendus à l'appel de la fonction sur un argument. L'expression $(\lambda\mathsf{x} : \mathsf{T}_0.\ \mathsf{E})\, \mathsf{V}$ a donc tous les

[13] Nous emploierons indifféremment les termes « annotation d'effets » et « contrainte d'effets », et abrègerons souvent en « effet ».

effets de E ; si E a le type T_0 et l'effet γ sous l'hypothèse que $\mathtt{x}$ a le type T_0 (en d'autres termes, si $\mathtt{x} : \mathsf{T}_0 \vdash \mathsf{E} :^{\gamma} \mathsf{T}$), alors $(\lambda \mathtt{x} : \mathsf{T}_0.\ \mathsf{E})$ a le type $\Pi \mathtt{x} : \mathsf{T}_0.\,{}^{\gamma}\mathsf{T}$.

On peut considérer les types T tels que nous les avons définis comme les types des valeurs, et les couples (γ, T) comme les types d'expressions, la première partie γ caractérisant les effets de l'évaluation de l'expression et la seconde partie T étant le type de l'éventuelle valeur renvoyée. Nous suivons notamment F. Henglein, H. Makholm et H. Niss [HMN05] en adoptant une notation qui fait apparaître ces « types d'expressions » sous la forme ${}^{\gamma}\,\mathsf{T}$. Dans cette optique, le sous-typage des expressions est l'ordre produit du sous-typage des valeurs et du sous-effectuage ; en appel par valeurs, l'argument d'une fonction est une valeur et a un type de valeur T_0, tandis que le corps est une expression et a un type d'expression ${}^{\gamma}\,\mathsf{T}$ (en appel par nom, l'argument aurait aussi un type d'expression ${}^{\gamma_0}\,\mathsf{T}_0$).

IV.4.2.2 Pureté

De toutes les constructions du langage que nous avons discuté jusqu'à maintenant, la seule qui ait un effet est le scellage. L'effet en question est la création d'un type abstrait (plus exactement d'une famille de types abstraits) ; il n'y a guère d'intérêt à raffiner plus, en conséquence de quoi notre système d'effets peut se limiter à deux annotations : `P` et `I`. L'annotation `P` dénote un programme **pur**, qui ne crée pas de type, tandis que l'annotation `I` dénote un programme **impur** dont l'évaluation peut comporter des opérations de scellage.

Nous avons vu à la section IV.3.1 que l'on peut former le singleton de n'importe quelle valeur, ou n'importe quelle variable — plus généralement, comme discuté à la section IV.3.1, l'on peut former $\mathsf{S}(\mathsf{E})$ lorsque E est parfaitement compréhensible statiquement (relativement à la connaissance du type de ses variables libres). L'on peut voir les choses ainsi : l'on peut former $\mathsf{S}(\mathsf{E})$ dès lors que E peut être évaluée lors de la vérification de typage à la compilation ; ceci signifie que E n'a pas d'effet de bord, c'est-à-dire que E est pure. Bref, **$\mathsf{S}(\mathsf{E})$ est bien formé si et seulement si E est pure**. De manière générale, une expression peut apparaître dans un type si et seulement si elle est pure.

Notons que si la pureté est une notion sémantique, puisque par exemple une application de fonction peut être pure ou impure, elle reste associée à une expression et non à un objet. Par exemple, si l'on réduit une expression impure, le résultat peut être pur (il l'est forcément si l'on arrive à une valeur). L'inverse est impossible : une expression pure ne peut se réduire qu'en une expression pure[14]. En revanche, on peut donner un nom au résultat de l'évaluation d'une expression impure : si l'on écrit $\mathsf{let}\,\mathtt{x} = \mathsf{E}\ !!\ \mathsf{T}\,\mathsf{in}\ \ldots$, l'expression $\mathtt{x}$ est pure. Ce n'est pas surprenant au regard de la justification que nous avons donné à l'impureté du scellage $\mathsf{E}\ !!\ \mathsf{T}$: le problème est de nommer le type abstrait, et le nom $\mathtt{x}$ le permet justement.

Si nous souhaitons étendre le système $\mathcal{E}$ à un langage de programmation réaliste, il faudra étendre le système d'effets pour gérer les effets de bords, au moins les effets externes du programme. Toutefois l'utilité du système d'effets pour notre propos se limite à savoir reconnaître les expressions pures, afin de les autoriser seules dans les types. Aussi nous limiterons-nous à un système à deux annotations d'effets seulement `P` et `I`.

Un point important à remarquer est que nous n'avons pas muni le langage des expressions d'un combinateur de point fixe. L'ajout d'un tel combinateur avec le type traditionnel $(\mathsf{T} \to \mathsf{T}) \to \mathsf{T}$ serait problématique : en effet, il permettrait d'introduire dans le monde des types des expressions dont l'évaluation ne termine pas. Bien sûr, un langage de programmation a besoin d'un tel combinateur (ou toute autre manière d'écrire des fonctions récursives quelconques) ; mais ce combinateur doit être considéré comme impur, ayant donc le type $(\mathsf{T} \to \mathsf{T}) \to^{\mathtt{I}} \mathsf{T}$.[15]

[14] Le théorème de préservation du typage nous l'assure.

[15] Notons au passage que considérer la non-terminaison comme un effet de bord est cohérent avec les intuitions

IV.4.2.3 Projetabilité, séparabilité et comparabilité

Dans le système que nous décrivons, un module « bien élevé » est un module pur. La pureté est une notion très forte : un module pur est complètement connu statiquement (puisqu'il a son type singleton, et que nos types singletons caractérisent complètement un objet). Il existe des notions plus fines pour déterminer les usages légitimes d'un module ; nous allons en toucher quelques mots.

Les systèmes de modules classiques, en particulier les sommes translucentes de R. Harper et M. Lillibridge [HL94, Lil97] et les types manifestes de X. Leroy [Ler94, Ler95], se concentrent sur la notion de **projetabilité** (voir la section I.2.2.2). Il s'agit, étant donné un module M, de savoir si l'on peut utiliser le type $\mathsf{M.t}$ pour former un type : si c'est le cas, M est dit **projetable** (avec nos notations, E est projetable lorsque $\mathsf{Typ}\,\pi_1\,\mathsf{E}$ est un type bien formé). Nous avons approché la notion de projetabilité par la pureté : $\mathsf{Typ}\,\pi_1\,\mathsf{E}$ est correct si et seulement si E est pure (et a une signature adéquate). Il s'agit bien d'une approximation, puisque la pureté est une notion plus forte : dans le fragment de code suivant, le module `A` est pur et projetable, tandis que le module `B` est impur mais tout de même projetable.

```
module A = struct  type t = int  let x = 3  end
module B = struct  type t = int  let x = ref 3  end
```

Une notion voisine de la projetabilité est celle de **comparabilité** : un module est dit **comparable** lorsque l'on peut tester son équivalence avec un autre module. Dans le calcul de D. Dreyer, K. Crary et R. Harper [DCH03], les notions de comparabilité et de projetabilité se confondent, puisque tester l'équivalence de deux modules revient à comparer leurs parties types. Dans notre calcul, la pureté tient aussi lieu de comparabilité (nous traitons à l'identique les composantes types et valeurs des modules dans les comparaisons).

Nous avons évoqué à la section I.3.1.1 la *séparation des phases*, qui consiste à différencier clairement la phase statique de la vie du programme, où l'on vérifie sa correction par le typage, de la phase dynamique, dans laquelle s'effectuent sans erreur les calculs spécifiés. Dans le noyau de ML, chaque phase est associée à une partie du langage : la phase statique s'intéresse particulièrement au monde des types, tandis que la phase dynamique concerne les expressions. En revanche, le propre des modules est de mélanger au niveau syntaxique types et expressions. On peut néanmoins chercher à séparer autant que possible dans l'étude sémantique les parties statiques, liées aux types, et des parties dynamiques, liées aux expressions.

Dans son analyse des modules pour ML [Dre05], D. Dreyer distingue deux niveaux de pureté dans les modules. Un module est dit **totalement pur** lorsqu'il est pur dans le sens que nous avons donné à ce terme (c'est-à-dire que son exécution ne déclenche pas d'effet de bord). Un module est dit **partiellement pur**[16] lorsque ses composantes types peuvent être entièrement déterminées sans déclencher d'effets. Ainsi le module `B` ci-dessus, bien que partiellement pur, ne l'est pas totalement. Pour qu'un module puisse être considéré comme projetable, il suffit qu'il soit partiellement pur.

Une difficulté de la notion de pureté partielle est de déterminer si les effets d'une expression influencent ses composantes types. La pureté totale est bien sûr une condition suffisante. Une autre condition suffisante est la **séparabilité**, une notion due à R. Harper, J. Mitchell et E. Moggi

traditionnelles de ce qu'est un effet. Par exemple, une expression pure peut être évaluée n'importe quel nombre de fois, alors que le nombre de fois qu'est évaluée une expression ayant un effet peut en général être observé ; si une expression ne termine pas, l'on peut observer une différence entre l'évaluer zéro fois et l'évaluer une fois. Dans le raisonnement « *theorems for free* » [Wad89], où l'on suppose un langage sans effet de bord au sens classique, la plupart des théorèmes doivent distinguer le cas où telle ou telle sous-expression ne termine pas.

[16]D. Dreyer utilise les termes « dynamiquement pur » et « statiquement pur » là où nous utilisons les adverbes respectifs « totalement » et « partiellement ». Nous préférons réserver aux expressions « dynamiquement pur » et « statiquement pur » le sens que leur donne les travaux précédents de D. Dreyer [DCH03].

[HMM90] et que D. Dreyer [Dre05] précise dans le cadre d'un système de modules avec foncteurs. Un module est **séparable** lorsque ses composantes types ne dépendent pas du tout de calculs pouvant avoir des effets de bord, en particulier de calculs effectués au niveau des expressions. Un module séparable est forcément partiellement pur, donc projetable, mais peut être impur ; c'est le cas par exemple du module `B` ci-dessus. Réciproquement, la présence dans le langage de modules de première classe permet d'écrire facilement des modules purs mais non séparables, comme le module `C` suivant :

```
let n = read_int ()
module A  = struct  type t = int  let x = 3  end
module B  = struct  type t = int  let x = ref 3  end
module A' = struct  type t = bool let x = true  end
module B' = struct  type t = bool let x = ref true  end
module C = if n >= 0 then A else A'
module D = if n >= 0 then B else B'
```

Le module `D` est quant à lui partiellement pur, mais ni séparable ni totalement pur.

Nous avons vu lors de l'élaboration des notions de types singletons et d'équivalences de modules (section IV.3.1) qu'en présence de foncteurs, la séparabilité est difficile à analyser : c'est justement ce qui nous a poussé à ne pas chercher à le faire, et à intégrer expressions et types dans les tests d'équivalence de modules. L'analyse de séparabilité nous apparaît d'autant moins tentante que nous souhaitons au bout du compte pouvoir comparer des types dynamiquement, ce qui signifie que notre notion d'équivalence doit être performante dans les cas inséparables. Dans le présent travail, nous n'irons pas plus loin que la notion de pureté totale ; plutôt que d'introduire une notion de séparabilité, nous suggérons comme raffinement un système d'effet plus sophistiqué, capable de reconnaître des cas de pureté partielle (en « déclassifiant » les effets de bords qui n'influencent pas la valeur d'une expression).

IV.4.3 Présentation formelle

Nous présentons formellement le système $\mathcal{E}$, obtenu à partir du système $\mathcal{S}$ en ajoutant la construction de scellage et surtout un système d'effets.

IV.4.3.1 Syntaxe

Donnons d'abord la syntaxe des effets :

γ ::=		**effet**
	`P`	pur
	`I`	impur

Rappelons que les effets sont munis d'une relation d'ordre, notée $\gamma_1 \sqsubseteq \gamma_2$, telle que $\mathtt{P} \sqsubseteq \mathtt{I}$. Nous notons $\gamma_1 \sqcup \gamma_2$ la borne supérieure de γ_1 et γ_2 et $\gamma_1 \sqcap \gamma_2$ leur borne inférieure.

La syntaxe du système $\mathcal{E}$ étend celle du système $\mathcal{S}$ en ajoutant une annotation d'effet aux endroits nécessaires :

- sur les types de fonctions, désormais notés $\Pi \mathsf{x} : \mathsf{T}_0.\, {}^{\gamma}\mathsf{T}_1$; nous abrégerons en $\mathsf{T}_0 \to^{\gamma} \mathsf{T}_1$ lorsque x n'est pas libre dans T_1 ;
- sur les jugements de typage d'expression, désormais notés $\mathsf{E} :^{\gamma} \mathsf{T}$.

Nous omettrons quelquefois l'annotation d'effet lorsque celle-ci est `P` ; ainsi nous pourrons noter $\Pi \mathsf{x} : \mathsf{T}_0.\, \mathsf{T}_1$ ou $\mathsf{T}_0 \to \mathsf{T}_1$ le type d'une fonction pure. De plus, la syntaxe des expressions comprend désormais le scellage.

$\mathsf{T} ::=$	**type**
$\quad \ldots$	
$\quad \Pi \mathsf{x} : \mathsf{T}_0 .\, ^{\gamma}\mathsf{T}_1$	produit dépendant (aussi noté $\mathsf{T}_1 \rightarrow^{\gamma} \mathsf{T}_2$ lorsque $\mathsf{x} \notin \mathbf{fv}\, \mathsf{T}_1$)
$\mathsf{E} ::=$	**expression (module)**
$\quad \ldots$	
$\quad \mathsf{E}\ !!\ \mathsf{T}$	module scellé
$\mathsf{J} ::=$	**membre droit de jugement local**
$\quad \ldots$	
$\quad \mathsf{E} :^{\gamma} \mathsf{T}$	typage d'une expression

IV.4.3.2 $\boxed{\mathsf{E} \longrightarrow \mathsf{E}'}$ Exécution

Au niveau de l'exécution, la seule nouveauté du système $\mathcal{E}$ est le besoin de réduire la construction de scellage $\mathsf{E}\ !!\ \mathsf{T}$. L'intuition dans les langages de la famille ML veut que les types n'influent que sur la compilation, et non sur l'exécution ; dans cette optique, $\mathsf{E}\ !!\ \mathsf{T}$ est à l'exécution équivalent à E, ce qu'exprime la règle ($\mathcal{E}$/ered.seal).

$$\mathsf{V}\ !!\ \mathsf{T} \longrightarrow \mathsf{V} \qquad (\mathcal{E}/\mathsf{ered.seal})$$

L'expression scellée est d'abord réduite en une valeur.

$\mathsf{C} ::=$	**contexte d'évaluation (de profondeur** 1**)**
$\quad \ldots$	
$\quad \_\ !!\ \mathsf{T}$	module scellé

Les règles ($\mathcal{E}$/ered.app), ($\mathcal{E}$/ered.proj), ($\mathcal{E}$/ered.let) et ($\mathcal{E}$/ered.context) sont reprises de $\mathcal{S}$, qui en hérite lui-même de $\mathcal{B}$.

IV.4.3.3 $\boxed{\Gamma \vdash \ldots}$ Typage : correction, équivalences, sous-typage

Le système $\mathcal{E}$ reprend toutes les règles de typage du système $\mathcal{S}$, et en ajoute une (le typage du scellage). Cependant la plupart des règles reprises le sont sous une forme modifiée, qui inclut des annotations d'effets. Aussi allons-nous réénoncer les règles affectées, en expliquant pour chacune les le traitement des effets.

Règles reprises Les règles suivantes sont reprises à l'identique du système $\mathcal{S}$ (qui les tient lui-même de $\mathcal{B}$) :
- ($\mathcal{E}$/envok.nil), ($\mathcal{E}$/envok.x) ;
- ($\mathcal{E}$/tok.base.unit), ($\mathcal{E}$/tok.base.bool), ($\mathcal{E}$/tok.base.int), ($\mathcal{E}$/tok.type), ($\mathcal{E}$/tok.pair) ;
- ($\mathcal{E}$/teq.refl), ($\mathcal{E}$/teq.sym), ($\mathcal{E}$/teq.trans), ($\mathcal{E}$/teq.conv) ;
- ($\mathcal{E}$/eeq.sym), ($\mathcal{E}$/eeq.trans), ($\mathcal{E}$/eeq.conv) ;
- ($\mathcal{E}$/tconv.cong.pair.1), ($\mathcal{E}$/tconv.cong.pair.2), ($\mathcal{E}$/econv.cong.field), ($\mathcal{E}$/tconv.cong.sing), ($\mathcal{E}$/tconv.field), ($\mathcal{E}$/tconv.unit) ;
- ($\mathcal{E}$/tsub.trans), ($\mathcal{E}$/tsub.eq), ($\mathcal{E}$/tsub.cong.pair).

En dehors du typage des expressions et de ($\mathcal{E}$/tsub.cong.fun), les modifications par rapport aux les règles de $\mathcal{S}$ consistent principalement à exiger la pureté des expressions imbriquées, et à autoriser les annotations d'effets sur les types produits. Les règles ($\mathcal{E}$/econv.cong.fun.arg) et ($\mathcal{E}$/econv.cong.fun.body) autorisent néanmoins le corps de la fonction à être impur (l'essentiel étant que l'objet réécrit soit pur, peu importe le contexte).

$$\frac{\Gamma \vdash \mathsf{T}' \,\mathsf{ok} \qquad \Gamma, x : \mathsf{T}' \vdash \mathsf{T}'' \,\mathsf{ok}}{\Gamma \vdash \Pi x : \mathsf{T}'.\, {}^{\gamma}\mathsf{T}'' \,\mathsf{ok}}\ (\mathcal{E}/\textbf{tok.fun}) \qquad \frac{\Gamma \vdash \mathsf{E} :^{\mathsf{P}} \textsc{type}}{\Gamma \vdash \mathsf{Typ}\,\mathsf{E}\,\mathsf{ok}}\ (\mathcal{E}/\textbf{tok.field}) \qquad \frac{\Gamma \vdash \mathsf{E} :^{\mathsf{P}} \mathsf{T}}{\Gamma \vdash \mathsf{S}(\mathsf{E})\,\mathsf{ok}}\ (\mathcal{E}/\textbf{tok.sing})$$

$$\frac{\Gamma \vdash \mathsf{E} :^{\mathsf{P}} \mathsf{T}}{\Gamma \vdash \mathsf{E} \equiv \mathsf{E}}\ (\mathcal{E}/\textbf{eeq.refl}) \qquad \frac{\Gamma \vdash \mathsf{E} :^{\mathsf{P}} \mathsf{T}}{\Gamma \vdash \mathsf{S}(\mathsf{E}) <: \mathsf{T}}\ (\mathcal{E}/\textbf{tsub.sing})$$

$$\frac{\Gamma \vdash \mathsf{T}_0 \longrightarrow \mathsf{T}_0' \qquad \Gamma, x : \mathsf{T}_0 \vdash \mathsf{T}_1 \,\mathsf{ok}}{\Gamma \vdash \Pi x : \mathsf{T}_0.\, {}^{\gamma}\mathsf{T}_1 \longrightarrow \Pi x : \mathsf{T}_0'.\, {}^{\gamma}\mathsf{T}_1}\ (\mathcal{E}/\textbf{tconv.cong.fun.arg})$$

$$\frac{\Gamma \vdash \mathsf{T}_0 \,\mathsf{ok} \qquad \Gamma, x : \mathsf{T}_0 \vdash \mathsf{T}_1 \longrightarrow \mathsf{T}_1'}{\Gamma \vdash \Pi x : \mathsf{T}_0.\, {}^{\gamma}\mathsf{T}_1 \longrightarrow \Pi x : \mathsf{T}_0.\, {}^{\gamma}\mathsf{T}_1'}\ (\mathcal{E}/\textbf{tconv.cong.fun.ret})$$

$$\frac{\Gamma \vdash \mathsf{E} \longrightarrow \mathsf{E}' \qquad \Gamma \vdash \mathsf{E} :^{\mathsf{P}} \textsc{type}}{\Gamma \vdash \mathsf{Typ}\,\mathsf{E} \longrightarrow \mathsf{Typ}\,\mathsf{E}'}\ (\mathcal{E}/\textbf{tconv.cong.field})$$

$$\frac{\Gamma \vdash \mathsf{T}_0 \longrightarrow \mathsf{T}_0' \qquad \Gamma, x : \mathsf{T}_0 \vdash \mathsf{E}_1 :^{\gamma} \mathsf{T}_1}{\Gamma \vdash (\lambda x : \mathsf{T}_0.\ \mathsf{E}_1) \longrightarrow (\lambda x : \mathsf{T}_0'.\ \mathsf{E}_1)}\ (\mathcal{E}/\textbf{econv.cong.fun.arg})$$

$$\frac{\Gamma, x : \mathsf{T}_0 \vdash \mathsf{E} \longrightarrow \mathsf{E}' \qquad \Gamma, x : \mathsf{T}_0, y : \mathsf{S}(\mathsf{E}) \vdash \mathsf{E}_1 :^{\gamma} \mathsf{T}_1}{\Gamma \vdash (\lambda x : \mathsf{T}_0.\ \{y \leftarrow \mathsf{E}\}\mathsf{E}_1) \longrightarrow (\lambda x : \mathsf{T}_0.\ \{y \leftarrow \mathsf{E}'\}\mathsf{E}_1)}\ (\mathcal{E}/\textbf{econv.cong.fun.body})$$

$$\frac{\Gamma \vdash \mathsf{E} \longrightarrow \mathsf{E}' \qquad \Gamma \vdash \mathsf{E}_2 :^{\mathsf{P}} \mathsf{T}_2}{\Gamma \vdash (\mathsf{E}, \mathsf{E}_2) \longrightarrow (\mathsf{E}', \mathsf{E}_2)}\ (\mathcal{E}/\textbf{econv.cong.pair.1}) \qquad \frac{\Gamma \vdash \mathsf{E} \longrightarrow \mathsf{E}' \qquad \Gamma \vdash \mathsf{E}_1 :^{\mathsf{P}} \mathsf{T}_1}{\Gamma \vdash (\mathsf{E}_1, \mathsf{E}) \longrightarrow (\mathsf{E}_1, \mathsf{E}')}\ (\mathcal{E}/\textbf{econv.cong.pair.2})$$

$$\frac{\Gamma \vdash \mathsf{E} \longrightarrow \mathsf{E}' \qquad \Gamma \vdash \mathsf{E} :^{\mathsf{P}} \Sigma x : \mathsf{T}_1.\, \mathsf{T}_2}{\Gamma \vdash \pi_i\, \mathsf{E} \longrightarrow \pi_i\, \mathsf{E}'}\ (\mathcal{E}/\textbf{econv.cong.proj})$$

$$\frac{\Gamma \vdash \mathsf{E} \longrightarrow \mathsf{E}' \qquad \Gamma \vdash \mathsf{E} :^{\mathsf{P}} \Pi x : \mathsf{T}_0.\, {}^{\mathsf{P}}\mathsf{T}_1 \qquad \Gamma \vdash \mathsf{E}_0 :^{\mathsf{P}} \mathsf{T}_0}{\Gamma \vdash \mathsf{E}\,\mathsf{E}_0 \longrightarrow \mathsf{E}'\,\mathsf{E}_0}\ (\mathcal{E}/\textbf{econv.cong.app.fun})$$

$$\frac{\Gamma \vdash \mathsf{E} \longrightarrow \mathsf{E}' \qquad \Gamma \vdash \mathsf{E} :^{\mathsf{P}} \mathsf{T}_0 \qquad \Gamma \vdash \mathsf{E}_1 :^{\mathsf{P}} \Pi x : \mathsf{T}_0.\, {}^{\mathsf{P}}\mathsf{T}_1}{\Gamma \vdash \mathsf{E}_1\,\mathsf{E} \longrightarrow \mathsf{E}_1\,\mathsf{E}'}\ (\mathcal{E}/\textbf{econv.cong.app.arg})$$

$$\frac{\Gamma \vdash \mathsf{E}_1 :^{\mathsf{P}} \mathsf{T}_1 \qquad \Gamma \vdash \mathsf{E}_2 :^{\mathsf{P}} \mathsf{T}_2}{\Gamma \vdash \pi_i\,(\mathsf{E}_1, \mathsf{E}_2) \longrightarrow \mathsf{E}_i}\ (\mathcal{E}/\textbf{econv.proj}) \qquad \frac{\Gamma, x : \mathsf{T}_0 \vdash \mathsf{E}_1 :^{\mathsf{P}} \mathsf{T}_1 \qquad \Gamma \vdash \mathsf{E}_0 :^{\mathsf{P}} \mathsf{T}_0}{\Gamma \vdash (\lambda x : \mathsf{T}_0.\ \mathsf{E}_1)\,\mathsf{E}_0 \longrightarrow \{x \leftarrow \mathsf{E}_0\}\mathsf{E}_1}\ (\mathcal{E}/\textbf{econv.app})$$

$$\frac{\Gamma \vdash \mathsf{E} :^{\mathsf{P}} \textsc{type}}{\Gamma \vdash \mathsf{E} \longrightarrow \langle \mathsf{Typ}\,\mathsf{E} \rangle}\ (\mathcal{E}/\textbf{econv.eta.field})$$

$$\frac{\Gamma \vdash \mathsf{E} :^{\mathsf{P}} \Pi x : \mathsf{T}_0.\, {}^{\gamma}\mathsf{T}_1}{\Gamma \vdash \mathsf{E} \longrightarrow (\lambda x : \mathsf{T}_0.\ \mathsf{E}\,x)}\ (\mathcal{E}/\textbf{econv.eta.fun}) \qquad \frac{\Gamma \vdash \mathsf{E} :^{\mathsf{P}} \Sigma x : \mathsf{T}_1.\, \mathsf{T}_2}{\Gamma \vdash \mathsf{E} \longrightarrow (\pi_1\,\mathsf{E}, \pi_2\,\mathsf{E})}\ (\mathcal{E}/\textbf{econv.eta.pair})$$

Sous-typage pour les fonctions La règle de congruence des types produits dépendants pour le sous-typage est enrichie. Un type fonctionnel est plus petit qu'un autre type fonctionnel lorsque le domaine du premier est plus grand, l'image du premier est plus petit, et le premier autorise moins d'effets lors de son exécution.

$$\frac{\Gamma \vdash \mathsf{T}_0' <: \mathsf{T}_0 \qquad \Gamma, x : \mathsf{T}_0' \vdash \mathsf{T}_1 <: \mathsf{T}_1' \qquad \Gamma, x : \mathsf{T}_0 \vdash \mathsf{T}_1 \,\mathsf{ok} \qquad \text{si } \gamma \sqsubseteq \gamma'}{\Gamma \vdash \Pi x : \mathsf{T}_0.\, {}^{\gamma}\mathsf{T}_1 <: \Pi x : \mathsf{T}_0'.\, {}^{\gamma'}\mathsf{T}_1'}\ (\mathcal{E}/\textbf{tsub.cong.fun})$$

IV.4.3.4 $\boxed{\Gamma \vdash \mathsf{E} :^{\gamma} \mathsf{T}}$ Typage des expressions

Les jugements de typage d'expressions portent désormais une annotation d'effets.

Constantes, variables, champs types Les constantes, les variables et les champs types sont forcément purs.

$$\frac{\Gamma \vdash \mathsf{ok}}{\Gamma \vdash () :^{\mathrm{P}} \textsc{unit}}\ (\mathcal{E}/\textbf{et.base.unit}) \qquad \frac{\Gamma \vdash \mathsf{ok}}{\Gamma \vdash \mathsf{b}\nu :^{\mathrm{P}} \textsc{bool}}\ (\mathcal{E}/\textbf{et.base.bool}) \qquad \frac{\Gamma \vdash \mathsf{ok}}{\Gamma \vdash \underline{\mathsf{n}} :^{\mathrm{P}} \textsc{int}}\ (\mathcal{E}/\textbf{et.base.int})$$

$$\frac{\Gamma \vdash \mathsf{ok} \qquad \text{si } \mathsf{x} : \mathsf{T} \in \Gamma}{\Gamma \vdash \mathsf{x} :^{\mathrm{P}} \mathsf{T}}\ (\mathcal{E}/\textbf{et.x}) \qquad \frac{\Gamma \vdash \mathsf{T}\ \mathsf{ok}}{\Gamma \vdash \langle \mathsf{T} \rangle :^{\mathrm{P}} \textsc{type}}\ (\mathcal{E}/\textbf{et.type})$$

Paires Les paires sont de simples structures de données : le type d'une paire indique seulement le type des composantes, sans donner d'indication sur l'éventuelle répartition des effets entre la fabrication des deux composantes. L'annotation d'effet d'une expression $(\mathsf{E}_1, \mathsf{E}_2)$ est donc l'annotation commune à E_1 et E_2 (obtenue grâce à la règle ($\mathcal{E}$/et.sub)). De même, les effets d'une première projection sont ceux du projeté. La deuxième projection n'est en revanche utilisable que sur une expression pure, parce que l'expression apparaît dans le type[17].

$$\frac{\Gamma \vdash \mathsf{E}_1 :^{\gamma} \mathsf{T}_1 \qquad \Gamma \vdash \mathsf{E}_2 :^{\gamma} \mathsf{T}_2}{\Gamma \vdash (\mathsf{E}_1, \mathsf{E}_2) :^{\gamma} \mathsf{T}_1 * \mathsf{T}_2}\ (\mathcal{E}/\textbf{et.pair})$$

$$\frac{\Gamma \vdash \mathsf{E} :^{\gamma} \Sigma \mathsf{x} : \mathsf{T}_1.\, \mathsf{T}_2}{\Gamma \vdash \pi_1 \mathsf{E} :^{\gamma} \mathsf{T}_1}\ (\mathcal{E}/\textbf{et.proj.1}) \qquad \frac{\Gamma \vdash \mathsf{E} :^{\mathrm{P}} \Sigma \mathsf{x} : \mathsf{T}_1.\, \mathsf{T}_2 \qquad \Gamma \vdash \mathsf{E}_1 :^{\mathrm{P}} \mathsf{S}(\pi_1 \mathsf{E})}{\Gamma \vdash \pi_2 \mathsf{E} :^{\mathrm{P}} \{\mathsf{x} \leftarrow \mathsf{E}_1\} \mathsf{T}_2}\ (\mathcal{E}/\textbf{et.proj.2})$$

Fonctions Une fonction immédiate est systématiquement pure. La construction lambda met en suspens les effets du corps de la fonction ; ceux-ci sont reflétés dans l'annotation d'effet dans le type fonctionnel. Lors de l'application, les effets du corps sont découverts et s'ajoutent à ceux directement visibles dans l'expression. Nous exigeons que l'argument d'une fonction soit pur ; en effet, celui-ci est substitué dans le type du résultat.

$$\frac{\Gamma, \mathsf{x} : \mathsf{T}_0 \vdash \mathsf{E} :^{\gamma} \mathsf{T}_1}{\Gamma \vdash \lambda \mathsf{x} : \mathsf{T}_0.\ \mathsf{E} :^{\mathrm{P}} \Pi \mathsf{x} : \mathsf{T}_0.\ {}^{\gamma}\mathsf{T}_1}\ (\mathcal{E}/\textbf{et.fun}) \qquad \frac{\Gamma \vdash \mathsf{E}_1 :^{\gamma_1} \Pi \mathsf{x} : \mathsf{T}_0.\ {}^{\gamma_2}\mathsf{T} \qquad \Gamma \vdash \mathsf{E}_0 :^{\mathrm{P}} \mathsf{T}_0}{\Gamma \vdash \mathsf{E}_1\, \mathsf{E}_0 :^{\gamma_1 \sqcup \gamma_2} \{\mathsf{x} \leftarrow \mathsf{E}_0\} \mathsf{T}}\ (\mathcal{E}/\textbf{et.app})$$

Liaison locale Dans la règle ($\mathcal{E}$/et.app), l'argument E_0 doit être pur. Pour lever cette restriction, on peut utiliser une liaison locale, comme mentionné à la section IV.2.1.4. Il faut alors éviter d'avoir à substituer E_0 dans le type du résultat, d'où dans la règle ($\mathcal{E}$/et.let) la prémisse imposant à T de ne pas mentionner la variable liée localement x (on peut voir T comme le type de l'expression $\mathsf{let}\ \mathsf{x} = \mathsf{E}_0\ \mathsf{in}\ \mathsf{E} : \mathsf{T}$ toute entière et non juste le type de T). Les effets de l'expression toute entière sont la réunion de ceux de E_0 et de E. Étant donné que notre système ne distingue que pur et impur, et que le cas où E est pure ne présente pas d'intérêt (puisque l'on peut alors utiliser une application de fonction), nous citons directement la règle dans le cas impur. En présence d'un système d'effets plus fin, nous ne permettrions pas au résultat d'être pur pour éviter d'alourdir le fragment pur du langage avec une construction superflue.

$$\frac{\Gamma \vdash \mathsf{E}_0 :^{\mathrm{I}} \mathsf{T}_0 \qquad \Gamma, \mathsf{x} : \mathsf{T}_0 \vdash \mathsf{E} :^{\mathrm{I}} \mathsf{T} \qquad \Gamma \vdash \mathsf{T}\ \mathsf{ok}}{\Gamma \vdash (\mathsf{let}\ \mathsf{x} = \mathsf{E}_0\ \mathsf{in}\ \mathsf{E} : \mathsf{T}) :^{\mathrm{I}} \mathsf{T}}\ (\mathcal{E}/\textbf{et.let})$$

Si E est une expression impure de type $\mathsf{T}_1 * \mathsf{T}_2$, la seconde projection de E pourra être codée $\mathsf{let}\ \mathsf{x} = \mathsf{E}\ \mathsf{in}\ \pi_2 \mathsf{x} : \mathsf{T}_2$. Si E_0 est une expression impure et que E a le type $\mathsf{T}_1 \to \mathsf{T}_2$, l'application de E

[17] Il suffirait d'exiger la pureté de E_1, E pouvant être quelconque, mais nous n'en avons pas l'utilité.

à E_0 pourra être codée $\mathsf{let}\, x = \mathsf{E}_0 \,\mathsf{in}\, \mathsf{E}\, x : \mathsf{T}_2$. On note que dans les deux cas, T_2 ne peut pas contenir x : le type de E ne doit pas être dépendant.

Sous-typage En plus du sous-typage implicite, la règle ($\mathcal{E}$/et.sub) assure la sous-effectuation implicite : toute expression qui a ses effets limités par γ les a à plus forte raison limités par γ' lorsque $\gamma \sqsubseteq \gamma'$.

$$\frac{\Gamma \vdash \mathsf{E} :^{\gamma} \mathsf{T} \qquad \Gamma \vdash \mathsf{T} <: \mathsf{T}' \qquad \text{si } \gamma \sqsubseteq \gamma'}{\Gamma \vdash \mathsf{E} :^{\gamma'} \mathsf{T}'}\ (\mathcal{E}/\textbf{et.sub})$$

Singletons Nous qualifions la règle ($\mathcal{E}$/et.sing) afin que seule une expression pure puisse donner lieu à un type singleton.

$$\frac{\Gamma \vdash \mathsf{E} :^{\mathsf{P}} \mathsf{T}}{\Gamma \vdash \mathsf{E} :^{\mathsf{P}} \mathsf{S}(\mathsf{E})}\ (\mathcal{E}/\textbf{et.sing})$$

Scellage Une nouvelle règle permet de typer la nouvelle construction de scellage. En général, l'expression E a « naturellement » un sous-type de T, et la règle ($\mathcal{E}$/et.sub) lui est appliquée. Les effets de l'expression E !! T sont ceux de E, plus l'effet du scellage lui-même ; comme nous ne distinguons pas celui-ci, l'annotation d'effet de E !! T est toujours $\mathtt{I}$.

$$\frac{\Gamma \vdash \mathsf{E} :^{\gamma} \mathsf{T}}{\Gamma \vdash (\mathsf{E} \mathbin{!!} \mathsf{T}) :^{\mathtt{I}} \mathsf{T}}\ (\mathcal{E}/\textbf{et.seal})$$

IV.4.4 Applicativité

IV.4.4.1 Foncteurs applicatifs

Considérons un foncteur dont le corps est scellé : $\lambda x : \mathsf{T}_0.\ (\mathsf{E} \mathbin{!!} \mathsf{T})$. Au vu de la description que nous avons faite du scellage, chaque application de ce foncteur déclenche l'évaluation de l'expression E !! T, donc la création d'un nouveau jeu de types abstraits. Dans le programme suivant, les types $\mathsf{Typ}\,\pi_1\, x_1$ et $\mathsf{Typ}\,\pi_1\, x_2$ sont donc incompatibles :

$$\begin{array}{l}\mathsf{let}\ \mathsf{f} = \lambda x : \text{UNIT}.\ ((\langle \mathsf{T}' \rangle, \mathsf{E}) \mathbin{!!} \Sigma x : \text{TYPE}.\ \mathsf{T}_1)\ \mathsf{in} \\ \mathsf{let}\ x_1 = \mathsf{f}\,()\ \mathsf{in}\ \mathsf{let}\ x_2 = \mathsf{f}\,()\ \mathsf{in}\ \ldots\end{array}$$

Ceci signifie que le foncteur f est **génératif** (voir la section I.2.2.3). Un foncteur dont le corps est scellé est forcément génératif ; cela est reflété par son type de la forme $\Pi x : \mathsf{T}_0.\ {}^{\mathtt{I}}\mathsf{T}$ indiquant que l'application produit un effet, par opposition aux types de la forme $\Pi x : \mathsf{T}_0.\ {}^{\mathsf{P}}\mathsf{T}$ conférés aux foncteurs **transparents**, qui ne contiennent pas de scellage.

Dans certains cas, par exemple celui d'un foncteur construisant une structure de donnée dont les arguments décrivent les éléments, il serait préférable que deux applications du même foncteur aux mêmes arguments définissent des types compatibles : nous voulons aussi des foncteurs **applicatifs** [Ler95]. Nous allons examiner deux méthodes pour définir des foncteurs applicatifs.

La première méthode consiste à ajouter au langage une nouvelle notion de scellage, telle que en un certain sens sceller deux fois le même module donne des résultats compatibles (au sens où leurs types abstraits sont équivalents). Une telle notion de scellage est appelée **scellage faible**, et nous la noterons $\mathsf{E} :: \mathsf{T}$, par opposition au **scellage fort** E !! T que nous avons étudié précédemment. Nous verrons à la section IV.4.4.2 comment définir un scellage faible de manière à avoir l'équivalence entre $\mathsf{Typ}\,\pi_1\, x_1$ et $\mathsf{Typ}\,\pi_1\, x_2$ dans le programme suivant :

$$\begin{array}{l}\mathsf{let}\ \mathsf{g} = \lambda x : \text{UNIT}.\ ((\langle \mathsf{T}' \rangle, \mathsf{E}) :: \Sigma x : \text{TYPE}.\ \mathsf{T}_1)\ \mathsf{in} \\ \mathsf{let}\ x_1 = \mathsf{f}\,()\ \mathsf{in}\ \mathsf{let}\ x_2 = \mathsf{f}\,()\ \mathsf{in}\ \ldots\end{array}$$

Il existe en fait un autre moyen de construire des foncteurs applicatifs qui ne nécessite pas d'étendre le langage. Il suffit de sceller le foncteur lui-même et non son corps. Du coup, la création d'abstraction se passe une fois pour toute au moment de la définition du foncteur : il n'y a pas d'abstraction supplémentaire lors de son application. Ainsi, dans le programme suivant, g est un foncteur applicatif, et $\mathsf{Typ}\,\pi_1\,y_1$ et $\mathsf{Typ}\,\pi_1\,y_2$ sont compatibles (nous supposons l'expression E pure).

$$\begin{array}{l}\mathsf{let}\ g = (\lambda x : \text{UNIT}.\ (\langle T' \rangle, E))\ !!\ (\Pi x : \text{UNIT}.{}^{\mathsf{P}}\Sigma x : \text{TYPE}.\ T_1)\ \mathsf{in} \\ \mathsf{let}\ y_1 = g\,()\ \mathsf{in}\ \mathsf{let}\ y_2 = g\,()\ \mathsf{in}\ \ldots\end{array}$$

Dans tous les cas, un foncteur applicatif se distingue d'un foncteur génératif par sa signature qui indique que l'application du foncteur n'a pas d'effet. Ceci n'empêche pas le foncteur de créer des types abstraits, comme le montre la présence de TYPE en position covariante dans la signature. La méthode de scellage du foncteur nous donne une intuition intéressante sur la façon dont sont créés les types abstraits : l'effet de bord que constitue le scellage a lieu au moment où le foncteur est défini (plus précisément, c'est lorsque le foncteur passe de transparent à applicatif, lorsqu'il est scellé, que l'effet a lieu). Si le foncteur est défini et scellé au niveau supérieur du programme, l'effet a lieu lors de l'initialisation du programme.

On notera que l'on peut transformer à tout moment un foncteur applicatif en un foncteur génératif (de même que l'on peut rendre applicatif un foncteur transparent), en le scellant à la signature de foncteur génératif idoine : $\Pi x : T_0.{}^{\mathsf{P}}T_1$ est un sous-type de $\Pi x : T_0.{}^{\mathsf{I}}T_1$ (ceci est contenu dans la règle ($\mathcal{E}$/tsub.cong.fun)). La transformation inverse est *a priori* indésirable : il y aurait perte de générativité donc perte d'abstraction. De fait, notre système d'effets l'interdit : appliquer un foncteur génératif produit un effet, et le seul moyen de « cacher » cet effet est de le protéger par une abstraction dont le type garde trace de l'effet.

IV.4.4.2 Scellage statique : formalisation $\boxed{\mathcal{W}}$

Introduction Nous allons définir une nouvelle notion de scellage « faible », telle que sceller deux fois le même module donne des résultats compatibles. Le but de cette notion est de définir des foncteurs applicatifs, et les foncteurs applicatifs sont un bon moyen de comprendre le scellage faible. Nous avons vu comment sceller un foncteur transparent pour le rendre applicatif, ce qui représente un effet lors de la création du foncteur. La famille de types abstraits créés par un foncteur applicatif (indexée par les arguments du foncteur) est déterminée une fois pour toutes. Nous avons vu que si l'on code un foncteur applicatif à l'aide du scellage génératif, l'effet a lieu lors de l'initialisation du programme. En fait, on peut considérer que l'effet du scellage faible se manifeste lors de la compilation du programme, d'où le nom de **scellage statique**, par opposition au **scellage dynamique** E !! T. (Nous évoquerons à la section IV.4.4.4 d'autres variantes de scellage faible.) Nous allons maintenans définir formellement le scellage statique.

Syntaxe Nous définissons le système $\mathcal{W}$ qui étend le système $\mathcal{E}$. Le langage des expressions comporte en plus le scellage statique $\mathsf{E} :: \mathsf{T}$. De plus, une annotation supplémentaire est nécessaire dans la liaison locale ; nous expliquerons sa motivation dans la présentation de la règle ($\mathcal{W}$/et.let) ; nous omettrons fréquemment cette annotation dans les exemple lorsqu'elle ne joue pas un rôle important. Le langage des types est inchangé.

$\mathsf{E} ::=$	**expression (module)**
$\quad \ldots$	
$\quad \mathsf{let}\, x = \mathsf{E}_0 \,\mathsf{in}\, \mathsf{E} :^{\gamma} \mathsf{T}$	liaison locale
$\quad \mathsf{E} :: \mathsf{T}$	scellage statique

La principale nouveauté est que le système d'effets est élargi :

$\gamma ::=$	**effet**
$\quad$ `P`	pur
$\quad$ `I`	effet dynamique
$\quad$ `S`	effet statique
$\quad$ `IS`	effet dynamique et effet statique

L'ordre sur les effets est donné par $\mathtt{P} \sqsubseteq \mathtt{I} \sqsubseteq \mathtt{IS}$ et $\mathtt{P} \sqsubseteq \mathtt{S} \sqsubseteq \mathtt{IS}$. Seules les annotations `P` et `I` peuvent apparaître dans les types de fonctions $\Pi x : \mathsf{T}_0.\ {}^{\gamma}\mathsf{T}_1$.

Le typage dans le système $\mathcal{W}$ est pour l'essentiel identique au système $\mathcal{E}$; seules changent certaines règles dans lesquelles apparaissent des expressions impures. Les règles paramétrées par une annotation d'effet peuvent instancier cette annotation d'effet par `S` ou `IS`, avec dans le cas de ($\mathcal{W}$/et.sub) la relation d'ordre étendue.

Le typage d'un scellage statique est similaire à celui d'un typage dynamique, en introduisant bien sûr l'effet approprié. Dans les deux cas, l'effet s'ajoute aux autres effets éventuels de l'expression scellée.

$$\frac{\Gamma \vdash \mathsf{E} :^{\gamma} \mathsf{T}}{\Gamma \vdash (\mathsf{E} \,!!\, \mathsf{T}) :^{\gamma \sqcup \mathtt{I}} \mathsf{T}}\ (\mathcal{W}/\textbf{et.seal.dyn}) \qquad \frac{\Gamma \vdash \mathsf{E} :^{\gamma} \mathsf{T}}{\Gamma \vdash (\mathsf{E} :: \mathsf{T}) :^{\gamma \sqcup \mathtt{S}} \mathsf{T}}\ (\mathcal{W}/\textbf{et.seal.stat})$$

L'abstraction fonctionnelle cache les effets dynamiques, qui apparaissent dans le type de la fonction, mais pas les effets statiques qui restent apparents dans le résultat — $\gamma \sqcap \mathtt{S}$ peut se lire « la partie statique de γ », et $\gamma \sqcap \mathtt{I}$ « la partie dynamique de γ ». Nous rappelons la règle de l'application de fonctions ; elle est formellement inchangée, mais signalons quand même que l'on a forcément $\gamma_2 \sqsubseteq \mathtt{I}$.

$$\frac{\Gamma, x : \mathsf{T}_0 \vdash \mathsf{E} :^{\gamma} \mathsf{T}_1}{\Gamma \vdash \lambda x : \mathsf{T}_0.\ \mathsf{E} :^{\gamma \sqcap \mathtt{S}} \Pi x : \mathsf{T}_0.\ {}^{\gamma \sqcap \mathtt{I}}\mathsf{T}_1}\ (\mathcal{W}/\textbf{et.fun}) \qquad \frac{\Gamma \vdash \mathsf{E}_1 :^{\gamma_1} \Pi x : \mathsf{T}_0.\ {}^{\gamma_2}\mathsf{T} \qquad \Gamma \vdash \mathsf{E}_0 :^{\mathtt{P}} \mathsf{T}_0}{\Gamma \vdash \mathsf{E}_1\, \mathsf{E}_0 :^{\gamma_1 \sqcup \gamma_2} \{x \leftarrow \mathsf{E}_0\}\mathsf{T}}\ (\mathcal{W}/\textbf{et.app})$$

Dans le système $\mathcal{E}$, nous avons exigé qu'une expression $\mathsf{let}\, x = \mathsf{E}_0 \,\mathsf{in}\, \mathsf{E} : \mathsf{T}$ ne soit pas pure, afin de ne pas alourdir l'ensemble des expressions à traiter à la compilation (une telle expression peut toujours s'écrire $(\lambda x : \mathsf{T}_0.\ \mathsf{E})\, \mathsf{E}_0$, voire même $\{x \leftarrow \mathsf{E}_0\}\mathsf{E}$, dès que E_0 est pure). Nous maintenons cette exigence. En revanche, nous ne pouvons plus directement déclarer le résultat comme impur, puisqu'il existe désormais deux annotations d'effets incomparables (`I` et `S`). Nous demandons donc au programmeur de spécifier l'effet de l'expression comme il spécifie son type (et pour la même raison : sans cela, le typage ne serait pas principal).

$$\frac{\Gamma \vdash \mathsf{E}_0 :^{\gamma} \mathsf{T}_0 \qquad \Gamma, x : \mathsf{T}_0 \vdash \mathsf{E} :^{\mathtt{I}} \mathsf{T} \qquad \Gamma \vdash \mathsf{T}\ \mathsf{ok} \qquad \text{si } \gamma \neq \mathtt{P}}{\Gamma \vdash (\mathsf{let}\, x = \mathsf{E}_0 \,\mathsf{in}\, \mathsf{E} :^{\gamma} \mathsf{T}) :^{\gamma} \mathsf{T}}\ (\mathcal{W}/\textbf{et.let})$$

Exécution La sémantique dynamique du système $\mathcal{W}$ est la même que celle du système $\mathcal{E}$; la règle ($\mathcal{W}$/ered.seal) étend ($\mathcal{E}$/ered.seal) pour autoriser le rédex à être au choix un scellage statique ou un scellage dynamique. Le degré de générativité du scellage n'a pas d'importance puisque sa réduction efface l'abstraction.

IV.4.4.3 Équivalences en présence de scellage statique

Nous avons pu remarquer à la section IV.4.4.1 que le scellage dynamique ne commute pas avec l'abstraction fonctionnelle : $\lambda x : T_0.\ (E\ !!\ T)$ et $(\lambda x : T_0.\ E)\ !!\ (\Pi x : T_0.\ {}^{\gamma}T)$ ne sont pas équivalents, quel que soit γ (si $\gamma = \mathtt{I}$, les deux expressions ont le même type — ce sont des foncteurs génératifs — mais la première est pure tandis que la seconde est (dynamiquement) impure). En revanche, le scellage statique commute avec les foncteurs : les expressions $E_1 = \lambda x : T_0.\ (E :: T)$ et $E_2 = (\lambda x : T_0.\ E) :: (\Pi x : T_0.\ {}^{\gamma}T)$ sont d'emploi équivalent — un fait qu'utilisent couramment (implicitement) les programmeurs ML. Nous dirons que E_1 et E_2 sont **équitypables**, c'est-à-dire que quels que soient Γ, γ' et T', le jugement de typage $\Gamma \vdash E_1 :^{\gamma'} T'$ est valide si et seulement si le jugement $\Gamma \vdash E_1 :^{\gamma'} T'$ l'est. L'équipabilité est la notion qui nous intéressera dans la présente section, dans laquelle nous allons examiner des fragments de programmes qui ont le même comportement à l'exécution en dehors de la manière dont ils appliquent le scellage, mais diffèrent au niveau du typage en ce qu'ils créent de l'abstraction différemment.

Pour démontrer l'équitypabilité de E_1 et E_2, notons d'abord que le bon typage de E_1 comme celui de E_2 requiert que le jugement $\Gamma, x : T_0 \vdash E :^{\gamma} T$ soit prouvable (faire l'analyse de cas sur les dérivations de typage de E_1 et E_2). L'expression E_1 est alors typable par application de la règle ($\mathcal{W}$/et.seal.stat) suivie de ($\mathcal{W}$/et.fun), tandis que E_2 est typable par application de la règle ($\mathcal{W}$/et.fun) suivie de ($\mathcal{W}$/et.seal.stat) (des applications de ($\mathcal{W}$/et.sub) concernant le type du résultat peuvent être intercalées ; leurs conséquences sont bénignes car les opérations en jeu sont covariantes par rapport au type du résultat). Les expressions E_1 et E_2 ont toutes deux comme type principal $\Pi x : T_0.\ {}^{\gamma \sqcap \mathtt{I}}T$ et comme annotation d'effet principale $\mathtt{S}$.

Le scellage statique commute également avec d'autres constructions. Par exemple, $(E_1, E_2) :: (T_1 * T_2)$ est équitypable avec $(E_1 :: T_1, E_2 :: T_2)$, ainsi qu'avec $(E_1 :: T_1, E_2)$ et $(E_1, E_2 :: T_2)$ si l'on suppose respectivement que E_2 a le type T_2 et E_1 a le type T_2 : dans tous les cas, la présence du scellage statique à l'une quelconque des positions entraîne l'impureté statique de toute l'expression. Les expressions $\pi_i\,(E :: T_1 * T_2)$ et $(\pi_i\,E) :: T_i$ sont également équitypables sous réserve que E ait le type $T_1 * T_2$.

Considérons maintenant une liaison locale $E' = \mathsf{let}\,x = E_0\,\mathsf{in}\,E :^{\gamma} T$. Sceller seulement la partie E ou sceller toute E' n'est manifestement pas équivalent, car l'ensemble des signatures par lesquels chacune de ces deux expressions peut être scellées est différent (seule E peut être scellée par un type mentionnant x). La différence n'a néanmoins guère d'importance. En effet, l'influence du scellage sur les effets est la même dans les deux cas (introduire $\mathtt{S}$ si l'on scelle statiquement, $\mathtt{I}$ si l'on scelle dynamiquement) ; quant au type de l'expression, c'est toujours T si l'on scelle E et que l'expression reste bien typée, et c'est un sous-type T' de T si l'on scelle E'. On note en particulier que $\mathsf{let}\,x = E_0\,\mathsf{in}\,(E :: T_1) : T$ est équitypable avec $(\mathsf{let}\,x = E_0\,\mathsf{in}\,E :^{\gamma} T) :: T$ du moment que E a le type T_1 dans l'environnement adéquat.

Un scellage, même statique, ne peut pas apparaître dans un argument de fonction. Toutes ces considérations montrent que nous pouvons donc sans perte d'expressivité limiter la présence d'un scellage statique à l'intérieur d'un programme à deux catégories d'emplacement : sur un module lié localement $\mathsf{let}\,x = (E_0 :: T_0)\,\mathsf{in}\,E :^{\gamma} T$ et sur un foncteur appliqué $(E_1 :: T_2)\,E_0$. Dans le premier cas, retirer le scellage pourrait autoriser l'expression E à utiliser un type plus précis pour x (n'importe quel type de E_0, et pas forcément T_0) — ce qui revient exactement à dire que l'abstraction conférée par le scellage disparaîtrait avec celui-ci. L'utilité du scellage sur un foncteur appliqué est de même ordre : pour que l'expression $(E_1 :: T_2)\,E_0$ soit bien typée, T_2 doit être un type de fonction[18] $\Pi x : T_0.\ {}^{\gamma}T_1$; si T_1 est plus petit que nécessaire, l'abstraction pourrait migrer au-dessus de l'application (on pourrait écrire $(E_1\,E_0) :: \{x \leftarrow E_0\}T_1$) ; si T_0 est plus grand que nécessaire, cela limite les types

[18] T_2 peut aussi être un singleton, mais le scellage n'introduirait alors aucune abstraction.

possibles pour E_0, autrement dit cela rend E_0 plus abstrait tel qu'il est vu par la fonction. En fait, $(E_1 :: \Pi x : T_0.\, {}^{\gamma}T_1)$ est équitypable avec $\mathsf{let}\, x = (E_0 :: T_0)\, \mathsf{in}\, E_1\, x :^{\gamma} T_1$. Finalement, *le scellage statique n'est utile que sur un module que l'on lie localement.*

IV.4.4.4 Autres formes de scellage

Nous avons formalisé deux formes de scellage : le scellage statique, qui crée une famille de types abstraits pour chaque occurence syntaxique de l'opérateur de scellage, et le scellage dynamique, qui crée une famille de types abstraits à chaque fois que l'opérateur de scellage est évalué. Ces deux formes correspondent au scellage faible (*weak sealing*, $E :: T$) et au scellage fort (*strong sealing*, $E :> T$) proposés par D. Dreyer, K. Crary et R. Harper [DCH03], et notre système d'effets reprend le leur[19] (ils déclarent le scellage fort comme ayant un effet statique en plus de son effet dynamique, mais cela n'a que peu d'incidence et guère d'utilité).

D. Dreyer [Dre05] distingue trois forme de scellage :

- le scellage impur $\mathsf{impure}(E :> T)$ est le scellage fort de D. Dreyer, K. Crary et R. Harper [DCH03] mentionné au paragraphe précédent, qui est quasiment équivalent à notre scellage dynamique ;
- le scellage séparable $E :> T$ et le scellage inséparable $\mathsf{pure}(E :> T)$ correspondent tous deux à notre scellage statique, se distinguant par leur séparabilité, trait que nous ne prenons pas en compte (voir la section IV.4.2.3).

De nombreuses autres variantes plus ou moins fortes sont concevables. L'**ascription** consiste à contraindre une expression par un type sans en restreindre la vue depuis l'extérieur : si E a le type T, l'ascription $E :_a T$ a n'importe quel type de E (en particulier, si E est pure, $E :_a T$ est pure et a le type $S(E)$). On peut voir l'ascription comme une forme dégénérée de scellage.

Le **scellage minimal** crée un type abstrait qui reste comparable : $E :_s T$ a le type T et la même pureté que E. Le scellage minimal crée donc une nouvelle forme d'expression pure, et deux expressions scellées minimalement sont comparables. Le scellage minimal crée des équivalences de types incidentales, lorsque la même expression est scellée deux fois au même type. Une variante du scellage minimal consiste à déclarer $E :_s T$ équivalent à $E :_s T'$ dès lors que les deux sont bien formés. Une autre variante consiste à distinguer une famille de scellages minimaux indexée par un nom, en considérant deux scellés d'une même expression comme équivalents seulement s'ils portent le même nom.

Remarquons que le scellage minimal peut être émulé à l'aide du scellage statique. Il suffit que la bibliothèque standard fournisse un foncteur applicatif

$$\begin{aligned} f_{\mathtt{minseal}} = \lambda t : \text{TYPE}.\ & \lambda x : \text{STRING}. \\ & ((t, ((\lambda x : \mathsf{Typ}\, t.\ x), (\lambda x : \mathsf{Typ}\, t.\ x))) :: \\ & \ \Sigma t' : \text{TYPE}. (\mathsf{Typ}\, t \to \mathsf{Typ}\, t') * (\mathsf{Typ}\, t' \to \mathsf{Typ}\, t)) \end{aligned}$$

soit dans la syntaxe d'Objective Caml

```
let MinSeal = functor (A : sig type t end) ->
  struct  type t = A.t  let a x = x     let c x = x     end :
  sig     type t        val a : t->A.t  val c : A.t->t  end
end
```

[19] Attention aux notations : nous voyons les annotations d'effet comme indiquant les effets présents, tandis qu'ils regardent les puretés, ce qui nous conduit à noter `S` (effet statique) ce qu'ils notent `D` (dynamiquement pur) et `I` (effet dynamique) ce qu'ils notent `S` (statiquement pur).

Pour chaque type T et chaque nom x, $f_{\mathtt{minseal}}\,\langle \mathsf{T}\rangle\, x$ fournit un type abstrait et des fonctions de conversion entre ce type abstrait et T. (En Objective Caml, il faudrait définir une fois pour toutes un module `module` $\mathtt{M}_{\mathsf{T}}^{nom}$ `= struct type t =` T `end` pour chaque type T et chaque nom *nom*, car les structures y sont forcément génératives.) Ceci définit la variante nominale du scellage minimal ; le scellage minimal de base s'obtient en supprimant le paramètre x. Comme nous l'avons remarqué à la section IV.4.4.1, $f_{\mathtt{minseal}}$ pourrait également être définie à l'aide du scellage dynamique.

Nous nous sommes limités dans notre exposition à deux formes de scellage seulement parce qu'à elles deux, plus le scellage minimal qui est facilement définissable, elles nous semblent correspondre précisément aux usages que l'on fait en ML des types abstraits. Nous avons en effet dégagé à la section I.1.2.6 trois catégories d'utilisation des types abstraits :

- les types de données abstraits, dans lesquels l'abstraction sert à assurer des propriétés algébriques qui dépassent les capacités du système de types : le scellage statique leur convient ;
- les variantes nommées des types isomorphes (par exemple `dollar` et `euro`) : le scellage minimal nominal y est adapté ;
- les types protégeant une ressource : le scellage dynamique leur correspond.

IV.4.4.5 Encodages mutuels des scellages statique et dynamique

Pouvons-nous aller encore plus loin, et nous limiter à une seule forme de scellage ? La réponse est « oui, mais », à double titre : scellage statique et scellage dynamique peuvent se coder l'un en fonction de l'autre, mais dans chaque cas une transformation globale du programme est nécessaire.

Commençons par exprimer le scellage statique en fonction du scellage dynamique. Nous partons de deux observations. D'une part, les deux formes de scellages sont équivalentes si elles sont exécutées exactement une fois. D'autre part, nous avons vu à la section IV.4.4.3 qu'il suffit d'étudier le scellage statique des expressions liées localement $\mathsf{let}\, x = \mathsf{E}_0 :: \mathsf{T}_0\, \mathsf{in}\, \mathsf{E} :^{\gamma} \mathsf{T}$.

Nous pouvons voir n'importe quel programme comme une suite de liaisons locales : $\mathsf{let}\, x_1 = \mathsf{E}_1 :: \mathsf{T}_1\, \mathsf{in} \ldots \mathsf{let}\, x_k = \mathsf{E}_k :: \mathsf{T}_k\, \mathsf{in}\, \mathsf{E}$ (le type retourné par chaque « let » est omis car sans importance). Nous appellerons E le corps du programme, et nous dirons également qu'une liaison locale ainsi mise en évidence est *au niveau supérieur*. Nous dirons que le programme est en forme *à scellages prénexes* si les expressions E_1, ..., E_k ne contiennent pas de scellage statique. Nous dirons enfin qu'un programme est *propre* si les seuls scellage statiques sont ceux mis en évidence dans les liaisons locales au niveau supérieur, c'est-à-dire que E ne contient pas de scellage statique. Dans un programme propre, les scellages statiques peuvent être remplacés par des scellages dynamiques sans affecter le typage. Nous allons montrer comment transformer tout programme en un programme propre équitypable.

Soit $\mathsf{E}_0 :: \mathsf{T}_0$ est une sous-expression du programme, c'est-à-dire que le corps du programme est $\mathsf{E}_0 :: \mathsf{T}_0$ dans un contexte C, ce que nous notons $\mathsf{E} = \mathsf{C} \cdot (\mathsf{E}_0 :: \mathsf{T}_0)$. Nous pouvons remplacer $\mathsf{E}_0 :: \mathsf{T}_0$ par $\mathsf{E}'_0 = (\lambda y_1 : \mathsf{T}_1.\ \ldots \lambda y_j : \mathsf{T}_j.\ \mathsf{E}_0 :: \mathsf{T}_0 \ldots)\, y_1 \ldots y_j$ où y_1, ..., y_j sont les variables liées par le contexte C, de l'extérieur vers l'intérieur, en omettant les variables liées au niveau supérieur. Si $j = 0$, nous prenons à la place $j = 1$, $\mathsf{T}_1 = \textsc{unit}$ et $\mathsf{E}'_0 = (\lambda y : \textsc{unit}.\ \mathsf{E}_0 :: \mathsf{T}_0)\,()$. Dans tous les cas, E'_0 est l'application d'une lambda-abstraction à un ou plusieurs paramètres. La lambda-abstraction en question ne contient de plus pas d'autre variable libre que celles liées au niveau supérieur. Nous pouvons l'extraire du contexte C pour la lier au-dessus, en réécrivant par bêta-expansion $\mathsf{E} = \mathsf{C} \cdot (\mathsf{E}_0 :: \mathsf{T}_0)$ en $\mathsf{E}' = (\mathsf{let}\, f = \mathsf{E}'_0\, \mathsf{in}\, \mathsf{C} \cdot (f\, y_1 \ldots y_j))$ où f est une variable fraîche. Or nous avons vu que l'on peut toujours sortir un scellage statique d'une lambda-abstraction : E'_0 est équitypable avec une expression de la forme $\mathsf{E}''_0 :: \mathsf{T}''_0$. Posons $\mathsf{E}'' = (\mathsf{let}\, f = \mathsf{E}''_0 :: \mathsf{T}''_0\, \mathsf{in}\, \mathsf{C} \cdot (f\, y_1 \ldots y_j))$. À condition que E_0 ne contienne pas de scellage statique, E'' est en forme à scellages prénexes, et nous avons augmenté de 1 le nombre de liaisons au niveau supérieur du programme initial.

En effectuant la transformation que nous venons de décrire autant de fois qu'il n'y a de scellage statique dans le programme initial (de l'intérieur vers l'extérieur), nous pouvons mettre tout programme en forme à scellages prénexes. Si nous remplaçons chaque scellage statique au niveau supérieur par un scellage dynamique, nous obtenons un programme équivalent au programme initial et qui ne contient pas de scellage statique. L'idée intuitive de cette transformation est que chaque scellage statique crée une unique famille de types abstraits, et nous le déplaçons afin qu'il soit exécuté une seule fois.

Nous allons maintenant chercher à exprimer le scellage dynamique en fonction du scellage statique. Le scellage dynamique se distingue du scellage statique par son effet $\mathtt{I}$. Or un effet peut aussi être forcé par l'application d'un foncteur génératif, et un foncteur génératif peut être créé par scellage (même statique) d'un foncteur quelconque. Ceci conduit D. Dreyer, K. Crary et R. Harper [DCH03] à proposer l'encodage suivant du scellage fort (presque notre scellage dynamique) dans le scellage faible (identique à notre scellage statique) :

$$\mathsf{E} :> \mathsf{T} = ((\lambda \mathsf{x} : \textsc{unit}.\ \mathsf{E}) :: (\Pi \mathsf{x} : \textsc{unit}.\ {}^{\mathtt{I}}\mathsf{T}))\,()$$

À la différence du scellage dynamique E !! T, le scellage fort $\mathsf{E} :> \mathsf{T}$ a un effet statique en plus de son effet dynamique : $\mathsf{E} :> \mathsf{T}$ est équitypable avec E !! $\mathsf{T} :: \mathsf{T}$. Cet effet ne peut pas être déchargé par une lambda-abstraction, ce qui fait que les foncteurs génératifs ne sont pas pur chez D. Dreyer, K. Crary et R. Harper [DCH03]. Nous pouvons toutefois effectuer déplacer les scellages statiques gênants au niveau supérieur, en appliquant la transformation décrite ci-dessus.

Pour résumer, scellage statique et scellage dynamique peuvent se réécrire l'un en l'autre, mais au prix d'une transformation globale. Cela signifie que nous pouvons nous contenter d'étudier en détail l'un d'entre eux. Nous avons retenu dans cette thèse le scellage dynamique, à nos yeux plus simple grâce à sa sémantique directement compositionnelle. Dans un langage de programmation, il serait judicieux de fournir les deux puisque l'expression de l'un en fonction de l'autre demanderait un travail certain au programmeur.

IV.4.4.6 Observations sur l'applicativité par scellage de foncteur

L'observation que la position du scellage détermine la nature applicative ou générative du foncteur est relativement rare dans la communauté ML. Elle nécessite que l'on puisse sceller un foncteur, et pas seulement une structure comme cela était le cas initialement en ML. En Objective Caml, où les foncteurs sont systématiquement applicatifs, l'usage veut que l'on scelle le corps du foncteur, et sceller le foncteur lui-même est généralement considéré comme équivalent (mais inutilement compliqué puisqu'il faut alors répéter la signature de l'argument) [Ler]. Notons que dès lors que l'on interprète le scellage comme un effet, il n'est pas surprenant que celui-ci ne commute pas avec la lambda-abstraction.

Les premiers systèmes de modules pour ML ne définissaient que le scellage d'une structure, et les possibilités offertes par le scellage d'un foncteur ne sont apparues que progressivement et avec un profil bas. C. Russo [Rus98] distingue les foncteurs applicatifs des foncteurs génératifs par leur définition plutôt que par leur signature, avec l'inconvénient qu'un foncteur génératif peut être directement vu comme un foncteur applicatif [Dre02] comme vu à la section I.2.2.4. Z. Shao [Sha99] remarque brièvement que sceller un foncteur transparent permet de construire un foncteur applicatif. Cette possibilité existe également chez D. Dreyer, K. Crary et R. Harper [DCH03, Dre05], mais ils recommandent plutôt l'utilisation du scellage faible pour former des foncteurs applicatifs.

Un argument en faveur du scellage faible ([DCH03], §2, p. 7) est qu'il peut être appliqué à un membre d'une structure dans le corps d'un foncteur, rendant ainsi des types abstraits dès les membres suivants, alors que sceller le foncteur ne rend les types abstraits qu'une fois le foncteur appliqué. Dans nos notations, la discussion concerne des foncteurs de la forme $\lambda x : T_0.\ \mathsf{let}\, y = E_1 :: T_1 \,\mathsf{in}\, E_2 : T_2$. Nous avons vu à la section IV.4.4.5 que c'est justement le cas où la transformation du scellage statique en un scellage dynamique nécessite une réorganisation du code.

D. Dreyer [Dre05] (§1.2.7) cite en particulier le cas d'un foncteur dont le corps définit et utilise un type déclaré (déclaration `datatype` en Standard ML, type génératif variant ordinaire ou enregistrement en Objective Caml). Si les types déclarés sont modélisés par un type abstrait obtenu par scellage dynamique, un tel foncteur est automatiquement génératif. Mais nous ne voyons aucune raison de choisir ici le scellage dynamique : le scellage minimal suffit, le caractère génératif des types déclarés servant essentiellement à ne pas confondre les noms de constructeurs et de destructeurs des différents types déclarés. Comme la modélisation du scellage minimal par le scellage dynamique est très simple, ce cas n'invalide pas à nos yeux l'absence de scellage autre que dynamique.

IV.5 Couleurs et crochets $\boxed{\mathcal{C}}$

Dans le système $\mathcal{E}$, la construction de scellage influence le typage, mais pas l'évaluation d'un programme, comme en témoigne la règle de réduction afférente :

$$E \,!!\, T \longrightarrow E \qquad (\mathcal{E}/\mathsf{ered.seal})$$

Cette règle est correcte, dans le sens où si le membre de gauche est bien typé, alors E a le type T, c'est-à-dire que le membre de droite est bien typé et a le type du membre de gauche. Elle présente toutefois un défaut : elle manifeste une perte d'information, puisque l'abstraction créée par le scellage n'est pas prise en compte dans la suite de la réduction. Ceci n'est pas un problème tant que l'on respecte une stricte séparation des phases entre typage et évaluation ; mais si nous voulons ajouter une construction de typage dynamique, une telle perte d'information est nuisible. Dans la présente section, nous allons construire le système $\mathcal{C}$, basé sur le système $\mathcal{E}$ et dans lequel la réduction du scellage préserve l'abstraction.

IV.5.1 Empreintes

IV.5.1.1 Génération d'apax

De fait, la règle $(\mathcal{E}/\mathsf{ered.seal})$ n'est pas fidèle à notre intention quant à la sémantique du scellage : nous avons en effet décrit celui-ci comme créant un nouveau type. Considérons l'expression $(\langle \text{INT} \rangle, 3) \,!!\, \Sigma t : \text{TYPE}.\ \mathsf{Typ}\, t$: elle se réduit par $(\mathcal{E}/\mathsf{ered.seal})$ en $(\langle \text{INT} \rangle, 3)$, qui admet le type $\Sigma t : \mathsf{S}(\langle \text{INT} \rangle).\ \mathsf{Typ}\, t = \mathsf{S}(\langle \text{INT} \rangle) * \text{INT}$, alors que nous souhaiterions un type $\Sigma t : \mathsf{S}(\langle T' \rangle).\ \mathsf{Typ}\, t = \mathsf{S}(\langle T' \rangle) * T'$ où le type T' est un type distinct de tous les types existant jusqu'alors, en particulier de INT.

Plus précisément, ce n'est pas un nouveau type que l'on doit créer, mais une nouvelle identité de module, comme on peut le voir en observant le scellage d'un module plus complexe. Par exemple, même si le module scellé définit plusieurs types abstraits, ces types sont rattachés à une même identité. Si le module scellé est un foncteur, une nouvelle identité de module est créée une seule fois lors de l'évaluation de la construction de scellage, et cette même identité est conservée lorsque le foncteur est appliqué. Chaque identité de module caractérise une instanciation de l'abstraction, qui peut produire un nombre quelconque de types (autant que de champs types dans une structure, un nombre non borné pour un foncteur...).

La figure IV.4 donne quelques exemples d'utilisation de nouvelles identités de modules. Ces identités de modules sont notées a et appelées **apax** (*nonce*). Ces apax ont la même propriété d'unicité universelle que ceux utilisés dans la sémantique de la sécurité. Les apax sont une généralisation de la notion de tampon (*stamp*) de D. MacQueen [AM91] ; par rapport à d'autres systèmes de modules

Si T est de la forme...	alors E !! T se réduit en une expression de type...
$\Sigma \mathsf{t} : \textsc{type}. (\textsc{int} \to \mathsf{Typ}\,\mathsf{t})$	$\mathsf{S}(\pi_1\,\mathsf{a}) * (\textsc{int} \to \mathsf{Typ}\,\pi_1\,\mathsf{a})$
$\Sigma \mathsf{t} : \textsc{type}. \Sigma \mathsf{t}' : \textsc{type}. (\mathsf{Typ}\,\mathsf{t} * \mathsf{Typ}\,\mathsf{t}' * \textsc{int})$	$\mathsf{S}(\pi_1\,\mathsf{a}) * \mathsf{S}(\pi_1\,\pi_2\,\mathsf{a}) * \mathsf{Typ}\,\mathsf{t} * \mathsf{Typ}\,\mathsf{t}' * \textsc{int}$
$\Pi \mathsf{x} : \mathsf{T}_0. \Sigma \mathsf{t} : \textsc{type}. (\mathsf{Typ}\,\mathsf{t} * \textsc{int})$	$\Pi \mathsf{x} : \mathsf{T}_0. \mathsf{S}(\pi_1\,(\mathsf{a}\,\mathsf{x})) * \mathsf{Typ}\,\mathsf{t} * \textsc{int}$

Dans chaque cas, a est l'apax (nouvelle identité de module) créé par le scellage.

Fig. IV.4 – Exemples de création d'apax

utilisant des tampons, nous attirons l'attention du lecteur sur le fait que nos apax peuvent désigner des modules de signature arbitraire, notamment des foncteurs, et que la création d'un tampon est attachée directement à l'évaluation de la construction de scellage. Les apax correspondent aux empreintes singularisées que nous avons décrites à la section II.6.1.2.

Au niveau syntaxique, nous allons pour l'instant admettre l'apax comme une nouvelle construction dans la syntaxe des modules, qui n'apparait en principe pas dans un programme source[20]. Néanmoins le rôle des apax est avant tout de désigner des types abstraits, ce qui nous conduira à encadrer leur présence syntaxique à la section IV.5.1.4. Nous supposons disposer d'une réserve infinie d'apax, similaire à la réserve de noms de variables.

IV.5.1.2 Lexiques

L'évaluation de E !! T doit faire apparaître un nouvel apax, c'est-à-dire un apax qui n'est pas présent dans l'expression originale. Par analogie avec l'allocation de cellules mémoires, nous maintenons un stockage d'apax, appelé **lexique** et noté B. Un lexique garde la trace des apax, ainsi que du module et de la signature qui en sont à l'origine ; un lexique a donc la forme

$$\mathsf{B} = (\mathsf{a}_1 = \mathsf{E}_1 : \mathsf{T}_1, \ldots, \mathsf{a}_k = \mathsf{E}_k : \mathsf{T}_k)$$

Nous traitons la concaténation de lexiques comme associative, et le lexique vide (noté nil) comme un élément neutre pour cette opération, comme nous le faisons pour les environnements (voir la section IV.2.3.3).

Les lexiques viennent annoter aussi bien les jugements d'évaluation que ceux de typage. Au niveau du typage, un apax a est valide, et a le type T, lorsque le lexique courant contient $\mathsf{a} = \mathsf{E} : \mathsf{T}$ pour un certain E (de même qu'une variable x est valide, et a le type T, lorsque l'environnement courant contient $\mathsf{x} : \mathsf{T}$). La relation de réduction du système $\mathcal{C}$ est définie sur les couples formés d'un apax et d'une expression : nous noterons une réduction $\mathsf{B} \vdash \mathsf{E} \longrightarrow \mathsf{B}' \vdash \mathsf{E}'$; cependant, lorsque le lexique n'intervient ni ne change dans la réduction, ce qui est le cas le plus courant, nous continuerons à noter simplement $\mathsf{E} \longrightarrow \mathsf{E}'$. La réduction d'un scellage ajoute un élément au lexique :

$$\mathsf{B} \vdash \mathsf{E} \mathbin{!!} \mathsf{T} \longrightarrow \mathsf{B}, \mathsf{a} = \mathsf{E} : \mathsf{T} \vdash \mathsf{E}'$$

L'apax a est choisi frais, au sens où il n'apparaît pas dans B. Comme il n'y a pas de lieur d'apax, les seuls apax qui apparaissent dans E ou T doivent être répertoriés par B.

Plutôt qu'un stockage global, nous pourrions utiliser une forme de lieur pour gérer la fraîcheur des apax, en notant classiquement $(\nu \mathsf{a} = \mathsf{E}_0 : \mathsf{T}_0)\mathsf{E}$. Nous préférons largement le stockage global parce que non seulement nous n'avons aucun besoin de lieur d'apax, mais ceux-ci seraient même difficiles d'emploi car ils devraient migrer au-dessus des environnements (un apax peut apparaître dans le type d'une variable liée).

[20] *A priori* ce dernier point pourrait être problématique, en ce qu'il existe des signatures que l'on ne peut pas écrire (un écueil fréquent dans les systèmes de modules), mais il n'en est rien parce que les apax sont inutiles pour typer des expressions qui ne contiennent pas elles-même d'apax.

IV.5.1.3 Du scellage aux crochets

Nous avons vu que la réduction d'un scellage $\mathsf{E} \mathbin{!!} \mathsf{T}$ crée un nouvel apax a, et que le résultat est une certaine expression E' de type T', le type T' utilisant l'apax a comme identité du module ainsi créé afin de préciser les parties de T qui y sont abstraites.

Le type T' est appelé **renforcement** (*strengthening*, ou plus particulièrement *selfification* ; voir la section I.2.2.2) du type T par l'apax a, et nous le noterons $\mathbf{self}^{\,\mathsf{T}}(\mathsf{a})$. Le principe général du renforcement est de conserver la structure du type original, mais de remplacer les parties inconnues par une référence à l'apax créé par le scellage. La figure IV.4 présente quelques exemples de renforcement ; nous aborderons la tâche d'une définition précise à la section IV.5.1.5.

Le scellage doit donc transformer l'expression E de type T en une expression E' qui a essentiellement le même comportement que E, mais un type différent $\mathbf{self}^{\,\mathsf{T}}(\mathsf{a})$. Le type $\mathbf{self}^{\,\mathsf{T}}(\mathsf{a})$ est plus précis que T, c'est donc un sous-type de T. Bien que T ne soit pas forcément le type le plus précis de E, il n'y a aucun espoir pour que E ait le type $\mathbf{self}^{\,\mathsf{T}}(\mathsf{a})$ en général : en effet, si l'on scelle deux fois la même expression, l'on veut obtenir des expressions E_1' et E_2' ayant respectivement les types $\mathbf{self}^{\,\mathsf{T}}(\mathsf{a}_1)$ et $\mathbf{self}^{\,\mathsf{T}}(\mathsf{a}_2)$, incompatibles. La construction de E' doit donc faire intervenir l'apax a.

La manière la plus évidente de construire E' est de partir de E et de lui appliquer une conversion de type : $\mathsf{E}' = \mathsf{coerce}\ \mathsf{E}\ \mathsf{to}\, \mathbf{self}^{\,\mathsf{T}}(\mathsf{a})$. Encore faut-il savoir gérer cette opération de conversion par rapport au reste du langage. D'une part, il faut savoir réduire une expression de la forme $\mathsf{coerce}\ \mathsf{E}\ \mathsf{to}\ \mathsf{T}'$. D'autre part, cette écriture porte très peu d'information — quand est-ce que $\mathsf{coerce}\ \mathsf{E}\ \mathsf{to}\ \mathsf{T}'$ est bien typé ?

Nous avons déjà abordé le problème dans le cadre plus simple du langage HAT : c'était l'objet de la section III.1.1. Notre solution d'alors, que nous adoptons également ici, est l'utilisation de **crochets colorés**. L'expression $[\mathsf{E}]_{\mathsf{a}}^{\mathbf{self}^{\,\mathsf{T}}(\mathsf{a})}$ désigne la conversion de E vers le type $\mathbf{self}^{\,\mathsf{T}}(\mathsf{a})$, en enregistrant le fait que cette conversion est justifiée par l'équivalence de a avec son implémentation E (cette dernière étant, rappelons-le, enregistrée dans le lexique ambiant). Plus généralement, si E_1 est une expression quelconque, et que T_2 est équivalent au type de E_1 sous réserve de considérer a et son implémentation comme équivalents, l'expression $[\mathsf{E}_1]_{\mathsf{a}}^{\mathsf{T}_2}$ (appelée crochet coloré entourant E_1 portant les annotations de couleur a et de type T_2) a le type T_2.

L'expression E_1 est vue comme à l'*intérieur* du crochet, tandis que a et T_2 sont des annotations portées par ledit crochet. Les apax a jouent ici le même rôle que les empreintes h dans HAT ; ils sont des empreintes singularisées, traduisent une vision générative du monde, contrairement à la vision applicative des empreintes structurelles de HAT. La couleur a traduit la possibilité d'utiliser une équation de typage supplémentaire, à savoir l'équivalence entre a et son implémentation E : nous dirons que a est transparent au niveau du crochet. Nous reviendrons plus longuement sur la syntaxe et la sémantique des couleurs dans le système $\mathcal{C}$ à la section IV.5.2.

Le scellage d'une expression prend ainsi la forme suivante :

$$\mathsf{B} \vdash \mathsf{E} \mathbin{!!} \mathsf{T} \longrightarrow \mathsf{B}, \mathsf{a} = \mathsf{E} : \mathsf{T} \vdash [\mathsf{E}]_{\mathsf{a}}^{\mathbf{self}^{\,\mathsf{T}}(\mathsf{a})}$$

La suite de la réduction, comme dans le langage HAT (voir la section III.1.2.2), consiste à pousser les crochets vers l'intérieur de E.

IV.5.1.4 Types abstraits

Dans l'expression $[\mathsf{E}]_{\mathsf{a}}^{\mathbf{self}^{\,\mathsf{T}}(\mathsf{a})}$, l'apax a est cantonné à deux positions : en annotation de couleur sur le crochet, et dans la fabrication de l'annotation de type sur ce même crochet. Il apparaît donc possible de limiter la présence syntaxique des apax. Le faut-il ?

Considérer un apax a comme une expression à part entière a un avantage net en termes de simplicité. Sous un lexique contenant $\mathsf{a} = \mathsf{E} : \mathsf{T}$, l'expression a a le type T, et la transparence de l'apax a peut s'exprimer simplement par la conversion $\mathsf{a} \longrightarrow \mathsf{E}$.

L'usage non restreint des apax a néanmoins deux défauts, l'un théorique, l'autre pratique. Les deux peuvent être vus comme un problème dans la réduction de certaines expressions contenant un apax.

Considérons par exemple le scellage $\mathsf{E} \mathbin{!!} \mathsf{T}$ où $\mathsf{E} = (\langle \textsc{int} \rangle, (2,3))$ et $\mathsf{T} = \Sigma \mathsf{t} : \textsc{type}.\ \mathsf{Typ}\,\mathsf{t} * \mathsf{Typ}\,\mathsf{t}$, qui produit une entrée de lexique $\mathsf{a} = \mathsf{E} : \Sigma \mathsf{t} : \mathsf{S}(\pi_1\,\mathsf{a}).\ \mathsf{Typ}\,\mathsf{t} * \mathsf{Typ}\,\mathsf{t}$. Si l'on peut admettre que des expressions comme a elle-même ou $\pi_1\,\mathsf{a}$ soient des valeurs — encore que $\pi_1\,\mathsf{a}$ serait une étrange valeur, ayant un destructeur en tête — il n'en va pas de même pour l'expression $\pi_2\,\mathsf{a}$. Au type près, celle-ci devrait avoir le même comportement que $(2,3)$. L'évaluation de $\pi_2(\mathsf{E} \mathbin{!!} \mathsf{T})$ passe par l'expression $[(2,3)]_{\mathsf{a}}^{\mathsf{Typ}\,\pi_1\,\mathsf{a} * \mathsf{Typ}\,\pi_1\,\mathsf{a}}$. Nous pouvons certes réduire a en $[\mathsf{E}]_{\mathsf{a}}^{\Sigma \mathsf{t}:\mathsf{S}(\pi_1\,\mathsf{a}).\ \mathsf{Typ}\,\mathsf{t} * \mathsf{Typ}\,\mathsf{t}}$ et laisser la poussée des crochets suivre son cours ; cependant, pour la préservation du typage, cette dernière expression doit avoir le type $\mathsf{S}(\mathsf{a})$.[21]. L'expérience d'une version préliminaire de ce système a montré que cela complique considérablement la métathéorie du langage car l'analyse des types équivalents à $\mathsf{Typ}\,\pi_1\,\mathsf{a}$ devient impraticable.

Sur le plan pratique, une telle réduction impose d'exhiber l'implémentation de a, qui doit donc être accessible en toute circonstance, alors qu'un apax est intuitivement une donnée opaque, sur laquelle seul un prédicat d'égalité est défini. En particulier, l'implémentation d'un apax pourrait être protégée cryptographiquement (nous y reviendrons à la section IV.5.2.1).

Nous avons vu que le principe du renforcement est de conserver la structure générale du type, mais de remplacer les champs laissés abstraits (type $\textsc{type}$) par la projection adéquate de l'apax. Ainsi un apax n'apparaît que sous forme d'une projection vers la signature $\textsc{type}$. Une telle projection a la forme $\mathsf{Typ}\,A$ où A peut être, outre un apax, une projection de paire $\pi_i\,A$ ou l'application d'un foncteur à un argument $A\,\mathsf{E}_0$. Nous dirons qu'un terme A formé suivant cette grammaire est une **composante** de module.

Nous n'autoriserons dorénavant plus a à être une expression, et remplaçons cette construction par une nouvelle forme de types, les **types composants**, notés $(\!|A|\!)$. Le type $(\!|A|\!)$ remplace $\mathsf{Typ}\,A$; si l'expression A était nécessaire, nous pouvons écrire à la place $\langle (\!|A|\!) \rangle$ (pourvu d'avoir suffisamment projeté pour arriver à un simple champ de signature $\textsc{type}$).

L'expression de la transparence d'un apax revêt une forme plus compliquée, puisque l'on ne peut plus directement exprimer l'équivalence $\mathsf{a} \equiv \mathsf{E}$ (où E est l'implentation de a). Elle doit s'appliquer séparément à chaque type composant : lorsque l'apax sous-jacent a de A_1, noté $\mathbf{underl}(A_1)$, est transparent, $(\!|A_1|\!)$ est équivalent à $\mathsf{Typ}\,\mathsf{E}_1$ où E_1 est la projection de E ayant la forme indiquée par A_1. Nous dirons que A_1 se **révèle** en E_1, et noterons $\mathsf{E}_1 = \mathbf{reveal}^{\mathsf{B}}(A_1)$ (B étant le lexique ambiant). Par exemple, si $\mathsf{T} = \Sigma \mathsf{t}_1 : \textsc{type}.\ \Sigma \mathsf{t}_2 : \textsc{type}.\ \Sigma \mathsf{t}_3 : \mathsf{S}(\langle \textsc{int} \rangle).\ \mathsf{T}_4$, les équivalences fournies par la transparence de a sont $(\!|\pi_1\,\mathsf{a}|\!) \equiv \mathsf{Typ}\,\pi_1\,\mathsf{E}$ et $(\!|\pi_1\,\pi_2\,\mathsf{a}|\!) \equiv \mathsf{Typ}\,\pi_1\,\pi_2\,\mathsf{E}$.

IV.5.1.5 Renforcement

L'opération de scellage consiste à remplacer dans un type les composantes qui désignent un type abstrait par un type concret qui est la projection adéquate d'un apax fixé. De manière générale, le renforcement est d'un type T par une composante A, et noté $\mathbf{self}^{\,\mathsf{T}}(A)$.

Cas de base Nous pouvons distinguer trois sortes de signatures élémentaires : les champs types manifestes $\mathsf{S}(\langle \mathsf{T}_1 \rangle)$ (`sig type t = T1 end`), les champs types abstraits $\textsc{type}$ (`sig type t end`), et

[21] notons que a doit bien être pur, étant destiné à être utilisé dans des types.

les champs termes (`sig val x : T2 end`). Le but du renforcement est de transformer les champs types abstraits en champs types manifestes : le renforcement de TYPE par A est $\mathsf{S}(\langle (\!|A|\!) \rangle)$. Les champs types déjà manifestes $\mathsf{S}(\langle \mathsf{T}_1 \rangle)$ restent inchangés ; il en est de même pour les champs termes, pour lesquelles une information plus précise n'est pas utile. Le renforcement donne un type précis (singleton) aux champs types, et ne modifie pas le typage sur les champs termes.

Structures Le renforcement d'une structure en conserve la décomposition en champs, renforçant seulement chaque champ. Par exemple, en notation ML :

```
sig
  type t1
  type t2
  type u = int * t1
  val f : int -> t1 -> u
end
```

renforcée au nom M donne

```
sig
  type t1 = M.t1
  type t2 = M.t2
  type u = int * t1
  val f : int -> t1 -> u
end
```

On remarque que le nom du module apparaît plusieurs fois : c'est le même nom M qui est utilisé dans la désignation des différents types abstraits de cette signature.

Le renforcement d'une paire $\mathsf{T}_1 * \mathsf{T}_2$ par A est naturellement $\textbf{self}^{\,\mathsf{T}_1}(\pi_1 A) * \textbf{self}^{\,\mathsf{T}_2}(\pi_2 A)$. Dans le cas d'une paire dépendante, une généralisation naturelle est de continuer à renforcer indépendamment chaque composante, soit $\textbf{self}^{\,\Sigma x : \mathsf{T}_1.\, \mathsf{T}_2}(A) = \Sigma x : \textbf{self}^{\,\mathsf{T}_1}(\pi_1 A).\, \textbf{self}^{\,\mathsf{T}_2}(\pi_2 A)$.

On pourrait toutefois aller plus loin : puisque x est désormais connu, pourquoi ne pas le substituer dans T_2 ? Ainsi, dans l'exemple ci-dessus, l'on pourrait remplacer les références à `t1` et `u` par `M.t1` et `M.u` respectivement. Mais ceci n'est pas possible dans un langage tel que le nôtre, qui inclut des signatures que ML ne saurait exprimer, en particulier des dépendances sur des champs termes. Si la première composante contient des champs termes, sa signature a des parties non entièrement spécifiées, et nous ne pouvons pas aller plus loin dans la deuxième composante — par exemple, le renforcement de $\Sigma x : \text{INT}.\, \mathsf{S}(x)$ par a est $\Sigma x : \text{INT}.\, \mathsf{S}(x)$. En général, le renforcement d'une paire dépendante reste donc une paire dépendante.

Foncteurs La notation TYPE a deux interprétations radicalement différentes dans l'argument et dans le résultat d'un foncteur. Dans l'argument, elle désigne un type qui ne sera connu que lors de l'application du foncteur, et qui peut varier d'une application à l'autre : le foncteur est polymorphe. Le renforcement d'un type foncteur $\Pi x : \mathsf{T}_0.\, {}^{\gamma}\mathsf{T}_1$ ne restreint pas le domaine autorisé aux arguments du foncteur[22] ; il a donc la forme $\Pi x : \mathsf{T}_0.\, {}^{\gamma}\mathsf{T}_1'$. L'interprétation de TYPE comme type résultat d'un foncteur dépend du caractère applicatif ou non du foncteur. S'agissant d'un foncteur applicatif, le type effectif est déterminé par l'argument passé au foncteur ; le renforcement fait alors appel à une êta-expansion du corps, ayant la forme $\Pi x : \mathsf{T}_0.\, {}^{\gamma}\textbf{self}^{\,\mathsf{T}_1}(A\, x)$. En revanche, dans le cas d'un foncteur génératif, chaque application du foncteur génère une identité différente ; le renforcement est en fait impossible tant que cette identité n'est pas connue, et T_1 doit conserver toute son abstraction.

Définition Formellement, le renforcement est défini par induction structurelle sur le type :

$$
\begin{array}{rcll}
\textbf{self}^{\,\mathsf{BT}}(A) & = & \mathsf{BT} & \text{si BT est un type de base (UNIT, BOOL, INT)} \\
\textbf{self}^{\,\Sigma x:\mathsf{T}_1.\,\mathsf{T}_2}(A) & = & \Sigma x : \textbf{self}^{\,\mathsf{T}_1}(\pi_1 A).\, \textbf{self}^{\,\mathsf{T}_2}(\pi_2 A) & \\
\textbf{self}^{\,\Pi x:\mathsf{T}_0.\,{}^{\mathsf{P}}\mathsf{T}_1}(A) & = & \Pi x : \mathsf{T}_0.\, {}^{\mathsf{P}}(\textbf{self}^{\,\mathsf{T}_1}(A\, x)) & \\
\textbf{self}^{\,\Pi x:\mathsf{T}_0.\,{}^{\mathsf{I}}\mathsf{T}_1}(A) & = & \Pi x : \mathsf{T}_0.\, {}^{\mathsf{I}}\mathsf{T}_1 & \\
\textbf{self}^{\,\mathsf{S}(E')}(A) & = & \mathsf{S}(E) & \\
\textbf{self}^{\,\text{TYPE}}(A) & = & \mathsf{S}(\langle (\!|A|\!) \rangle) &
\end{array}
$$

[22]Le renforcement d'un type $\Pi x : \mathsf{T}_0.\, {}^{\gamma}\mathsf{T}_1$ conserve de toutes façon la structure, donc a la forme $\Pi x : \mathsf{T}_0'.\, {}^{\gamma}\mathsf{T}_1'$.

Évaluation des champs types La table ci-dessus n'indique pas comment calculer **self** $^{\mathsf{Typ}\,\mathsf{E}}(\mathsf{A})$. La raison est que cela dépend de la valeur de E : la définition ne saurait être purement syntaxique. Intuitivement, les renforcements de deux types équivalents devraient être équivalents, ce qui implique en particulier **self** $^{\mathsf{Typ}\,\langle\mathsf{T}\rangle}(\mathsf{A}) =$ **self** $^{\mathsf{T}}(\mathsf{A})$. Il faudra donc évaluer E en une valeur (forcément de la forme $\langle\mathsf{T}\rangle$ pour des raisons de typage) pour calculer le renforcement de $\mathsf{Typ}\,\mathsf{E}$.

IV.5.2 Couleurs

IV.5.2.1 Colorisation

Crochet coloré Un crochet coloré $[\mathsf{E}]^{\mathsf{T}}_{\mathsf{a}}$ permet d'après leur introduction à la section IV.5.1.3 d'attribuer à l'expression E le type T en utilisant les équations de typage fournies par la connaissance de a. Comme dans HAT, nous aurons une vision plus large des crochets colorés. Ceux-ci maintiennent durant l'exécution du programme la barrière entre l'intérieur et l'extérieur du module initialement exprimée par le programmeur au moyen de la construction de scellage.

Syntaxe colorée Une manière de décrire la syntaxe du système $\mathcal{C}$ à partir de la syntaxe du système $\mathcal{E}$ serait d'associer à chaque nœud de la syntaxe une couleur. Cette couleur représente l'origine du nœud de syntaxe — de quel module scellé il vient. Les crochets donnent cette indication sous une forme différente : la couleur d'un nœud de syntaxe est celle portée par le crochet environnant.

En l'absence d'un crochet environnant, la couleur est la **couleur ambiante**, qui est portée par le jugement dans lequel le terme considéré est placé. Tout compte fait, un jugement de typage d'une expression a dans le système $\mathcal{C}$ la forme suivante :

$$\mathsf{B}; \Gamma \vdash_{\mathsf{c}} \mathsf{E} :^{\gamma} \mathsf{T}$$

Les couleurs régulent également l'évaluation d'expressions contenant des crochets ; une réduction dans $\mathcal{C}$ a finalement la forme suivante :

$$\mathsf{B} \vdash \mathsf{E} \longrightarrow_{\mathsf{c}} \mathsf{B}' \vdash \mathsf{E}'$$

Les couleurs apparaissent aussi à d'autres endroits de la syntaxe — en règle générale, à chaque fois que l'on attribue un type à une expression, il nous faut désormais lui attribuer également une couleur. Une entrée dans un lexique a par exemple la forme définitive $(\mathsf{a} = \mathsf{E} :_{\mathsf{c}} \mathsf{T})$, et une entrée d'environnement a la forme $(\mathsf{x} :_{\mathsf{c}} \mathsf{T})$.

Sémantique de la couleur Au niveau du typage, la couleur détermine quels sont les types abstraits qui peuvent être révélés : conformément à l'intuition présentée à la section IV.5.1.4, ce sont les projections d'apax transparents dans la couleur indiquée. Durant la réduction, une règle ($\mathcal{C}$/ered.colAbs) permet également de dévoiler les apax transparents.

Réduction du scellage Nous sommes désormais en mesure d'exprimer la réduction d'un scellage. Voici par exemple comment s'évalue notre scellage fétiche, celui du module `struct type t = int let x = 3 end` à la signature `sig type t val x : t end` :

$$\begin{array}{l} \mathsf{nil} \vdash ((\langle\text{INT}\rangle, 3) \mathbin{!!} (\Sigma \mathsf{t} : \text{TYPE}.\, \mathsf{Typ}\,\mathsf{t}) \quad \longrightarrow_{\bullet} \\ \qquad \mathsf{a} = ((\langle\text{INT}\rangle, 3) :_{\bullet} (\Sigma \mathsf{t} : \text{TYPE}.\, \mathsf{Typ}\,\mathsf{t}) \vdash [(\langle\text{INT}\rangle, 3)]_{\mathsf{a}}^{\Sigma \mathsf{t}:\mathsf{S}(\langle(\!|\pi_1\, \mathsf{a}|\!)\rangle).\, \mathsf{Typ}\,\mathsf{t}} \end{array}$$

La suite de la réduction consiste à pousser les crochets vers l'intérieur de la valeur, comme dans HAT (voir la section III.1.2.2).

En général, un scellage $\mathsf{V} \mathbin{!!} \mathsf{T}$ s'évalue dans la couleur ambiante c en $[\mathsf{V}]^{\mathsf{T}'}_{\mathsf{c}'}$, où $\mathsf{c}' = \mathsf{c} \cup \{\mathsf{a}\}$ et $\mathsf{T}' = \mathbf{self}^{\,\mathsf{T}}(\mathsf{a})$, a étant un apax frais. Soit la règle de réduction ($\mathcal{C}$/ered.seal)[23] :

$$\mathsf{B} \vdash \mathsf{V} \mathbin{!!} \mathsf{T} \longrightarrow_{\mathsf{c}} \mathsf{B}, \mathsf{a} = \mathsf{V} :_{\mathsf{c}} \mathsf{T} \vdash [\mathsf{V}]^{\mathbf{self}^{\,\mathsf{T}}(\mathsf{a})}_{\mathsf{c}\cup\{\mathsf{a}\}}$$

On remarque que la couleur ambiante c est déjà nécessaire au typage de V par T ; le typage de V par $\mathbf{self}^{\,\mathsf{T}}(\mathsf{a})$ nécessite en plus la transparence de a[24]. Une couleur apparaît ainsi comme un ensemble (fini) d'apax, que nous noterons $\mathsf{c} = \{\mathsf{a}_1, \ldots, \mathsf{a}_k\}$. La sémantique d'une couleur est de rendre ses éléments transparents. Jusqu'à présent, nous n'avions vu que des couleurs singletons $\{\mathsf{a}\}$ abrégées en a ; nous appellerons **couleur primaire** une couleur singleton, ou par synecdoque un élément d'une couleur. La **couleur vide**, notée de préférence $\bullet$, ne rend aucun apax transparent.

Transparence Nous dirons d'un apax qu'il est **transparent** ou **opaque** (dans une couleur qui sera souvent sous-entendue) suivant s'il est ou non un élément de la couleur. (Nous généraliserons la transparence à une définition sémantique pour une forme plus générale de couleurs à la section IV.5.2.3).

Affaiblissement On remarque que la réduction d'un scellage fait passer l'expression scellée d'une couleur à une couleur plus grande. Intuitivement, ceci ne devrait pas poser de problème de typage : la couleur plus grande donne plus d'équations de typage, et qui peut le plus peut le moins. Nous énoncerons effectivement un lemme d'**affaiblissement de couleur** : si E a le type T dans la couleur c, et que c' est une couleur bien formée qui contient c, alors E a le type T dans c'.

Couleur frontière Notons que dans un crochet coloré $[\mathsf{E}]^{\mathsf{T}}_{\mathsf{c}'}$, l'annotation de type T vit à la frontière entre la couleur intérieure c' et la couleur ambiante c. Nous pouvons exprimer cela en disant que T doit être à la fois valide dans c et dans c'. En vertu du lemme d'affaiblissement, une condition suffisante est que T soit valide dans $\mathsf{c} \cap \mathsf{c}'$. C'est cette condition que nous retiendrons.[25]

Couleur et sécurité Signalons ici que les crochets colorés peuvent avoir une interprétation en termes de sécurité. On peut considérer les couleurs comme des capabilités accordées aux expressions ; dans notre cas, ces capabilités autorisent des équations de typage. Les crochets marquent et maintiennent les blocs de code privilégiés. Les apax correspondent alors aux apax habituelles en cryptographie abstraite ; le lien est connu [PS00] et a été notamment étudié sous le nom $\lambda_{\texttt{seal}}$ [SP04].

IV.5.2.2 Sémantique d'un type et dépendances d'un apax

Sémantique d'un type dans une couleur Une couleur exprime des équations entre types : il y a donc potentiellement plusieurs moyens d'exprimer un même type dans une couleur donnée. Si le

[23] Spécialisée au cas d'un scellage incolore — voir la section IV.5.3.6.

[24] Une telle situation est possible ici parce que les scellages peuvent être imbriqués, et qu'un foncteur peut s'interposer empêchant la poussée des crochets introduits par le scellage extérieur avant que l'application du foncteur n'active la réduction du second. Le cas ne se présentait pas dans HAT où les définitions de modules étaient forcément séquentielles.

[25] L'utilisation de $\mathsf{c}\cap\mathsf{c}'$ plutôt que de la duplication d'hypothèses dans c et dans c' présente des avantages techniques, essentiellement parce que le fait que les jugements $\mathsf{B}; \Gamma \vdash_{\mathsf{c}} \mathsf{J}$ et $\mathsf{B}; \Gamma \vdash_{\mathsf{c}'} \mathsf{J}$ soient tous les deux dérivables ne permet pas de conclure que ces jugements aient la même forme. Nous soupçonnons que cela est vrai, et que l'on pourrait également dériver $\mathsf{B}; \Gamma \vdash_{\mathsf{c}\cap\mathsf{c}'} \mathsf{J}$ mais la démonstration dans notre approche syntaxique apparaît très difficile. De plus, en présence de variables (voir la section IV.5.2.3), il faudrait utiliser une intersection sémantique, qui tient compte des fermetures transparentes des variables (voir la section B.1.2).

lexique contient $a_1 = (\langle \text{INT} \rangle, E_1) :_\bullet \Sigma t : \text{TYPE}.\, T_1$, alors $(\!|\pi_1\, a_1|\!)$ est équivalent à INT dans la couleur $\{a_1\}$ mais pas dans la couleur vide $\bullet$. C'est cette possibilité qu'un terme désigne un type valide dans des couleurs différentes mais n'y ayant pas la même sémantique qui donne leur puissance aux crochets colorés : l'annotation de type est considérée d'une part dans la couleur intérieure, d'autre part dans la couleur extérieure. (Si l'on considère que la sémantique d'un type est l'ensemble des termes qui constituent sa classe d'équivalence, la sémantique d'un terme est une fonction croissante de la couleur ambiante ; elle contient toujours le terme de départ, pourvu bien sûr que celui-ci soit valide.)

Sémantique et validité Étant donné qu'un type peut contenir des expressions imbriquées, l'on peut construire des types dont la validité même, et non seulement la sémantique, fait appel à une couleur donnée. En se plaçant toujours sous le lexique ci-dessus, le type $S((\lambda x : (\!|\pi_1\, a_1|\!).\ x)\, 3)$ n'est bien valide que si la couleur ambiante rend a_1 transparent. En apparence, notre système de types est prêt à accomoder ce phénomène : il suffit d'annoter par une couleur chaque jugement de typage, y compris notamment la validité d'un type $(B; \Gamma \vdash_c T\, \mathsf{ok})$.

Entrées de lexique Considérons en particulier une réduction de scellage $B \vdash V \mathbin{!!} T \longrightarrow_c B' \vdash [V]^{\mathbf{self}^{T}(a)}_{c\cup\{a\}}$. Puisque V et T sont ne valides que dans la couleur c, cette dernière doit être enregistrée avec ceux-ci dans le lexique B' : il faut donc écrire $B' = B, a = V :_c T$. Nous appellerons les éléments de c les **dépendances** de l'apax a.

Bonne formation d'une couleur Lorsque a est utilisé dans une couleur, l'équivalence entre projections de a et V doit être valide. La couleur en question doit donc contenir c, en tant que couleur dans laquelle V est connu comme valide. Autrement dit, un apax ne peut être transparent que si ses dépendances le sont également. Nous exprimons cette contrainte par une condition de bord sur la bonne formation d'une couleur (règle ($\mathcal{C}$/$\mathsf{envok.c.a}$)). Une autre possibilité serait de rendre automatiquement les dépendances transparentes (les couleurs $\{a\}$ et $\{a\} \cup c$ auraient donc la même sémantique) ; nous préférons la condition de bonne formation car elle est plus simple et similaire à la condition d'utilisation d'un apax dans une expression.

Utilisation d'un apax dans une expression Lorsque a est utilisé dans une expression, son type T doit être valide. La couleur de l'expression doit donc contenir c, en tant que couleur dans laquelle T est connu comme valide. Autrement dit, un apax ne peut être utilisé que si ses dépendances sont transparentes ; cela se traduit par une hypothèse de transparence dans la règle ($\mathcal{C}$/$\mathsf{ac.a}$).

Concrétisation Nous pouvons envisager une autre approche de l'utilisation d'un apax dans une expression. Plutôt que de lui donner le type T enregistré dans le lexique, qui n'est valide que dans c, nous pouvons construire un type T' valide dans n'importe quelle couleur mais dont la sémantique dans c est la même que T. Dans la couleur c, nous pouvons remplacer n'importe quelle dépendance de a par son implémentation : le type obtenu remplit la condition demandée. Cette opération est appelée **concrétisation** du type T ; l'opération de concrétisation est définie également sur les expressions. Formellement, la concrétisation d'un type ou d'une expression recopie son argument en effectuant les remplacements suivants :

$$\begin{array}{lcll}
\mathbf{conc}^B_c((\!|A_1|\!)) & = & \mathsf{Typ}\,\mathbf{reveal}^B(A_1) & \text{si } \mathbf{underl}(A_1) \in c \\
\mathbf{conc}^B_c((\!|A_1|\!)) & = & (\!|A_1|\!) & \text{si } \mathbf{underl}(A_1) \notin c \\
\mathbf{conc}^B_c([E]^T_{c'}) & = & [E]^{\mathbf{conc}^B_{c\cap c'}(T)}_{c'} &
\end{array}$$

(simple induction dans les autres cas)

On a $\mathsf{B}; \Gamma \vdash_{\mathsf{c}'} \mathsf{T} \equiv \mathbf{conc}_{\mathsf{c}}^{\mathsf{B}}(\mathsf{T})$ dès que c' contient c.

L'intérêt de la concrétisation est que l'apax a (de dépendances c) puisse être utilisé dans n'importe quelle couleur c' (et non seulement si $\mathsf{c} \subseteq \mathsf{c}'$), en lui attribuant le type $\mathbf{conc}_{\mathsf{c}}^{\mathsf{B}}(\mathsf{T})$. Malheureusement, lorsque c n'est pas incluse dans la couleur ambiante, $\mathbf{conc}_{\mathsf{c}}^{\mathsf{B}}(\mathsf{T})$ n'est en général pas équivalent à T même si celui-ci est bien formé, ce qui est source de confusion. Cela nous a conduit à préférer la restriction d'utilisation à une concrétisation forcée.

Concrétisation au scellage Plutôt que de concrétiser le type d'un apax lors de son utilisation, nous pourrions tenter de forcer la concrétisation à la source, lors de la création de l'entrée de lexique. La réduction du scellage s'exprimerait ainsi :

$$\mathsf{B} \vdash \mathsf{V} \mathrel{!!} \mathsf{T} \longrightarrow_{\mathsf{c}} \mathsf{B}, \mathsf{a} = [\mathsf{V}]_{\mathsf{c}}^{\mathbf{conc}_{\mathsf{c}}^{\mathsf{B}}(\mathsf{T})} : \mathbf{conc}_{\mathsf{c}}^{\mathsf{B}}(\mathsf{T}) \vdash [\mathsf{V}]_{\mathsf{c} \cup \{\mathsf{a}\}}^{\mathbf{self}^{\mathsf{T}}(\mathsf{a})}$$

Le type de l'expression scellée est alors forcément universel, c'est-à-dire valide dans n'importe quelle couleur. L'entrée de lexique est également universelle, ce qui dispense de l'annoter par une couleur. La légère perte d'expressivitée occasionnée par la concrétisation forcée est ainsi compensée par une simplification du système.

Malheureusement nous utilisons les crochets colorés hors de leur domaine d'application. En effet, le type T est en général destiné à rendre le module scellé partiellement abstrait, c'est-à-dire que certaines composantes types du module sont spécifiées comme TYPE ; or pour utiliser le module nous devons pouvoir spécifier complètement ce type — nous restreindrons les annotations de types sur les crochets à être monomorphes, c'est-à-dire complètement spécifiées, sans champ TYPE (la section IV.5.3 sera consacrée à une discussion de ce concept).

IV.5.2.3 Couleurs de variables

Nous avons vu que dans HAT, il n'est pas vrai en général que la substitution d'une variable par une valeur bien typée préserve le typage. En effet, il faut exiger que la valeur soit bien typée dans toutes les couleurs auxquelles la variable est utilisée. Une manière de s'assurer de cette propriété est d'associer à chaque variable une couleur, la couleur du nœud de syntaxe qui la lie, et d'exiger que la variable soit utilisée dans cette même couleur ; la préservation du typage par substitution est alors vraie pourvu que la valeur de remplacement soit elle aussi typée dans la couleur en question. Des règles de réduction similaires à celles de HAT assurent la poussée de crochets autour des constructeurs de données (nous examinerons la poussée d'un crochet autour d'une lambda-abstraction à la section IV.5.3.3) ; elles assurent que les occurences d'une variable ne changent pas de couleur au cours de la réduction. Par rapport à HAT, nous gagnons la possibilité d'écrire la bêta-réduction ($\mathcal{C}$/ered.app) directement ; de plus, l'annotation de couleur sur les sites de liaison de variables est cohérente avec le principe énoncé ci-dessus que l'on attribue désormais une couleur à chaque élément auquel on attribue un type. En revanche, nous allons voir que l'attribution d'une couleur à chaque variable a un coût qui dépasse celui des annotations qu'elle entraîne directement.

La couleur d'un site de liaison est indiquée comme d'habitude par le crochet environnant (ou en son absence par la couleur ambiante). Lorsqu'une variable est ajoutée à un environnement, il faut sauvegarder la couleur de sa liaison : une entrée d'environnement dans le système $\mathcal{C}$ a donc la forme $(\mathsf{x} :_{\mathsf{c}} \mathsf{T})$.

A priori le bon typage d'une variable est donné par $\mathsf{x} :_{\mathsf{c}} \mathsf{T} \vdash_{\mathsf{c}} \mathsf{x} :^{\mathsf{P}} \mathsf{T}$ — la variable x est utilisable dans sa couleur de définition. Afin que l'affaiblissement de couleur puisse être déduit des règles de typage, nous devons relâcher cette condition : la variable x est utilisable dans toute couleur contenant sa couleur de définition, soit

$$\mathsf{x} :_{\mathsf{c}} \mathsf{T} \vdash_{\mathsf{c}'} \mathsf{x} :^{\mathsf{P}} \mathsf{T} \quad \text{si } \mathsf{c} \subseteq \mathsf{c}'$$

Malheureusement, cette règle ne suffit pas à assurer l'affaiblissement. Un contre-exemple est le suivant :

$$\mathsf{B}; \mathsf{nil} \vdash_{\mathsf{c}} (\lambda \mathsf{x} : \text{INT}.\ [[\mathsf{x}]_{\mathsf{c}}^{\text{INT}}]_{\mathsf{c}_1}^{\text{INT}}) :^{\mathsf{P}} \text{INT} \to \text{INT}$$

Sous réserve que B, c et c_1 soient bien formés, ce jugement est correct. En particulier, c et c_1 n'ont aucune raison d'avoir un lien particulier. Si nous élargissons la couleur c en une couleur c' (vérifiant $\mathsf{c} \subseteq \mathsf{c}'$), le jugement devient

$$\mathsf{B}; \mathsf{nil} \vdash_{\mathsf{c}'} (\lambda x : \text{INT}.\ [[x]_{\mathsf{c}}^{\text{INT}}]_{\mathsf{c}_1}^{\text{INT}}) :^{\mathsf{P}} \text{INT} \to \text{INT}$$

La couleur du crochet intérieur reste c : en effet, rien n'indique que cette couleur est c pour que x y soit valide, et non par simple coïncidence, donc il n'y a pas lieu de la modifier lors du changement de couleur ambiante.

Une manière de voir ce problème est de considérer que la couleur de l'occurence d'une variable doit être explicitement liée à la couleur de la liaison : les couleurs d'occurences doivent être calculées par nom et non par valeur. Chaque variable a donc une couleur primaire symbolique associée, une « couleur primaire variable » par opposition aux « couleurs primaires constantes » que sont les apax ; cette couleur primaire est notée comme la variable elle-même. Une couleur a donc la forme $\mathsf{c} = \{a_1, \ldots, a_k, x_1, \ldots, x_n\}$: un ensemble fini d'apax et de couleurs[26]. Une variable ne peut être utilisée que si elle est présente dans la couleur ambiante, soit

$$x :_{\mathsf{c}} \mathsf{T} \vdash_{\mathsf{c}'} x :^{\mathsf{P}} \mathsf{T} \quad \text{si } x \in \mathsf{c}'$$

On remarque que la couleur c de définition de x peut elle-même contenir d'autres variables, et elle peut dès le départ contenir des apax. Ces apax et variables sont automatiquement considérés comme transparents dès lors que x est transparente. Autrement dit, si $x \in \mathsf{c}'$ alors tous les éléments de $\mathsf{c} \cup \{x\}$ sont transparents dans c', ce que nous noterons par le jugement $x :_{\mathsf{c}} \mathsf{T} \vdash_{\mathsf{c}'} \mathsf{c} \cup \{x\}\,\mathsf{transparent}$. La condition d'utilisation d'une variable dans une couleur devient finalement

$$\Gamma_0, x :_{\mathsf{c}} \mathsf{T}, \Gamma_1 \vdash_{\mathsf{c}'} x :^{\mathsf{P}} \mathsf{T} \quad \text{si } \Gamma_0, x :_{\mathsf{c}} \mathsf{T}, \Gamma_1 \vdash_{\mathsf{c}'} x\,\mathsf{transparent}$$

Puisqu'il y a désormais des occurences de variables dans les couleurs, celles-ci font l'objet d'alpha-conversion et de substitution. Une substitution indique la couleur cible en plus d'une expression cible ; nous notons $\{x \leftarrow_{\mathsf{c}} \mathsf{E}\}$ la substitution de x par E dans la couleur c, et nous aurons par exemple l'égalité suivante, sous réserve que x appartienne à c :

$$\{x \leftarrow_{\mathsf{c}_0} \mathsf{E}_0\}[x]_{\mathsf{c}}^{\mathsf{S}(x)} = [\mathsf{E}_0]_{(\mathsf{c} \setminus \{x\}) \cup \mathsf{c}_0}^{\mathsf{S}(\mathsf{E}_0)}$$

IV.5.2.4 Crochets absolus, crochets additifs

Un crochet $[\mathsf{E}]_{\mathsf{c}'}^{\mathsf{T}}$ permet d'utiliser l'expression E de couleur c' dans n'importe quelle couleur c. Nous pourrions généraliser la notation à n'importe quelle relation entre la couleur interne et la couleur externe. Si $\mathscr{R}$ est une relation binaire sur l'espace des couleurs, nous noterions $[\mathsf{E}]_{\mathscr{R}}^{\mathsf{T}}$ le crochet coloré de relation $\mathscr{R}$, qui est bien typé dans la couleur c si et seulement si il existe une couleur c' telle que E ait le type T dans c' et $(\mathsf{c}, \mathsf{c}') \in \mathscr{R}$.

Lorsque $\mathscr{R}$ est une relation arbitraire, nous perdons une indication importante : la couleur interne devient ambiguë. Aussi limiterons-nous l'analyse au cas où la relation est une fonction (partielle) de la couleur extérieure à la couleur intérieure : $[\mathsf{E}]_{\mathsf{f}}^{\mathsf{T}}$ est bien typé dans la couleur c si et seulement si E a le type T dans $\mathsf{f}(\mathsf{c})$. Une propriété intéressante pour que l'affaiblissement de couleur soit valide est qu'un affaiblissement de la couleur interne corresponde à un affaiblissement externe, c'est-à-dire que f soit croissante ($\mathsf{c}_1 \subseteq \mathsf{c}_2$ implique $\mathsf{f}(\mathsf{c}_1) \subseteq \mathsf{f}(\mathsf{c}_2)$).

[26] On pourrait noter symboliquement $\mathsf{c} = \{a_1, \ldots, a_k\} \cup \mathsf{col}(x_1) \cup \cdots \cup \mathsf{col}(x_n)$ où le symbole col note la couleur de définition de son argument et le symbole $\cup$ est interprété par l'union ensembliste.

Les crochets colorés tels que nous les avons étudiés jusqu'à présent correspondent au cas particulier où la fonction f est partout définie et constante. De tels crochets sont appelés **crochets absolus**. Pour l'instant, la seule manière par laquelle un crochet est introduit dans une expression est la réduction d'un scellage ; le crochet a une couleur externe arbitraire c_0 et une couleur interne de la forme $c_0 \cup \{a\}$. Plutôt qu'un crochet absolu, nous pourrions utiliser un **crochet additif**, de relation $c \mapsto c \cup \{a\}$ (une injection totale de la couleur intérieure vers la couleur extérieure, donc une injection partielle de la couleur extérieure vers la couleur intérieure[27]). Un tel crochet additif sera noté $[E]^{T}_{+a}$.

Les crochets additifs ont l'avantage de se marier bien avec le reste du langage. En particulier, les occurences de variables sous un crochet additif ne posent pas les problèmes que nous avons soulevé à la section IV.5.2.3 : dans une expression comme $\lambda x : T.\ [[x]^{T}_{+a_2}]^{T}_{+a_1}$, la couleur de l'occurence est automatiquement un sur-ensemble de la couleur de la liaison. Du coup, si nous limitons le langage aux crochets additifs, nous pouvons nous passer de la complication que représentent les variables dans les couleurs.

L'inconvénient des crochets additifs est leur faible expressivité. Ils ne permettent pas de restreindre les équations de typage vues par une sous-expression ; en particulier, ils ne donnent aucun moyen de spécifier qu'une sous-expression est indépendante de son contexte. Si cela n'est pas gênant pour la viabilité intrinsèque du système $\mathcal{C}$, cela en interdit de nombreuses applications. Dans l'interprétation sécuritaire des crochets, un crochet additif ne fait qu'élever les privilèges du code qu'il entoure, ce qui exclut non seulement toute modélisation de bac à sable (*sandbox*) les rappels à du code ordinaire que ferait du code privilégié. Dans le cadre de la présente thèse, nous aurons besoin de crochets universels, qui assurent que leur contenu est typable sans équation particulière, c'est-à-dire des crochets absolus de couleur $\bullet$ (voir la section IV.6.3.1) ; ce besoin motive notre choix des crochets absolus (en présence desquels les autres formes de crochets sont superflues).

IV.5.3 Polymorphisme

IV.5.3.1 Coloration d'un type

Nous avons vu qu'un crochet coloré peut servir à former une valeur d'un type abstrait, selon l'annotation de type sur le crochet. Ainsi, si a est un apax d'implémentation $(\langle \text{INT} \rangle, 3)$ et de signature $\Sigma t : \text{TYPE}.\ \mathsf{Typ}\, t$, l'expression $[3]^{(\!|\pi_1\, a|\!)}_{\{a\}}$ désigne une valeur du type abstrait $(\!|\pi_1\, a|\!)$, tandis que l'expression $[3]^{\text{INT}}_{\{a\}}$ s'évalue en 3. Considérons un crochet coloré autour d'un champ type : $[\langle \text{INT} \rangle]^{\mathsf{S}(\langle (\!|\pi_1\, a|\!) \rangle)}_{\{a\}}$ désigne le champ type abstrait que l'on peut plus simplement écrire $\langle (\!|\pi_1\, a|\!) \rangle$, tandis que $[\langle \text{INT} \rangle]^{\mathsf{S}(\langle \text{INT} \rangle)}_{\{a\}}$ est équivalent au simple $\langle \text{INT} \rangle$. Une autre possibilité qui apparaît est d'écrire $[\langle \text{INT} \rangle]^{\text{TYPE}}_{\{a\}}$. Que signifie ce terme ?

Dans une expression de la forme $[\langle T \rangle]^{\text{TYPE}}_{c'}$, l'annotation portée par le crochet ne permet pas de décider entre la version concrète et la version abstraite du type. L'annotation TYPE n'est pas ici une source d'abstraction — l'abstraction viendrait de l'utilisation d'un nom de type abstrait, c'est-à-dire d'un apax. Il faut plutôt voir dans cette annotation une spécification incomplète, une simple indication de signature et non de type.

L'écriture $[\langle T \rangle]^{\text{TYPE}}_{c'}$ dénote un champ type, le type en question étant décrit comme le terme T vu dans la couleur c'. Or nous avons vu à la section IV.5.2.2 que la couleur a une double influence sur un type. D'une part, elle régit sa bonne formation (la valeur de c' détermine la validité du jugement $B; \Gamma \vdash_c T\, \mathsf{ok}$). D'autre part, la couleur régit la sémantique du type, en ce qu'elle détermine l'ensemble des expressions qui ont ce type. Ainsi, a étant l'apax décrit ci-dessus, le type $(\!|\pi_1\, a|\!)$ est

[27] La fraîcheur de a fait que c_0 ne peut pas contenir a.

bien formé quelle que soit la couleur ; dans la couleur a, les valeurs 3 et $[3]_{\{\mathsf{a}\}}^{\mathsf{Typ}\,\pi_1\,\mathsf{a}}$ ont toutes les deux le type $(\!|\pi_1\,\mathsf{a}|\!)$, tandis que 3 n'a pas ce type dans la couleur $\bullet$. En général, une couleur plus grande rend valide plus de types[28], et augmente également l'ensemble des expressions qui ont ce type.

Nous avons défini un moyen de transformer un type en un type équivalent (donc de même sémantique au sens de l'ensemble des expressions ayant ce type dans la couleur ambiante) et valide dans n'importe quelle couleur : l'opération de concrétisation $\mathbf{conc}_{\mathsf{c}'}^{\mathsf{B}}(\mathsf{T})$ vue à la section IV.5.2.2. Cependant, bien que $\mathbf{conc}_{\mathsf{c}'}^{\mathsf{B}}(\mathsf{T})$ soit valide dans n'importe quelle couleur c, il n'y a pas la même sémantique ; nous ne saurions donc remplacer $[\langle\mathsf{T}\rangle]_{\mathsf{c}'}^{\mathrm{TYPE}}$ par $\langle\mathbf{conc}_{\mathsf{c}'}^{\mathsf{B}}(\mathsf{T})\rangle$.

Nous voyons là que l'expression $[\langle\mathsf{T}\rangle]_{\mathsf{c}'}^{\mathrm{TYPE}}$ a une sémantique nouvelle, qui ne peut essentiellement être exprimée autrement. L'intérêt d'accepter ce terme n'est toutefois pas évident. Nous allons évaluer les implications qu'ont l'exclusion et l'inclusion de tels termes dans le langage. Avant cela, nous allons voir comment décrire l'ensemble des termes concernés, à l'aide de sortes.

IV.5.3.2 Sortage des types

Nous souhaitons reconnaître les types spécifiant complètement leurs valeurs. Des exemples de tels types sont les types singletons : toutes les expressions pures qui ont un type singleton donné sont essentiellement équivalentes. Suivant une interprétation stricte, on pourrait dire qu'un type complètement spécifié est un type singleton. Cependant, si l'on admet l'équivalence extensionnelle, un type peut être un singleton sémantiquement sans l'être sémantiquement : par exemple le type $\mathsf{S}(3) * \mathsf{S}(4)$ ne contient pas plus de valeurs que $\mathsf{S}((3,4))$. Il peut même exister des types qui ne contiennent qu'une valeur à équivalence observationnelle près pour des raisons tenant au langage dans sa totalité ; par exemple, en ML, un résultat de paramétricité [Wad89] fait que la seule fonction de type $\forall\alpha, \alpha \to \alpha$ (ce que nous noterions $\Pi\mathsf{t} : \mathrm{TYPE}.\ \mathsf{Typ}\,\mathsf{t} \to \mathsf{Typ}\,\mathsf{t}$) est l'identité.

En fait, nous cherchons plus précisément à caractériser les signatures qui spécifient complètement les champs types qu'elles contiennent. En effet, même si le type BOOL possède deux valeurs qui sont bien distinctes observationnellement et structurellement, cette absence de précision ne vaut pas abstraction : s'agissant d'un langage de programmation, et non d'un langage de preuve, nous limitons l'abstraction aux types, et acceptons d'exposer $[\mathsf{true}]_{\mathsf{c}'}^{\mathrm{BOOL}}$ en true.

Le paradigme de la signature incomplète est TYPE, qui désigne un champ type non spécifié ; plus largement, tout type qui comporte TYPE en position covariante, tel que $\mathrm{INT} * \mathrm{TYPE}$ ou $\mathrm{INT} \to \mathrm{TYPE}$, est incomplètement spécifié. La présence de TYPE en position contravariante n'est pas indicatrice d'incomplétude, comme le montre l'exemple du type de fonction constante $\Pi\mathsf{t} : \mathrm{TYPE}.\ \mathsf{S}(\mathsf{V})$.

Pour formaliser cette notion, nous munissons le système $\mathcal{C}$ de deux **sortes** (*kinds*). La sorte $\wr$ comporte les types suffisamment spécifiés comme $\mathsf{S}(\mathsf{E})$, BOOL, ou encore $\Pi\mathsf{t} : \mathrm{TYPE}.\mathsf{T}$ avec T de sorte $\wr$. Nous dirons qu'un type de sorte $\wr$ est **complètement spécifié**. La sorte $*$ comporte tous les types, quel que soit leur niveau de spécification ; un type qui n'a pas la sorte $\wr$ sera dit **partiellement spécifié**. Une sorte est génériquement notée K. Nous munissons les sortes de la relation d'ordre $\wr \leqslant *$; nous utiliserons également les opérations de borne inférieure $\mathsf{K}_1 \wedge \mathsf{K}_2$ ($\wr \wedge * = \wr$) et de borne supérieure $\mathsf{K}_1 \vee \mathsf{K}_2$ ($\wr \vee * = *$). Le sortage des types est assorti d'une relation de sous-sortage très simple : si T a la sorte K_1 et si $\mathsf{K}_1 \leqslant \mathsf{K}_2$ alors T a la sorte K_2.

Il est tentant de qualifier un type complètement spécifié de **monomorphe** (par opposition à **polymorphe**) ; il est également tentant de parler de type concret (par opposition à abstrait). Ces terminologies seraient toutefois trompeuses en l'absence de contexte, comme le montre d'ailleurs leur simple multiplicité. Le type TYPE est en quelque sorte une variable de type (ce qui fait de la sorte $\wr$ une caractérisation des types clos), et l'interprétation de sa présence dépend de l'interprétation que

[28] La couleur influe sur les expressions imbriquées. Par exemple, $\mathsf{S}((\lambda\mathsf{x}{:}\,\mathsf{Typ}\,\pi_1\,\mathsf{a}.\ \mathsf{x})\,3)$ n'est bien formé que dans une couleur exposant a.

l'on fait de ces variables. Liées universellement, la distinction est entre monomorphe et polymorphe ; liées existentiellement, elle est entre concret et abstrait. Par souci d'esthétique, nous emploierons les termes monomorphe et polymorphe (ce dernier signifiant souvent en fait non-monomorphe) ; le lecteur est néanmoins invité à conserver cet avertissement à l'esprit.

Les règles de sortage des types sont essentiellement simples : TYPE est polymorphe, tout autre constructeur produit un type monomorphe si et seulement si ses arguments covariants le sont — les types de base tels que BOOL et INT, ainsi que les singletons, sont monomorphes ; un type produit est monomorphe lorsque toutes ses composantes le sont ; un type de fonction est monomorphe si et seulement si le type résultat l'est.

Le cas qui reste à traiter est celui de la projection d'un champ type d'une expression : quand $\mathsf{Typ}\,\mathsf{E}$ est-il monomorphe ? Il faut que $\mathsf{Typ}\,\langle\mathsf{T}\rangle$ soit monomorphe si et seulement si T l'est. La présence dans le langage de types dépendants nous amène à refléter la multiplicité des sortes dans le typage des champs types. Nous remplaçons donc le type TYPE par deux types $\text{TYPE}^{\wr}$ et TYPE^{*} : le type TYPE^{K} caractérise les champs types contenant un type de sorte K. Par exemple $\langle\text{INT}\rangle$ et $\langle\text{TYPE}^{*}\rightarrow\text{INT}\rangle$ ont le type $\text{TYPE}^{\wr}$, et il en est de même pour $\langle\text{INT} * \mathsf{Typ}\,\mathsf{x}\rangle$; tandis que $\langle\text{TYPE}^{\mathsf{K}}\rangle$ et $\langle\text{INT}\rightarrow\text{TYPE}^{\wr}\rangle$ n'ont que le type TYPE^{*}. On veillera ici à ne pas confondre le type donné à l'expression et le type contenu dans le champ : par exemple $\langle\text{TYPE}^{*}\rangle$ est une expression contenant un champ type, qui se trouve être celui des champs types quelconques (en ML, il s'agirait d'un champ signature dans un module, ce qui existe par exemple en Objective Caml avec la syntaxe `struct module type S : sig end end`). L'expression $\langle\text{TYPE}^{*}\rangle$ admet le type monomorphe $\mathsf{S}(\langle\text{TYPE}^{*}\rangle)$, mais n'a pas le type des champs monomorphes $\text{TYPE}^{\wr}$, pas plus que ne l'a $\langle\text{TYPE}^{\wr}\rangle$ puisque le type $\text{TYPE}^{\wr}$ n'a pas la sorte $\wr$.

IV.5.3.3 Crochets et application de fonction ; fonctions polymorphes

La réduction d'un scellage qui introduit durant l'évaluation des crochets colorés, annote ces crochets par des types monomorphes, produits par l'opération de renforcement. C'est même son rôle fondamental : elle achève de spécifier le type apposé sur un scellage, en nommant la partie non spécifiée. Lorsqu'un crochet coloré est poussé vers l'intérieur d'une structure de données, le caractère monomorphe des annotations de types sur les crochets est préservé — nous dirons que les termes manipulés sont des crochets monomorphes (par opposition aux crochets polymorphes dont l'annotation de type est incomplète). L'aspect du système $\mathcal{C}$ qui reste à préciser vis-à-vis de la sorte des annotations de types sur les crochets est la poussée des crochets autour d'une fonction.

Considérons l'expression $[\lambda\mathsf{x}:\mathsf{T}_2.\ \mathsf{E}]_{\mathsf{c}'}^{\Pi\mathsf{x}:\mathsf{T}_0.\,{}^{\mathsf{P}}\mathsf{T}_1}$ dans une couleur ambiante c. Pour son bon typage, T_2 doit être un sous-type de T_0 et E doit avoir le type T_1 dans la couleur c'. Lorsque cette expression est appliqué à un argument V de type T_0, le résultat doit être celui de la bêta-réduction $\{\mathsf{x}{\leftarrow}\mathsf{V}\}\mathsf{E}$, aux couleurs près. Il reste à déterminer comment gérer les crochets au cours de l'évaluation.

Nous avons annoncé à la section IV.5.2.3 que a bêta-réduction resterait valide, ce qui nous impose de régler le sort des crochets dès leur poussée sous le lambda. Cette contrainte ne limite pas notre latitude à choisir la sémantique : elle revient à donner un nom symbolique x à l'argument effectif (ainsi qu'à la couleur extérieure c, qui est également la couleur (visible) de l'argument). La question est donc : comment réduire $[\lambda\mathsf{x}:\mathsf{T}_2.\ \mathsf{E}]_{\mathsf{c}'}^{\Pi\mathsf{x}:\mathsf{T}_0.\,{}^{\mathsf{P}}\mathsf{T}_1}$?

La possibilité la plus évidente est d'utiliser des crochets colorés à la fois autour du corps de la fonction pour en protéger la sortie et autour de chaque occurence du paramètre pour protéger l'argument. L'argument doit également être protégé dans le type de retour.

$$[\lambda\mathsf{z}:\mathsf{T}_2.\ \mathsf{E}]_{\mathsf{c}'}^{\Pi\mathsf{y}:\mathsf{T}_0.\,{}^{\mathsf{P}}\mathsf{T}_1} \longrightarrow_{\mathsf{c}} \lambda\mathsf{x}:\mathsf{T}_0.\ [\{\mathsf{z}{\leftarrow}_{\{\mathsf{x}\}}[\mathsf{x}]_{\mathsf{c}\cup\{\mathsf{x}\}}^{\mathsf{T}_0}\}\mathsf{E}]_{\mathsf{c}'}^{\{\mathsf{y}\leftarrow_{\{\mathsf{x}\}}[\mathsf{x}]_{\mathsf{c}\cup\{\mathsf{x}\}}^{\mathsf{T}_0}\}\mathsf{T}_1} \quad \text{(ered.col.fun.P-POLY)}$$

Le crochet qui entoure le paramètre x dans le corps de la fonction et dans le type de retour doit autoriser x à l'intérieur, donc la couleur portée par ce crochet doit contenir x (l'ajout de c est techniquement inutile, puisque x amène automatiquement sa couleur de liaison c). Le choix de l'annotation de type à poser sur ce crochet n'est pas forcément évident. On sait que E et T_1 sont bien typés dès que leur variable a le type T_0 (rappelons que T_2 est un sous-type de T_0) ; T_0 est correct dans la couleur c de par la correction de l'annotation du crochet dans le contracté ; donc T_0 est un choix possible.

Toutefois rien n'impose que le type T_0 soit monomorphe, même si $\Pi y : T_0.\,{}^{\mathsf{P}}T_1$ l'est : cette règle introduit donc des crochets colorés portant une annotation de type polymorphe. De fait, un type T_0 incomplètement spécifié signifie que la fonction $\lambda z : T_0.\ E$ est une **fonction polymorphe**[29]. Cette terminologie correspond à celle de ML : une fonction polymorphe en ML a un schéma de types de la forme $\forall\alpha, T_0 \to T_1$, ce qui se traduit dans le système $\mathcal{C}$ en $\Pi t : \text{TYPE}^l.\ T_0 \to T_1$ ($\mathsf{Typ}\,t$ correspondant à α).

Si nous souhaitons adopter la poussée des crochets autour des fonctions telle que décrite ci-dessus, nous devons donc accepter des crochets polymorphes. Nous allons maintenant réexaminer ceux-ci à la lumière de la manière dont ils apparaissent. Nous proposerons ensuite une manière de nous en passer.

IV.5.3.4 Types et valeurs polymorphes

Nous supposons toujours la règle (ered.col.fun.P-POLY) définie comme dans la section IV.5.3.3. Les crochets polymorphes sont formés lors de l'application d'une fonction polymorphe provenant d'une couleur étrangère : un crochet $[E]_{c'}^{\text{TYPE}^l}$ provient de l'application d'une fonction $[f]_{c'}^{\Pi t:\text{TYPE}^l.\ T_1}$. Considérons l'exemple le plus simple possible : la fonction identité spécialisée au type TYPE^l, publiée de c' sous le type $\Pi t : \text{TYPE}^l.\ \mathsf{S}(t)$.

$$\left([\lambda t : \text{TYPE}^l.\ t]_{c'}^{\Pi t:\text{TYPE}^l.\ \mathsf{S}(t)}\right) \langle\text{INT}\rangle \longrightarrow_c \left(\lambda t : \text{TYPE}^l.\ [[t]_{c\cup\{t\}}^{\text{TYPE}^l}]_{c'}^{\mathsf{S}([t]_{c\cup\{t\}}^{\text{TYPE}^l})}\right) \langle\text{INT}\rangle$$

$$\longrightarrow_c [[\langle\text{INT}\rangle]_c^{\text{TYPE}^l}]_{c'}^{\mathsf{S}([\langle\text{INT}\rangle]_c^{\text{TYPE}^l})}$$

Dans ce cas particulier, la valeur finale est $[\langle\text{INT}\rangle]_c^{\text{TYPE}^l}$ — l'identité renvoie donc son argument protégé par un crochet superflu, dont la couleur est la couleur ambiante.

Considérons maintenant l'identité polymorphe $\lambda t : \text{TYPE}^l.\ \lambda x : \mathsf{Typ}\,t.\ x$, publiée de c' sous le type $\Pi t : \text{TYPE}^l.\ \mathsf{Typ}\,t \to \mathsf{Typ}\,t$.

$$\left([\lambda t : \text{TYPE}^l.\ \lambda x : \mathsf{Typ}\,t.\ x]_{c'}^{\Pi t:\text{TYPE}^l.\ \mathsf{Typ}\,t \to \mathsf{Typ}\,t}\right) \langle\text{INT}\rangle\ 3$$

$$\longrightarrow_c \left(\lambda t : \text{TYPE}^l.\ [\lambda x : \mathsf{Typ}\,[t]_{c\cup\{t\}}^{\text{TYPE}^l}.\ x]_{c'}^{\mathsf{Typ}\,[t]_{c\cup\{t\}}^{\text{TYPE}^l} \to \mathsf{Typ}\,[t]_{c\cup\{t\}}^{\text{TYPE}^l}}\right) \langle\text{INT}\rangle\ 3$$

$$\longrightarrow_c [\lambda x : \mathsf{Typ}\,[\langle\text{INT}\rangle]_c^{\text{TYPE}^l}.\ x]_{c'}^{\mathsf{Typ}\,[\langle\text{INT}\rangle]_c^{\text{TYPE}^l} \to \mathsf{Typ}\,[\langle\text{INT}\rangle]_c^{\text{TYPE}^l}}\ 3$$

$$\longrightarrow_c \left(\lambda x : \mathsf{Typ}\,[\langle\text{INT}\rangle]_c^{\text{TYPE}^l}.\ [[x]_{c\cup\{x\}}^{\mathsf{Typ}\,[\langle\text{INT}\rangle]_c^{\text{TYPE}^l}}]_{c'}^{\mathsf{Typ}\,[\langle\text{INT}\rangle]_c^{\text{TYPE}^l}}\right)\ 3$$

$$\longrightarrow_c [[3]_c^{\mathsf{Typ}\,[\langle\text{INT}\rangle]_c^{\text{TYPE}^l}}]_{c'}^{\mathsf{Typ}\,[\langle\text{INT}\rangle]_c^{\text{TYPE}^l}}$$

L'argument 3 se retrouve entouré de deux crochets : le crochet interne le plonge dans la couleur c' avec un type polymorphe ; le crochet externe, bien que d'apparence très proche, a un rôle bien

[29] On retrouve l'association entre incomplétude de la spécification et polymorphisme en présence de quantification universelle mentionnée à la section IV.5.3.2.

différent puisque le type apparemment polymorphe $\mathsf{Typ}\,[\langle\textsc{int}\rangle]_{\mathsf{c}}^{\textsc{type}^l}$ est en fait monomorphe dans la couleur extérieure c.

Si nous cherchons à généraliser ces exemples, nous pouvons identifier plusieurs notions dignes d'intérêt. Un crochet $[\langle\mathsf{T}\rangle]_{\mathsf{c}}^{\textsc{type}^l}$ représente un **paramètre type polymorphe** de la fonction. Un crochet $[\mathsf{V}]_{\mathsf{c}}^{\mathsf{Typ}\,[\langle\mathsf{T}\rangle]_{\mathsf{c}}^{\textsc{type}^l}}$ est une **valeur polymorphe**. Une valeur polymorphe n'a pas de structure apparente, puisqu'elle est protégée par un crochet, crochet inamovible puisque son annotation de type d'a elle-même pas de structure exposée. Le corps de la fonction ne peut donc avec les outils que nous avons vu jusqu'à présent manipuler une valeur polymorphe que de façon paramétrique. Cette approche devrait donc convenir au polymorphisme paramétrique de ML, mais pas à un langage capable de programmation générique ou de typage dynamique (que nous introduirons dans le système $\mathcal{D}$).

Nous ne mènerons pas plus loin dans cette thèse l'étude des valeurs polymorphes. Il resterait au moins à déterminer les règles de simplification des crochets polymorphes : comment l'identité polymorphe peut-elle renvoyer son argument sans crochets superflus (noter qu'il y a un passage par la couleur c' : comment s'assurer qu'il demeure anodin) ?

IV.5.3.5 Fusion des couleurs

Une solution existe pour gérer les appels de fonctions polymorphes qui, si elle manque d'expressivité et de finesse, n'en conserve pas moins un certain attrait, ne serait-ce que de par sa simplicité. Elle consiste à fusionner la couleur de l'argument et la couleur du corps de la fonction.

$$[\lambda\mathsf{x}:\mathsf{T}_2.\ \mathsf{E}]_{\mathsf{c}'}^{\Pi\mathsf{x}:\mathsf{T}_0.\,^{\mathsf{P}}\mathsf{T}_1} \longrightarrow_{\mathsf{c}} \lambda\mathsf{x}:\mathsf{T}_0.\ [\mathsf{E}]_{\mathsf{c}'\cup\{\mathsf{x}\}}^{\mathsf{T}_1} \qquad (\mathcal{C}/\mathsf{ered.col.fun.P})$$

Toute idée de protéger l'argument est abandonnée : toutes les équations nécessaires à son typage sont autorisées dans l'ensemble du corps de la fonction. On constate l'extrême simplicité technique de cette règle : conservation du nombre de crochets, diminution de la taille de l'annotation de types aussi bien que de celle de l'expression à l'intérieur des crochets.

Cette règle présente une certaine symétrie : l'application de $[\lambda\mathsf{x}:\mathsf{T}_2.\ \mathsf{E}]_{\mathsf{c}'}^{\Pi\mathsf{x}:\mathsf{T}_0.\,^{\mathsf{P}}\mathsf{T}_1}$ à un argument V donne

$$[\{\mathsf{x}\leftarrow_{\mathsf{c}}\mathsf{V}\}\mathsf{E}]_{\mathsf{c}\cup\mathsf{c}'}^{\{\mathsf{x}\leftarrow_{\mathsf{c}}\mathsf{V}\}\mathsf{T}_1}$$

c'est-à-dire les calculs sont simplement effectués dans la réunion des couleurs des opérandes mises en contact V et E.

Nous pouvons vérifier le résultat de l'application de l'identité polymorphe en utilisant cette règle.

$$\begin{aligned}
&\left([\lambda\mathsf{t}:\textsc{type}^l.\ \lambda\mathsf{x}:\mathsf{Typ}\,\mathsf{t}.\ \mathsf{x}]_{\mathsf{c}'}^{\Pi\mathsf{t}:\textsc{type}^l.\ \mathsf{Typ}\,\mathsf{t}\to\mathsf{Typ}\,\mathsf{t}}\right)\ \langle\textsc{int}\rangle\ 3\\
&\qquad\longrightarrow_{\mathsf{c}}\ \left(\lambda\mathsf{t}:\textsc{type}^l.\ [\lambda\mathsf{x}:\mathsf{Typ}\,\mathsf{t}.\ \mathsf{x}]_{\mathsf{c}'\cup\{\mathsf{t}\}}^{\mathsf{Typ}\,\mathsf{t}\to\mathsf{Typ}\,\mathsf{t}}\right)\ \langle\textsc{int}\rangle\ 3\\
&\qquad\longrightarrow_{\mathsf{c}}\ [\lambda\mathsf{x}:\mathsf{Typ}\,\langle\textsc{int}\rangle.\ \mathsf{x}]_{\mathsf{c}\cup\mathsf{c}'}^{\mathsf{Typ}\,\langle\textsc{int}\rangle\to\mathsf{Typ}\,\langle\textsc{int}\rangle}\ 3\\
&\qquad\longrightarrow_{\mathsf{c}}\ \left(\lambda\mathsf{x}:\mathsf{Typ}\,\langle\textsc{int}\rangle.\ [\mathsf{x}]_{\mathsf{c}\cup\mathsf{c}'\cup\{\mathsf{x}\}}^{\mathsf{Typ}\,\langle\textsc{int}\rangle}\right)\ 3\\
&\qquad\longrightarrow_{\mathsf{c}}\ [3]_{\mathsf{c}\cup\mathsf{c}'}^{\mathsf{Typ}\,\langle\textsc{int}\rangle}\ \longrightarrow_{\mathsf{c}}\ [3]_{\mathsf{c}\cup\mathsf{c}'}^{\textsc{int}}\ \longrightarrow_{\mathsf{c}}\ 3
\end{aligned}$$

Dans l'étude formelle du système $\mathcal{C}$, nous retiendrons la forme fusionnelle présentée ici de la poussée des crochets autour d'une abstraction fonctionnelle.

IV.5.3.6 Foncteurs génératifs

Nous avons vu à la section IV.5.1.5 que le renforcement laisse les types foncteurs génératifs inchangés. Or le renforcement produit un type destiné à être l'annotation d'un crochet (dans la règle ($\mathcal{C}$/ered.seal)) ; ce type doit donc être monomorphe. Un type foncteur génératif $\Pi x : T_0.\ ^{\mathrm{I}}T_1$ est donc monomorphe quel que soit T_1, même polymorphe[30].

Nous avons énoncé une règle ($\mathcal{C}$/ered.col.fun.P) permettant la poussée d'un crochet portant un type foncteur applicatif. Le passage à la règle ($\mathcal{C}$/ered.col.fun.I) similaire correspondant aux types foncteurs génératifs n'est pas immédiat. Une transposition naïve serait la suivante :

$$\frac{}{[\lambda x : T_2.\ E]_{c'}^{\Pi x:T_0.\ ^{\mathrm{I}}T_1} \longrightarrow_c \lambda x : T_0.\ [E]_{c'\cup\{x\}}^{T_1}}$$

Mais puisque T_1 peut être polymorphe, le membre de droite n'est pas bien typé. Intuitivement, la règle ci-dessus ne peut convenir : le membre de gauche est un foncteur génératif, dont chaque application doit générer un nouvel apax ; il faut faire apparaître cet aspect dans le membre de droite.

La génération d'un apax est actuellement assurée par la réduction d'un scellage par la règle ($\mathcal{C}$/ered.seal). Introduisons-donc un scellage dans le membre de droite. Deux possibilités s'offrent à nous :

$$\frac{}{[\lambda x : T_2.\ E]_{c'}^{\Pi x:T_0.\ ^{\mathrm{I}}T_1} \longrightarrow_c \lambda x : T_0.\ ([E]_{c'\cup\{x\}}^{T_1} \,!!\, T_1)}$$

$$\frac{}{[\lambda x : T_2.\ E]_{c'}^{\Pi x:T_0.\ ^{\mathrm{I}}T_1} \longrightarrow_c \lambda x : T_0.\ [E \,!!\, T_1]_{c'\cup\{x\}}^{T_1}}$$

Dans les deux cas, si l'intuition est sauve, la correction formelle n'est pas au rendez-vous : le membre de droite contient un crochet polymporphe, mais c'est un crochet polymorphe bénin, dont l'annotation pourra être rendue monomorphe avant de le pousser plus avant.

Un scellage constitue une frontière statique entre domaines d'abstraction, tandis qu'un crochet coloré est une frontière dynamique. Nous avons ici la combinaison d'une frontière statique et dynamique. Nous la matérialiserons par un **scellage coloré** noté $E \,!!_{c'}\, T$. Ce scellage agit comme un scellage normal, sauf qu'il confère en plus à l'expression E les équations de typage fournies par la couleur c'. Un scellage coloré a donc l'effet d'un crochet additif (voir la section IV.5.2.4. Un scellage normal est un cas particulier de scellage coloré n'ajoutant aucune connaissance : $E \,!!\, T = E \,!!_{\bullet}\, T$.

La poussée d'un crochet portant une signature de foncteur génératif crée un scellage coloré :

$$[\lambda x : T_2.\ E]_{c'}^{\Pi x:T_0.\ ^{\mathrm{P}}T_1} \longrightarrow_c \lambda x : T_0.\ (E \,!!_{c'}\, T_1) \qquad (\mathcal{C}/\mathsf{ered.col.fun.I})$$

Noter que l'additivité de la couleur c' dispense de lui adjoindre x (comparer avec ($\mathcal{C}$/ered.col.fun.P)). Un scellage coloré est réduit par la règle ($\mathcal{C}$/ered.seal) dont nous pouvons finalement donner la forme générale :

$$\frac{}{B \vdash V \,!!_{c'}\, T \longrightarrow_c B, a = V :_{c\cup c'} T \vdash [V]_{c\cup c'\cup\{a\}}^{\mathbf{self}^{T}(a)}}\ (\mathcal{C}/\mathsf{ered.seal})$$

IV.5.4 Évaluation

IV.5.4.1 Syntaxe

La syntaxe du système $\mathcal{C}$ comprend par rapport au système $\mathcal{E}$ deux nouvelles constructions (qui ne sont pas en principe utilisées dans les programmes sources), les types abstraits et les crochets

[30] En un sens, c'est un type amorphe, qui n'est pas encore déterminé mais donnera naissance à un type monomorphe lorsque le foncteur sera appliqué.

colorés. De plus, la signature TYPE porte désormais une annotation de sorte, et le scellage porte désormais une annotation de couleur (la notation $E \mathbin{!!} T$ étant conservée comme abréviation de $E \mathbin{!!_\bullet} T$).

$K ::=$	**sorte**
$\wr$	monomorphe (complètement spécifié)
$*$	polymorphe (partiellement spécifié)
$T ::=$	**type**
$\ldots$	
TYPE^K	champ type abstrait
$(\!|A|\!)$	type abstrait
$E ::=$	**expression**
$\ldots$	
$E \mathbin{!!_c} T$	module scellé et coloré
$[E]_c^T$	crochet coloré
$A ::=$	**composante**
a	apax
$A\,E$	application
$\pi_i A$	projection ($i \in \{1,2\}$)
$\xi ::=$	**couleur primaire**
a	apax
x	variable
$c ::=$	**couleur**
$\bullet$	couleur vide (aussi notée $\{\}$)
$\{a_1,\ldots,a_k,\ x_1,\ldots,x_k\}$	ensemble fini de couleurs élémentaires

Rappelons que les sortes sont munies d'une relation d'ordre, notée $K_1 \leqslant K_2$, telle que $\wr \leqslant *$. Nous notons $K_1 \vee K_2$ la borne supérieure de K_1 et K_2 et $K_1 \wedge K_2$ leur borne inférieure.

Si A est une composante, son **apax sous-jacent underl**(A) est défini formellement par

$$\begin{aligned}\mathbf{underl}(a) &= a\\ \mathbf{underl}(A\,E) &= \mathbf{underl}(A)\\ \mathbf{underl}(\pi_i A) &= \mathbf{underl}(A)\end{aligned}$$

La révélation d'une composante est quant à elle définie formellement comme suit :

$$\begin{aligned}\mathbf{reveal}^B(a) &= E && \text{où } a = E :_c T \in B\\ \mathbf{reveal}^B(A\,E) &= (\mathbf{reveal}^B(A))\,E\\ \mathbf{reveal}^B(\pi_i A) &= \pi_i(\mathbf{reveal}^B(A))\end{aligned}$$

Les jugements de typage comprennent maintenant un lexique et une couleur. Les membres de droite sont, en plus de ceux du système $\mathcal{E}$, la transparence d'une couleur, la révélation d'une composante et la conversion et la convertibilité d'une composante.

$\mathscr{J} ::=$	**jugement de typage**
$B;\Gamma \vdash_c J$	jugement local coloré

$J ::=$	**membre de droite de jugement local**
$\quad\dots$	
$\quad T : K$	sortage d'un type (généralise $T\,\mathsf{ok}$)
$\quad c_0\,\mathsf{transparent}$	transparence d'une couleur
$\quad A \triangleright E : T$	révélation d'une composante
$\quad A \longrightarrow A'$	conversion d'une composante
$\quad A \equiv A'$	équivalence par conversion des composantes

Nous notons simplement $\xi\,\mathsf{transparent}$ pour $\{\xi\}\,\mathsf{transparent}$.

Les environnements contiennent désormais des annotations de couleur. Nous précisons également la syntaxe des lexiques.

$B ::=$	**lexique**
$\quad \mathsf{nil}$	vide
$\quad B, a = E :_{c_0} T$	apax a d'implémentation E et de signature T

$\Gamma ::=$	**environnement**
$\quad \mathsf{nil}$	vide
$\quad \Gamma, x :_c T$	liaison d'une variable x

Sur le modèle des environnements, le domaine d'un lexique B, c'est-à-dire l'ensemble des apax qu'il répertorie, est noté **dom** B.

Puisque les couleurs peuvent contenir des variables, elles sont affectées par les substitutions. Une substitution spécifie par quelle expression remplacer une variable lorsqu'elle apparaît dans une expression ; elle doit de même spécifier par quelle couleur remplacer la variable lorsqu'elle apparaît dans une couleur. La substitution de x par E sous c dans $\aleph$ sera notée $\{x \leftarrow_c E\}\aleph$.

IV.5.4.2 Valeurs et composantes abstraites

Valeurs et crochets Comme dans HAT, la notion de valeur dépend de la couleur ambiante. Nous commençons donc par définir une notion de *quasi-valeur*, qui est une valeur aux couleurs près. Aux valeurs du système $\mathcal{E}$ (qui sont celles de $\mathcal{B}$) viennent s'ajouter les crochets colorés. Pour qu'une expression de la forme $[V]_{c'}^{T}$ soit une valeur, il faut que V soit une valeur (dans la couleur c'), et que T ait une forme adéquate. Si T a une structure apparente, les règles de poussée des crochets permettent de réduire $[V]_{c'}^{T}$. Le seul cas où $[V]_{c'}^{T}$ peut être une valeur est donc celui où T est un type abstrait $(\!|A|\!)$. Même certaines expressions de la forme $[V]_{c'}^{(\!|A|\!)}$ peuvent être réduites dans certaines couleurs.

Quasi-valeurs Nous appelons **quasi-valeur** une expression qui a la forme générale d'une valeur, mais qui peut éventuellement être réduite par un moyen dépendant des couleurs en jeu. Les quasi-valeurs du système $\mathcal{C}$ sont formées des valeurs du système $\mathcal{E}$ et de certains crochets colorés. Une **composante valeur** est une composante dans laquelle les arguments de foncteurs sont des valeurs.

$V ::=$	**quasi-valeur**
$\quad\dots$	
$\quad [V]_{c'}^{(\!|A^V|\!)}$	crochet coloré potentiellement abstrayant

$A^V ::=$	**composante valeur**
$\quad a$	apax
$\quad A^V\,V$	application à une quasi-valeur
$\quad \pi_i\,A^V$	projection ($i \in \{1, 2\}$)

Crochets colorés irréductibles Une quasi-valeur de la forme $[V]_{c'}^{(\!|A^V|\!)}$ est une valeur si le crochet est indispensable. Intuitivement, un crochet coloré n'est indispensable que s'il apporte effectivement de l'abstraction, ce qui se traduit par la condition que l'apax sous-jacent à A^V soit opaque dans la couleur ambiante mais transparent dans c'. Nous décrirons le comportement d'un crochet coloré suivant les relations entre l'apax sous-jacent au type abstrait et les deux couleurs en jeu lorsque nous présenterons les règles d'élimination des crochets à la section IV.5.4.2.

Valeurs et composantes abstraites La notion de valeur dépend de la couleur ambiante. Nous noterons V^c une valeur dans la couleur c. Les valeurs sont engendrées par une grammaire paramétrée par la couleur ambiante et comportant des conditions relative aux couleurs. Une quasi-valeur $[V]_{c'}^{(\!|A^V|\!)}$ n'est une valeur que si le crochet coloré est irréductible au sens de la discussion ci-dessus, et que les valeurs éventuelles contenues dans A^V sont elles-mêmes des valeurs de la couleur ambiante.

$V^c ::=$	**valeur dans** c
$() \mid bv \mid \underline{n}$	constante
$\langle T \rangle$	champ type
(V_1^c, V_2^c)	paire
$\lambda x : T.\ E$	lambda-abstraction
$[V^{c'}]_{c'}^{(\!|A^{V^{c \cap c'}}|\!)}$	crochet coloré, si $A^{V^{c \cap c'}}$ est abstraite dans c mais concrète dans c'

$A^{V^c} ::=$	**composante abstraite dans** c
a	apax opaque dans c
$A^{V^c}\, V^c$	application d'un foncteur à une valeur
$\pi_i\, A^{V^c}$	projection ($i \in \{1, 2\}$)

Notons que bien que la transparence d'un apax dans une couleur soit une notion sémantique dépendant d'un lexique et d'un environnement, nous n'incluons pas ces deux éléments dans la syntaxe car ceux-ci seront toujours évidents d'après le contexte.

IV.5.4.3 $\boxed{B \vdash E \longrightarrow_c B' \vdash E'}$ Réduction

Contextes d'évaluation Nous réduisons les expressions sous les crochets. Bien que les crochets soient au départ apposés sur des valeurs, cette propriété n'est pas invariante ; en particulier, lorsqu'un crochet est poussé sous une abstraction fonctionnelle, il peut entourer une expression arbitraire. Lorsque l'annotation de type sur un crochet est un champ type de module, l'expression de module concernée doit aussi être réduite.

$C ::=$	**contexte d'évaluation (de profondeur** 1**)**
$\ldots$	
$\_\_ \,!!_{c_1}\, T$	scellage
$[\_\_]_{c_1}^{T}$	crochet coloré
$[V^{c_1}]_{c_1}^{\mathsf{Typ}\,\_\_}$	champ type sur un crochet

Formellement, l'ensemble des contextes d'évaluation dépend d'un lexique et d'une couleur, en raison des parties qui doivent être des valeurs. Dans le contexte $[\_\_]_{c_1}^{T}$, l'expression intérieure est réduite dans la couleur c_1. Dans le contexte $[V^{c_1}]_{c_1}^{\mathsf{Typ}\,\_\_}$, l'expression est réduite dans la couleur $c \cap c_1$, c'est-à-dire l'intersection des deux couleurs de part et d'autre de la frontière sur laquelle l'expression est située.

Règles calculatoires Le système $\mathcal{C}$ hérite des règles déjà présentes dans le système $\mathcal{B}$: ($\mathcal{C}$/ered.app), ($\mathcal{C}$/ered.proj), ($\mathcal{C}$/ered.let), ($\mathcal{C}$/ered.context). Ces règles sont valables dans n'importe quelle couleur et n'importe quel lexique, et cette couleur annote les substitutions le cas échéant. Les valeurs doivent être prises dans leur couleur ambiante. L'annexe A récapitule toutes les règles du système $\mathcal{C}$, y compris les règles héritées.

La règle de réduction du scellage est modifiée pour entourer la valeur d'un crochet coloré. De plus, le scellage porte désormais une annotation de couleur.

$$B \vdash V^{c \cup c'} \,!!_{c'}\, T \longrightarrow_c B, a = V^{c \cup c'} :_{c \cup c'} T \vdash [V^{c \cup c'}]^{\mathbf{self}^{T}(a)}_{c \cup c' \cup \{a\}} \quad (\mathcal{C}\text{/ered.seal})$$

$$\text{où } a \text{ est frais (c'est-à-dire que } a \notin \mathbf{dom}\, B)$$

Réductions dans les types Jusqu'au système $\mathcal{E}$, la réduction n'était pas influencée par les types contenus dans les expressiosn, ce qui permettait de les traiter comme des boîtes noires. Ce n'est plus le cas dans le système $\mathcal{C}$ où la réduction d'un crochet coloré dépend de l'annotation de type qu'il porte, plus particulièrement du constructeur de tête de ce type. Néanmoins, comme nos besoins calculatoires sur les types se limite à une mise en forme normale de tête faible, et ce dans un seul contexte dans la syntaxe des expressions, nous n'avons pas besoin de définir une réduction sur les types. Le seul destructeur dans la syntaxe des types est $\mathsf{Typ}\,\_\_$; son argument peut être réduit via le contexte $[V^{c_1}]^{\mathsf{Typ}\,\_\_}_{c_1}$, et la règle ($\mathcal{C}$/ered.colTyp) consacre son élimination.

$$[V^{c'}]^{\mathsf{Typ}\,\langle T\rangle}_{c'} \longrightarrow_c [V^{c'}]^{T}_{c'} \qquad (\mathcal{C}\text{/ered.colTyp})$$

Le cas des types abstraits est particulier : un type abstrait $(\!|A|\!)$ est en forme normale de tête faible si l'empreinte sous-jacente est opaque, mais doit être révélé si elle est transparente.

$$B \vdash [V^{c'}]^{(\!|A|\!)}_{c'} \longrightarrow_c B \vdash [V^{c'}]^{\mathsf{Typ}\,\mathbf{reveal}^{B}(A)}_{c'} \qquad (\mathcal{C}\text{/ered.colAbs})$$

$$\text{si } \mathbf{underl}(A) \in c \cap c'$$

Poussée de crochets Lorsqu'un crochet entoure une valeur, et que l'annotation de type sur le crochet n'est pas un type abstrait, le crochet est poussé à l'intérieur de la valeur. Les règles à cet effet suivent la même ligne directrice que dans HAT (voir les sections III.1.2.2 et III.2.5). La poussée des crochets est basée sur le type apparent du crochet, et le crochet traverse le constructeur pour ce type.

$$\begin{aligned}
[()]^{\mathrm{UNIT}}_{c'} &\longrightarrow_c () && (\mathcal{C}\text{/ered.col.base.unit})\\
[bv]^{\mathrm{BOOL}}_{c'} &\longrightarrow_c bv && (\mathcal{C}\text{/ered.col.base.bool})\\
[\underline{n}]^{\mathrm{INT}}_{c'} &\longrightarrow_c \underline{n} && (\mathcal{C}\text{/ered.col.base.int})\\
[V^{c'}]^{S(E)}_{c'} &\longrightarrow_c E && (\mathcal{C}\text{/ered.col.sing})\\
[(V_1^{c'}, V_2^{c'})]^{\Sigma x : T_1.\, T_2}_{c'} &\longrightarrow_c ([V_1^{c'}]^{T_1}_{c'}, [V_2^{c'}]^{\{x \leftarrow_c [V_1^{c'}]^{T_1}_{c'}\} T_2}_{c'}) && (\mathcal{C}\text{/ered.col.pair})
\end{aligned}$$

Nous adoptons la fusion systématique de la couleur d'une fonction avec la couleur de son argument, telle que vue à la section IV.5.3.5. Dans le cas d'un foncteur génératif, de nouveaux types doivent être créés à chaque application du foncteur ; nous nous en assurons en scellant le corps du foncteur, et le scellage assure également une fonction de crochet coloré.

$$\begin{aligned}
[\lambda x : T_2.\ E]^{\Pi x : T_0.\, {}^{P} T_1}_{c'} &\longrightarrow_c \lambda x : T_0.\ [E]^{T_1}_{c' \cup \{x\}} && (\mathcal{C}\text{/ered.col.fun.P})\\
[\lambda x : T_2.\ E]^{\Pi x : T_0.\, {}^{I} T_1}_{c'} &\longrightarrow_c \lambda x : T_0.\ E \,!!_{c' \cup \{x\}}\, T_1 && (\mathcal{C}\text{/ered.col.fun.I})
\end{aligned}$$

Lorsqu'un crochet entoure un autre crochet et qu'aucun ne peut être réduit par une des règles vues jusqu'à présent, c'est-à-dire que l'on a une expression de la forme $[[V^{c_2}]_{c_2}^{(\!|A_2|\!)}]_{c_1}^{(\!|A_1|\!)}$, et que $[V^{c_2}]_{c_2}^{(\!|A_2|\!)}$ est une valeur, trois comportements sont possibles :

- si le crochet extérieur porte une annotation simplifiable, il disparaît — c'est dans HAT le rôle de ($\mathcal{H}$/ered.col.le), ici assuré par la règle ($\mathcal{C}$/ered.colAbs) suivie par des calculs mettant en jeu l'expression révélée et le cas échéant de la poussée du crochet ;
- si le crochet extérieur porte une annotation abstraite à l'extérieur mais pas à l'intérieur, l'expression considérée est une valeur ;
- enfin le crochet extérieur peut porter une annotation de type abstraite à l'extérieur comme dans la couleur intermédiaire : dans HAT, le typage fait que A_1 et A_2 sont alors les mêmes, et le crochet extérieur est effacé par la règle ($\mathcal{H}$/ered.col.col).

Ce dernier cas est compliqué par la présence de foncteurs et de couleurs d'intersection non triviale. En effet, une nouvelle possibilité est que l'on ait $A_1 = a_1\, V_1$ et $A_2 = a_2\, V_2$; le typage entraîne certes dans $\mathcal{C}$ comme dans HAT que $a_1 = a_2$, mais l'on ne peut dire des arguments que de ce qu'ils sont équivalents dans la couleur intermédiaire c_1 : ils ne sont pas forcément équivalents dans c, si bien que l'on ne peut pas faire disparaître le passage par c_1. Nous adopterons donc une règle plus faible, qui ici encore fusionne les deux couleurs mises en jeu.

$$[[V^{c_2}]_{c_2}^{(\!|A_2|\!)}]_{c_1}^{(\!|A_1|\!)} \longrightarrow_c [V^{c_2}]_{c_1 \cup c_2}^{(\!|A_1|\!)} \qquad (\mathcal{C}/\text{ered.col.merge})$$

si A_1 et A_2 sont toutes deux opaques dans c_1 mais que A_2 est concrète dans c_2

IV.5.5 Typage

Le système $\mathcal{C}$ hérite du système $\mathcal{E}$, mais toutes les règles doivent être modifiées pour ajouter des annotations de couleur. Pour la plupart des règles, il suffit de reprendre les règles du système $\mathcal{E}$ en ajoutant mécaniquement un lexique et une couleur arbitraires. Chaque jugement $\Gamma \vdash J$ devient $B; \Gamma \vdash_c J$. Lorsqu'une variable est liée par l'environnement, elle doit être introduite dans la couleur : $\Gamma, x : T \vdash J$ devient $B; \Gamma, x :_c T \vdash_{c \cup \{x\}} J$. Les substitutions doivent être également décorées de la couleur idoine. À titre d'exemple, nous explicitons ici les règles ($\mathcal{C}$/et.fun) et ($\mathcal{C}$/et.app) :

$$\frac{B; \Gamma, x :_c T_0 \vdash_{c \cup \{x\}} E :^{\gamma} T_1}{B; \Gamma \vdash_c \lambda x : T_0.\ E :^{P} \Pi x : T_0.\ {}^{\gamma}T_1}\ (\mathcal{C}/\text{et.fun}) \qquad \frac{B; \Gamma \vdash_c E_1 :^{\gamma_1} \Pi x : T_0.\ {}^{\gamma_2}T \qquad B; \Gamma \vdash_c E_0 :^{P} T_0}{B; \Gamma \vdash_c E_1\, E_0 :^{\gamma_1 \sqcup \gamma_2} \{x \leftarrow_c E_0\}T}\ (\mathcal{C}/\text{et.app})$$

En plus de l'ajout des couleurs, les hypothèses de correction d'un type T ok deviennent $T : *$, et les contextes exigeant une expression de type TYPE exigent désormais TYPE*. La liste complète des règles ainsi adaptées est la suivante :

- toutes les règles de conversion et d'équivalence des types et des expressions, et de sous-typage : ($\mathcal{C}$/econv.cong.fun.arg), ($\mathcal{C}$/econv.cong.fun.body), ($\mathcal{C}$/econv.cong.app.fun), ($\mathcal{C}$/econv.cong.app.arg), ($\mathcal{C}$/econv.cong.pair.1), ($\mathcal{C}$/econv.cong.pair.2), ($\mathcal{C}$/econv.cong.field), ($\mathcal{C}$/econv.cong.proj), ($\mathcal{C}$/econv.app), ($\mathcal{C}$/econv.proj), ($\mathcal{C}$/econv.eta.field), ($\mathcal{C}$/econv.eta.fun), ($\mathcal{C}$/econv.eta.pair), ($\mathcal{C}$/tconv.cong.pair.1), ($\mathcal{C}$/tconv.cong.pair.2), ($\mathcal{C}$/tconv.cong.fun.arg), ($\mathcal{C}$/tconv.cong.fun.ret), ($\mathcal{C}$/tconv.cong.sing), ($\mathcal{C}$/tconv.cong.field), ($\mathcal{C}$/tconv.field), ($\mathcal{C}$/tconv.unit), ($\mathcal{C}$/eeq.refl), ($\mathcal{C}$/eeq.sym), ($\mathcal{C}$/eeq.trans), ($\mathcal{C}$/eeq.conv), ($\mathcal{C}$/teq.refl), ($\mathcal{C}$/teq.sym), ($\mathcal{C}$/teq.trans), ($\mathcal{C}$/teq.conv), ($\mathcal{C}$/tsub.trans), ($\mathcal{C}$/tsub.eq), ($\mathcal{C}$/tsub.cong.fun), ($\mathcal{C}$/tsub.cong.pair), ($\mathcal{C}$/tsub.sing) ;
- la plupart des règles de typage des expressions : ($\mathcal{C}$/et.base.unit), ($\mathcal{C}$/et.base.bool), ($\mathcal{C}$/et.base.int), ($\mathcal{C}$/et.fun), ($\mathcal{C}$/et.app), ($\mathcal{C}$/et.pair), ($\mathcal{C}$/et.proj.1), ($\mathcal{C}$/et.proj.2), ($\mathcal{C}$/et.let), ($\mathcal{C}$/et.sub), ($\mathcal{C}$/et.sing).

L'annexe A récapitule toutes les règles du système $\mathcal{C}$, y compris les règles héritées.

IV.5.5.1 $\boxed{\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{ok}}$ Formation de l'environnement

Nous décrivons dans cette section les règles de formation des lexiques, environnements et couleurs. Ces composants sont construits de gauche à droite, aussi bien au sens ou les lexiques et les environnements sont formés par ajouts élémentaires successifs à droite (les couleurs sont aussi construites élément par élément), et en ce que la formation d'un jugement commence par assembler le lexique, puis l'environnement, et enfin la couleur.

La correction d'un lexique et celle d'un environnement procèdent selon un mécanisme similaire : les informations associées à chaque entrée doivent être correctes, et un nom frais doit être utilisé comme étiquette de l'entrée. On remarque que la couleur d'une entrée de lexique peut contenir des apax déjà existants, et la couleur d'une entrée d'environnement peut contenir des apax ainsi que des variables présentes auparavant dans l'environnement.

$$\frac{}{\mathsf{nil};\mathsf{nil} \vdash_{\bullet} \mathsf{ok}}\,(\mathcal{C}/\mathsf{envok.nil}) \qquad \frac{\mathsf{B};\mathsf{nil} \vdash_{\mathsf{c}_0} \mathsf{E} :^{\mathsf{P}} \mathsf{T} \qquad \text{si } a \notin \mathbf{dom}\,\mathsf{B}}{\mathsf{B}, a = \mathsf{E} :_{\mathsf{c}_0} \mathsf{T};\mathsf{nil} \vdash_{\bullet} \mathsf{ok}}\,(\mathcal{C}/\mathsf{envok.a})$$

$$\frac{\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{T} : * \qquad \text{si } x \notin \mathbf{dom}\,\Gamma}{\mathsf{B};\Gamma, x :_{\mathsf{c}} \mathsf{T} \vdash_{\bullet} \mathsf{ok}}\,(\mathcal{C}/\mathsf{envok.x})$$

Une couleur peut contenir des apax et des variables provenant respectivement du lexique et de l'environnement. Comme une couleur est un ensemble, les ajouts à la couleur ne sont pas contraints par un ordre particulier (sauf par le fait que chaque couleur intermédiaire doit être valide). Une couleur est un ensemble quelconque d'apax et de variables ; les apax doivent être ajoutés après leurs dépendances (voir la section IV.5.2.2) tandis que les variables amènent implicitement leurs dépendances (voir la section IV.5.2.3 et les règles de transparence).

$$\frac{\mathsf{B};\Gamma \vdash_{\mathsf{c}'} \mathsf{ok} \qquad \text{si } a = \mathsf{E} :_{\mathsf{c}_0} \mathsf{T} \in \mathsf{B} \wedge \mathsf{c}_0 \subseteq \mathsf{c}'}{\mathsf{B};\Gamma \vdash_{\mathsf{c}' \cup \{a\}} \mathsf{ok}}\,(\mathcal{C}/\mathsf{envok.c.a})$$

$$\frac{\mathsf{B};\Gamma \vdash_{\mathsf{c}'} \mathsf{ok} \qquad \text{si } x :_{\mathsf{c}_0} \mathsf{T} \in \Gamma}{\mathsf{B};\Gamma \vdash_{\mathsf{c}' \cup \{x\}} \mathsf{ok}}\,(\mathcal{C}/\mathsf{envok.c.x})$$

IV.5.5.2 $\boxed{\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{T} : \mathsf{K}}$ Sortage des types

Des règles de sortage des types raffinent les règles de correction des types du système $\mathcal{E}$. L'ajout des couleurs ne présente pas de difficulté. Comme mentionné à la section IV.5.3.6, un foncteur génératif est toujours monomorphe (alors qu'un foncteur applicatif a la sorte de son résultat) ; la règle ($\mathcal{E}$/tok.fun) est scindée afin de traiter les deux cas correctement.

$$\frac{\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{ok}}{\mathsf{B};\Gamma \vdash_{\mathsf{c}} \textsc{bool} : \wr}\,(\mathcal{C}/\mathsf{tok.base.bool}) \qquad \frac{\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{ok}}{\mathsf{B};\Gamma \vdash_{\mathsf{c}} \textsc{int} : \wr}\,(\mathcal{C}/\mathsf{tok.base.int}) \qquad \frac{\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{ok}}{\mathsf{B};\Gamma \vdash_{\mathsf{c}} \textsc{unit} : \wr}\,(\mathcal{C}/\mathsf{tok.base.unit})$$

$$\frac{\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{E} :^{\mathsf{P}} \textsc{type}^{\mathsf{K}}}{\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{Typ}\,\mathsf{E} : \mathsf{K}}\,(\mathcal{C}/\mathsf{tok.field})$$

$$\frac{\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{T}' : \mathsf{K}' \qquad \mathsf{B};\Gamma, x :_{\mathsf{c}} \mathsf{T}' \vdash_{\mathsf{c} \cup \{x\}} \mathsf{T}'' : \mathsf{K}''}{\mathsf{B};\Gamma \vdash_{\mathsf{c}} \Pi x : \mathsf{T}'.\,^{\mathsf{P}}\mathsf{T}'' : \mathsf{K}''}\,(\mathcal{C}/\mathsf{tok.fun.P})$$

$$\frac{\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{T}' : \mathsf{K}' \qquad \mathsf{B};\Gamma, x :_{\mathsf{c}} \mathsf{T}' \vdash_{\mathsf{c} \cup \{x\}} \mathsf{T}'' : \mathsf{K}''}{\mathsf{B};\Gamma \vdash_{\mathsf{c}} \Pi x : \mathsf{T}'.\,^{\mathsf{I}}\mathsf{T}'' : \wr}\,(\mathcal{C}/\mathsf{tok.fun.I})$$

$$\frac{\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{T}' : \mathsf{K}' \qquad \mathsf{B};\Gamma, x :_{\mathsf{c}} \mathsf{T}' \vdash_{\mathsf{c} \cup \{x\}} \mathsf{T}'' : \mathsf{K}''}{\mathsf{B};\Gamma \vdash_{\mathsf{c}} \Sigma x : \mathsf{T}'.\,\mathsf{T}'' : \mathsf{K}' \vee \mathsf{K}''}\,(\mathcal{C}/\mathsf{tok.pair})$$

$$\frac{B;\Gamma \vdash_c \mathsf{ok}}{B;\Gamma \vdash_c \textsc{type}^K : *}\ (\mathcal{C}/\textbf{tok.type}) \qquad \frac{B;\Gamma \vdash_c E :^P T}{B;\Gamma \vdash_c S(E) : \wr}\ (\mathcal{C}/\textbf{tok.sing})$$

Une règle supplémentaire indique que tout type monomorphe est *a fortiori* un type polymorphe. De même, tout champ type contenant un type monomorphe peut être vu comme contenant un type polymorphe, ce que nous imposons en énonçant que le type du premier est un sous-type du type du second.

$$\frac{B;\Gamma \vdash_c T : K' \qquad \text{si } K' \leqslant K}{B;\Gamma \vdash_c T : K}\ (\mathcal{C}/\textbf{tok.sub}) \qquad \frac{B;\Gamma \vdash_c \mathsf{ok} \qquad \text{si } K_1 \leqslant K_2}{B;\Gamma \vdash_c \textsc{type}^{K_1} <: \textsc{type}^{K_2}}\ (\mathcal{C}/\textbf{tsub.cong.type})$$

Donnons également la règle de formation des champs types, modifiée pour tenir compte de la sorte du type.

$$\frac{B;\Gamma \vdash_c T : K}{B;\Gamma \vdash_c \langle T\rangle :^P \textsc{type}^K}\ (\mathcal{C}/\textbf{et.type})$$

IV.5.5.3 $\boxed{B;\Gamma \vdash_c c_0\ \mathsf{transparent}}$ Transparence d'une couleur

Une couleur primaire (apax ou variable) peut être transparente si elle est directement présente dans la couleur ambiante. Elle peut également l'être indirectement, via une variable présente dans la couleur ambiante et dont la couleur de définition rend la couleur élémentaire considérée transparente.

$$\frac{B;\Gamma \vdash_c \mathsf{ok} \qquad \text{si } \xi \in c}{B;\Gamma \vdash_c \xi\ \mathsf{transparent}}\ (\mathcal{C}/\textbf{vis.in})$$

$$\frac{B;\Gamma \vdash_c \mathsf{ok} \qquad B;\Gamma_0 \vdash_{c_0} \xi\ \mathsf{transparent} \qquad \text{si } \Gamma = (\Gamma_0, x :_{c_0} T, \Gamma_1) \wedge x \in c}{B;\Gamma \vdash_c \xi\ \mathsf{transparent}}\ (\mathcal{C}/\textbf{vis.env})$$

Une couleur est transparent si et seulement si tous ses éléments le sont.

$$\frac{B;\Gamma \vdash_c \mathsf{ok}}{B;\Gamma \vdash_c \bullet\ \mathsf{transparent}}\ (\mathcal{C}/\textbf{vis.o}) \qquad \frac{B;\Gamma \vdash_c c_1\ \mathsf{transparent} \qquad B;\Gamma \vdash_c c_2\ \mathsf{transparent}}{B;\Gamma \vdash_c c_1 \cup c_2\ \mathsf{transparent}}\ (\mathcal{C}/\textbf{vis.union})$$

La transparence d'un apax est utilisée dans la règle $(\mathcal{C}/\mathsf{tconv.abs})$ pour justifier sa révélation. La transparence d'une couleur est utilisée dans plusieurs règles ($(\mathcal{C}/\mathsf{ac.a})$, $(\mathcal{C}/\mathsf{et.x})$) pour exprimer la transparence des dépendances d'une couleur primaire.

IV.5.5.4 $\boxed{B;\Gamma \vdash_c A \triangleright E : T\ ;\ \ldots}$ Composantes

Les jugements de révélation $B;\Gamma \vdash_c A \triangleright E : T$ définissent deux attributs d'une composante A : l'expression E vers laquelle elle se révèle, et la signature apparente T issue mécaniquement de la signature donnée à l'apax sous-jacent par le lexique. La structure de la preuve de révélation suit celle de cette signature.

$$\frac{B;\Gamma \vdash_c c_0\ \mathsf{transparent} \qquad \text{si } a = E :_{c_0} T \in B}{B;\Gamma \vdash_c a \triangleright E : T}\ (\mathcal{C}/\textbf{ac.a}) \qquad \frac{B;\Gamma \vdash_c A \triangleright E : \Sigma x : T_1.\, T_2}{B;\Gamma \vdash_c \pi_1 A \triangleright \pi_1 E : T_1}\ (\mathcal{C}/\textbf{ac.proj.1})$$

$$\frac{B;\Gamma \vdash_c E_1 :^P S(\pi_1 E) \qquad B;\Gamma \vdash_c A \triangleright E : \Sigma x : T_1.\, T_2}{B;\Gamma \vdash_c \pi_2 A \triangleright \pi_2 E : \{x \leftarrow_c E_1\} T_2}\ (\mathcal{C}/\textbf{ac.proj.2})$$

$$\frac{B;\Gamma \vdash_c A \triangleright E : \Pi x : T_0.\, {}^P T_1 \qquad B;\Gamma \vdash_c E_0 :^P T_0}{B;\Gamma \vdash_c A\, E_0 \triangleright E\, E_0 : \{x \leftarrow_c E_0\} T_1}\ (\mathcal{C}/\textbf{ac.app})$$

Lorsqu'une composante a la signature apparente $\textsc{type}^K$, elle peut servir à former un type abstrait. Si l'apax sous-jacent est de plus transparent, ce type abstrait peut être transformé en l'implémentation révélée.

$$\frac{B; \Gamma \vdash_c A \rhd E : \text{TYPE}^K}{B; \Gamma \vdash_c (\!|A|\!) : K} \ (\mathcal{C}/\textbf{tok.abs})$$

$$\frac{B; \Gamma \vdash_c A \rhd E : \text{TYPE}^K \qquad B; \Gamma \vdash_c \mathbf{underl}(A) \ \mathsf{transparent}}{B; \Gamma \vdash_c (\!|A|\!) \longrightarrow \mathsf{Typ}\ E} \ (\mathcal{C}/\textbf{tconv.abs})$$

Une composante est presque inerte, la seule conversion qui puisse significativement l'affecter étant sa révélation. Des règles contextuelles permettent néanmoins la conversion des expressions imbriquées.

$$\frac{B; \Gamma \vdash_c E_0 \longrightarrow E_0' \qquad B; \Gamma \vdash_c E_0 :^{P} T_0 \qquad B; \Gamma \vdash_c A \rhd E : \Pi x : T_0.\,^{P}T_1}{B; \Gamma \vdash_c A\,E_0 \longrightarrow A\,E_0'} \ (\mathcal{C}/\textbf{aconv.cong.app.arg})$$

$$\frac{B; \Gamma \vdash_c A \longrightarrow A' \qquad B; \Gamma \vdash_c A \rhd E : \Pi x : T_0.\,^{P}T_1 \qquad B; \Gamma \vdash_c E_0 :^{P} T_0}{B; \Gamma \vdash_c A\,E_0 \longrightarrow A'\,E_0} \ (\mathcal{C}/\textbf{aconv.cong.app.fun})$$

$$\frac{B; \Gamma \vdash_c A \longrightarrow A' \qquad B; \Gamma \vdash_c A \rhd E : \Sigma x : T_1.\, T_2}{B; \Gamma \vdash_c \pi_i A \longrightarrow \pi_i A} \ (\mathcal{C}/\textbf{aconv.cong.proj})$$

$$\frac{B; \Gamma \vdash_c A \longrightarrow A' \qquad B; \Gamma \vdash_c A \rhd E : \text{TYPE}^K}{B; \Gamma \vdash_c (\!|A|\!) \longrightarrow (\!|A'|\!)} \ (\mathcal{C}/\textbf{tconv.cong.abs})$$

L'équivalence de convertibilité des composantes abstraites est définie par quatre règles ($\mathcal{C}$/aeq.refl), ($\mathcal{C}$/aeq.sym), ($\mathcal{C}$/aeq.trans) et ($\mathcal{C}$/aeq.conv) sur le modèle des règles (teq.*) et (eeq.*).

IV.5.5.5 $\boxed{B; \Gamma \vdash_c E :^{\gamma} T}$ Coloration des expressions

La règle de typage des variables a une nouvelle condition qui impose la transparence de la variable dans la couleur ambiante. Cette transparence assure que ses dépendances seront toujours présentes si la couleur ambiante est affaiblie.

$$\frac{B; \Gamma \vdash_c x \ \mathsf{transparent} \qquad \text{si } x :_{c'} T \in \Gamma}{B; \Gamma \vdash_c x :^{P} T} \ (\mathcal{C}/\textbf{et.x})$$

Les crochets colorés entourent une expression de couleur différente. L'annotation de type sur le crochet doit être valable à la fois à l'intérieur et à l'extérieur, ce pour quoi nous utilisons l'intersection des deux couleurs.

$$\frac{B; \Gamma \vdash_{c'} E :^{\gamma} T \qquad B; \Gamma \vdash_{c \cap c'} T : \wr \qquad B; \Gamma \vdash_c \mathsf{ok}}{B; \Gamma \vdash_c [E]^{T}_{c'} :^{\gamma} T} \ (\mathcal{C}/\textbf{et.col})$$

La règle de typage du scellage doit tenir compte de l'annotation de couleur, qui est additive.

$$\frac{B; \Gamma \vdash_{c \cup c'} E :^{\gamma} T \qquad B; \Gamma \vdash_c T : *}{B; \Gamma \vdash_c (E \,!!_{c'}\, T) :^{I} T} \ (\mathcal{C}/\textbf{et.seal})$$

IV.5.5.6 $\boxed{B; \Gamma \vdash_c E \longrightarrow E'}$ Conversion et crochets colorés

De nouvelles règles de conversion reflètent les nouvelles règles de réduction concernant les crochets : les nouveaux contextes de réduction, et les règles de poussée.

$$\frac{B; \Gamma \vdash_{c'} E \longrightarrow E' \qquad B; \Gamma \vdash_{c'} E :^{P} T \qquad B; \Gamma \vdash_{c \cap c'} T : \wr \qquad B; \Gamma \vdash_c \mathsf{ok}}{B; \Gamma \vdash_c [E]^{T}_{c'} \longrightarrow [E']^{T}_{c'}} \ (\mathcal{C}/\textbf{econv.cong.col.e})$$

$$\frac{B; \Gamma \vdash_{c'} E :^{P} T_1 \qquad B; \Gamma \vdash_{c \cap c'} T_1 \longrightarrow T_2 \qquad B; \Gamma \vdash_{c \cap c'} T_1 : \wr \qquad B; \Gamma \vdash_c \mathsf{ok}}{B; \Gamma \vdash_c [E]^{T_1}_{c'} \longrightarrow [E]^{T_2}_{c'}} \ (\mathcal{C}/\textbf{econv.cong.col.t})$$

$$\frac{B;\Gamma \vdash_{c'} \mathsf{ok} \qquad B;\Gamma \vdash_{c} \mathsf{ok}}{B;\Gamma \vdash_{c} [()]_{c'}^{\mathrm{UNIT}} \longrightarrow ()}\ (\mathcal{C}/\textbf{econv.col.base.unit}) \qquad \frac{B;\Gamma \vdash_{c'} \mathsf{ok} \qquad B;\Gamma \vdash_{c} \mathsf{ok}}{B;\Gamma \vdash_{c} [bv]_{c'}^{\mathrm{BOOL}} \longrightarrow bv}\ (\mathcal{C}/\textbf{econv.col.base.bool})$$

$$\frac{B;\Gamma \vdash_{c'} \mathsf{ok} \qquad B;\Gamma \vdash_{c} \mathsf{ok}}{B;\Gamma \vdash_{c} [\underline{n}]_{c'}^{\mathrm{INT}} \longrightarrow \underline{n}}\ (\mathcal{C}/\textbf{econv.col.base.int})$$

$$\frac{\begin{array}{c} B;\Gamma \vdash_{c'} T_0 <: T_2 \qquad B;\Gamma, x :_{c'} T_2 \vdash_{c' \cup \{x\}} E :^{P} T_1 \\ B;\Gamma \vdash_{c} \mathsf{ok} \qquad B;\Gamma, x :_{c \cap c'} T_0 \vdash_{(c\cap c')\cup\{x\}} T_1 : \wr \end{array}}{B;\Gamma \vdash_{c} [\lambda x : T_2.\ E]_{c'}^{\Pi x : T_0.\,^{P} T_1} \longrightarrow \lambda x : T_0.\ [E]_{c' \cup \{x\}}^{T_1}}\ (\mathcal{C}/\textbf{econv.col.fun.P})$$

$$\frac{\begin{array}{c} B;\Gamma \vdash_{c'} T_0 <: T_2 \qquad B;\Gamma, x :_{c'} T_2 \vdash_{c' \cup \{x\}} E :^{I} T_1 \\ B;\Gamma \vdash_{c} \mathsf{ok} \qquad B;\Gamma, x :_{c \cap c'} T_0 \vdash_{(c\cap c')\cup\{x\}} T_1 : \wr \end{array}}{B;\Gamma \vdash_{c} [\lambda x : T_2.\ E]_{c'}^{\Pi x : T_0.\,^{I} T_1} \longrightarrow \lambda x : T_0.\ E\ !!_{c' \cup \{x\}}\ T_1}\ (\mathcal{C}/\textbf{econv.col.fun.I})$$

$$\frac{\begin{array}{ccc} B;\Gamma \vdash_{c'} E_1 :^{P} T_1 & B;\Gamma \vdash_{c \cap c'} T_1 : \wr & B;\Gamma, x :_{c'} T_1 \vdash_{c' \cup \{x\}} E_2 :^{P} T_2 \\ B;\Gamma \vdash_{c} \mathsf{ok} & B;\Gamma, x :_{c \cap c'} T_1 \vdash_{(c\cap c')\cup\{x\}} T_2 : \wr & B;\Gamma \vdash_{c'} E_2 :^{P} \{x \leftarrow_{c'} [E_1]_{c'}^{T_1}\} T_2 \end{array}}{B;\Gamma \vdash_{c} [(E_1, E_2)]_{c'}^{\Sigma x : T_1.\ T_2} \longrightarrow ([E_1]_{c'}^{T_1}, [E_2]_{c'}^{\{x \leftarrow_{c'} [E_1]_{c'}^{T_1}\} T_2})}\ (\mathcal{C}/\textbf{econv.col.pair})$$

$$\frac{\begin{array}{c} B;\Gamma \vdash_{c_2} E :^{P} T_2 \qquad B;\Gamma \vdash_{c_1 \cap c_2} T_2 : \wr \\ B;\Gamma \vdash_{c_1} T_2 <: T_1 \qquad B;\Gamma \vdash_{c \cap c_1} T_1 : \wr \qquad B;\Gamma \vdash_{c} \mathsf{ok} \end{array}}{B;\Gamma \vdash_{c} [[E]_{c_2}^{T_2}]_{c_1}^{T_1} \longrightarrow [E]_{c_1 \cup c_2}^{T_1}}\ (\mathcal{C}/\textbf{econv.col.merge})$$

$$\frac{B;\Gamma \vdash_{c'} E' :^{P} S(E) \qquad B;\Gamma \vdash_{c \cap c'} E :^{P} T \qquad B;\Gamma \vdash_{c} \mathsf{ok}}{B;\Gamma \vdash_{c} [E']_{c'}^{S(E)} \longrightarrow E}\ (\mathcal{C}/\textbf{econv.col.sing})$$

$$\frac{\begin{array}{c} B;\Gamma, x :_{c} T_0 \vdash_{c'} E_1 :^{\gamma} T_1 \quad B;\Gamma, x :_{c \cap c'} T_0 \vdash_{(c\cap c')\cup\{x\}} T_1 \longrightarrow T_1' \\ B;\Gamma, x :_{c \cap c'} T_0 \vdash_{(c\cap c')\cup\{x\}} T_1 : \wr \end{array}}{B;\Gamma \vdash_{c} (\lambda x : T_0.\ E_1\ !!_{c'}\ T_1) \longrightarrow (\lambda x : T_0.\ E_1\ !!_{c'}\ T_1')}\ (\mathcal{C}/\textbf{econv.cong.fun.seal})$$

IV.6 Typage dynamique et distribution $\boxed{\mathcal{D}}$

IV.6.1 Typage dynamique

IV.6.1.1 Introduction

Jusqu'à présent, bien que nous mélangions sans vergogne types et termes, nous avons respecté une stricte *séparation des phases* [HMM90], au sens où nous avons décrit d'une part un système de typage sans référence à l'exécution d'un programme (seuls les termes purs sont évalués, et la pureté est définie de telle sorte que ceci soit possible) et d'autre part une notion d'exécution (réduction) qui n'échoue pas pour les programmes bien typés. (De plus l'exécution ne dépend des types que par les crochets colorés, dont la présence sert seulement à assurer des propriétés de typage.)

Nous allons maintenant ajouter à notre langage un moyen de tester à l'exécution le type d'une valeur, inspiré de ceux que nous avons décrit à la section I.3.2. Une telle fonctionnalité a plusieurs intérêts :

- le typage dynamique permet de contourner le typage statique, pour écrire un programme dont la correction dépasse les capacités du système de typage statique ;
- certains usages, comme la programmation répartie (voir la section II.1), ne se prêtent pas à une séparation en deux phases, requérant des vérifications de typage après le début de l'exécution ;

– certains algorithmes ou idiomes bénéficient d'une discrimination sur le type d'une valeur : le typage dynamique est un cas particulier de programmation générique.

Dans le premier cas, le test de typage doit toujours réussir ; un échec indiquerait une erreur du programmeur. Dans le deuxième cas, le test peut réussir ou échouer, traduisant une condition externe (par exemple, la bonne configuration du déploiement d'un programme sur un réseau, ou le chargement d'une version acceptable d'une bibliothèque). Dans le troisième cas, le test n'est pas booléen, et il n'y a pas forcément de notion d'échec : il s'agit d'une analyse de cas.

Nous allons nous concentrer sur le deuxième cas, et mettre en place un moyen de tester à l'exécution si une valeur a un type donné. L'ajout des construction supplémentaires forme le système $\mathcal{D}$.

IV.6.1.2 Dynamiques

Fidèle à la tradition, nous équipons notre langage de **dynamiques**, c'est-à-dire de valeurs dont le type est déterminé à l'exécution. La construction d'un dynamique est notée $\mathsf{dyn\,E\,at\,T}$, l'expression E devant (statiquement) avoir le type T. Le type d'un dynamique est toujours DYN, indépendamment du type effectif de l'expression sous-jacente. Le décodage d'un dynamique doit spécifier le type attendu, et échoue si le type effectif diffère du type attendu ; le destructeur des dynamiques a donc la forme $\mathsf{undyn\,E\,at\,T\,else\,E'}$, dans laquelle le dynamique E devant avoir le type DYN, et la clause $\mathsf{E'}$ doit avoir le type T et est utilisée en cas d'échec. Nous utiliserons la notation simplifiée $\mathsf{undyn\,E\,at\,T}$ dans lorsque le comportement en cas d'échec ne nous importe pas (dans un langage de programmation, cette syntaxe correspondrait à lever une exception en cas d'échec du typage dynamique).

Notre langage étant équipé d'une riche équivalence entre types, cette équivalence doit être invoquée lors d'une vérification dynamique de type. C'est donc le jugement $\mathsf{nil} \vdash \mathsf{T} \equiv \mathsf{T'}$ qui doit être vérifié lors de l'évaluation de l'expression $\mathsf{undyn\,(dyn\,E\,at\,T)\,at\,T'\,else\,E'}$. Si nous sommes capables de définir une représentation canonique pour les types, il suffit tester l'égalité des représentations ; sinon une partie du typeur doit se retrouver dans l'environnement d'exécution. En fait, notre système est muni de sous-typage ; il est donc logique d'autoriser $\mathsf{undyn\,(dyn\,E\,at\,T)\,at\,T'\,else\,E'}$ à réussir dès lors que le jugement $\mathsf{nil} \vdash \mathsf{T} <: \mathsf{T'}$ est vérifié, ce qui est la condition optimale pour garantir que E a le type $\mathsf{T'}$ du moment qu'elle a le type T.

Par rapport à ce qui se fait en ML, notons qu'une expression E peut être un module, contenant en particulier des champs types, ce qui est utile dans de nombreuses situations. Ainsi notre système sait dynamiser une structure de données polymorphe (dont le type a la forme $\Pi\mathsf{t} : \text{TYPE}.\,\mathsf{T}$ où $\mathsf{Typ\,t}$ est le type des éléments) ; un serveur de calcul générique attendra des valeurs de type $\Sigma\mathsf{t} : \text{TYPE}.\,\Sigma\mathsf{t'} : \text{TYPE}.\,\mathsf{Typ\,t} * (\mathsf{Typ\,t} \to \mathsf{Typ\,t'})$; le chargement dynamique de module pourra lire des valeurs de type DYN.

IV.6.1.3 Correspondance entre le typage statique et le typage dynamique

Une question qui se pose naturellement puisque nous disposons désormais de deux notions de comparaison de type — statique et dynamique — est si ces deux notions sont vraiment identiques. C'est en apparence le cas puisque nous avons défini la vérification dynamique de typage par référence à un jugement de typage statique : que E ait dynamiquement le type $\mathsf{T'}$ signifie que E a statiquement le type T (pour que $\mathsf{dyn\,E\,at\,T}$ soit bien typé) et que $\mathsf{T} <: \mathsf{T'}$ (ce qui implique que E a statiquement le type $\mathsf{T'}$, et réciproquement on peut choisir $\mathsf{T} = \mathsf{T'}$).

En fait, cette correspondance n'est plus valable *a priori* si $\mathsf{dyn\,E\,at\,T}$ n'est pas une valeur : il faut aussi vérifier que la réduction de $\mathsf{dyn\,E\,at\,T}$ (en $\mathsf{dyn\,E_1\,at\,T_1}$) ne change pas l'ensemble des types qu'a l'expression ni l'ensemble des expressions ayant le type. La préservation du typage par réduction apporte une garantie dans un sens : si un programme est bien typé, on peut introduire une vérification dynamique de typage en n'importe quel point en remplaçant une expression E utilisée dans un contexte exigeant un type T par $\mathsf{undyn}\,(\mathsf{dyn}\,\mathsf{E}\,\mathsf{at}\,\mathsf{T})\,\mathsf{at}\,\mathsf{T}$, et la vérification dynamique réussit forcément. Dans l'autre sens, en revanche, le test dynamique est quelquefois plus permissif, car la réduction peut conduire deux types à « devenir » égaux. S'agissant de types abstraits, leur représentation à l'exécution par des apax vise à préserver l'abstraction, ce qui signifie notamment que deux apax correspondant à des types statiquement distincts sont distincts. Nous discuterons en revanche à la section IV.6.3.4 d'empreintes structurelles (par opposition aux empreintes singularisées que constituent les apax ; voir la section II.6.1.2) qui servent précisément à rendre compatibles certains types abstraits que le typage statique distingue.

Un cas où le typage statique est par essence insuffisant est celui où des types apparaissent durant l'exécution : non pas des types abstraits, qui sont créés lors de l'exécution mais en suivant des règles décrite statiquement, mais des types dont l'origine même n'est pas connue à l'avance. Ce phénomène se produit lorsque la contrainte de type à vérifier dynamiquement est polymorphe (incomplètement spécifiée), au sens de la section IV.5.3.2. Il s'agit de champs types de modules dont le type a été vérifié dynamiquement : leur archétype est $\mathsf{undyn}\,\mathsf{E}\,\mathsf{at}\,\textsc{type}$. Considérons par exemple le fragment programme suivant :

$$\mathsf{let}\ \mathsf{t}_1 = \mathsf{undyn}\ \mathsf{E}_1\ \mathsf{at}\ \textsc{type}\ \mathsf{in}\ \ \mathsf{let}\ \mathsf{t}_2 = \mathsf{undyn}\ \mathsf{E}_2\ \mathsf{at}\ \textsc{type}\ \mathsf{in}\ \ \ldots$$

Les types $\mathsf{Typ}\,\mathsf{t}_1$ et $\mathsf{Typ}\,\mathsf{t}_2$ ne peuvent en aucun cas être égaux statiquement (au moins lorsque E_1 et E_2 ont des effets de bord). Nous choisirons pourtant de permettre qu'ils soient égaux, par exemple, si E_1 et E_2 ont la même valeur. Ainsi la construction $\mathsf{undyn}\,\mathsf{E}\,\mathsf{at}\,\mathsf{T}$ ne contribue pas à l'abstraction : l'annotation de type qu'elle contient est une pure ascription et pas un scellage — si l'abstraction est désirée, il suffit d'écrire $(\mathsf{undyn}\,\mathsf{E}\,\mathsf{at}\,\mathsf{T})\ !!\ \mathsf{T}$. Ce choix de conception est motivé par le désir de pouvoir manipuler des dynamiques sans perdre d'information. Considérons par exemple le programme suivant qui fabrique un dynamique en recopiant partiellement un autre :

$$\begin{array}{ll}
\mathsf{let}\ x = \mathsf{dyn}\ (0, (\langle\textsc{bool}\rangle, \mathsf{true}\ ))\ \mathsf{at}\ \textsc{int} * (\mathsf{S}(\langle\textsc{bool}\rangle) * \textsc{bool})\ \mathsf{in} & \\
\mathsf{let}\ x' = & \mathsf{let}\ y = (\mathsf{undyn}\ x\ \mathsf{at}\ \textsc{int} * \Sigma \mathsf{t} : \textsc{type}.\ \mathsf{Typ}\ \mathsf{t})\ \mathsf{in} \\
& \mathsf{dyn}\ (\mathsf{add}\,(\pi_1 y, 1), \pi_2 y)\ \mathsf{at}\ \textsc{int} * \Sigma \mathsf{t} : \mathsf{S}(\pi_1 \pi_2 y).\ \mathsf{Typ}\ \mathsf{t} \quad \mathsf{in} \\
\mathsf{undyn}\ x'\ \mathsf{at}\ \textsc{int} * (\mathsf{S}(\langle\textsc{bool}\rangle) * \textsc{bool}) &
\end{array}$$

Le champ type porté par le dynamique reste compatible avec BOOL, et la vérification dynamique de typage finale réussit.

Au moment de la dynamisation, l'intérêt d'une annotation de type polymorphe n'est pas clair. Si le programmeur désire dynamiser E en abstrayant une partie de l'information de typage, il a toujours le loisir d'utiliser explicitement le scellage, et nous jugeons ceci préférable pour la clarté. Aussi restreyons-nous dans une expression $\mathsf{dyn}\,\mathsf{E}\,\mathsf{at}\,\mathsf{T}$ le type T à être monomorphe.

IV.6.1.4 Dynamiques et couleurs

Un dynamique porte une annotation de type. Comme nous l'avons vu à la section IV.5.3.1, ce type est influencé par la couleur ambiante de deux manières : d'une part, une partie de la couleur est nécessaire à la bonne formation du type ; d'autre part, le reste de la couleur élargit la sémantique du type en lui rendant plus de types équivalents. La couleur de la dynamisation et la couleur de la vérification dynamique de typage (que nous désignerons respectivement par « couleur de production » et « couleur d'utilisation ») n'ont aucun lien *a priori*. Plusieurs stratégies sont envisageables pour gérer la différence.

Une possibilité est d'exiger que la couleur d'utilisation soit un sur-ensemble de la couleur de production. C'est en fait ce qui se passe naturellement si l'on ne se préoccupe pas de l'interaction entre le typage dynamique et les crochets colorés. Dans ce cas, l'interface externe du dynamique doit comprendre une couleur en plus du type. Un inconvénient majeur de cette approche est que la couleur de production peut contenir des éléments inutiles, difficilement contrôlables, que le code d'utilisation n'a pas moyen de connaître. Une telle approche n'est donc tenable qu'en présence d'un moyen de transférer des couleurs — l'idée n'est pas absurde mais constituerait un ajout majeur au langage.

À l'inverse, on peut exiger que les dynamiques soient munis d'une interface valable dans n'importe quelle couleur — une interface **universelle**. Ceci revient à exiger que ladite interface soit valable dans la couleur vide. Sur le modèle de HAT (voir la section III.2.5), notons $\mathsf{dynned}\,\mathsf{E}\,\mathsf{at}\,\mathsf{T}$ un dynamique dont l'interface T est garantie typable dans la couleur vide. Un tel dynamique est appelé **dynamique universel**.

L'intégration de $\mathsf{dynned}\,\mathsf{E}\,\mathsf{at}\,\mathsf{T}$ dans le langage sans casser les différentes propriétés liées aux couleurs que nous avons pu voir et utiliser dans l'élaboration du système $\mathcal{C}$ est fortement contrainte. En particulier, comme T est typable dans la couleur vide, T ne peut pas contenir de variable libre (section IV.5.2.3 : une variable ne peut être utilisée que dans une couleur dont elle est membre). Cela ne limite pas le pouvoir expressif du langage, à condition que $\mathsf{dynned}\,\mathsf{E}\,\mathsf{at}\,\mathsf{T}$ soit destiné à être produit par évaluation d'une expression de la forme $\mathsf{dyn}\,\mathsf{E}_0\,\mathsf{at}\,\mathsf{T}_0$ — si celle-ci est en position d'être réduite, elle est nécessairement close.

À partir de E_0 ayant le type T_0 dans la couleur ambiante c, nous devons produire E ayant le type T dans la couleur vide, avec E et T équivalents à E_0 et T_0 dans c. Nous avons vu à la section IV.5.2.2 un moyen d'extirper un type de la couleur ambiante : l'opération de concrétisation. Posons donc $\mathsf{T} = \mathbf{conc}^{\mathsf{B}}_{\mathsf{c}}(\mathsf{T}_0)$ (où B est le lexique courant). Nous savons de même mémoire produire une expression de type T à partir de E_0 : il suffit d'utiliser un crochet coloré. Nous posons donc $\mathsf{E} = [\mathsf{E}_0]^{\mathsf{T}}_{\mathsf{c}} = [\mathsf{E}_0]^{\mathbf{conc}^{\mathsf{B}}_{\mathsf{c}}(\mathsf{T}_0)}_{\mathsf{c}}$. La règle de formation des dynamiques universels est donc

$$\mathsf{dyn}\,\mathsf{E}_0\,\mathsf{at}\,\mathsf{T}_0 \longrightarrow_{\mathsf{c}} \mathsf{dynned}\,[\mathsf{E}_0]^{\mathbf{conc}^{\mathsf{B}}_{\mathsf{c}}(\mathsf{T}_0)}_{\mathsf{c}}\,\mathsf{at}\,\mathbf{conc}^{\mathsf{B}}_{\mathsf{c}}(\mathsf{T}_0)$$

Dans cette approche, la couleur de production du dynamique ne l'infecte pas inutilement : les règles de poussée des crochets du système $\mathcal{C}$ conduisent à leur élimination chaque fois qu'ils sont inutiles.

D'un point de vue technique, l'apparition de c dans le résidu, notamment comme annotation du crochet, fait que cette règle peut être utilisée pour l'évaluation durant l'exécution mais pas en tant que règle de conversion durant le typage. En effet, cette règle ne vérifie pas la propriété d'affaiblissement de la couleur (voir la section IV.5.2.1) : si la couleur ambiante est affaiblie en c', le résidu change. L'intégration de la couleur ambiante dans l'expression en cours de calcul apparaît comme un effet de bord d'un genre particulier. En conséquence, une expression $\mathsf{dyn}\,\mathsf{E}\,\mathsf{at}\,\mathsf{T}$ ne saurait être pure ; en revanche, rien ne s'oppose à ce que $\mathsf{dynned}\,\mathsf{E}\,\mathsf{at}\,\mathsf{T}$ le soit (pourvu bien sûr que E soit pure) — et cela est heureux, puisque la valeur d'une telle expression a la forme $\mathsf{dynned}\,\mathsf{V}\,\mathsf{at}\,\mathsf{T}$ et doit être pure.

IV.6.2 Formalisation

IV.6.2.1 Syntaxe

Nous définissons un nouveau calcul, le système $\mathcal{D}$. Ce système est une extension conservative du système $\mathcal{C}$. Au niveau syntaxique, nous ajoutons un type ainsi que deux constructeurs et un destructeur pour ce type.

T ::=	**type**
...	
DYN	valeur dynamiquement typée
E ::=	**expression**
...	
$\mathsf{dyn\,E\,at\,T}$	dynamique
$\mathsf{dynned\,E\,at\,T}$	dynamique universel
$\mathsf{undyn\,E\,at\,T\,else\,E'}$	vérification de typage dynamique

IV.6.2.2 Réduction

Un dynamique universel d'une valeur est une valeur. Les nouvelles constructions sont des contextes d'évaluation.

V ::=	**quasi-valeur**
...	
$\mathsf{dynned\,V^\bullet\,at\,T}$	dynamique universel
$\mathsf{V^c}$ ::=	**valeur dans c**
...	
$\mathsf{dynned\,V^\bullet\,at\,T}$	dynamique universel
$\mathsf{C^c_{c'}}$ ::=	**contexte d'évaluation (de couleurs intérieure $\mathsf{c'}$ et extérieure c)**
...	
dyn __ $\mathsf{at\,T}$	dynamique
dynned __ $\mathsf{at\,T}$	dynamique universel, lorsque $\mathsf{c'} = \bullet$
undyn __ $\mathsf{at\,T\,else\,E'}$	vérification de typage dynamique

Nous avons vu à la section IV.6.1.4 comment est produit un dynamique universel à partir d'un dynamique. L'évaluation d'une vérification de typage dynamique conduit soit à accepter la valeur sous-jacente si les types sont compatibles, soit à évaluer l'expression de remplacement. Une nouvelle règle permet de pousser un crochet sous un dynamique. Un crochet coloré autour d'un dynamique universel peut simplement être effacé : son contenu est déjà protégé contre tous les assauts.

$$\mathsf{dyn\,V^c\,at\,T} \longrightarrow_{\mathsf{c}} \mathsf{dynned}\,[\mathsf{V^c}]_{\mathsf{c}}^{\mathbf{conc}_{\mathsf{c}}^{\mathsf{B}}(\mathsf{T})}\,\mathsf{at}\,\mathbf{conc}_{\mathsf{c}}^{\mathsf{B}}(\mathsf{T}) \qquad (\mathcal{D}/\mathsf{ered.dyn})$$

$$\mathsf{B} \vdash \mathsf{undyn\,(dyn\,V^c\,at\,T)\,at\,T'\,else\,E'} \longrightarrow_{\mathsf{c}} \mathsf{B} \vdash \begin{cases} \mathsf{V^c} & \text{si } \mathsf{B;nil} \vdash_{\mathsf{c}} \mathsf{T} <: \mathsf{T'} \\ \mathsf{E'} & \text{sinon} \end{cases} \qquad (\mathcal{D}/\mathsf{ered.undyn})$$

$$[\mathsf{dynned\,V^\bullet\,at\,T}]_{\mathsf{c'}}^{\text{DYN}} \longrightarrow_{\mathsf{c}} \mathsf{dynned\,V^\bullet\,at\,T} \qquad (\mathcal{D}/\mathsf{ered.col.dynned})$$

IV.6.2.3 Typage

Comme nous l'avons vu à la section IV.6.1.4, les dynamiques ordinaires $\mathsf{dyn\,E\,at\,T}$ sont impurs, mais les dynamiques universels $\mathsf{dynned\,E\,at\,T}$ (utiles seulement avec E pure) sont purs. Nous ne souhaitons avoir à véritablement gérer le typage dynamique au sein du compilateur[31] ; aussi déclarons-nous toute expression de la forme $\mathsf{undyn\,E\,at\,T\,else\,E'}$ comme impure.

$$\frac{\mathsf{B;\Gamma \vdash_c\ ok}}{\mathsf{B;\Gamma \vdash_c}\ \text{DYN} : \mathsf{K}}\ (\mathcal{D}/\textbf{tok.base.dyn}) \qquad \frac{\mathsf{B;\Gamma \vdash_c E :^{\Upsilon} T} \qquad \mathsf{B;\Gamma \vdash_c T : \wr}}{\mathsf{B;\Gamma \vdash_c dyn\,E\,at\,T :^{I}}\ \text{DYN}}\ (\mathcal{D}/\textbf{et.dyn})$$

[31]Non seulement ce serait inutile, mais cela permettrait de contourner une éventuelle stratification (voir la section V.3.1.1).

$$\frac{\mathsf{B};\Gamma \vdash_\bullet \mathsf{E} :^{\mathtt{P}} \mathsf{T} \qquad \mathsf{B};\Gamma \vdash_\bullet \mathsf{T} : \wr \qquad \mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{ok}}{\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{dynned}\ \mathsf{E}\ \mathsf{at}\ \mathsf{T} :^{\mathtt{P}} \textsc{dyn}}\ (\mathcal{D}/\textbf{et.dynned})$$

$$\frac{\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{E} :^{\Upsilon} \textsc{dyn} \qquad \mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{E}' :^{\Upsilon} \mathsf{T}}{\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{undyn}\ \mathsf{E}\ \mathsf{at}\ \mathsf{T}\ \mathsf{else}\ \mathsf{E}' :^{\mathtt{I}} \mathsf{T}}\ (\mathcal{D}/\textbf{et.undyn})$$

Puisque nous avons ajouté un nouveau constructeur au langage, nous devons ajouter des règles de conversion : des règles de congruence pour réécrire les arguments du constructeur, et une règle de poussée de crochet (reflètant la règle de réduction ($\mathcal{D}$/ered.col.dynned)).

$$\frac{\mathsf{B};\Gamma \vdash_\bullet \mathsf{E} \longrightarrow \mathsf{E}' \qquad \mathsf{B};\Gamma \vdash_\bullet \mathsf{T} : \wr \qquad \mathsf{B};\Gamma \vdash_\bullet \mathsf{E} :^{\mathtt{P}} \mathsf{T} \qquad \mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{ok}}{\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{dynned}\ \mathsf{E}\ \mathsf{at}\ \mathsf{T} \longrightarrow \mathsf{dynned}\ \mathsf{E}'\ \mathsf{at}\ \mathsf{T}}\ (\mathcal{D}/\textbf{econv.cong.dynned.e})$$

$$\frac{\mathsf{B};\Gamma \vdash_\bullet \mathsf{T} \longrightarrow \mathsf{T}' \qquad \mathsf{B};\Gamma \vdash_\bullet \mathsf{T} : \wr \qquad \mathsf{B};\Gamma \vdash_\bullet \mathsf{E} :^{\mathtt{P}} \mathsf{T} \qquad \mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{ok}}{\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{dynned}\ \mathsf{E}\ \mathsf{at}\ \mathsf{T} \longrightarrow \mathsf{dynned}\ \mathsf{E}\ \mathsf{at}\ \mathsf{T}'}\ (\mathcal{D}/\textbf{econv.cong.dynned.t})$$

$$\frac{\mathsf{B};\Gamma \vdash_\bullet \mathsf{E} :^{\mathtt{P}} \mathsf{T} \qquad \mathsf{B};\Gamma \vdash_\bullet \mathsf{T} : \wr \qquad \mathsf{B};\Gamma \vdash_{\mathsf{c}'} \mathsf{ok} \qquad \mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{ok}}{\mathsf{B};\Gamma \vdash_{\mathsf{c}} [\mathsf{dynned}\ \mathsf{E}\ \mathsf{at}\ \mathsf{T}]_{\mathsf{c}'}^{\textsc{dyn}} \longrightarrow \mathsf{dynned}\ \mathsf{E}\ \mathsf{at}\ \mathsf{T}}\ (\mathcal{D}/\textbf{econv.col.dynned})$$

Fonction de dynamisation Nos règles de typage permettent de typer la fonction $\lambda \mathsf{x}{:}\mathsf{T}.\,\mathsf{dyn}\ \mathsf{x}\ \mathsf{at}\ \mathsf{T}$ (fonction de dynamisation monomorphe), avec le type $\mathsf{T} \rightarrow^{\mathtt{I}} \textsc{dyn}$. Elles permettent également de typer la fonction de dynamisation polymorphe $\lambda \mathsf{t} : \textsc{type}^{\wr}.\ \lambda \mathsf{x} : \mathsf{Typ}\,\mathsf{t}.\ \mathsf{dyn}\,\mathsf{x}\,\mathsf{at}\,\mathsf{Typ}\,\mathsf{t}$, avec le type $\Pi \mathsf{t} : \textsc{type}^{\wr}.\ {}^{\mathtt{P}}\mathsf{Typ}\ \mathsf{t} \rightarrow^{\mathtt{I}} \textsc{dyn}$. L'application de ces fonctions de dynamisation résulte en une valeur de la forme $\mathsf{dynned}\ [\mathsf{V}]_{\mathsf{c}}^{\mathbf{conc}_{\mathsf{c}}^{\mathsf{B}}(\mathsf{T})}\ \mathsf{at}\ \mathbf{conc}_{\mathsf{c}}^{\mathsf{B}}(\mathsf{T})$ où c est la couleur ambiante.

IV.6.3 Communication inter-machines

IV.6.3.1 Introduction

Le problème qui a motivé la présente thèse est une vérification dynamique de typage dans les programmes répartis supportant les types abstraits. L'essentiel du présent chapitre était consacré aux types abstraits ; nous venons d'introduire la vérification dynamique de typage. Il nous reste à intégrer la communication inter-machines.

Comme au chapitre II, nous supposons acquis un mécanisme de sérialisation permettant d'échanger des valeurs entre des programmes qui s'exécutent sur des machines différentes. Nous utilisons le vocabulaire des réseaux, mais nos propos s'appliquent également au cas de données résidant sur un moyen de stockage persistant et manipulées successivement par plusieurs programmes.

Pour qu'une valeur envoyée par une machine A soit correctement reçue et décodée par une machine B, il faut que les programmes tournant sur ces deux machines soient d'accord sur l'interprétation donnée aux suites de bits échangées. Nous supposons que tous les programmes communicants sont écrits dans le même langage et utilisent la même bibliothèque de sérialisation ; il suffit donc de s'assurer que les valeurs échangées ne dépendent pas d'un quelconque environnement qui ne serait pas partagé par les deux machines. Or, si notre langage ne permet de construire des valeurs que suivant des règles bien définies et de manière reproductible, cette propriété n'est pas vraie au niveau des types : les types abstraits d'une machine ne sont pas forcément disponibles sur l'autre machine.

Dans un premier temps, nous allons faire l'hypothèse sûre qu'un type abstrait défini sur une machine est toujours distinct d'un type abstrait défini sur une autre machine. Nous avons consacré une part importante du chapitre II à l'étude de relâchements de cette restriction, et nous verrons à la section IV.6.3.5 comment intégrer ces idées au système $\mathcal{D}$.

IV.6.3.2 Communication et couleurs

Nous supposons ici disposer de deux primitives `send` et `recv` servant respectivement à envoyer et à recevoir une valeur, la communication pouvant s'effectuer sur un réseau, via un disque, ou par tout autre moyen. Plus précisément, puisque nous travaillons dans un langage typé, nous supposons disposer de deux familles de primitives $\mathtt{send}^{\mathsf{T}}$ et $\mathtt{recv}^{\mathsf{T}}$ où T est le type des valeurs communiquées ; leurs types sont $\mathtt{send}^{\mathsf{T}} : \mathsf{T} \rightarrow^{\mathtt{I}} \textsc{unit}$ et $\mathtt{recv}^{\mathsf{T}} : \textsc{unit} \rightarrow^{\mathtt{I}} \mathsf{T}$. Afin que la communication soit bien typée, le protocole doit ne mettre en regard un envoi par $\mathtt{send}^{\mathsf{T}}$ et une réception par $\mathtt{recv}^{\mathsf{T}'}$ que si les types T et T' sont compatibles, plus précisément s'il est garanti que toute valeur de type T ait également le type T', ce que nous traduisons par la contrainte $\mathsf{T} <: \mathsf{T}'$.

Mais attention, les types T et T' « vivent » dans des contextes différents : le point d'envoi et le point de réception peuvent disposer de types abstraits différents. D'un point de vue technique, la couleur ambiante des points d'envoi et de réception peuvent être différents. Or comme nous l'avons vu à la section IV.5.3.1, la couleur influence à la fois la validité et la sémantique d'un type. Ceci s'applique également à la valeur communiquée : elle peut avoir le type T dans la couleur d'envoi c sans pour autant l'avoir, ni même avoir un type quelconque, dans la couleur de réception c'.

Une possibilité pour assurer la sûreté de la communication est d'indexer les primitives par la couleur en plus du type, soit $\mathtt{send}_{\mathsf{c}}^{\mathsf{T}}$ et $\mathtt{recv}_{\mathsf{c}'}^{\mathsf{T}'}$, et d'exiger du protocole de communication qu'il vérifie la compatibilité des couleurs $\mathsf{c} \subseteq \mathsf{c}'$ (ou plus exactement $\Gamma \vdash_{\mathsf{c}'} \mathsf{c}\,\mathsf{transparent}$) en plus de celle des types $\Gamma \vdash_{\mathsf{c}} \mathsf{T} <: \mathsf{T}'$. Il n'est toutefois pas forcément immédiat d'adapter un protocole pour assurer la compatibilité des couleurs : en effet, les annotations de type T et T' sont la plupart du temps statiques, ou au moins fixées lorsque l'argument est réduit à une valeur, alors que la couleur ambiante peut changer au grès des manipulations effectuées sur cette valeur.

Une manière d'éviter toute incompatibilité due à une différence de couleurs est d'exiger que la couleur d'envoi soit vide, c'est-à-dire que le type d'envoi T ainsi que la valeur envoyée soient universels. Nous explorerons cette possibilité à la section IV.6.3.3.

Si l'on souhaite envoyer une valeur depuis une couleur quelconque vers une couleur quelconque, il faut protéger cette valeur. Nous avons rencontré une situation similaire à la section IV.5.2.2 : étant donnés une valeur V et un type T dans une couleur c, il s'agit de construire une valeur « équivalente » à V et ayant un type « équivalent » à T dans la couleur vide $\bullet$. La solution est d'utiliser la concrétisation pour réduire l'envoi de V à l'envoi de $[\mathsf{V}]_{\mathsf{c}}^{\mathbf{conc}_{\mathsf{c}}^{\mathsf{B}}(\mathsf{T})}$, dans laquelle $\mathbf{conc}_{\mathsf{c}}^{\mathsf{B}}(\mathsf{T})$ est le type T dans lequel les usages de c ont été développés, le crochet protégeant la valeur V en lui conférant les équations de typage dont elle a besoin. Une fois la valeur universelle $[\mathsf{V}]_{\mathsf{c}}^{\mathbf{conc}_{\mathsf{c}}^{\mathsf{B}}(\mathsf{T})}$ obtenue, nous sommes ramené au cas précédent.

IV.6.3.3 Universels

Introduction Dans cette section, nous étudions l'ajout au système $\mathcal{D}$ d'une nouvelle famille de types, les types universels : le type $\textsc{Univ}\,\mathsf{T}$ contient les valeurs universelles de type T. De telles valeurs sont formées par un constructeur $\mathsf{univ}\,\mathsf{V}$; plus généralement, si E a le type T dans la couleur vide, alors $\mathsf{univ}\,\mathsf{E}$ a le type $\textsc{Univ}\,\mathsf{T}$. Le constructeur univ sert seulement de témoin de l'universalité de son argument. Un destructeur permet d'annuler l'effet du constructeur univ : si E a le type $\textsc{Univ}\,\mathsf{T}$ dans une couleur c', alors $\mathsf{welcome}\,\mathsf{E}$ a le type T dans n'importe quelle couleur c (en effet, la valeur de E a la forme $\mathsf{univ}\,\mathsf{V}$ où V est universelle, et $\mathsf{welcome}\,\mathsf{E}$ se réduit en V qui a bien le type T dans une couleur c quelconque).

Récapitulons la formalisation des universels suivant la discussion qui précède.

Syntaxe

$\mathsf{T} ::=$	**type**
...	
$\textsc{Univ}\,\mathsf{T}$	type universel
$\mathsf{E} ::=$	**expression**
...	
$\mathsf{univ}\,\mathsf{E}$	universel
$\mathsf{welcome}\,\mathsf{E}$	décapsulation d'un universel

Typage

$$\frac{\mathsf{B};\Gamma \vdash_\bullet \mathsf{T} : \mathsf{K} \qquad \mathsf{B};\Gamma \vdash_\mathsf{c} \mathsf{ok}}{\mathsf{B};\Gamma \vdash_\mathsf{c} \textsc{Univ}\,\mathsf{T} : \mathsf{K}}\ (\mathcal{D}/\mathsf{tok.univ}) \qquad \frac{\mathsf{B};\Gamma \vdash_\bullet \mathsf{T} <: \mathsf{T}' \qquad \mathsf{B};\Gamma \vdash_\mathsf{c} \mathsf{ok}}{\mathsf{B};\Gamma \vdash_\mathsf{c} \textsc{Univ}\,\mathsf{T} <: \textsc{Univ}\,\mathsf{T}'}\ (\mathcal{D}/\mathsf{tsub.univ})$$

$$\frac{\mathsf{B};\Gamma \vdash_\bullet \mathsf{E} :^\gamma \mathsf{T} \qquad \mathsf{B};\Gamma \vdash_\mathsf{c} \mathsf{ok}}{\mathsf{B};\Gamma \vdash_\mathsf{c} \mathsf{univ}\,\mathsf{E} :^\gamma \textsc{Univ}\,\mathsf{T}}\ (\mathcal{D}/\mathsf{et.univ}) \qquad \frac{\mathsf{B};\Gamma \vdash_\mathsf{c} \mathsf{E} :^\gamma \textsc{Univ}\,\mathsf{T}}{\mathsf{B};\Gamma \vdash_\mathsf{c} \mathsf{welcome}\,\mathsf{E} :^\gamma \mathsf{T}}\ (\mathcal{D}/\mathsf{et.welcome})$$

À ces règles s'ajoutent des règles de conversion ($\mathcal{D}$/tconv.univ), ($\mathcal{D}$/econv.welcome), ($\mathcal{D}$/econv.col.univ), ($\mathcal{D}$/econv.cong.univ) et ($\mathcal{D}$/econv.cong.welcome).

Évaluation

$\mathsf{V}^\mathsf{c} ::=$	**valeur (sous B) dans c**
...	
$\mathsf{univ}\,\mathsf{V}^\bullet$	universel
$\mathsf{C} ::=$	**contexte d'évaluation**
...	
$\mathsf{univ}\,\_$	universel
$\mathsf{welcome}\,\_$	décapsulation d'un universel

$$\mathsf{welcome}\,(\mathsf{univ}\,\mathsf{V}) \longrightarrow_\mathsf{c} \mathsf{V} \qquad (\mathcal{D}/\mathsf{ered.welcome})$$

$$[\mathsf{univ}\,\mathsf{V}^\bullet]_{\mathsf{c}'}^{\textsc{Univ}\,\mathsf{T}} \longrightarrow_\mathsf{c} \mathsf{univ}\,\mathsf{V}^\bullet \qquad (\mathcal{D}/\mathsf{ered.col.univ})$$

Limitations Malheureusement les règles que nous venons d'énoncer ont un défaut qui limite nettement l'expressivité du langage obtenu : l'argument du constructeur univ doit être typable dans la couleur vide, donc ne peut pas contenir de variable libre non protégée par un crochet. Ainsi l'expression $\lambda x\!:\!\mathsf{T}.\ \mathsf{univ}\,x$ n'est pas typable. L'expression $\lambda x\!:\!\mathsf{T}.\ \mathsf{univ}\,[x]_{\{x\}}^{\mathsf{T}}$ est en revanche bien typée pourvu que T soit bien typé dans la couleur vide (donc en particulier clos).

Cette restriction concernant le typage du constructeur $\mathsf{univ}\,\_$ est similaire à celle subie par le constructeur $\mathsf{dynned}\,\_\,\mathsf{at}\,\mathsf{T}_0$. Ce dernier n'est pas destiné à être écrit directement par le programmeur : il provient de la réduction d'une construction dyn E at T par la règle ($\mathcal{D}$/ered.dyn). Ceci suggère d'ajouter au langage une construction supplémentaire protect E at T

$$\mathsf{protect}\,\mathsf{V}^\mathsf{c}\,\mathsf{at}\,\mathsf{T} \longrightarrow_\mathsf{c} \mathsf{univ}\,[\mathsf{V}^\mathsf{c}]_\mathsf{c}^{\mathbf{conc}_\mathsf{c}^\mathsf{B}(\mathsf{T})} \qquad (\mathcal{D}/\mathsf{ered.protect})$$

On note que la présence du type T est nécessaire, puisque celui-ci doit orner le crochet protégeant E dans le membre de droite. Comme la couleur ambiante c apparaît dans le membre de droite, la règle ($\mathcal{D}$/ered.protect) ne peut pas donner lieu à une règle de conversion, pas plus que la règle ($\mathcal{D}$/ered.dyn) (voir la section IV.6.1.4).

Conclusion Nous n'avons pas inclus les universels généraux dans le développement formel, nous contentant des dynamiques dont la formalisation est un peu moins lourde et qui sont suffisants par rapport à notre objectif.

IV.6.3.4 Partage d'empreintes

Nous avons discuté à la section IV.6.3.2 de l'échange de valeurs entre couleurs différentes. Or une couleur est définie par référence à un lexique : comparer deux couleurs, ou transmettre un terme contenant une couleur, n'a de sens qu'au sein d'un lexique bien défini. Si nous considérons plusieurs machines distinctes, chacune dispose *a priori* de son propre lexique.

Nous avons modélisé l'évaluation d'un programme (avec un unique fil d'exécution, sur une machine fixée) par une relation de réduction de la forme $\mathsf{B} \vdash \mathsf{E} \longrightarrow_{\bullet} \mathsf{B}' \vdash \mathsf{E}'$ (à l'extérieur de tout crochet, la couleur ambiante est vide). Une généralisation immédiate à plusieurs machines conduit à considérer une famille de réductions $\mathsf{B}_i \vdash \mathsf{E}_i \longrightarrow_{\bullet} \mathsf{B}'_i \vdash \mathsf{E}'_i$ où l'indice i représente la machine. Dans ce modèle, la communication doit prendre en compte non seulement le changement de couleur ambiante mais aussi le changement de lexique B_i en B_j.

Rappelons qu'un lexique est une collection d'apax (contenant également des informations supplémentaires concernant ces apax), et que chaque apax qui est ajouté au lexique vient d'être créé (par la règle ($\mathcal{C}$/ered.seal)) et est globalement unique. Deux lexiques B_1 et B_2 formés sur deux machines différentes ont donc des domaines disjoints ; on voit facilement que $(\mathsf{B}_1, \mathsf{B}_2)$ — ou plus généralement n'importe quel mélange de B_1 et B_2 respectant l'ordre au sein de B_1 et au sein de B_2 — constitue un lexique tout aussi valide et associant à chaque apax de B_1 ou de B_2 les mêmes informations que B_1 ou B_2. Rien ne s'oppose au niveau théorique à ce que nous fusionnions les lexiques ; l'évaluation d'un réseau de machines peut donc être modélisée par une relation de réduction de la forme

$$\mathsf{B} \vdash \mathsf{E}_1 \parallel \ldots \parallel \mathsf{E}_n \longrightarrow \mathsf{B}' \vdash \mathsf{E}'_1 \parallel \ldots \parallel \mathsf{E}'_n$$

(la notation $\mathsf{E}_1 \parallel \ldots \parallel \mathsf{E}_n$ désigne la mise en parallèle de n expressions, chacune tournant sur une machine).

Ce modèle facilite la théorie en rendant caduques les considérations de transport entre lexiques différents. En revanche, il ne correspond pas à une pratique optimale, puisqu'il nécessiterait la diffusion de chaque apax à l'ensemble du réseau lors de sa création (ce qui est très coûteux dans un réseau statique, et impossible dans des arrangements spatio-temporels de machines plus complexes). Il est cependant facile de fournir une implémentation raisonnable d'un tel modèle à lexique partagé, en considérant que chaque machine dispose d'une copie partielle du lexique, et que chaque envoi d'une valeur doit être accompagné des informations nécessaires à la reconstruction de la partie du lexique qu'elle utilise (les apax présents dans la valeur, ainsi que leurs dépendances, récursivement). Le lexique est ainsi diffusé paresseusement. À noter que bien que la création d'un apax nécessite la génération d'un nom globalement unique, cette opération n'impose pas de synchroniser toutes les machines, pourvu que chaque machine dispose d'un nom propre qu'elle peut inclure dans l'apax, ce qui est possible dans la plupart des systèmes distribués.

IV.6.3.5 Scellage statique et empreintes structurelles

Les apax sont des empreintes singularisées au sens de la section II.6.1.2, puisqu'un nouvel apax est généré à chaque création d'une famille de types abstraits par évaluation d'un scellage dynamique $\mathsf{E} \mathbin{!!} \mathsf{T}$ via la règle ($\mathcal{C}$/ered.seal). Nous avons présenté à la section IV.4.4.2 le système $\mathcal{W}$ muni d'une autre notion de scellage, le scellage statique $\mathsf{E} :: \mathsf{T}$, qui se distingue par le fait que la nouvelle famille de types abstraits (donc l'empreinte qui la désigne) est créée une fois pour toutes durant la compilation ou l'initialisation du programme.

Dans un environnement réparti, plusieurs choix sont possibles quant au moment de génération des empreintes correspondant aux scellages statiques. Deux généralisations immédiates du cas local sont d'effectuer la génération lors de la compilation ou lors de l'initialisation du programme.

La génération de nouvelles identités de types à la compilation est une manière traditionnelle de présenter les types abstraits [Mac84]. Elle n'est pas adaptée lorsque l'identité d'un type dépend de facteurs liés à l'exécution du programme, mais ce n'est jamais le cas pour notre scellage statique. Certaines théories des modules conçues pour un environnement réparti [Sew01] prévoient explicitement la possibilité de générer des types abstraits lors de la compilation du programme. Mais cette possibilité présente un inconvénient pratique rédhibitoire : il n'est plus possible de reconstruire le programme à partir de ses sources ! En effet, si l'on met en communication deux instances du même programme, celles-ci auront des types abstraits compatibles si le programme a été déployé en recopiant les exécutables, et incompatibles si le déploiement s'est fait par les sources. Si des données ont été archivées sur un disque, elles risquent d'être irrécupérables si, quelques années plus tard, le programme a été recompilé à partir de ses sources. Aussi ne proposons-nous à dessein pas de générer des identités de types à la compilation.

La génération de nouvelles identités lors de l'initialisation du programme permet moins de compatibilité que si elle était faite lors de la compilation. En revanche, le comportement est aisément prévisible et reproductible ; un excès de générativité sera rapidement repéré durant une phase de test du programme. Il s'agit donc d'une sémantique valable pour le scellage statique.

Aucune des sémantiques présentées jusqu'ici ne permettent le partage des types abstraits entre instances indépendantes du même programme (à plus forte raison entre instances indépendantes du même composant). Or les cas d'utilisation du scellage statique — avant tout les structures de données dont on veut garantir les invariants — correspondent aux cas d'utilisation des empreintes structurelles proposées à la section II.3. Il est donc naturel d'attribuer à chaque module scellé statiquement une empreinte structurelle. Par construction, cette empreinte peut être générée indépendamment sur chaque machine où le module est défini. De même que dans HAT (voir la section III.2.7.5), on peut voir cela comme une unification des différentes définitions du « même » module sur des machines différentes.

Nous ne formaliserons pas ici la construction des empreintes structurelles pour le système $\mathcal{W}$. Notons seulement que cette construction nécessite l'extrusion du module scellé statiquement hors de son contexte, comme décrit à la section IV.4.4.5. Remarquons également que dans le système $\mathcal{W}$, contrairement à HAT, l'identité d'un type est un terme de taille arbitraire, qui peut mentionner plusieurs empreintes : par exemple, si f est un foncteur scellé statiquement à la signature $\Pi x : T_0. \Sigma t : \text{TYPE}. T_1$ et que l'on applique f à un module obtenu par un scellage dynamique ayant produit l'apax a, le champ type du résultat a pour identité $\pi_1(h\, a)$ où h est l'empreinte structurelle créée par le scellage statique du foncteur.

IV.7 Conclusion

Bilan Nous avons présenté dans ce chapitre un langage destiné à décrire un système de modules pour un langage de la famille ML. Les principales fonctionnalités de ce langages sont les suivantes :

- des structures et des foncteurs, dont les types sont respectivement des sommes et des produits dépendants ;
- la possibilité de vérifier l'équivalence de deux modules, et de propager cette connaissance, grâce aux signatures singletons ;
- la définition de types abstraits par scellage d'un module, avec un système d'effets servant à déterminer quelles expressions sont comparables ;

– la conservation de l'abstraction au cours de l'exécution grâce aux crochets colorés ;
– une vérification de typage dynamique qui ne dépend pas du contexte du programme.

Sûreté Le minimum que l'on puisse exiger d'un système de types est qu'il soit sûr (*sound*) pour le mécanisme d'exécution proposé. L'annexe B est consacrée à la démonstration de la sûreté de TOPHAT, que nous énonçons classiquement via deux théorèmes : la préservation du typage par réduction (Th. B.3.6 (préservation du typage)), et le progrès des expressions bien typées (Th. B.3.7 (progrès)).

Décidabilité du typage Une autre propriété attendue d'un système de typage destiné à un langage de programmation est la décidabilité : nous souhaitons un algorithme qui décide si une expression a un type donné[32], ce qui en particulier nécessite de savoir quand deux types sont équivalents. Un algorithme de décision est connu pour des systèmes plus simples que le nôtre, par exemple celui proposé par D. Dreyer, C. Crary et R. Harper [DCH03]. La généralisation de leur algorithme à notre système n'a malheureusement rien d'immédiat, et nous laissons ici la question ouverte.

[32]L'**inférence de types** serait même souhaitable, mais il est connu que celle-ci est déjà impossible pour des systèmes beaucoup plus simples que le nôtre, par exemple le système F. Avec les annotations de types que nous exigeons, notamment sur les arguments de fonction, il est néanmoins probable que la reconstruction du type d'une expression ne soit pas substantiellement plus compliquée que sa vérification.

Chapitre V

Conclusion

V.1 Bilan

Les contributions de cette thèse s'articulent autour de trois axes :
- une notion de types abstraits pour les systèmes répartis ;
- une sémantique opérationnelle des types abstraits à la ML ;
- un système de modules plus expressif pour ML.

Types abstraits pour les systèmes répartis

La notion habituelle de type abstrait (§I.1) fait intervenir la création de nouveaux types durant la compilation ou l'exécution du programme. Dans le cas d'un programme réparti, constitué de plusieurs composantes démarrées (et potentiellement compilées, voire écrites) indépendamment, une généralisation naturelle conduit à considérer comme incompatibles les types abstraits définis par des composantes différentes. Or ceci est souvent trop restrictif, interdisant par exemple la communication de structures de données désignées par un type abstrait servant à assurer le maintien d'invariants.

La caractérisation exacte d'un type abstrait dépend de sa sémantique, qui n'est pas forcément indiquée sans ambiguïté dans le programme. Mais l'on peut quand même dégager des notions formelles de compatibilité. Le principe de base est de prendre l'*empreinte* d'un type abstrait (§II.3), qui est une abstraction du code qui définit le type abstrait. Nous proposons deux notions d'empreintes, l'une partageable (empreinte structurelle), l'autre non (empreinte générative) (§II.6.1). Ces notions d'empreintes s'étendent aux méthodes de production de familles de types abstraits, matérialisées en ML par les foncteurs (§II.5).

Une sémantique opérationnelle préservant l'abstraction

L'implémentation d'un type abstrait manipule des données dont le type est la représentation du type abstrait. À l'interface entre cette implémentation et le reste du programme, la conversion entre le type abstrait et sa représentation est autorisée. Nous matérialisons cette interface par des *crochets colorés* (§III.1), qui délimitent une expression dans laquelle un ou plusieurs types abstraits (indiqués par la couleur) sont équivalents à leur représentation.

Nous proposons deux langages utilisant les crochets colorés. Le premier, HAT (§III.2), consiste en l'ajout au lambda-calcul simplement typé d'une notion simple de types abstraits et de crochets colorés. Le second, TOPHAT (§IV), étend ces notions (§IV.5) à un système de modules à la ML (§I.2) très expressif. Dans les deux cas, les crochets colorés permettent de donner une sémantique opérationnelle aux types abstraits, le résultat classique de préservation du typage incluant en sus la préservation de l'abstraction.

Un système de modules plus expressif

Nous proposons un système de module qui généralise les systèmes couramment proposés pour ML. La principale innovation est l'utilisation d'une équivalence calculatoire entre modules de signature arbitraire, qui permet d'exprimer la compatibilité de deux fragments de code aussi bien que celle des représentations de types. Cette équivalence prend la forme de signatures singletons (§IV.3) dans un système admettant des types dépendants (§IV.2). Un système d'effets (§IV.4.2) restreint les expressions apparaissant dans les types à un fragment décidable du langage.

Outre son adaptation à la programmation répartie (§IV.6), le langage TOPHAT offre une plus grande expressivité grâce à une équivalence de types plus puissante (§IV.3.1). De plus, ce langage permet de contrôler finement le degré de générativité de chaque abstraction du programme : la cohabitation de *foncteurs applicatifs* et de *foncteurs génératifs* y est naturelle, et nous montrons comment différentes formes de *scellage* peuvent s'y exprimer (§IV.4.4).

V.2 Autres approches

V.2.1 Origines théoriques

Si les types abstraits ont fait couler beaucoup d'encre, ainsi que la communication de valeurs respectant un système de types structurels, la combinaison de l'abstraction et de la programmation répartie n'a que rarement été abordée jusqu'à présent.

L. Cardelli et D. MacQueen [CM88] suggèrent trois modèles pour le traitement de types abstraits annotant des valeurs persistantes :
- dans le modèle avec témoin transparent, la représentation du type abstrait est exposée : l'abstraction est ignorée lors de la communication ;
- dans le modèle avec témoin hypothétique, une valeur ayant un type abstrait n'est jamais utilisable : en dehors du contexte qui le définit, le type abstrait est considéré comme distinct de tout autre type, y compris lui-même ;
- dans le modèle avec témoin abstrait, l'identité d'un type abstrait est associée au composant logiciel qui le définit.

Seul ce dernier modèle propose une notion utile d'abstraction pour un système réparti ; malheureusement il n'est qu'esquissé.

A. Ohori, I. Tabkha, R. Connor et P. Philbrow [OTCP90] présentent un langage muni d'une sémantique qui suit le modèle avec témoin abstrait. Ce langage permet de sérialiser un module contenant la définition d'un type abstrait et du code manipulant des valeurs de ce type, et de le désérialiser en choisissant soit de créer un nouveau type soit de réutiliser une identité existante. Les témoins (identités de types) sont des objets de première classe. Le contrôle de la générativité est ainsi laissé à l'utilisateur du type abstrait, échappant nécessairement à son producteur.

V.2.2 Pratique

Plusieurs langages de programmation combinent une notion de type abstrait et un mécanisme de sérialisation. L'identité du type est sérialisée sous une forme qui dépend du langage.

Modula-3 [CDG+92] permet à l'identité (*brand*) d'un type abstrait d'être soit générée automatiquement de manière globalement unique lors de la compilation, soit choisie par le programmeur. Dans le premier cas, des types abstraits définis dans des instances compilées indépendamment d'un même programme sont incompatibles, ce qui bride la communication. Dans le second cas, le maintien de l'abstraction est de la seule responsabilité du programmeur.

La norme de Java inclut un mécanisme de sérialisation [Sun06]. À l'intérieur d'un programme, chaque classe définit de fait un type abstrait. La compatibilité entre la classe déclarée et la classe attendue lors de la sérialisation ne respecte pas l'abstraction : seuls sont vérifiés les types des champs et des méthodes non privées, ainsi que le nom de la classe (qui donne un contrôle indicatif au programmeur).

L'équivalence entre classes dans le cadre de .NET [MS01] est plus stricte : deux classes ne sont équivalentes que si elles ont le même nom et la même implémentation, cette dernière condition étant matérialisée par l'identité des bibliothèques (DLL) contenant les implémentations de la classe. Notre mécanisme d'empreinte peut être vu comme un raffinement de la vérification effectuée dans .NET : notre vérification d'égalité ne concerne que le code qui contribue effectivement au type abstrait considéré et non tout autre code qui se trouve dans la même unité de liaison dynamique ; de plus nous vérifions la compatibilité au niveau du code source, ce qui autorise l'utilisation de compilateurs différents (les compilateurs doivent être compatibles au niveau de la représentation des données, des types et des empreintes, mais conservent toute latitude quant à la génération du code). La vérification faite dans .NET est notamment impossible si le code est réparti entre différentes machines reposant sur des architectures matérielles ou des systèmes d'exploitation différents.

La sérialisation d'une valeur en Objective Caml [L$^+$] n'est accompagnée d'aucune information de typage, et la désérialisation n'est pas sûre. Plusieurs mécanismes ont été proposés pour rendre la désérialisation sûre. Une proposition faible, mais d'implémentation peu coûteuse, est de vérifier lors de la désérialisation que la représentation de la valeur est bien dans le type attendu [HMC06] ; cette méthode évite les erreurs arbitraires d'exécution mais ne respecte pas le typage même concret (un booléen peut par exemple être lu comme un entier). Le langage G'Caml [Fur02] ajoute la programmation générique, et en particulier le typage dynamique, à Objective Caml, ce qui permet de coder la bibliothèque de sérialisation dans un langage sûr. L'auteur a proposé[1] une bibliothèque de sérialisation sûre pour Objective Caml, reposant sur un ajout localisé au compilateur qui fournit deux primitives pour réifier un type et comparer deux types réifiés. Aucune des implémentations mentionnées ici ne s'attaque aux types abstraits.

V.2.3 Acute et HashCaml

Les travaux sur HAT présentés dans cette thèse s'inscrivent dans le cadre d'une étude plus large sur l'adaptation d'un langage basé sur ML à la programmation répartie. Outre le langage HAT [LPSW03a], dans lequel l'on s'intéresse aux types abstraits, citons le langage Proteus [SHB$^+$05], qui permet la mise à jour à chaud de composants d'un programme. Trois lignées sont issues (entres autres) de HAT : TOPHAT (qui est l'objet d'une part importante de la présente thèse), Acute et HashCaml.

Le langage Acute [SLW$^+$05] réunit de nombreuses fonctionnalités utiles à la programmation répartie. Il dispose notamment d'une construction de sérialisation sûre, qui respecte le typage, y compris l'abstraction entre des programmes compilés séparéments, grâce à un mécanisme d'empreintes dérivé de celui de HAT. Comme HAT, Acute utilise des crochets colorés pour garder trace des abstractions (c'est à notre connaissance la seule implémentation (publique) des crochets colorés). D'autres fonctionnalités saillantes sont la liaison dynamique et la possibilité de mise à jour à chaud de modules, une notion de compatibilité entre différentes versions d'un même programme, et la possibilité de migrer un fil d'exécution actif d'une machine à une autre. Le système de types

[1] Cette implémentation est malheureusement confinée à une branche morte du développement de JoCaml.

inclut des types abstraits et un polymorphisme explicite basé sur le système F. Le langage Acute dispose à la fois d'une implémentation et d'une sémantique formelle détaillée [SLW+04].

Le compilateur HashCaml [BSSS06] est une extension du compilateur Objective Caml supportant la sérialisation sûre respectant l'abstraction, grâce à un mécanisme d'empreintes. Une valeur sérialisée est accompagnée d'une représentation de son type ; si cette représentation n'est pas celle du type attendu lors de la désérialisation, une exception est levée. Les types sont réifiés sous la forme d'une somme de contrôle de taille fixe. Le polymorphisme à la ML est traduit en polymorphisme explicite avec passage aux fonctions polymorphes des types réifiés, ce qui permet d'écrire la fonction polymorphe `fun x -> Marshall.to_string x` (voir la section I.3.2.2). L'empreinte d'un module est structurelle par défaut, mais le programmeur peut demander lorsqu'il écrit une structure à ce qu'une empreinte singularisée soit générée (grâce au mot-clé `fresh`). L'empreinte de l'application d'un foncteur est une fonction de l'empreinte de l'argument, traduisant le caractère applicatif des foncteur d'Objective Caml.

V.2.4 Alice ML

Les travaux d'A. Rossberg constituent à notre connaissance le seul autre traitement approfondi du problème central de la présente thèse. Il est intéressant de constater que nos approches indépendantes nous ont souvent conduit à utiliser les mêmes outils.

Dans un premier temps, A. Rossberg [Ros03], comme nous [LPSW03a], propose l'utilisation de crochets colorés [ZGM99] pour garder la trace de types abstraits, permettant à la préservation du typage lors de l'exécution d'exprimer également la préservation de l'abstraction. Cependant, la théorie qu'il propose est purement générative : deux types abstraits définis sur des machines différentes sont forcément incompatibles.

A. Rossberg s'est également intéressé à la généralisation de ces travaux à un langage ML complet [Ros07], à savoir Alice ML [PSL]. Le $\lambda^{\omega}_{\mathrm{SA}\,\Psi}$-calcul, qui modélise le cœur d'Alice ML, comprend une construction qui définit un type abstrait (§10.5–10.7), dont l'auteur montre l'équivalence avec le scellage des modules à la ML (spécifiquement le scellage dynamique, voir notre section IV.4.4.2). Un type abstrait est désigné par une variable de type α dont la sorte est une sorte abstraction $\mathsf{A}(\tau)$ (§11.3), similaire à une sorte singleton $\mathsf{S}(\tau)$ mais ne permettant de conversion qu'explicite ; la différence avec nos apax répertoriés dans un lexique semble du seul domaine syntaxique. Le système de types du $\lambda^{\omega}_{\mathrm{SA}\,\Psi}$-calcul autorise les conversions explicites entre un type abstrait et sa représentation à n'importe quel endroit du programme ; nous matérialisons ces conversions par des crochets colorés.

A. Rossberg prouve une propriété d'opacité (§12.9) : un programme qui ne contient pas de conversion explicite entre un type abstrait et sa représentation est paramétrique en ladite représentation. Il propose également un mécanisme de scellage de foncteurs (§13), qui permet de coder des foncteurs tant applicatifs que génératifs. Étant donnée la complexité des deux systèmes, nous ne nous prononcerons pas ici quant à l'expressivité relative de TOPHAT et du $\lambda^{\omega}_{\mathrm{SA}\,\Psi}$-calcul avec foncteurs.

V.3 Perspectives

V.3.1 Raffinements théoriques

V.3.1.1 Stratification

Dans le langage TOPHAT, nous avons choisi d'unifier le langage des modules avec le langage de base (voir la section IV.2.2). Néanmoins certains des termes de notre langage appartiennent manifestement à l'une ou l'autre **strate** ; par exemple, 3 est un terme du noyau (strate 0) tandis que $\langle\textsc{int}\rangle$ relève de la strate des modules (strate 1). Le caractère compositionnel du langage permet de monter à des strates supérieures : TYPE est une signature, c'est-à-dire un type de strate 1, donc $\langle\textsc{type}\rangle$ est un terme de strate 2. La strate 2 correspond à un calcul sur les signatures ; il est rare, en pratique, de monter au-delà. En fait, TYPE est un type polystratique, puisqu'il peut être le type de n'importe quel expression $\langle\mathsf{T}\rangle$ où T peut appartenir à n'importe quelle strate.

Notre système de types est dit **imprédicatif** : il autorise le terme $\langle\textsc{type}\rangle$ à avoir le type TYPE. D'un point de vue logique, cette propriété est néfaste parce qu'elle fait apparaître le paradoxe de Burali-Forti [Coq86], qui rend le système de types logiquement incohérent. En particulier, la bêta-conversion devient non normalisante [HM93].

Certains dialectes de ML, notamment Alice ML et Objective Caml, autorisent les signatures abstraites. Cet ajout fait boucler les typeurs concernés sur certains programmes pathologiques, comme le suivant, tiré du manuel d'Alice ML [PSL] (mais donné ici dans la syntaxe d'Objective Caml) :

```
module type I = sig
  module type A
  module F(X : sig module type A = A module F(X:A) : sig end end) : sig end
end;;
module type J = sig
  module type A = I
  module F(X:I) : sig end
end;;
module Loop(X : J) = (X : I);;
```

La solution classique pour éviter ces problèmes tant logiques que pratiques est de stratifier le langage en **univers**. Le type TYPE devient une famille de types $\textsc{type}_\mathsf{s}$ où s est une strate (ou univers), l'ensemble des strates étant muni d'une relation d'ordre bien fondée (en général les strates sont des entiers naturels). Par exemple, un champ type « ordinaire » tel que $\langle\textsc{int}\rangle$ a le type $\textsc{type}_\mathsf{s}$; lorsque TYPE caractérise non pas un type mais une signature, il porte une strate plus élevée : en particulier, $\textsc{type}_0$ a le type $\textsc{type}_1$.

La qualification de la strate de TYPE est orthogonale à la caractérisation du niveau de précision ($\wr$ ou $*$). Aussi suggérons-nous d'étendre les sortes pour inclure également une strate. Notons Pr un niveau de précision ; une sorte a alors la forme ${}^{\mathsf{Pr}}\mathsf{s}$. Les sortes sont munies de l'ordre produit (${}^{\mathsf{Pr}_1}\mathsf{s}_1 \leqslant {}^{\mathsf{Pr}_2}\mathsf{s}_2$ si et seulement si $\mathsf{Pr}_1 \leqslant \mathsf{Pr}_2$ et $\mathsf{s}_1 \leqslant \mathsf{s}_2$).

Les types de base autres que $\textsc{type}_\mathsf{s}^\mathsf{Pr}$ (UNIT, BOOL, INT, DYN) ont la sorte minimale ${}^{\wr}0$. Le type $\textsc{type}_\mathsf{s}^\mathsf{Pr}$ est non spécifié (précision $*$) et monte dans l'échelle des strates : sa sorte ne peut être que ${}^{*}\mathsf{s}'$ où s' est strictement supérieure à s. Du coup, $\langle\textsc{type}_\mathsf{s}^\mathsf{Pr}\rangle$ n'a pas le type $\textsc{type}_\mathsf{s}^\mathsf{Pr}$, seulement $\textsc{type}_{\mathsf{s}'}^\mathsf{Pr}$ avec $\mathsf{s} < \mathsf{s}'$: il faut monter d'un cran dans l'échelle des univers.

Notons que l'ensemble des strates pourrait ne pas être les entiers naturels. Par exemple, on peut choisir d'interdire les signatures abstraites en se limitant aux deux strates 0 et 1 — mais cela empêcherait également les signatures calculées.

V.3.1.2 Niveaux de langue

Supposant l'ajout d'une stratification des types, se pose la question de distinguer ou non le niveau du noyau de celui ou ceux des modules (la strate 0 par rapport aux autres). Conserver une syntaxe et une sémantique unifiées est un avantage tant théorique que pratique : il évite d'avoir à répéter essentiellement à l'identique les règles définissant le langage, ainsi que les programmes de calcul pur qui ont un sens quelle que soit la strate.

D'un autre côté, si les différents niveaux partagent un potentiel calculatoire, il n'est pas clair que leur typage doive être le même. En effet, une propriété très importante des langages à la ML est l'inférence de types ; or celle-ci est contradictoire avec la présence de types dépendants. Il existe des propositions de types dépendants pour ML [XP99, XCC03], mais ceux-ci ne sont utilisés qu'en présence d'annotations donnant suffisamment d'indications de typage, formant une extension conservative de ML. Au niveau des modules, les types dépendants sont bien plus fréquents — ils apparaissent dès que l'on utilise des foncteurs — et les annotations de types sont souvent présentes de toutes façons pour permettre la compilation séparée. Il apparaît donc légitime de se baser sur le polymorphisme à la ML dans le langage du noyau et d'abandonner l'inférence au niveau des modules.

Nous ne nous prononcerons pas sur l'opportunité de distinguer au niveau syntaxique le noyau des modules[2].

V.3.1.3 Analyse d'effets

Nous nous sommes limités dans le système TOPHAT à un système d'effets réduit à sa plus simple expression : il distingue seulement les expressions pures des expressions impures. Nous avons évoqué l'ajout au système d'effets statiques pour modéliser le scellage statique (section IV.4.4.2). Plus classiquement, nous pourrions distinguer de nombreux effets dynamiques : effets du scellage, effets externes (entrées-sorties), effets liés aux modifications de la mémoire...

Une fois cette distinction effectuée, de nombreux raffinements sont envisageables. Par exemple, si un module scellé est défini localement et que les types abstraits n'échappent pas à la portée du module, par exemple dans le fragment $\mathsf{let}\,\mathsf{x} = \mathsf{E}\ !!\ \mathsf{T}\,\mathsf{in}\ \ldots : \textsc{int}$, l'expression pourrait être considérée comme pure. Une telle expression est en effet comparable à un fragment de programme qui alloue une zone de mémoire dans laquelle il travaille impérativement et qui libère cette mémoire avant de renvoyer un résultat[3] Notons toutefois que la construction d'une valeur dynamiquement typée pourrait laisser échapper le type.

Dans une autre direction, nous pouvons enrichir notre système d'effets par du polymorphisme, éventuellement passant par des effets dépendants. Considérons par exemple la fonction `twice = λf.(λx.f(fx))`, dont le (schéma de) type en ML est $(\alpha \to \alpha) \to (\alpha \to \alpha)$. Nous pouvons encoder son polymorphisme en abstrayant sur la variable de type, ce qui nous donne le type $\Pi\mathsf{t} : \textsc{type}.(\mathsf{Typ}\,\mathsf{t} \to^{\gamma} \mathsf{Typ}\,\mathsf{t}) \to^{\mathsf{P}} (\mathsf{Typ}\,\mathsf{t} \to^{\gamma} \mathsf{Typ}\,\mathsf{t})$ dans lequel γ peut être n'importe quel effet ; ces types sont deux à deux incomparables. Il serait logique d'encoder le polymorphisme d'effets désiré par un système d'effets dépendants, soit en introduisant des variables d'effets et une abstraction pour ces variables [LG88, HMN05], soit comme pour les types en introduisant une injection des effets dans les expressions et la projection réciproque.

V.3.1.4 Couleurs et crochets

Nous avons décrit dans cette thèse un unique langage muni de crochets colorés. Nous avons tout au long de la section IV.5 évoqué diverses variations possibles, qui mériteraient d'être étudiées.

Nous avons tenu compte des révélations de connaissance dues aux arguments de fonctions en considérant des couleurs symboliques, pouvant contenir des variables auxquelles seront substituées des couleurs concrètes lors de la bêta-réduction. J.-J. Lévy remarque que dans le lambda-calcul, les liaisons de variables sont successives, ce qui suggère que toute couleur contenant une variable est équivalente au singleton constitué de la variable la plus interne. Une couleur serait donc soit une

[2]Les MLeurs ont l'habitude mais les lispiens n'apprécient pas.

[3]En Haskell, il s'agirait d'une utilisation locale de la monade d'état.

variable x, soit une couleur constante $\{a_1, \ldots, a_k\}$. Cette simplification nous limiterait aux crochets additifs.

Nous avons évoqué à la section IV.5.2.4 la possibilité d'imposer différentes restrictions sur les couleurs annotant les crochets (crochets additifs, crochets absolus)... Des crochets non additifs seraient notamment nécessaires pour modéliser des boîtes noires dont le système de type (guidé par les crochets) vérifie l'intégrité.

Lors de l'application d'une fonction à son argument, la couleur du corps de la fonction et celle de l'argument sont fusionnées, ce qui limite la précision offerte par le typage. Un traitement plus précis, notamment, des fonctions polymorphes, nécessiterait l'utilisation de paramètres types polymorphes $[\langle \mathsf{T} \rangle]_{\mathsf{c}}^{\mathrm{TYPE}}$ donnant lieu à des valeurs polymorphes $[\mathsf{V}]_{\mathsf{c}}^{\mathsf{Typ}\,[\langle \mathsf{T} \rangle]_{\mathsf{c}}^{\mathrm{TYPE}}}$ (voir la section IV.5.3.4). Un théorème de paramétricité pourrait aider à contrôler le flux de ces valeurs polymorphes.

V.3.1.5 Décidabilité du typage

Nous avons laissé en suspens la question de la décidabilité du typage de TOPHAT (voir la section IV.7). Il est probable que celle-ci exige de stratifier le système tel que décrit à la section V.3.1.1 [HM93].

Le principal contenu calculatoire du fragment pur de TOPHAT est la bêta-réduction (règle (econv.app), en particulier) et l'êta-expansion (règles (econv.eta.*)). Ces règles sont contraintes par un système de types qui est essentiellement une restriction du calcul des constructions inductives[4] [Coq], dans lequel l'équivalence est décidable. H. Goguen a proposé [Gog05] un algorithme de vérification du typage pour un lambda-calcul muni de types sommes et produits dépendants et de singletons. Nous conjecturons que cet algorithme peut être adapté à notre système.

V.3.1.6 Paramétricité

Le maintien de l'abstraction dans la sémantique opérationnelle confère au théorème de préservation du typage une valeur de préservation de l'abstraction. Une modélisation plus traditionnelle de la préservation de l'abstraction est la **paramétricité** (voir la section II.6.2.2) : si l'implémentation d'une abstraction change sans que sa sémantique observable extérieurement change, la sémantique du programme ne doit pas être affectée. Il serait intéressant de relier ces deux approches.

V.3.2 Fonctionnalités supplémentaires

V.3.2.1 Nommage des champs et sous-typage en largeur

Un manque, au premier abord syntaxique, de TOPHAT par rapport aux systèmes de modules habituels est l'absence de champs nommés dans les structures. Nous ne disposons en effet que de paires (E_1, E_2). Nous pouvons facilement donner un encodage théorique des champs nommés, en choisissant une énumération des noms de champs et en codant une structure par la liste de ses champs $(E_0, (E_1, (E_2, \ldots)))$ dans laquelle E_i est le champ dont le nom porte le numéro i (ou $E_i = ()$ si la structure ne contient pas de champ ayant ce nom). Un tel encodage n'est pas raisonnable pour une implémentation, mais il permet d'étendre la théorie, ce qui assure de la correction d'une implémentation plus classique.

Un premier défaut de cet encodage est que l'ordre des champs est forcé par leur nom ; or si nos paires sont symétriques, leurs types ne le sont pas : nous n'avons pas en général le choix de l'ordre des dépendances. Une autre manière de gérer les noms de champs est de précéder le typage par une phase d'élaboration qui associe à chaque structure une numérotation des champs. Ceci correspond à

[4] Dans cette perspective, nous ne disposons comme seule définition inductive que de l'égalité, écrite via les singletons.

la manière utilisée pour représenter une structure en mémoire dans de nombreuses implémentations, par exemple Objective Caml. L'élaboration doit dans certains cas introduire une permutation de l'ordre des champs ; par exemple les structures `struct type t=int type u=bool let x=3 end` et `struct type u=bool type t=int let x=3 end` ont toutes deux la signature `sig type t type u val x:t end`, soit après élaboration $\Sigma \mathsf{t} : \textsc{type}.\ \Sigma \mathsf{u} : \textsc{type}.\ \mathsf{Typ}\,\mathsf{t}$.

Dans tous les cas, la relation de sous-signaturage de TOPHAT permet, outre éventuellement le réordonnancement des champs, de remplacer le type d'un champ par un type plus abstrait : il s'agit de **sous-typage en profondeur**. Elle ne permet en revanche pas de supprimer un champ : nous ne disposons pas de **sous-typage en largeur**. Nous ne prévoyons pas de difficulté majeure à ajouter ce dernier à TOPHAT, dans le style habituel de $\mathsf{F}_{<:}$ avec enregistrements. En revanche les interactions du sous-typage en largeur avec les empreintes structurelles semblent problématiques, comme nous l'avons vu à la section II.6.3.

V.3.2.2 Vers un langage de programmation

Le système TOPHAT prétend modéliser le système de modules d'un langage de la famille ML, voire également le noyau du langage. Outre le nommage des champs des modules, abordé ci-dessus (section V.3.2.1), un certain nombre de fonctionnalités manquent à TOPHAT pour en faire un « vrai » langage de programmation ; nous allons brièvement étudier comment les ajouter.

La présence de types dépendants confère à notre langage un polymorphisme explicite. Le polymorphisme implicite habituel de ML est essentiellement équivalent au polymorphisme explicite [HM93] ; une traduction devrait être possible. Si l'on choisit de cantonner TOPHAT aux modules, et d'utiliser un autre modèle de langage pour le noyau, ce dernier peut être équipé directement de polymorphisme à la ML.

Nous pouvons sans affecter les propriétés du système de typage ajouter des constantes dont le type a la forme $\mathsf{T}_0 \rightarrow^{\mathtt{I}} \mathsf{T}_1$ (ou même $\Pi \mathsf{x} : \mathsf{T}_0.{}^{\mathtt{I}}\mathsf{T}_1$), c'est-à-dire des fonctions dont l'application engendre un effet de bord. La sémantique opérationnelle d'une telle constante f doit prendre la forme de delta-règles de la forme $\mathsf{f}\,(\mathsf{V}) \longrightarrow_{\mathsf{c}} \mathsf{E}$ où V est une valeur quelconque de type T_0. Notre langage est ainsi à même d'exprimer un combinateur de point fixe, des entrées-sorties, des mutations de structures de données, et toute autre fonctionnalité de bibliothèque. Lorsque la fonction désirée est polymorphe, nous pouvons la considérer comme une famille de fonctions et introduire une unique constante ayant un type idoine, par exemple $\Pi \mathsf{t} : \textsc{type}.\ {}^{\mathtt{P}}(\mathsf{Typ}\,\mathsf{t} \rightarrow^{\mathtt{I}} \mathsf{Typ}\,\mathsf{t}) \rightarrow^{\mathtt{I}} \mathsf{Typ}\,\mathsf{t}$ pour un combinateur de point fixe.

L'ajout de constantes permettant de former de nouvelles expressions pures doit être effectué avec précaution, puisque notre système de typage permet d'exprimer l'égalité de deux expressions pures via un jugement tel que $\mathsf{E}_1 :^{\mathtt{P}} \mathsf{S}(\mathsf{E}_2)$. Ainsi, la simple attribution à la fonction d'addition du type $\textsc{int} \rightarrow^{\mathtt{P}} \textsc{int} \rightarrow^{\mathtt{P}} \textsc{int}$, utile pour vérifier statiquement que les accès aux tableaux n'en dépassent pas les bornes [XP99], nécessite d'équiper un algorithme de typage d'une procédure de décision pour l'arithmétique de Presburger.

L'ajout de types de base et de constantes couvrant les valeurs de ces types peut se faire sur le modèle de BOOL et INT. L'ajout de constructions sur les types telles que des types sommes ou des types récursifs est également possible, à condition d'équiper le système de règles permettant de décider de l'équivalence de deux types et d'attribuer un typage suffisamment fin aux constructeurs et aux destructeurs. Sur ce dernier point, notons que la présence de types dépendants peut rendre complexe le typage des destructeurs lorsque le système de types doit encapsuler l'égalité des expressions — en présence de types récursifs, le destructeur habituel est un combinateur de points fixe que nous ne pouvons admettre comme fonction pure que s'il est accompagné de restrictions suffisantes pour le rendre normalisant (il s'agirait essentiellement d'incorporer le calcul des constructions inductives [Coq]). Notons toutefois que la présence de types mutuellement récursifs (ou pire, de modules récursifs [Dre05]) interagit subtilement avec la notion de sous-signature en profondeur [CHC⁺98].

V.3.2.3 Programmation générique

Dans le langage TOPHAT, la construction $\mathsf{undyn}\ \mathsf{E}\ \mathsf{at}\ \mathsf{T}\ \mathsf{else}\ \mathsf{E}'$ effectue une vérification dynamique de typage. Dans certaines utilisations, l'expression E est censée avoir le type T et le cas contraire signifie un non-respect du protocole ; la construction proposée est alors parfaitement satisfaisante. Il est également possible d'enchaîner plusieurs tentatives de dédynamisation, par exemple pour des versions successives du protocole.

Toutefois cette construction ne permet pas un accès optimal à l'information. Dès lors que le vrai type du contenu du dynamique est accessible pour vérification, il apparaît désirable de permettre son analyse : l'on peut alors utiliser la programmation générique (voir la section I.3.2.4). Si l'expression E est considérée comme un module, il s'agirait d'analyser sa signature. Une construction adaptée permettrait par exemple à un protocole de communication d'offrir des options dont l'utilisation pourrait être détectée orthogonalement.

L'analyse du type du contenu du dynamique peut être rendue impossible par certaines formes de codage de ce type, par exemple si celui-ci est réduit à une somme de contrôle, soit pour en assurer la confidentialité, soit à fin de compression. Notons toutefois qu'une telle restriction n'est compatible avec notre système en l'état que pour un type qui n'a pas de sur-type, puisque la destruction d'un dynamique est autorisée dès lors que le type attendu est un sur-type du type effectif. La dynamisation peut bien sûr lister les sommes de contrôle de tous les sur-types dès lors que ceux-ci sont en nombre fini.

V.3.2.4 Sécurité

Le scellage créant un type abstrait donne à celui-ci un caractère confidentiel et authentifié [Mor73a]. Notre sémantique opérationnelle garde une trace du scellage sous la forme d'un crochet coloré $[\mathsf{V}]^{\mathsf{T}}_{\mathsf{c}}$. Nous avons décrit ces crochets comme représentant les frontières entre des domaines de connaissance, spécifiquement de connaissance d'égalités entre types. Une vision plus large est que les domaines sont ceux des participants d'une communication [ZGM99], et la connaissance peut être celle de clés cryptographiques. Nos apax apparaissent naturellement comme les clés en question ; encadrer une valeur par un crochet coloré signifie alors que l'on crypte cette valeur avec la ou les clés contenues dans la couleur. Cette interprétation des crochets colorés a été étudiée formellement par B. Pierce et E. Sumii [PS00].

V.3.3 Implémentation

V.3.3.1 Calcul d'empreintes

Nous avons discuté des aspects implémentatoires des empreintes structurelles à la section II.3.3. Nous renvoyons le lecteur à cette discussion, ainsi qu'à l'article idoine [LPSW03a] (§3) au sujet de l'implémentation de HAT.

Les langages Acute [SLW⁺04] et HashCaml [BSSS06] implémentent une vérification dynamique de typage qui respecte les types abstraits, pour des langages d'expressivité croissante en ce qui nous concerne. Les types abstraits d'Acute sont générés globalement dans chaque unité de compilation, tandis que HashCaml dispose de foncteurs.

V.3.3.2 Typage de TOPHAT

Nous avons évoqué à la section V.3.1.5 la question de la décidabilité du typage de TOPHAT. À supposer que la réponse soit positive, restera le problème de la rendre pratique — obtenir des messages d'erreur clairs pour un programme mal typé est notoirement difficile. Une possibilité est de simplifier le système de types pour le rendre plus facile à manipuler tant pour l'auteur du compilateur que le programmeur ; la relative orthogonalité des fonctionnalités centrales (foncteurs, types dépendants, singletons, effets) n'offre pas de partie clairement excisable.

V.3.3.3 Intégration à Objective Caml : le système de modules

Si nous ajoutons à TOPHAT des champs nommés dans les structures et un sous-typage en largeur (voir la section V.3.2.1), nous obtenons un langage qui dispose de toutes les fonctionnalités du système de modules d'Objective Caml [L^+]. Mais est-il compatible, autrement dit, s'agit-il d'une extension conservative d'Objective Caml ?

La réponse est « non, mais ». En effet, Objective Caml autorise certains programmes que nous rejetons, puisque tous les foncteurs y sont applicatifs, y compris si leur corps comporte des effets de bord. Cela est inacceptable dans TOPHAT puisque nous devons pouvoir évaluer statiquement l'application d'un foncteur applicatif. Une solution pourrait être d'introduire une notion de séparation (au sens de la séparabilité [Dre05] mentionnée à la section IV.4.2.3). Toutefois, rendre applicatif un foncteur dont l'évaluation provoque des effets de bord peut apparaître de mauvais goût — même si le typage structurel, donc la sûreté du langage, est sauf, il s'agit d'une rupture de l'abstraction. *A contrario*, considérer comme génératif tout foncteur dont le corps contient un effet de bord nous apparaît légitime ; dans la plupart des cas où l'applicativité est désirée, le corps est pur, ([Dre05, RRS]). On notera par exemple que tous les foncteurs de la bibliothèque standard d'Objective Caml ont un corps pur (constitué essentiellement de définitions de types et de fonctions, ainsi que quelques constructions de structures de données qui sont syntaxiquement des valeurs).

Le scellage existant d'Objective Caml doit être considéré comme un scellage statique (voir la section IV.4.4.2). Il serait bien sûr judicieux de prévoir une forme supplémentaire de scellage dynamique. Une syntaxe pour indiquer qu'un foncteur est génératif est nécessaire afin que toutes les signatures soient expressibles.

Outre le langage des modules, nous devons considérer le langage de base. Nous avons évoqué le polymorphisme à la section V.3.2.2. Il reste à étendre notre analyse d'effets à Objective Caml. Nous pouvons étendre sans difficulté TOPHAT avec des constructions impures ; un choix sûr est de rendre presque toutes les expressions impures. Les principales exigences raisonnables sont qu'une fonction immédiate, et la projection d'un champ d'un module, doivent être considérés comme purs. De fait, Objective Caml, comme tout dialecte de Standard ML, contient déjà une analyse de pureté satisfaisante, à savoir la vérification de la restriction aux valeurs [Wri95, Gar04] pour le polymorphisme de la liaison locale. Un système d'effets suffisant pour nos besoins s'intègre ainsi facilement à Objective Caml.

V.3.4 Applications du typage dynamique

Nous avons argué à la section I.3.1 de l'opportunité de fournir un moyen de vérification dynamique de typage, rompant la séparation des phases, dans un langage typé statiquement. Un cas notable où la séparation des phases est intrinsèquement inapplicable est celui où plusieurs programmes démarrés séparément communiquent. Cette circonstance couvre de nombreux systèmes que l'on peut qualifier de répartis : répartis dans l'espace, lorsque l'on fait communiquer des programmes tournant sur des machines différentes ; répartis dans le temps, lorsque des données sont écrites sur un moyen de stockage persistant et relues ultérieurement. L'application essentielle du typage dynamique est ainsi la sérialisation, qui agit naturellement sur des valeurs portant une annotation de type, qui doit être vérifiée avant utilisation.

V.3.4.1 JoCaml : Serveur de noms

Une application qui a participé à motiver la présente thèse est le « serveur de noms (*name server*) » de JoCaml [FLFS07, MM01]. Le langage JoCaml est muni d'un système de types statique qui rend les communications sûres [FLMR97]. Mais ce résultat ne s'applique qu'au sein d'une instance donnée d'un programme : lorsque deux programmes indépendants cherchent à communiquer, le fait que la valeur envoyée par le programme émetteur ait bien le type attendu par le programme récepteur ne peut découler que de l'adhésion à un protocole extérieur à tout système de vérification qui se limite à une seule instance, ce qu'est nécessairement un typage effectué lors de la compilation.

Le modèle recommandé de programmation en JoCaml limite au minimum l'interaction non sûre : l'un des programmes envoie à l'autre un canal de communication d'un type convenu à l'avance et généralement d'ordre supérieur, à la suite de quoi les programmes peuvent s'échanger d'autres canaux et toutes données de façon sûre. La bibliothèque standard de JoCaml (par son module `Ns`) facilite l'utilisation d'un serveur de noms, c'est-à-dire un programme qui est dépositaire de canaux de communications (les fameux noms) fournis par les différentes instances en présence. Un nouveau participant peut interroger le serveur de noms et récupérer des canaux typés qui lui permettront de communiquer avec les autres intervenants en bénéficiant du typage statique : la seule vérification dynamique de typage est effectuée lors de la réponse à la requête du nouveau participant (soit par le serveur de noms lui-même si la requête est accompagnée de l'indication du type attendu, soit lors de la réception de la réponse ; dans tous les cas la valeur stockée sur le serveur de noms doit être accompagnée d'une information de typage).

Annexe A

Tables récapitulatives de TOPHAT

Cette annexe constitue un précis du language TOPHAT, c'est-à-dire le système $\mathcal{D}$ du chapitre IV.

$\mathsf{E} ::=$	**expression**
$x \mid y \mid t \mid \ldots$	variables
$()$	valeur unité
$\mathsf{false} \mid \mathsf{true}$	booléen (génériquement bv)
$0 \mid 1 \mid \ldots$	entier (génériquement $\underline{n}$)
$\langle \mathsf{T} \rangle$	champ type
$(\mathsf{E}_1, \mathsf{E}_2)$	paire
$\pi_i\, \mathsf{E}$	projection ($i \in \{1, 2\}$)
$\lambda x : \mathsf{T}.\ \mathsf{E}$	lambda-abstraction
$\mathsf{E}_1\, \mathsf{E}_2$	application
$\mathsf{let}\ x = \mathsf{E}_0\ \mathsf{in}\ \mathsf{E} : \mathsf{T}$	liaison locale
$\mathsf{E}\ !!_c\ \mathsf{T}$	module scellé et coloré
$[\mathsf{E}]_c^{\mathsf{T}}$	crochet coloré
$\mathsf{dyn}\ \mathsf{E}\ \mathsf{at}\ \mathsf{T}$	dynamique
$\mathsf{dynned}\ \mathsf{E}\ \mathsf{at}\ \mathsf{T}$	dynamique universel
$\mathsf{undyn}\ \mathsf{E}\ \mathsf{at}\ \mathsf{T}\ \mathsf{else}\ \mathsf{E}'$	vérification de typage dynamique
$\mathsf{T} ::=$	**type**
UNIT	unité
BOOL	booléens
INT	entiers
$\mathsf{Typ}\ \mathsf{E}$	projection d'un champ type
$\Sigma x : \mathsf{T}_1.\ \mathsf{T}_2$	somme dépendante (aussi notée $\mathsf{T}_1 * \mathsf{T}_2$ lorsque $x \notin \mathbf{fv}\, \mathsf{T}_2$)
$\Pi x : \mathsf{T}_0.\ {}^{\gamma}\mathsf{T}_1$	produit dépendant (aussi noté $\mathsf{T}_1 \to^{\gamma} \mathsf{T}_2$ lorsque $x \notin \mathbf{fv}\, \mathsf{T}_1$)
$\mathsf{S}(\mathsf{E})$	singleton
$\mathrm{TYPE}^{\mathsf{K}}$	champ type abstrait
$(\!\mid \mathsf{A} \mid\!)$	type abstrait
DYN	valeur dynamiquement typée
$\mathsf{K} ::=$	**sorte**
$\wr$	monomorphe (complètement spécifié)
$*$	polymorphe (partiellement spécifié)

$A ::=$	**composante**
$\quad a$	apax
$\quad A\ E$	application
$\quad \pi_i\, A$	projection ($i \in \{1, 2\}$)
$\gamma ::=$	**effet**
$\quad \mathsf{P}$	pur
$\quad \mathsf{I}$	impur
$\xi ::=$	**couleur primaire**
$\quad a$	apax
$\quad x$	variable
$c ::=$	**couleur**
$\quad \bullet$	couleur vide (aussi notée $\{\}$)
$\quad \{a_1, \ldots, a_k,\ x_1, \ldots, x_k\}$	ensemble fini de couleurs élémentaires
$B ::=$	**lexique**
$\quad \mathsf{nil}$	vide
$\quad B, a = E :_{c_0} T$	apax a d'implémentation E et de signature T
$\Gamma ::=$	**environnement**
$\quad \mathsf{nil}$	vide
$\quad \Gamma, x :_c T$	liaison d'une variable x
$J ::=$	**membre de droite de jugement local**
$\quad \mathsf{ok}$	correction de l'environnement
$\quad T : K$	sortage d'un type (généralise $T\,\mathsf{ok}$)
$\quad T \longrightarrow T'$	conversion des types
$\quad T \equiv T'$	équivalence par conversion des types
$\quad E \longrightarrow E'$	conversion des expressions
$\quad E \equiv E'$	équivalence par conversion des expressions
$\quad T_1 <: T_2$	sous-typage
$\quad c_0\ \mathsf{transparent}$	transparence d'une couleur
$\quad A \triangleright E : T$	révélation d'une composante
$\quad A \longrightarrow A'$	conversion d'une composante
$\quad A \equiv A'$	équivalence par conversion des composantes
$\quad E :^{\gamma} T$	typage d'une expression
$V ::=$	**quasi-valeur**
$\quad () \mid bv \mid \underline{n}$	constante
$\quad \langle T \rangle$	champ type
$\quad (V_1, V_2)$	paire
$\quad \lambda x : T.\ E$	lambda-abstraction
$\quad [V]_{c'}^{(\!\mid A^V \mid\!)}$	crochet coloré potentiellement abstrayant
$\quad \mathsf{dynned}\ V^{\bullet}\ \mathsf{at}\ T$	dynamique universel

$V^c ::=$	**valeur dans c**
$() \mid bv \mid \underline{n}$	constante
$\langle T \rangle$	champ type
(V^c_1, V^c_2)	paire
$\lambda x : T.\ E$	lambda-abstraction
$[V^{c'}]_{c'}^{(\!\|A^{V^{c \cap c'}}\|\!)}$	crochet coloré, si $A^{V^{c \cap c'}}$ est abstraite dans c mais concrète dans c'
$\mathsf{dynned}\ V^\bullet\ \mathsf{at}\ T$	dynamique universel

$A^V ::=$	**composante valeur**
a	apax
$A^V\ V$	application à une quasi-valeur
$\pi_i\, A^V$	projection $(i \in \{1, 2\})$

$A^{V^c} ::=$	**composante abstraite dans c**
a	apax opaque dans c
$A^{V^c}\ V^c$	application d'un foncteur à une valeur
$\pi_i\, A^{V^c}$	projection $(i \in \{1, 2\})$

$C^c_{c'} ::=$	**contexte d'évaluation (de couleurs intérieure c' et extérieure c)**
$E_1\ \_\_$	argument de fonction
$\_\_\ V_2$	fonction appliquée
$(\_\_, E_2)$	première composante d'une paire
$(V_1, \_\_)$	seconde composante d'une paire
$\pi_i\ \_\_$	projection $(i \in \{1, 2\})$
$\mathsf{let}\ x = \_\_\ \mathsf{in}\ E : T$	lié local
$\_\_\ !!_{c_1}\ T$	scellage
$[\_\_]^T_{c'}$	crochet coloré
$[V^{c_1}]^{\mathsf{Typ}\,\_\_}_{c_1}$	champ type sur un crochet, lorsque $c' = c \cap c_1$
$\mathsf{dyn}\ \_\_\ \mathsf{at}\ T$	dynamique
$\mathsf{dynned}\ \_\_\ \mathsf{at}\ T$	dynamique universel, lorsque $c' = \bullet$
$\mathsf{undyn}\ \_\_\ \mathsf{at}\ T\ \mathsf{else}\ E'$	vérification de typage dynamique

$$
\begin{array}{rcll}
\mathbf{self}^{\,BT}(A) & = & BT & \text{si BT est un type de base (UNIT, BOOL, INT, DYN)} \\
\mathbf{self}^{\,\Sigma x:T_1.\,T_2}(A) & = & \Sigma x : \mathbf{self}^{\,T_1}(\pi_1 A).\ \mathbf{self}^{\,T_2}(\pi_2 A) & \\
\mathbf{self}^{\,\Pi x:T_0.\,{}^{P}T_1}(A) & = & \Pi x : T_0.\ {}^{P}(\mathbf{self}^{\,T_1}(A\,x)) & \\
\mathbf{self}^{\,\Pi x:T_0.\,{}^{I}T_1}(A) & = & \Pi x : T_0.\ {}^{I}T_1 & \\
\mathbf{self}^{\,S(E')}(A) & = & S(E) & \\
\mathbf{self}^{\,\text{TYPE}^K}(A) & = & S(\langle (\!|A|\!) \rangle) & \\
\mathbf{conc}^B_c((\!|A_1|\!)) & = & \mathsf{Typ}\,\mathbf{reveal}^B(A_1) & \text{si } \mathbf{underl}(A_1) \in c \\
\mathbf{conc}^B_c((\!|A_1|\!)) & = & (\!|A_1|\!) & \text{si } \mathbf{underl}(A_1) \notin c \\
\mathbf{conc}^B_c([E]^T_{c'}) & = & [E]^{\mathbf{conc}^B_{c \cap c'}(T)}_{c'} &
\end{array}
$$

(simple induction dans les autres cas)

$$\begin{array}{rcll}
\{x \leftarrow_{c_0} E_0\}x & = & E_0 & \\
\{x \leftarrow_{c_0} E_0\}y & = & y & \text{si } y \neq x \\
\{x \leftarrow_{c_0} E_0\}[E]_c^T & = & [\{x \leftarrow_{c_0} E_0\}E]_{\{x \leftarrow_{c_0} E_0\}c}^{\{x \leftarrow_{c_0} E_0\}T} & \\
\{x \leftarrow_{c_0} E_0\}c & = & (c \setminus \{x\}) \cup c_0 & \text{si } x \in c \\
\{x \leftarrow_{c_0} E_0\}c & = & c & \text{si } x \notin c
\end{array}$$

(les autres cas suivent la notion habituelle de substitution sans capture)

$$\begin{array}{rcl}
\mathbf{underl}(a) & = & a \\
\mathbf{underl}(A\ E) & = & \mathbf{underl}(A) \\
\mathbf{underl}(\pi_i\, A) & = & \mathbf{underl}(A)
\end{array}$$

$$\begin{array}{rcll}
\mathbf{reveal}^B(a) & = & E & \text{où } a = E :_c T \in B \\
\mathbf{reveal}^B(A\ E) & = & (\mathbf{reveal}^B(A))\ E & \\
\mathbf{reveal}^B(\pi_i\, A) & = & \pi_i\,(\mathbf{reveal}^B(A)) &
\end{array}$$

$$\frac{B; \mathsf{nil} \vdash_{c_0} E :^{\mathsf{P}} T \qquad \text{si } a \notin \mathbf{dom}\, B}{B, a = E :_{c_0} T; \mathsf{nil} \vdash_{\bullet} \mathsf{ok}}\ \textbf{(envok.a)} \qquad \frac{B; \Gamma \vdash_{c'} \mathsf{ok} \qquad \text{si } a = E :_{c_0} T \in B \wedge c_0 \subseteq c'}{B; \Gamma \vdash_{c' \cup \{a\}} \mathsf{ok}}\ \textbf{(envok.c.a)} \qquad \frac{B; \Gamma \vdash_{c'} \mathsf{ok} \qquad \text{si } x :_{c_0} T \in \Gamma}{B; \Gamma \vdash_{c' \cup \{x\}} \mathsf{ok}}\ \textbf{(envok.c.x)}$$

$$\frac{}{\mathsf{nil}; \mathsf{nil} \vdash_{\bullet} \mathsf{ok}}\ \textbf{(envok.nil)} \qquad \frac{B; \Gamma \vdash_c T : * \qquad \text{si } x \notin \mathbf{dom}\, \Gamma}{B; \Gamma, x :_c T \vdash_{\bullet} \mathsf{ok}}\ \textbf{(envok.x)}$$

$$\frac{B; \Gamma \vdash_c \mathsf{ok} \qquad B; \Gamma_0 \vdash_{c_0} \xi\ \mathsf{transparent} \qquad \text{si } \Gamma = (\Gamma_0, x :_{c_0} T, \Gamma_1) \wedge x \in c}{B; \Gamma \vdash_c \xi\ \mathsf{transparent}}\ \textbf{(vis.env)} \qquad \frac{B; \Gamma \vdash_c \mathsf{ok} \qquad \text{si } \xi \in c}{B; \Gamma \vdash_c \xi\ \mathsf{transparent}}\ \textbf{(vis.in)} \qquad \frac{B; \Gamma \vdash_c \mathsf{ok}}{B; \Gamma \vdash_c \bullet\ \mathsf{transparent}}\ \textbf{(vis.o)}$$

$$\frac{B; \Gamma \vdash_c c_1\ \mathsf{transparent} \qquad B; \Gamma \vdash_c c_2\ \mathsf{transparent}}{B; \Gamma \vdash_c c_1 \cup c_2\ \mathsf{transparent}}\ \textbf{(vis.union)}$$

$$\frac{B; \Gamma \vdash_c c_0\ \mathsf{transparent} \qquad \text{si } a = E :_{c_0} T \in B}{B; \Gamma \vdash_c a \triangleright E : T}\ \textbf{(ac.a)} \qquad \frac{B; \Gamma \vdash_c A \triangleright E : \Pi x : T_0.\,^{\mathsf{P}} T_1 \qquad B; \Gamma \vdash_c E_0 :^{\mathsf{P}} T_0}{B; \Gamma \vdash_c A\ E_0 \triangleright E\ E_0 : \{x \leftarrow_c E_0\} T_1}\ \textbf{(ac.app)}$$

$$\frac{B; \Gamma \vdash_c A \triangleright E : \Sigma x : T_1.\ T_2}{B; \Gamma \vdash_c \pi_1 A \triangleright \pi_1 E : T_1}\ \textbf{(ac.proj.1)} \qquad \frac{B; \Gamma \vdash_c E_1 :^{\mathsf{P}} S(\pi_1 E) \qquad B; \Gamma \vdash_c A \triangleright E : \Sigma x : T_1.\ T_2}{B; \Gamma \vdash_c \pi_2 A \triangleright \pi_2 E : \{x \leftarrow_c E_1\} T_2}\ \textbf{(ac.proj.2)}$$

$$\frac{B; \Gamma \vdash_c A \triangleright E : \mathrm{TYPE}^K}{B; \Gamma \vdash_c (\!|A|\!) : K}\ \textbf{(tok.abs)} \qquad \frac{B; \Gamma \vdash_c \mathsf{ok}}{B; \Gamma \vdash_c \mathrm{BOOL} : \wr}\ \textbf{(tok.base.bool)} \qquad \frac{B; \Gamma \vdash_c \mathsf{ok}}{B; \Gamma \vdash_c \mathrm{DYN} : K}\ \textbf{(tok.base.dyn)}$$

$$\frac{B; \Gamma \vdash_c \mathsf{ok}}{B; \Gamma \vdash_c \mathrm{INT} : \wr}\ \textbf{(tok.base.int)} \qquad \frac{B; \Gamma \vdash_c \mathsf{ok}}{B; \Gamma \vdash_c \mathrm{UNIT} : \wr}\ \textbf{(tok.base.unit)} \qquad \frac{B; \Gamma \vdash_c E :^{\mathsf{P}} \mathrm{TYPE}^K}{B; \Gamma \vdash_c \mathsf{Typ}\ E : K}\ \textbf{(tok.field)}$$

$$\frac{B;\Gamma \vdash_c T' : K' \qquad B;\Gamma, x :_c T' \vdash_{c\cup\{x\}} T'' : K''}{B;\Gamma \vdash_c \Pi x : T'.\,{}^{I}T'' : \wr}\;\textbf{(tok.fun.I)}$$

$$\frac{B;\Gamma \vdash_c T' : K' \qquad B;\Gamma, x :_c T' \vdash_{c\cup\{x\}} T'' : K''}{B;\Gamma \vdash_c \Pi x : T'.\,{}^{P}T'' : K''}\;\textbf{(tok.fun.P)}$$

$$\frac{B;\Gamma \vdash_c T' : K' \qquad B;\Gamma, x :_c T' \vdash_{c\cup\{x\}} T'' : K''}{B;\Gamma \vdash_c \Sigma x : T'.\, T'' : K' \vee K''}\;\textbf{(tok.pair)}$$

$$\frac{B;\Gamma \vdash_c \mathsf{ok}}{B;\Gamma \vdash_c \textsc{type}^{K} : *}\;\textbf{(tok.type)} \qquad \frac{B;\Gamma \vdash_c E :^{P} T}{B;\Gamma \vdash_c S(E) : \wr}\;\textbf{(tok.sing)}$$

$$\frac{B;\Gamma \vdash_c T : K' \qquad \text{si } K' \leqslant K}{B;\Gamma \vdash_c T : K}\;\textbf{(tok.sub)}$$

$$\frac{B;\Gamma \vdash_c E_0 \longrightarrow E_0' \qquad B;\Gamma \vdash_c E_0 :^{P} T_0 \qquad B;\Gamma \vdash_c A \triangleright E : \Pi x : T_0.\,{}^{P}T_1}{B;\Gamma \vdash_c A\, E_0 \longrightarrow A\, E_0'}\;\textbf{(aconv.cong.app.arg)}$$

$$\frac{B;\Gamma \vdash_c A \longrightarrow A' \qquad B;\Gamma \vdash_c A \triangleright E : \Pi x : T_0.\,{}^{P}T_1 \qquad B;\Gamma \vdash_c E_0 :^{P} T_0}{B;\Gamma \vdash_c A\, E_0 \longrightarrow A'\, E_0}\;\textbf{(aconv.cong.app.fun)}$$

$$\frac{B;\Gamma \vdash_c A \longrightarrow A' \qquad B;\Gamma \vdash_c A \triangleright E : \Sigma x : T_1.\, T_2}{B;\Gamma \vdash_c \pi_i A \longrightarrow \pi_i A}\;\textbf{(aconv.cong.proj)}$$

$$\frac{B;\Gamma \vdash_c A_1 \longrightarrow A_2}{B;\Gamma \vdash_c A_1 \equiv A_2}\;\textbf{(aeq.conv)} \qquad \frac{B;\Gamma \vdash_c A \triangleright E : T}{B;\Gamma \vdash_c A \equiv A}\;\textbf{(aeq.refl)} \qquad \frac{B;\Gamma \vdash_c A_2 \equiv A_1}{B;\Gamma \vdash_c A_1 \equiv A_2}\;\textbf{(aeq.sym)}$$

$$\frac{B;\Gamma \vdash_c A_1 \equiv A_2 \qquad B;\Gamma \vdash_c A_2 \equiv A_3}{B;\Gamma \vdash_c A_1 \equiv A_3}\;\textbf{(aeq.trans)}$$

$$\frac{B;\Gamma \vdash_c A \longrightarrow A' \qquad B;\Gamma \vdash_c A \triangleright E : \textsc{type}^{K}}{B;\Gamma \vdash_c (\!|A|\!) \longrightarrow (\!|A'|\!)}\;\textbf{(tconv.cong.abs)}$$

$$\frac{B;\Gamma \vdash_c E \longrightarrow E' \qquad B;\Gamma \vdash_c E :^{P} \textsc{type}^{*}}{B;\Gamma \vdash_c \mathsf{Typ}\, E \longrightarrow \mathsf{Typ}\, E'}\;\textbf{(tconv.cong.field)}$$

$$\frac{B;\Gamma \vdash_c T_0 \longrightarrow T_0' \qquad B;\Gamma, x :_c T_0 \vdash_{c\cup\{x\}} T_1 : *}{B;\Gamma \vdash_c \Pi x : T_0.\,{}^{\Upsilon}T_1 \longrightarrow \Pi x : T_0'.\,{}^{\Upsilon}T_1}\;\textbf{(tconv.cong.fun.arg)}$$

$$\frac{B;\Gamma \vdash_c T_0 : * \qquad B;\Gamma, x :_c T_0 \vdash_{c\cup\{x\}} T_1 \longrightarrow T_1'}{B;\Gamma \vdash_c \Pi x : T_0.\,{}^{\Upsilon}T_1 \longrightarrow \Pi x : T_0.\,{}^{\Upsilon}T_1'}\;\textbf{(tconv.cong.fun.ret)}$$

$$\frac{B;\Gamma \vdash_c T_1 \longrightarrow T_1' \qquad B;\Gamma, x :_c T_1 \vdash_{c\cup\{x\}} T_2 : *}{B;\Gamma \vdash_c \Sigma x : T_1.\, T_2 \longrightarrow \Sigma x : T_1'.\, T_2}\;\textbf{(tconv.cong.pair.1)}$$

$$\frac{B;\Gamma, x :_c T_1 \vdash_{c\cup\{x\}} T_2 \longrightarrow T_2' \qquad B;\Gamma \vdash_c T_1 : *}{B;\Gamma \vdash_c \Sigma x : T_1.\, T_2 \longrightarrow \Sigma x : T_1.\, T_2'}\;\textbf{(tconv.cong.pair.2)} \qquad \frac{B;\Gamma \vdash_c E \longrightarrow E'}{B;\Gamma \vdash_c S(E) \longrightarrow S(E')}\;\textbf{(tconv.cong.sing)}$$

$$\frac{B;\Gamma \vdash_c A \triangleright E : \textsc{type}^{K} \qquad B;\Gamma \vdash_c \mathbf{underl}(A)\ \mathsf{transparent}}{B;\Gamma \vdash_c (\!|A|\!) \longrightarrow \mathsf{Typ}\, E}\;\textbf{(tconv.abs)} \qquad \frac{B;\Gamma \vdash_c T : *}{B;\Gamma \vdash_c \mathsf{Typ}\,\langle T\rangle \longrightarrow T}\;\textbf{(tconv.field)}$$

$$\frac{B;\Gamma \vdash_c \mathsf{ok}}{B;\Gamma \vdash_c S(()) \longrightarrow \textsc{unit}}\;\textbf{(tconv.unit)}$$

$$\frac{B;\Gamma \vdash_c T_1 \longrightarrow T_2}{B;\Gamma \vdash_c T_1 \equiv T_2}\,\textbf{(teq.conv)} \qquad \frac{B;\Gamma \vdash_c T : *}{B;\Gamma \vdash_c T \equiv T}\,\textbf{(teq.refl)} \qquad \frac{B;\Gamma \vdash_c T_2 \equiv T_1}{B;\Gamma \vdash_c T_1 \equiv T_2}\,\textbf{(teq.sym)} \qquad \frac{\begin{array}{c}B;\Gamma \vdash_c T_1 \equiv T_2\\ B;\Gamma \vdash_c T_2 \equiv T_3\end{array}}{B;\Gamma \vdash_c T_1 \equiv T_3}\,\textbf{(teq.trans)}$$

$$\frac{\begin{array}{c}B;\Gamma \vdash_c E \longrightarrow E'\\ B;\Gamma \vdash_c E :^{P} T_0\\ B;\Gamma \vdash_c E_1 :^{P} \Pi x : T_0.\ {}^{P}T_1\end{array}}{B;\Gamma \vdash_c E_1\, E \longrightarrow E_1\, E'}\,\textbf{(econv.cong.app.arg)} \qquad \frac{\begin{array}{c}B;\Gamma \vdash_c E \longrightarrow E'\\ B;\Gamma \vdash_c E :^{P} \Pi x : T_0.\ {}^{P}T_1\\ B;\Gamma \vdash_c E_0 :^{P} T_0\end{array}}{B;\Gamma \vdash_c E\, E_0 \longrightarrow E'\, E_0}\,\textbf{(econv.cong.app.fun)}$$

$$\frac{\begin{array}{c}B;\Gamma \vdash_{c'} E \longrightarrow E'\\ B;\Gamma \vdash_{c'} E :^{P} T\\ B;\Gamma \vdash_{c\cap c'} T : \wr\\ B;\Gamma \vdash_c \mathsf{ok}\end{array}}{B;\Gamma \vdash_c [E]^{T}_{c'} \longrightarrow [E']^{T}_{c'}}\,\textbf{(econv.cong.col.e)} \qquad \frac{\begin{array}{c}B;\Gamma \vdash_{c'} E :^{P} T_1\\ B;\Gamma \vdash_{c\cap c'} T_1 \longrightarrow T_2\\ B;\Gamma \vdash_{c\cap c'} T_1 : \wr\\ B;\Gamma \vdash_c \mathsf{ok}\end{array}}{B;\Gamma \vdash_c [E]^{T_1}_{c'} \longrightarrow [E]^{T_2}_{c'}}\,\textbf{(econv.cong.col.t)}$$

$$\frac{\begin{array}{c}B;\Gamma \vdash_{\bullet} E \longrightarrow E'\\ B;\Gamma \vdash_{\bullet} T : \wr\\ B;\Gamma \vdash_{\bullet} E :^{P} T\\ B;\Gamma \vdash_c \mathsf{ok}\end{array}}{B;\Gamma \vdash_c \mathsf{dynned}\ E\ \mathsf{at}\ T \longrightarrow \mathsf{dynned}\ E'\ \mathsf{at}\ T}\,\textbf{(econv.cong.dynned.e)}$$

$$\frac{\begin{array}{c}B;\Gamma \vdash_{\bullet} T \longrightarrow T'\\ B;\Gamma \vdash_{\bullet} T : \wr\\ B;\Gamma \vdash_{\bullet} E :^{P} T\\ B;\Gamma \vdash_c \mathsf{ok}\end{array}}{B;\Gamma \vdash_c \mathsf{dynned}\ E\ \mathsf{at}\ T \longrightarrow \mathsf{dynned}\ E\ \mathsf{at}\ T'}\,\textbf{(econv.cong.dynned.t)} \qquad \frac{B;\Gamma \vdash_c T \longrightarrow T'}{B;\Gamma \vdash_c \langle T\rangle \longrightarrow \langle T'\rangle}\,\textbf{(econv.cong.field)}$$

$$\frac{\begin{array}{c}B;\Gamma \vdash_c T_0 \longrightarrow T_0'\\ B;\Gamma, x :_c T_0 \vdash_{c\cup\{x\}} E_1 :^{\gamma} T_1\end{array}}{B;\Gamma \vdash_c (\lambda x : T_0.\ E_1) \longrightarrow (\lambda x : T_0'.\ E_1)}\,\textbf{(econv.cong.fun.arg)}$$

$$\frac{\begin{array}{c}B;\Gamma, x :_c T_0 \vdash_{c\cup\{x\}} E \longrightarrow E'\\ B;\Gamma, x :_c T_0, y :_{c\cup\{x\}} S(E) \vdash_{c\cup\{y\}\cup\{x\}} E_1 :^{\gamma} T_1\end{array}}{B;\Gamma \vdash_c (\lambda x : T_0.\ \{y \leftarrow_{c\cup\{x\}} E\}E_1) \longrightarrow (\lambda x : T_0.\ \{y \leftarrow_{c\cup\{x\}} E'\}E_1)}\,\textbf{(econv.cong.fun.body)}$$

$$\frac{\begin{array}{c}B;\Gamma, x :_c T_0 \vdash_{c'} E_1 :^{\gamma} T_1\\ B;\Gamma, x :_{c\cap c'} T_0 \vdash_{(c\cap c')\cup\{x\}} T_1 \longrightarrow T_1'\\ B;\Gamma, x :_{c\cap c'} T_0 \vdash_{(c\cap c')\cup\{x\}} T_1 : \wr\end{array}}{B;\Gamma \vdash_c (\lambda x : T_0.\ E_1 \,!!_{c'}\, T_1) \longrightarrow (\lambda x : T_0.\ E_1 \,!!_{c'}\, T_1')}\,\textbf{(econv.cong.fun.seal)}$$

$$\frac{\begin{array}{c}B;\Gamma \vdash_c E \longrightarrow E'\\ B;\Gamma \vdash_c E_2 :^{P} T_2\end{array}}{B;\Gamma \vdash_c (E, E_2) \longrightarrow (E', E_2)}\,\textbf{(econv.cong.pair.1)} \qquad \frac{\begin{array}{c}B;\Gamma \vdash_c E \longrightarrow E'\\ B;\Gamma \vdash_c E_1 :^{P} T_1\end{array}}{B;\Gamma \vdash_c (E_1, E) \longrightarrow (E_1, E')}\,\textbf{(econv.cong.pair.2)}$$

$$\frac{\begin{array}{c}B;\Gamma \vdash_c E \longrightarrow E'\\ B;\Gamma \vdash_c E :^{P} \Sigma x : T_1.\ T_2\end{array}}{B;\Gamma \vdash_c \pi_i\, E \longrightarrow \pi_i\, E'}\,\textbf{(econv.cong.proj)} \qquad \frac{\begin{array}{c}B;\Gamma, x :_c T_0 \vdash_{c\cup\{x\}} E_1 :^{P} T_1\\ B;\Gamma \vdash_c E_0 :^{P} T_0\end{array}}{B;\Gamma \vdash_c (\lambda x : T_0.\ E_1)\, E_0 \longrightarrow \{x \leftarrow_c E_0\}E_1}\,\textbf{(econv.app)}$$

$$\frac{\begin{array}{c}B;\Gamma \vdash_{c'} \mathsf{ok}\\ B;\Gamma \vdash_c \mathsf{ok}\end{array}}{B;\Gamma \vdash_c [bv]^{\mathrm{BOOL}}_{c'} \longrightarrow bv}\,\textbf{(econv.col.base.bool)} \qquad \frac{\begin{array}{c}B;\Gamma \vdash_{c'} \mathsf{ok}\\ B;\Gamma \vdash_c \mathsf{ok}\end{array}}{B;\Gamma \vdash_c [\underline{n}]^{\mathrm{INT}}_{c'} \longrightarrow \underline{n}}\,\textbf{(econv.col.base.int)}$$

$$\frac{\begin{array}{c} B;\Gamma \vdash_{c'} \ \mathsf{ok} \\ B;\Gamma \vdash_{c} \ \mathsf{ok} \end{array}}{B;\Gamma \vdash_{c} [()]_{c'}^{\mathrm{UNIT}} \longrightarrow ()}\ \textbf{(econv.col.base.unit)}$$

$$\frac{\begin{array}{c} B;\Gamma \vdash_{\bullet} E :^{\mathsf{P}} T \\ B;\Gamma \vdash_{\bullet} T : \wr \\ B;\Gamma \vdash_{c'} \ \mathsf{ok} \\ B;\Gamma \vdash_{c} \ \mathsf{ok} \end{array}}{B;\Gamma \vdash_{c} [\mathsf{dynned}\ E\ \mathsf{at}\ T]_{c'}^{\mathrm{DYN}} \longrightarrow \mathsf{dynned}\ E\ \mathsf{at}\ T}\ \textbf{(econv.col.dynned)}$$

$$\frac{\begin{array}{c} B;\Gamma \vdash_{c'} T_0 <: T_2 \\ B;\Gamma, x :_{c'} T_2 \vdash_{c' \cup \{x\}} E :^{\mathrm{I}} T_1 \\ B;\Gamma, x :_{c \cap c'} T_0 \vdash_{(c \cap c') \cup \{x\}} T_1 : \wr \\ B;\Gamma \vdash_{c} \ \mathsf{ok} \end{array}}{B;\Gamma \vdash_{c} [\lambda x : T_2.\ E]_{c'}^{\Pi x : T_0.\ ^{\mathrm{I}} T_1} \longrightarrow \lambda x : T_0.\ E\ !!_{c' \cup \{x\}}\ T_1}\ \textbf{(econv.col.fun.I)}$$

$$\frac{\begin{array}{c} B;\Gamma \vdash_{c'} T_0 <: T_2 \\ B;\Gamma, x :_{c'} T_2 \vdash_{c' \cup \{x\}} E :^{\mathsf{P}} T_1 \\ B;\Gamma, x :_{c \cap c'} T_0 \vdash_{(c \cap c') \cup \{x\}} T_1 : \wr \\ B;\Gamma \vdash_{c} \ \mathsf{ok} \end{array}}{B;\Gamma \vdash_{c} [\lambda x : T_2.\ E]_{c'}^{\Pi x : T_0.\ ^{\mathsf{P}} T_1} \longrightarrow \lambda x : T_0.\ [E]_{c' \cup \{x\}}^{T_1}}\ \textbf{(econv.col.fun.P)}$$

$$\frac{\begin{array}{c} B;\Gamma \vdash_{c_2} E :^{\mathsf{P}} T_2 \\ B;\Gamma \vdash_{c_1 \cap c_2} T_2 : \wr \\ B;\Gamma \vdash_{c_1} T_2 <: T_1 \\ B;\Gamma \vdash_{c \cap c_1} T_1 : \wr \\ B;\Gamma \vdash_{c} \ \mathsf{ok} \end{array}}{B;\Gamma \vdash_{c} [[E]_{c_2}^{T_2}]_{c_1}^{T_1} \longrightarrow [E]_{c_1 \cup c_2}^{T_1}}\ \textbf{(econv.col.merge)}$$

$$\frac{\begin{array}{c} B;\Gamma \vdash_{c'} E_1 :^{\mathsf{P}} T_1 \\ B;\Gamma \vdash_{c \cap c'} T_1 : \wr \\ B;\Gamma, x :_{c'} T_1 \vdash_{c' \cup \{x\}} E_2 :^{\mathsf{P}} T_2 \\ B;\Gamma \vdash_{c'} E_2 :^{\mathsf{P}} \{x \leftarrow_{c'} [E_1]_{c'}^{T_1}\} T_2 \\ B;\Gamma, x :_{c \cap c'} T_1 \vdash_{(c \cap c') \cup \{x\}} T_2 : \wr \\ B;\Gamma \vdash_{c} \ \mathsf{ok} \end{array}}{B;\Gamma \vdash_{c} [(E_1, E_2)]_{c'}^{\Sigma x : T_1.\ T_2} \longrightarrow ([E_1]_{c'}^{T_1}, [E_2]_{c'}^{\{x \leftarrow_{c'} [E_1]_{c'}^{T_1}\} T_2})}\ \textbf{(econv.col.pair)}$$

$$\frac{\begin{array}{c} B;\Gamma \vdash_{c'} E' :^{\mathsf{P}} S(E) \\ B;\Gamma \vdash_{c \cap c'} E :^{\mathsf{P}} T \\ B;\Gamma \vdash_{c} \ \mathsf{ok} \end{array}}{B;\Gamma \vdash_{c} [E']_{c'}^{S(E)} \longrightarrow E}\ \textbf{(econv.col.sing)}$$

$$\frac{\begin{array}{c} B;\Gamma \vdash_{c} E_1 :^{\mathsf{P}} T_1 \\ B;\Gamma \vdash_{c} E_2 :^{\mathsf{P}} T_2 \end{array}}{B;\Gamma \vdash_{c} \pi_i (E_1, E_2) \longrightarrow E_i}\ \textbf{(econv.proj)}$$

$$\frac{B;\Gamma \vdash_{c} E :^{\mathsf{P}} \mathrm{TYPE}^*}{B;\Gamma \vdash_{c} E \longrightarrow \langle \mathsf{Typ}\ E \rangle}\ \textbf{(econv.eta.field)}$$

$$\frac{B;\Gamma \vdash_{c} E :^{\mathsf{P}} \Pi x : T_0.\ ^{\gamma} T_1}{B;\Gamma \vdash_{c} E \longrightarrow (\lambda x : T_0.\ E\ x)}\ \textbf{(econv.eta.fun)}$$

$$\frac{B;\Gamma \vdash_{c} E :^{\mathsf{P}} \Sigma x : T_1.\ T_2}{B;\Gamma \vdash_{c} E \longrightarrow (\pi_1 E, \pi_2 E)}\ \textbf{(econv.eta.pair)}$$

$$\frac{B;\Gamma\vdash_c E_1 \longrightarrow E_2}{B;\Gamma\vdash_c E_1 \equiv E_2}\,\textbf{(eeq.conv)}\qquad \frac{B;\Gamma\vdash_c E :^{P} T}{B;\Gamma\vdash_c E \equiv E}\,\textbf{(eeq.refl)}\qquad \frac{B;\Gamma\vdash_c E_2 \equiv E_1}{B;\Gamma\vdash_c E_1 \equiv E_2}\,\textbf{(eeq.sym)}\qquad \frac{\begin{array}{c}B;\Gamma\vdash_c E_1 \equiv E_2\\ B;\Gamma\vdash_c E_2 \equiv E_3\end{array}}{B;\Gamma\vdash_c E_1 \equiv E_3}\,\textbf{(eeq.trans)}$$

$$\frac{\begin{array}{c}B;\Gamma\vdash_c T_0' <: T_0\\ B;\Gamma, x :_c T_0' \vdash_{c\cup\{x\}} T_1 <: T_1'\\ B;\Gamma, x :_c T_0 \vdash_{c\cup\{x\}} T_1 : *\\ \text{si } \gamma \sqsubseteq \gamma'\end{array}}{B;\Gamma\vdash_c \Pi x : T_0.\ {}^{\gamma}T_1 <: \Pi x : T_0'.\ {}^{\gamma'}T_1'}\,\textbf{(tsub.cong.fun)}\qquad \frac{\begin{array}{c}B;\Gamma\vdash_c T_1 <: T_1'\\ B;\Gamma, x :_c T_1 \vdash_{c\cup\{x\}} T_2 <: T_2'\\ B;\Gamma, x :_c T_1' \vdash_{c\cup\{x\}} T_2' : *\end{array}}{B;\Gamma\vdash_c \Sigma x : T_1.\ T_2 <: \Sigma x : T_1'.\ T_2'}\,\textbf{(tsub.cong.pair)}$$

$$\frac{\begin{array}{c}B;\Gamma\vdash_c \mathsf{ok}\\ \text{si } K_1 \leqslant K_2\end{array}}{B;\Gamma\vdash_c \text{TYPE}^{K_1} <: \text{TYPE}^{K_2}}\,\textbf{(tsub.cong.type)}\qquad \frac{B;\Gamma\vdash_c T \equiv T'}{B;\Gamma\vdash_c T <: T'}\,\textbf{(tsub.eq)}\qquad \frac{\begin{array}{c}B;\Gamma\vdash_c T <: T'\\ B;\Gamma\vdash_c T' <: T''\end{array}}{B;\Gamma\vdash_c T <: T''}\,\textbf{(tsub.trans)}$$

$$\frac{B;\Gamma\vdash_c E :^{P} T}{B;\Gamma\vdash_c S(E) <: T}\,\textbf{(tsub.sing)}$$

$$\frac{\begin{array}{c}B;\Gamma\vdash_c E_1 :^{\gamma_1} \Pi x : T_0.\ {}^{\gamma_2}T\\ B;\Gamma\vdash_c E_0 :^{P} T_0\end{array}}{B;\Gamma\vdash_c E_1\ E_0 :^{\gamma_1\sqcup\gamma_2} \{x \leftarrow_c E_0\}T}\,\textbf{(et.app)}\qquad \frac{B;\Gamma\vdash_c \mathsf{ok}}{B;\Gamma\vdash_c bv :^{P} \text{BOOL}}\,\textbf{(et.base.bool)}\qquad \frac{B;\Gamma\vdash_c \mathsf{ok}}{B;\Gamma\vdash_c \underline{n} :^{P} \text{INT}}\,\textbf{(et.base.int)}$$

$$\frac{B;\Gamma\vdash_c \mathsf{ok}}{B;\Gamma\vdash_c () :^{P} \text{UNIT}}\,\textbf{(et.base.unit)}\qquad \frac{\begin{array}{c}B;\Gamma\vdash_{c'} E :^{\gamma} T\\ B;\Gamma\vdash_{c\cap c'} T : \wr\\ B;\Gamma\vdash_c \mathsf{ok}\end{array}}{B;\Gamma\vdash_c [E]^{T}_{c'} :^{\gamma} T}\,\textbf{(et.col)}\qquad \frac{\begin{array}{c}B;\Gamma\vdash_c E :^{\gamma} T\\ B;\Gamma\vdash_c T : \wr\end{array}}{B;\Gamma\vdash_c \mathsf{dyn}\ E\ \mathsf{at}\ T :^{I} \text{DYN}}\,\textbf{(et.dyn)}$$

$$\frac{\begin{array}{c}B;\Gamma\vdash_{\bullet} E :^{P} T\\ B;\Gamma\vdash_{\bullet} T : \wr\\ B;\Gamma\vdash_c \mathsf{ok}\end{array}}{B;\Gamma\vdash_c \mathsf{dynned}\ E\ \mathsf{at}\ T :^{P} \text{DYN}}\,\textbf{(et.dynned)}\qquad \frac{B;\Gamma, x :_c T_0 \vdash_{c\cup\{x\}} E :^{\gamma} T_1}{B;\Gamma\vdash_c \lambda x : T_0.\ E :^{P} \Pi x : T_0.\ {}^{\gamma}T_1}\,\textbf{(et.fun)}$$

$$\frac{\begin{array}{c}B;\Gamma\vdash_c E_0 :^{I} T_0\\ B;\Gamma, x :_c T_0 \vdash_{c\cup\{x\}} E :^{I} T\\ B;\Gamma\vdash_c T : *\end{array}}{B;\Gamma\vdash_c (\mathsf{let}\ x = E_0\ \mathsf{in}\ E : T) :^{I} T}\,\textbf{(et.let)}\qquad \frac{\begin{array}{c}B;\Gamma\vdash_c E_1 :^{\gamma} T_1\\ B;\Gamma\vdash_c E_2 :^{\gamma} T_2\end{array}}{B;\Gamma\vdash_c (E_1, E_2) :^{\gamma} T_1 * T_2}\,\textbf{(et.pair)}$$

$$\frac{B;\Gamma\vdash_c E :^{\gamma} \Sigma x : T_1.\ T_2}{B;\Gamma\vdash_c \pi_1 E :^{\gamma} T_1}\,\textbf{(et.proj.1)}\qquad \frac{\begin{array}{c}B;\Gamma\vdash_c E :^{P} \Sigma x : T_1.\ T_2\\ B;\Gamma\vdash_c E_1 :^{P} S(\pi_1 E)\end{array}}{B;\Gamma\vdash_c \pi_2 E :^{P} \{x \leftarrow_c E_1\}T_2}\,\textbf{(et.proj.2)}\qquad \frac{\begin{array}{c}B;\Gamma\vdash_{c\cup c'} E :^{\gamma} T\\ B;\Gamma\vdash_c T : *\end{array}}{B;\Gamma\vdash_c (E\ !!_{c'}\ T) :^{I} T}\,\textbf{(et.seal)}$$

$$\frac{B;\Gamma\vdash_c T : K}{B;\Gamma\vdash_c \langle T\rangle :^{P} \text{TYPE}^{K}}\,\textbf{(et.type)}\qquad \frac{\begin{array}{c}B;\Gamma\vdash_c E :^{\gamma} \text{DYN}\\ B;\Gamma\vdash_c E' :^{\gamma} T\end{array}}{B;\Gamma\vdash_c \mathsf{undyn}\ E\ \mathsf{at}\ T\ \mathsf{else}\ E' :^{I} T}\,\textbf{(et.undyn)}\qquad \frac{\begin{array}{c}B;\Gamma\vdash_c x\ \mathsf{transparent}\\ \text{si } x :_{c'} T \in \Gamma\end{array}}{B;\Gamma\vdash_c x :^{P} T}\,\textbf{(et.x)}$$

$$\frac{B;\Gamma\vdash_c E :^{P} T}{B;\Gamma\vdash_c E :^{P} S(E)}\,\textbf{(et.sing)}\qquad \frac{\begin{array}{c}B;\Gamma\vdash_c E :^{\gamma} T\\ B;\Gamma\vdash_c T <: T'\\ \text{si } \gamma \sqsubseteq \gamma'\end{array}}{B;\Gamma\vdash_c E :^{\gamma'} T'}\,\textbf{(et.sub)}$$

$$(\lambda x : T.\ E)\, V^{c} \longrightarrow_{c} \{x \leftarrow_{c} V^{c}\}E \qquad \text{(ered.app)}$$

$$[bv]^{\mathrm{BOOL}}_{c'} \longrightarrow_{c} bv \qquad \text{(ered.col.base.bool)}$$

$$[\underline{n}]^{\mathrm{INT}}_{c'} \longrightarrow_{c} \underline{n} \qquad \text{(ered.col.base.int)}$$

$$[()]^{\mathrm{UNIT}}_{c'} \longrightarrow_{c} () \qquad \text{(ered.col.base.unit)}$$

$$[\mathsf{dynned}\ V^{\bullet}\ \mathsf{at}\ T]^{\mathrm{DYN}}_{c'} \longrightarrow_{c} \mathsf{dynned}\ V^{\bullet}\ \mathsf{at}\ T \qquad \text{(ered.col.dynned)}$$

$$[\lambda x : T_2.\ E]^{\Pi x : T_0.\ ^{\mathrm{I}}T_1}_{c'} \longrightarrow_{c} \lambda x : T_0.\ E\ !!_{c' \cup \{x\}}\ T_1 \qquad \text{(ered.col.fun.I)}$$

$$[\lambda x : T_2.\ E]^{\Pi x : T_0.\ ^{\mathrm{P}}T_1}_{c'} \longrightarrow_{c} \lambda x : T_0.\ [E]^{T_1}_{c' \cup \{x\}} \qquad \text{(ered.col.fun.P)}$$

$$[[V^{c_2}]^{(\!|A_2|\!)}_{c_2}]^{(\!|A_1|\!)}_{c_1} \longrightarrow_{c} [V^{c_2}]^{(\!|A_1|\!)}_{c_1 \cup c_2} \qquad \text{(ered.col.merge)}$$

si A_1 et A_2 sont toutes deux opaques dans c_1 mais que A_2 est concrète dans c_2

$$[(V_1^{c'}, V_2^{c'})]^{\Sigma x : T_1.\ T_2}_{c'} \longrightarrow_{c} ([V_1^{c'}]^{T_1}_{c'}, [V_2^{c'}]^{\{x \leftarrow_{c} [V_1^{c'}]^{T_1}_{c'}\}T_2}_{c'}) \qquad \text{(ered.col.pair)}$$

$$[V^{c'}]^{\mathsf{S}(E)}_{c'} \longrightarrow_{c} E \qquad \text{(ered.col.sing)}$$

$$B \vdash [V^{c'}]^{(\!|A|\!)}_{c'} \longrightarrow_{c} B \vdash [V^{c'}]^{\mathsf{Typ}\,\mathbf{reveal}^{B}(A)}_{c'} \qquad \text{(ered.colAbs)}$$

si $\mathbf{underl}(A) \in c \cap c'$

$$[V^{c'}]^{\mathsf{Typ}\,\langle T\rangle}_{c'} \longrightarrow_{c} [V^{c'}]^{T}_{c'} \qquad \text{(ered.colTyp)}$$

$$\frac{E \longrightarrow_{c'} E'}{C^{c}_{c'} \cdot E \longrightarrow_{c} C^{c}_{c'} \cdot E'}\ \textbf{(ered.context)}$$

$$\mathsf{dyn}\ V^{c}\ \mathsf{at}\ T \longrightarrow_{c} \mathsf{dynned}\ [V^{c}]^{\mathbf{conc}^{B}_{c}(T)}_{c}\ \mathsf{at}\ \mathbf{conc}^{B}_{c}(T) \qquad \text{(ered.dyn)}$$

$$\mathsf{let}\ x = V^{c}\ \mathsf{in}\ E : T \longrightarrow_{c} \{x \leftarrow_{c} V^{c}\}E \qquad \text{(ered.let)}$$

$$\pi_i\,(V_1^{c}, V_2^{c}) \longrightarrow_{c} V_i \qquad \text{(ered.proj)}$$

$$B \vdash V^{c \cup c'}\ !!_{c'}\ T \longrightarrow_{c} B, a = V^{c \cup c'} :_{c \cup c'} T \vdash [V^{c \cup c'}]^{\mathbf{self}^{\,T}(a)}_{c \cup c' \cup \{a\}} \qquad \text{(ered.seal)}$$

où a est frais (c'est-à-dire que $a \notin \mathbf{dom}\,B$)

$$B \vdash \mathsf{undyn}\,(\mathsf{dyn}\ V^{c}\ \mathsf{at}\ T)\ \mathsf{at}\ T'\ \mathsf{else}\ E' \longrightarrow_{c} B \vdash \begin{cases} V^{c} & \text{si } B; \mathsf{nil} \vdash_{c} T <: T' \\ E' & \text{sinon} \end{cases} \qquad \text{(ered.undyn)}$$

Annexe B

Sûreté de TOPHAT

Cette annexe est consacré à la démonstration de la sûreté du langage TOPHAT présenté au chapitre IV. Le lecteur est invité à se reporter à l'annexe A pour un précis de la syntaxe et de la sémantique du langage.

Avertissement Notre démonstration s'appuie sur trois conjectures pour lesquelles nous ne ferons qu'indiquer le principe qui serait à la base d'une démonstration.

B.1 Typage

B.1.1 Correction

Lemme B.1.1 (les variables sont liées) Si $B; \Gamma, \Gamma' \vdash_c J$ alors B n'a pas de variable libre, les variables libres de Γ' sont contenues dans $\mathbf{dom}\,\Gamma$, et les variables libres de c et J sont contenues dans $\mathbf{dom}\,(\Gamma, \Gamma')$.

Démonstration. Par induction sur la dérivation. $\square$

Lemme B.1.2 (correction du lexique) Si $B, B'; \Gamma \vdash_c J$ alors $B; \mathsf{nil} \vdash_\bullet \mathsf{ok}$ par une sous-dérivation.

Démonstration. Par induction sur la dérivation. Dans la plupart des cas, il y a une prémisse de la forme $B, B_1; \Gamma_1 \vdash_{c_1} J_1$. Les exceptions sont les règles (envok.nil) ainsi que (envok.a) lorsque $B' = \mathsf{nil}$, et dans ces cas la conclusion est l'hypothèse. $\square$

Définition B.1.3 (numéro d'un apax) Le numéro de a_i dans le lexique $a_0 = E_0 :_{c_0} T_0, \ldots, a_k = E_k :_{c_k} T_k$ est i.

Lemme B.1.4 (fraîcheur des apax) Si $B; \Gamma \vdash_c J$ et $a_1 = E_1 :_{c_1} T_1 \in B$ et $a_2 = E_2 :_{c_2} T_2 \in B$ alors $a_1 = a_2$ si et seulement si a_1 et a_2 ont le même numéro dans B.

Un lexique peut donc être vu comme une fonction partielle des apax dans les triplets (expression, couleur, type), ce que nous ferons implicitement par la suite.

Démonstration. La numérotation est une fonction, d'où l'implication rétrograde. L'implication directe signifie que les apax sont choisis frais. Supposons par exemple que a_1 a un numéro strictement inférieur à celui de a_2, et notons $B = B_2, a_2 = E_2 :_{c_2} T_2, B_2'$. D'après le Lem. B.1.2 (correction du lexique), $B_2, a_2 = E_2 :_{c_2} T_2; \mathsf{nil} \vdash_\bullet \mathsf{ok}$. Ce jugement est forcément obtenu par la règle (envok.a), avec une hypothèse $a_2 \notin \mathbf{dom}\,B_2$: en particulier $a_2 \neq a_1$. $\square$

Corollaire B.1.5 (extraction directe d'un lexique) Si $B_0, a = E_0 :_{c_0} T_0, B_0'; \Gamma \vdash_c J$ alors $B_0; \mathsf{nil} \vdash_\bullet E :^P T$ par une sous-dérivation.

Démonstration. D'après le Lem. B.1.2 (correction du lexique), $B_0, a_2 = E_2 :_{c_2} T_2; \mathsf{nil} \vdash_\bullet \mathsf{ok}$. Ce jugement est forcément obtenu par la règle (envok.a), avec une hypothèse $B_0; \mathsf{nil} \vdash_\bullet E_0 :^P T_0$. □

Lemme B.1.6 (correction de l'environnement) Si $B; \Gamma, \Gamma' \vdash_c J$ alors $B; \Gamma \vdash_\bullet \mathsf{ok}$.

Démonstration. Par induction sur la dérivation. Dans la plupart des cas, il y a une prémisse de la forme $B; \Gamma, \Gamma_1 \vdash_{c_1} J_1$. (On note que Γ_1 peut être différent de Γ', par exemple il peut l'étendre dans le cas de (et.fun)). Le seul cas où ce raisonnement échoue est (envok.x) lorsque $\Gamma' = \mathsf{nil}$ (si $\Gamma' \neq \mathsf{nil}$, Γ_1 est Γ' privé de son dernier élément) ; mais ce cas est trivial. □

Lemme B.1.7 (fraîcheur des variables) Si $B; \Gamma \vdash_c J$ et $x_1 :_{c_1} T_1 \in \Gamma$ et $x_2 :_{c_2} T_2 \in \Gamma$ alors $x_1 = x_2$ si et seulement si x_1 et x_2 ont le même numéro dans Γ.

Un environnement peut donc être vu comme une fonction partielle des variables dans les paires (couleur, type), ce que nous ferons implicitement par la suite.

Démonstration. Le raisonnement est le même que pour les apax (Lem. B.1.4 (fraîcheur des apax)), en utilisant le Lem. B.1.6 (correction de l'environnement). □

Définition B.1.8 (correction d'une couleur par rapport à un lexique) La couleur c est dite bien formée par rapport au lexique B si et seulement si pour tout apax a dans c, il existe c_0, E_0 et T_0 tels que $a = E_0 :_{c_0} T_0 \in B$ et $c_0 \subseteq c$.

Définition B.1.9 (correction d'une couleur par rapport à un environnement) La couleuur c est dite bien formée par rapport à l'environnement Γ si et seulement si toute variable x dans c est liée par Γ.

Lemme B.1.10 (bonne formation des couleurs) Si $B; \Gamma \vdash_c J$ alors c est bien formée par rapport à B et Γ. De plus, si $\Gamma = (\Gamma_0, x :_{c_0} T_0, \Gamma_1)$ alors c_0 est bien formée par rapport à B et Γ_0.

Démonstration. Par induction sur la dérivation.

Pour ce qui est de la couleur du jugement dans tous les cas autres que les règles (envok.*), il y a une prémisse de même lexique, et dont l'environnement et la couleur sont soit Γ et c, soit $\Gamma, x :_c T$ et $c \cup \{x\}$; dans tous ces cas le résultat est vrai par induction (dans le deuxième cas, c'est parce que c annote l'entrée pour x dans l'environnement de la prémisse). Pour les règles (envok.c.*), l'hypothèse de récurrence combinée avec les prémisses supplémentaires donne le résultat. Les règles restantes sont triviales car $c = \bullet$.

Pour ce qui est de la couleur d'une entrée dans l'environnement, le résultat vient de même par une prémisse de même environnement, sauf dans le cas de (envok.x) si $\Gamma_1 = \mathsf{nil}$, qui résulte de la première partie de l'hypothèse d'induction. □

Lemme B.1.11 (dérivation d'une couleur bien formée) Si $B; \Gamma \vdash_\bullet \mathsf{ok}$ et si c est bien formée par rapport à B et par rapport à Γ alors $B; \Gamma \vdash_c \mathsf{ok}$.

Démonstration. Appliquer (envok.c.a) pour chaque apax et (envok.c.x) pour chaque variable de c (les conditions de bonne formation assurent que les conditions de bord sont remplies). □

Lemme B.1.12 (correction de la couleur et de l'environnement) Si $B; \Gamma \vdash_c J$ alors $B; \Gamma \vdash_c \mathsf{ok}$.

Démonstration. D'après le Lem. B.1.6 (correction de l'environnement), $B; \Gamma \vdash_{\bullet} \mathsf{ok}$. D'après le Lem. B.1.10 (bonne formation des couleurs), c est bien formée par rapport à B et Γ. D'après le Lem. B.1.11 (dérivation d'une couleur bien formée), $B; \Gamma \vdash_{c} \mathsf{ok}$. □

Lemme B.1.13 (correction d'une sous-couleur) Si $B; \Gamma \vdash_{c} \mathsf{ok}$ et $c \subseteq c'$ alors $B; \Gamma \vdash_{c'} \mathsf{ok}$.

Démonstration. Par le Lem. B.1.6 (correction de l'environnement), on a $B; \Gamma \vdash_{\bullet} \mathsf{ok}$. Par le Lem. B.1.10 (bonne formation des couleurs), c est bien formée par rapport à B et Γ. Les conditions de bonne formation étant croissantes, c' est toute aussi bien formée. Par le Lem. B.1.10 (bonne formation des couleurs), on a $B; \Gamma \vdash_{c'} \mathsf{ok}$. □

Corollaire B.1.14 (extraction directe d'un environnement) Si $B; \Gamma_0, x :_{c_0} T_0, \Gamma_0' \vdash_{c} J$ alors $B; \Gamma_0 \vdash_{c_0} T_0 : *$ par une sous-dérivation.

Démonstration. D'après le Lem. B.1.6 (correction de l'environnement), $B; \Gamma_0, x :_{c_0} T_0 \vdash_{\bullet} \mathsf{ok}$. Ce jugement est forcément obtenu par la règle (envok.x), avec une hypothèse $B; \Gamma_0 \vdash_{c_0} T_0 : *$. □

B.1.2 Transparence

Lemme B.1.15 (transparence point par point) Soit c_0 une couleur close. On a $B; \Gamma \vdash_{c} c_0\,\mathsf{transparent}$ si et seulement si pour tout élément ξ de c_0, on a $B; \Gamma \vdash_{c} \xi\,\mathsf{transparent}$.

Dans le sens direct, on a $B; \Gamma \vdash_{c} \xi\,\mathsf{transparent}$ par une sous-dérivation de $B; \Gamma \vdash_{c} c_0\,\mathsf{transparent}$.

Démonstration. Analyse triviale des règles (vis.*). □

Définition B.1.16 (fermeture transparente) La **fermeture transparente** de la couleur c par rapport à l'environnement Γ, notée $\mathbf{clos}^{\Gamma}(c)$, est définie ainsi :

$$\begin{array}{rcll} \mathbf{clos}^{\Gamma}(c) & = & c & \text{si } c \text{ est close} \\ \mathbf{clos}^{\Gamma}(c \cup \{x\}) & = & \mathbf{clos}^{\Gamma}(c) \cup \{x\} \cup \mathbf{clos}^{\Gamma_0}(c_0) & \text{si } \Gamma = (\Gamma_0, x :_{c_0} T, \Gamma_1) \end{array}$$

On vérifie trivialement que cette définition est non contradictoire (elle ne dépend pas de l'ordre dans lequel on ajoute les variables) et définie dès lors que c est bien formée par rapport à Γ. On vérifie également qu'à Γ fixé, la fermeture transitive est compatible avec l'union des couleurs et croissante pour l'inclusion des couleurs. Enfin, $\mathbf{clos}^{\Gamma}(c) = \mathbf{clos}^{\Gamma,\Gamma'}(c)$ dès que le membre de gauche est défini.

Lemme B.1.17 (dénotation de la transparence) Si $B; \Gamma \vdash_{c} \mathsf{ok}$ alors $B; \Gamma \vdash_{c} c'\,\mathsf{transparent}$ si et seulement si $\mathbf{clos}^{\Gamma}(c') \subseteq \mathbf{clos}^{\Gamma}(c)$.

Démonstration. Montrons d'abord l'implication directe, par induction sur la démonstration de $B; \Gamma \vdash_{c} c'\,\mathsf{transparent}$. En vertu du Lem. B.1.15 (transparence point par point), il suffit de prouver que si $B; \Gamma \vdash_{c} \xi\,\mathsf{transparent}$ alors $\xi \in \mathbf{clos}^{\Gamma}(c)$. Si ce jugement est obtenu par (vis.in), on a $\xi \in c$, d'où $\xi \in \mathbf{clos}^{\Gamma}(c)$. Sinon le jugement est obtenu par (vis.env) avec la prémisse $B; \Gamma_0 \vdash_{c_0} \xi\,\mathsf{transparent}$, via une couleur c_0 qui vérifie $\mathbf{clos}^{\Gamma_0}(c_0) \subseteq \mathbf{clos}^{\Gamma}(c)$ par induction.

Réciproquement, supposons $B; \Gamma \vdash_{c} \mathsf{ok}$ et $\mathbf{clos}^{\Gamma}(c') \subseteq \mathbf{clos}^{\Gamma}(c)$. Grâce au Lem. B.1.15 (transparence point par point), il suffit de prouver que si $\xi \in \mathbf{clos}^{\Gamma}(c)$ alors $B; \Gamma \vdash_{c} \xi\,\mathsf{transparent}$. On décompose inductivement $\mathbf{clos}^{\Gamma}(c)$ suivant la définition de la fermeture transparente, en appliquant (vis.in) ou (vis.env) suivant la partie où se trouve ξ. □

Corollaire B.1.18 (transitivité de la transparence) Si $B; \Gamma \vdash_{c} c'\,\mathsf{transparent}$ et $B; \Gamma \vdash_{c'} c''\,\mathsf{transparent}$ alors $B; \Gamma \vdash_{c} c''\,\mathsf{transparent}$.

Démonstration. Notons que $B; \Gamma \vdash_c \mathsf{ok}$ par le Lem. B.1.12 (correction de la couleur et de l'environnement). On a immédiatement le résultat souhaité par le Lem. B.1.17 (dénotation de la transparence). $\square$

B.1.3 Affaiblissement

Dans les raisonnements génériques sur les règles de typage, nous notons $\widehat{\aleph}$ une métavariable et $\tilde{\aleph}$ un méta-motif (formé de termes du langage ou de la métalangue, pouvant contenir des métavariables).

Lemme B.1.19 (affaiblissement du lexique) Si $B; \Gamma \vdash_c J$ et $B, B'; \mathsf{nil} \vdash_\bullet \mathsf{ok}$ alors $B, B'; \Gamma \vdash_c J$.

Démonstration. Nous décrivons une transformation de la dérivation de $B; \Gamma \vdash_c J$ en une dérivation de $B, B'; \Gamma \vdash_c J$, construite par induction sur son argument. La plupart des règles de typage autorisent un lexique arbitraire $\widehat{B}$ dans leur conclusion, que l'on peut donc instancier en B, B' au lieu de B. Changer B en B, B' dans une prémisse de la forme $B; \Gamma_1 \vdash_{c_1} J_1$ correspond à appeler récursivement la transformation, pourvu que la métavariable $\widehat{B}$ n'apparaisse pas ailleurs dans la prémisse. Une prémisse de la forme $a_1 = E_1 :_{c_1} T_1 \in B$ reste vraie si l'on remplace B par B'.

Les seules règles que la méthode générale ne traite pas sont (envok.a) et (envok.nil), qui exigent une forme particulière pour le lexique B ; mais dans ces deux cas la conclusion désirée est $B, B'; \mathsf{nil} \vdash_\bullet \mathsf{ok}$ qui est vrai par hypothèse. Ainsi avons-nous construit une transformation dont le résultat est une dérivation de $B, B'; \Gamma \vdash_c J$. $\square$

Lemme B.1.20 (affaiblissement de l'environnement) Si $B; \Gamma, \Gamma'' \vdash_c J$ et $B; \Gamma, \Gamma' \vdash_\bullet \mathsf{ok}$ alors $B; \Gamma, \Gamma', \Gamma'' \vdash_c J$.

Démonstration. Avant toute chose, noter que quitte à effectuer une alpha-conversion, on peut supposer les domaines de Γ' et de Γ'' disjoints ; par le Lem. B.1.1 (les variables sont liées), ceci implique que les domaines de Γ, Γ' et de Γ'' sont disjoints.

Nous décrivons une transformation de la dérivation de $B; \Gamma, \Gamma'' \vdash_c J$ en une dérivation de $B; \Gamma, \Gamma', \Gamma'' \vdash_c J$, construite par induction sur son argument. La plupart des règles de typage autorisent un environnement arbitraire $\widehat{\Gamma}$ dans leur conclusion ; s'il a pu être instancié en Γ, Γ'', il peut aussi l'être en $\Gamma, \Gamma', \Gamma''$. La plupart des prémisses ont un environnement qui est un motif de la forme $\widehat{\Gamma}, \tilde{\Gamma}$, et n'ont que cette seule occurence de la métavariable $\widehat{\Gamma}$. Y instancier $\widehat{\Gamma}$ en $\Gamma, \Gamma', \Gamma''$ au lieu de Γ, Γ'' correspond à appeler récursivement la transformation (l'environnemnt passe de $\Gamma, (\Gamma'', \Gamma''')$ à $\Gamma, \Gamma', (\Gamma'', \Gamma''')$ où Γ''' est l'instanciation de Γ'''). Une prémisse de la forme $\tilde{x} :_{\tilde{c}} \tilde{T} \in \widehat{\Gamma}$ reste vraie si $\widehat{\Gamma}$ est instancié par un environnement plus grand. Il en est de même pour toute prémisse de la forme $\widehat{\Gamma} = \tilde{\Gamma}$ où $\tilde{\Gamma}$ est une concaténation de métavariables et d'environnements de longueur 1 — Γ' peut être inséré à n'importe quelle position. Bien sûr, une prémisse ne mentionnant pas $\widehat{\Gamma}$ peut être reprise telle qu'elle.

Les seules règles que la méthode générale ne traitent pas sont (envok.nil), (envok.a) et (envok.x). Pour les règles (envok.nil) et (envok.a), on a forcément $\Gamma = \Gamma'' = \mathsf{nil}$, $c = \bullet$ et $J = \mathsf{ok}$, donc la conclusion est l'hypothèse $B; \Gamma, \Gamma' \vdash_\bullet \mathsf{ok}$. Quant à la règle (envok.x), il convient de distinguer deux cas, suivant si Γ'' est vide ou non. Si $\Gamma'' = \mathsf{nil}$, comme dans le cas des règles (envok.nil) et (envok.a), la conclusion est l'hypothèse $B; \Gamma, \Gamma' \vdash_\bullet \mathsf{ok}$. Sinon, la conclusion étant de la forme $\widehat{\Gamma}, \tilde{\Gamma}$ où l'instanciation de la métavariable $\widehat{\Gamma}$ recouvre entièrement Γ, le cas général s'applique (on peut remplacer Γ par Γ, Γ' et obtenir encore une instance de la règle) — l'hypothèse $x \notin \mathbf{dom}\,\widehat{\Gamma}$ reste vraie parce que $x \in \mathbf{dom}\,\Gamma''$ si bien que $x \notin \mathbf{dom}\,\Gamma'$. Ainsi avons-nous construit une transformation dont le résultat est une dérivation de $B; \Gamma, \Gamma', \Gamma'' \vdash_c J$. $\square$

Lemme B.1.21 (affaiblissement du type d'une variable) Si $B; \Gamma, x :_{c_0} T_0, \Gamma' \vdash_c J$ et $B; \Gamma \vdash_{c_0} T'_0 <: T_0$ alors $B; \Gamma, x :_{c_0} T'_0, \Gamma' \vdash_c J$.

Remarque B.1.22 En vertu des règles (tsub.eq), (teq.conv) et (teq.refl), l'hypothèse de sous-typage peut être remplacée par $B; \Gamma \vdash_{c_0} T'_0 \equiv T_0$ ou $B; \Gamma \vdash_{c_0} T_0 \equiv T'_0$ ou $B; \Gamma \vdash_{c_0} T_0 \equiv T'_0$ ou $B; \Gamma \vdash_{c_0} T_0 \longrightarrow T'_0$.

Démonstration. Nous suivons le même principe que dans la démonstration du Lem. B.1.20 (affaiblissement de l'environnement), mais en remplaçant $\Gamma, x :_{c_0} T_0$ par $\Gamma, x :_{c_0} T'_0$ au lieu de remplacer Γ par Γ, Γ'. Ce remplacement affecte les prémisses de la forme $\widehat{x} :_{\widehat{c}} \widehat{T} \in \widehat{\Gamma}$ ou $\widehat{\Gamma} = \widehat{\Gamma}_0, \widehat{x} :_{\widehat{c}} \widehat{T}, \widehat{\Gamma}_1$ (si $\widehat{x}$ est instancié en x, $\widehat{T}$ est instancié en T_0 dans la dérivation originale et T'_0 dans le candidat dérivation obtenu), nous devons donc examiner instanciations problématiques des règles concernées. Trois règles sont concernées : (envok.c.x), (vis.env) et (et.x). Dans les deux premières règles, la métavariable $\widehat{T}$ n'a pas d'autre occurence, donc son instanciation est sans importance. Dans la règle (et.x), instancier $\widehat{T}$ en T'_0 au lieu de T_0 donne une nouvelle instance de la règle, mais change la conclusion en $B; \Gamma, x :_{c_0} T'_0, \Gamma' \vdash_c x :^P T'_0$. Par le Lem. B.1.6 (correction de l'environnement) et le Lem. B.1.20 (affaiblissement de l'environnement), on a $B; \Gamma, x :_{c_0} T'_0, \Gamma' \vdash_{c_0} T'_0 <: T_0$, d'où $B; \Gamma, x :_{c_0} T'_0, \Gamma' \vdash_c x :^P T_0$ par la règle (et.sub). □

Lemme B.1.23 (affaiblissement de la couleur) Si $B; \Gamma \vdash_c J$ et $B; \Gamma \vdash_{c'} c\,\mathsf{transparent}$ alors $B; \Gamma \vdash_{c'} J$.

Démonstration. Nous décrivons une transformation d'une dérivation de $B; \Gamma \vdash_c J$ et d'une dérivation de $B; \Gamma \vdash_{c'} c\,\mathsf{transparent}$ en une dérivation de $B; \Gamma \vdash_{c'} J$, construite par induction sur son premier argument. La plupart des règles de typage autorisent une couleur arbitraire $\widehat{c}$ dans leur conclusion ; si elle a pu être instanciée en c, elle peut aussi l'être en c'.

La plupart des prémisses ont une couleur qui est un motif de la forme $\widehat{c} \cup c_0$, et n'ont que cette seule occurence de la métavariable $\widehat{c}$. Suivant la prémisse, on a $c_0 = \bullet$ ou $c_0 = \{x\}$ avec $x \notin c$ parce que $x \notin \mathbf{dom}\,\Gamma$ alors que les variables présentes dans c appartiennent à $\mathbf{dom}\,\Gamma$ en vertu du Lem. B.1.1 (les variables sont liées). On obtient encore une instance de la prémisse en instanciant $\widehat{c}$ en c' au lieu de c, et cette prémisse est dérivable par le résultat d'un appel récursif à la transformation. Bien sûr, une prémisse ne mentionnant pas $\widehat{c}$ peut être reprise telle qu'elle. Nous examinons maintenant les règles non traitées par la méthode générale.

Règles (envok.*) : La conclusion n'autorise pas une couleur arbitraire. Mais dans ces cas, la conclusion est l'hypothèse $B; \Gamma \vdash_{c'} \mathsf{ok}$.

Règles (vis.env) et (vis.in) : La conclusion est $B; \Gamma \vdash_c \xi\,\mathsf{transparent}$. Par le Cor. B.1.18 (transitivité de la transparence), on a $B; \Gamma \vdash_{c'} \xi\,\mathsf{transparent}$.

Ainsi avons-nous construit une transformation dont le résultat est une dérivation de $B; \Gamma \vdash_{c'} J$. □

Corollaire B.1.24 (extraction d'un lexique) Si $B; \Gamma \vdash_c J$ et $a = E_0 :_{c_0} T_0 \in B$ et $B; \Gamma \vdash_c c_0\,\mathsf{transparent}$ alors $B; \Gamma \vdash_c E_0 :^P T_0$.

Démonstration. Posons $B = B_0, a = E_0 :_{c_0} T_0, B'_0$. D'après le Cor. B.1.5 (extraction directe d'un lexique), $B_0; \mathsf{nil} \vdash_\bullet E_0 :^P T_0$. D'après le Lem. B.1.2 (correction du lexique) et le Lem. B.1.19 (affaiblissement du lexique), $B; \mathsf{nil} \vdash_\bullet E_0 :^P T_0$. D'après le Lem. B.1.6 (correction de l'environnement) et le Lem. B.1.20 (affaiblissement de l'environnement), $B; \Gamma \vdash_\bullet E_0 :^P T_0$. D'après le Lem. B.1.23 (affaiblissement de la couleur), $B; \Gamma \vdash_c E_0 :^P T_0$. □

Corollaire B.1.25 (extraction d'un environnement) Si $B; \Gamma \vdash_c J$ et $x :_{c_0} T_0 \in \Gamma$ et $B; \Gamma \vdash_c c_0\,\mathsf{transparent}$ alors $B; \Gamma \vdash_c T_0 : *$.

Démonstration. Posons $\Gamma = \Gamma_0, x :_{c_0} T_0, \Gamma_0'$. D'après le Cor. B.1.14 (extraction directe d'un environnement), $B; \Gamma_0 \vdash_{c_0} T_0 : *$. D'après le Lem. B.1.6 (correction de l'environnement) et le Lem. B.1.20 (affaiblissement de l'environnement), $B; \Gamma \vdash_{c_0} T_0 : *$. D'après le Lem. B.1.12 (correction de la couleur et de l'environnement) et le Lem. B.1.23 (affaiblissement de la couleur), $B; \Gamma \vdash_c T_0 : *$. □

Corollaire B.1.26 (nommage d'une couleur) Si $B; \Gamma \vdash_c J$ et $x :_c T \in \Gamma$ alors $B; \Gamma \vdash_{\{x\}} J$.

Démonstration. Par le Lem. B.1.12 (correction de la couleur et de l'environnement), on a $B; \Gamma \vdash_c \mathsf{ok}$. Grâce au Lem. B.1.15 (transparence point par point) et en appliquant (vis.env) à chaque élément de c, on a $B; \Gamma \vdash_{\{x\}} c\,\mathsf{transparent}$. Par le Lem. B.1.23 (affaiblissement de la couleur), on a $B; \Gamma \vdash_{\{x\}} J$. □

B.1.4 Substitution

Theorème B.1.27 (préservation du typage par substitution) Si $B; \Gamma, x :_{c'} T, \Gamma' \vdash_c J$ et $B; \Gamma \vdash_{c'} E :^P T$ alors $B; \Gamma, \{x \leftarrow_{c'} E\}\Gamma' \vdash_{\{x \leftarrow_{c'} E\}c} \{x \leftarrow_{c'} E\}J$.

Démonstration. Posons $\sigma = \{x \leftarrow_{c'} E\}$.

Nous décrivons une transformation de la dérivation de $B; \Gamma, x :_{c'} T, \Gamma' \vdash_c J$ en une dérivation de $B; \Gamma, \sigma\Gamma' \vdash_{\sigma c} \sigma J$, construite par induction sur son argument. Considérons la règle à la racine de la dérivation. Nous allons chercher à construire une nouvelle instance de cette règle dont la conclusion soit $B; \Gamma, \sigma\Gamma' \vdash_{\sigma c} \sigma J$.

La plupart des règles de typage autorisent un lexique arbitraire $\widehat{B}$ environnement arbitraire $\widehat{\Gamma}$ dans leur conclusion ; s'il a pu être instancié en $\Gamma, x :_{c'} T, \Gamma'$, il peut aussi l'être en $\Gamma, \sigma\Gamma'$. Les seules exceptions sont (envok.nil), (envok.a) et (envok.x) ; les deux premières sont exclues car elles imposent un environnement vide. Nous instancions toutes les métavariables $\widehat{\aleph}$ autres que $\widehat{B}$ et $\widehat{\Gamma}$ en $\sigma\aleph$ où $\aleph$ est l'instanciation originale de $\widehat{\aleph}$. Nous obtenons ainsi une nouvelle instance de la règle ; nous allons maintenant nous pencher sur la dérivabilité des prémisses.

Dans la plupart des cas, nous pouvons appliquer l'hypothèse de récurrence sur la prémisse. Ceci concerne toutes les prémisses de la forme $\widehat{B}; \widehat{\Gamma}, \tilde{\Gamma} \vdash_{\tilde{c}} \tilde{J}$ dans lesquelles $\widehat{B}$ et $\widehat{\Gamma}$ ont seulement leur occurence évidente, ce qui est le cas de toutes les prémisses concernées qui sont des jugements colorés. Nous nous penchons maintenant sur les règles qui ont des prémisses qui ne sont pas des jugements colorés, et que la syntaxe autorise à contenir des variables. Suivant les cas, nous allons soit montrer que la transformation décrite jusqu'ici convient, soit proposer une autre transformation.

Règle (envok.c.a) : Il s'agit d'examiner les conditions de bord $a = E_0 :_{c_0} T_0 \in B$ et $c_0 \subseteq c'$. Par le Lem. B.1.1 (les variables sont liées), B n'a pas de variable libre, en particulier pas x, donc la première condition est inchangée. Comme $x \notin c_0$, $\sigma c_0 = c_0 \subseteq c' \setminus \{x\} \subseteq \sigma c'$.

Règle (envok.c.x) : La conclusion originale est $B; \Gamma \vdash_{c_0 \cup \{y\}} \mathsf{ok}$, et les prémisses originales sont $B; \Gamma \vdash_{c_0} \mathsf{ok}$ et $y :_{c_1} T_1 \in (\Gamma, x :_{c'} T, \Gamma')$. Si x vient avant y dans $(\Gamma, x :_{c'} T, \Gamma')$, la proposition courante convient. Si x vient après y, il convient de garder c_1 et T_1 intacts, et le contracté a bien ses prémisses dérivables. Enfin, si $x = y$, le but est $B; \Gamma, \sigma\Gamma' \vdash_{c_0 \cup c'} \mathsf{ok}$, qui est exactement la transformée de la première prémisse.

Règles (vis.env), (vis.in) : Une prémisse originale est $B; \Gamma, x :_{c'} T, \Gamma' \vdash_c \mathsf{ok}$, qui se transforme récursivement en $B; \Gamma, \sigma\Gamma' \vdash_{\sigma c} \mathsf{ok}$. Grâce au Lem. B.1.17 (dénotation de la transparence), on se ramène à démontrer que si $\mathbf{clos}^{\Gamma, x :_{c'} T, \Gamma'}(\{\xi\}) \subseteq \mathbf{clos}^{\Gamma, x :_{c'} T, \Gamma'}(c)$ alors $\mathbf{clos}^{\Gamma, \sigma\Gamma'}(\sigma\{\xi\}) \subseteq \mathbf{clos}^{\Gamma, \sigma\Gamma'}(\sigma c)$. Ceci résulte de considérations ensemblistes, en distinguant suivant si $\xi = x$ et suivant si ξ ou x appartient ou non à c.

Règle (et.x) **:** La conclusion originale est $\mathsf{B};\Gamma, x :_{c'} \mathsf{T}, \Gamma' \vdash_c y :^{\mathsf{P}} \mathsf{T}$, la prémisse originale est $\mathsf{B};\Gamma, x :_{c'} \mathsf{T}, \Gamma' \vdash_c y\,\mathsf{transparent}$, et on a la condition de bord $y :_{c_1} \mathsf{T}_1 \in \Gamma, x :_{c'} \mathsf{T}, \Gamma'$. Comme dans le cas de (envok.c.x), nous discriminons suivant la position relative de x et de y dans Γ, et seul le cas où $x = y$ est problématique. La transformation de la prémisse donne $\mathsf{B};\Gamma, \sigma\Gamma' \vdash_{\sigma c} c'\,\mathsf{transparent}$, ce qui permet d'affaiblir l'hypothèse $\mathsf{B};\Gamma \vdash_{c'} \mathsf{E} :^{\mathsf{P}} \mathsf{T}$ en $\mathsf{B};\Gamma, \sigma\Gamma' \vdash_{\sigma c} \mathsf{E} :^{\mathsf{P}} \mathsf{T}$ par le Lem. B.1.20 (affaiblissement de l'environnement) et le Lem. B.1.23 (affaiblissement de la couleur).

Il reste à traiter la règle (envok.x). Si $\Gamma' \neq \mathsf{nil}$, le raisonnement générique convient (la prémisse $y \notin \mathbf{dom}\,\widehat{\Gamma}$ reste vraie car le changement d'instanciation de $\widehat{\Gamma}$ ne fait que retirer x du domaine). Sinon, la conclusion désirée est $\mathsf{B};\Gamma \vdash_{\bullet} \mathsf{ok}$, que l'on obtient par le Lem. B.1.6 (correction de l'environnement). □

Comme annoncé à la section IV.3.3, montrons l'admissibilité des deux versions courantes du typage de la seconde projection :

$$\frac{\mathsf{B};\Gamma \vdash_c \mathsf{E} :^{\mathsf{P}} \Sigma x : \mathsf{T}_1.\,\mathsf{T}_2}{\mathsf{B};\Gamma \vdash_c \pi_2\,\mathsf{E} :^{\mathsf{P}} \{x{\leftarrow_c}\pi_1\,\mathsf{E}\}\mathsf{T}_2}\ \textbf{(et.proj.2s)} \qquad \frac{\mathsf{B};\Gamma \vdash_c \mathsf{E} :^{\mathsf{P}} \Sigma x : \mathsf{T}_1.\,\mathsf{T}_2}{\mathsf{B};\Gamma, x :_c \mathsf{S}(\pi_1\,\mathsf{E}) \vdash_{c\cup\{x\}} \pi_2\,\mathsf{E} :^{\mathsf{P}} \mathsf{T}_2}\ \textbf{(et.proj.2x)}$$

Lemme B.1.28 (seconde projection) Les règles (et.proj.2s) et (et.proj.2x) sont admissibles.

Démonstration. Supposons $\mathsf{B};\Gamma \vdash_c \mathsf{E} :^{\mathsf{P}} \Sigma x : \mathsf{T}_1.\,\mathsf{T}_2$. Par (et.proj.1), on a $\mathsf{B};\Gamma \vdash_c \pi_1\,\mathsf{E} :^{\mathsf{P}} \mathsf{T}_1$. Par (et.proj.2), on a $\mathsf{B};\Gamma \vdash_c \pi_2\,\mathsf{E} :^{\mathsf{P}} \{x{\leftarrow_c}\pi_1\,\mathsf{E}\}\mathsf{T}_2$.

De plus, par (et.sing) et (envok.x), on a $\mathsf{B};\Gamma, x :_c \mathsf{S}(\pi_1\,\mathsf{E}) \vdash_{\bullet} \mathsf{ok}$. Par le Lem. B.1.12 (correction de la couleur et de l'environnement), on a $\mathsf{B};\Gamma \vdash_c \mathsf{ok}$, d'où $\mathsf{B};\Gamma, x :_c \mathsf{S}(\pi_1\,\mathsf{E}) \vdash_c \mathsf{ok}$ par le Lem. B.1.20 (affaiblissement de l'environnement), puis $\mathsf{B};\Gamma, x :_c \mathsf{S}(\pi_1\,\mathsf{E}) \vdash_{c\cup\{x\}} \mathsf{ok}$ par (envok.c.x). Par le Cor. B.1.26 (nommage d'une couleur), on a $\mathsf{B};\Gamma, x :_c \mathsf{S}(\pi_1\,\mathsf{E}) \vdash_{\{x\}} \mathsf{E} :^{\mathsf{P}} \Sigma x : \mathsf{T}_1.\,\mathsf{T}_2$. La règle (et.x) donne $\mathsf{B};\Gamma, x :_c \mathsf{S}(\pi_1\,\mathsf{E}) \vdash_{\{x\}} x :^{\mathsf{P}} \mathsf{S}(\pi_1\,\mathsf{E})$. La règle (et.proj.2) donne alors $\mathsf{B};\Gamma, x :_c \mathsf{S}(\pi_1\,\mathsf{E}) \vdash_{\{x\}} \pi_2\,\mathsf{E} :^{\mathsf{P}} \mathsf{T}_2$. Par le Lem. B.1.23 (affaiblissement de la couleur), sachant que $\mathsf{B};\Gamma, x :_c \mathsf{S}(\pi_1\,\mathsf{E}) \vdash_{\{x\}} c \cup \{x\}\,\mathsf{transparent}$ par (vis.env) et (vis.union), on obtient $\mathsf{B};\Gamma, x :_c \mathsf{S}(\pi_1\,\mathsf{E}) \vdash_{c\cup\{x\}} \pi_2\,\mathsf{E} :^{\mathsf{P}} \mathsf{T}_2$. □

B.1.5 Validité

Lemme B.1.29 (validité d'un apax transparent) Si $\mathsf{B};\Gamma \vdash_c a\,\mathsf{transparent}$ alors il existe E_0, c_0 et T_0 tels que $a = \mathsf{E}_0 :_{c_0} \mathsf{T}_0 \in \mathsf{B}$ et $\mathsf{B};\Gamma \vdash_c c_0\,\mathsf{transparent}$.

Démonstration. Par récurrence sur la dérivation de $\mathsf{B};\Gamma \vdash_c a\,\mathsf{transparent}$. Nous utiliserons sans le détailler le Lem. B.1.15 (transparence point par point).

Règle (vis.in) **:** On a $a = \mathsf{E}_0 :_{c_0} \mathsf{T}_0 \in \mathsf{B}$. Par le Lem. B.1.10 (bonne formation des couleurs), $c_0 \subseteq c$. En appliquant (vis.in) pour chaque élément de c_0, on a $\mathsf{B};\Gamma \vdash_c c_0\,\mathsf{transparent}$.

Règle (vis.env) **:** Une prémisse est $\mathsf{B};\Gamma_1 \vdash_{c_1} a\,\mathsf{transparent}$. Par récurrence, on a $a = \mathsf{E}_0 :_{c_0} \mathsf{T}_0 \in \mathsf{B}$ et $\mathsf{B};\Gamma \vdash_{c_1} c_0\,\mathsf{transparent}$. En appliquant (vis.env) à chaque élément de c_0, on a $\mathsf{B};\Gamma \vdash_c c_0\,\mathsf{transparent}$.

Règle (vis.o) **:** Impossible.

Règle (vis.union) **:** Trivial par récurrence.

□

Définition B.1.30 (équivalence substitutive) On dit que $\aleph_1$ et $\aleph_2$ sont substitutivement équivalents ssi il existe une chaîne de substitutions par des expressions convertibles à partir d'une origine commune qui résulte en $\aleph_1$ et $\aleph_2$. On note $\aleph_1 \doteq \aleph_2$.

Autrement dit, l'équivalence substitutive est la plus petite relation d'équivalence telle que $\{x \leftarrow_c E\}\aleph \doteq \{x \leftarrow_c E'\}\aleph$ dès lors que $E \equiv E'$ (on suppose un lexique, un environnement et une couleur ambiants fixés).

Remarque B.1.31 (forme de types substitutivement équivalents) Si $T_1 \doteq T_2$ alors T_1 et T_2 ont la même structure à l'exclusion des expressions imbriquées.

Lemme B.1.32 (conversion d'une composante abstraite) Si $B; \Gamma \vdash_c A \longrightarrow A'$ et $B; \Gamma \vdash_c A \triangleright E : T$ alors il existe E' et T' tels que $B; \Gamma \vdash_c A' \triangleright E' : T'$ et $B; \Gamma \vdash_c E \longrightarrow E'$ et $T \doteq T'$.

Démonstration. Simultanément avec le lemme suivant.

On note que les règles (ac.*) sont structurelles : la seule connaissance de A entraîne l'unicité de E et T tels que $B; \Gamma \vdash_c A \triangleright E : T$ ainsi que la forme de la preuve de ce jugement en ce qui concerne les règles (ac.*). □

Lemme B.1.33 (validité) Si $B; \Gamma \vdash_c A \triangleright E : T$ alors $B; \Gamma \vdash_c E :^{P} T$.
Si $B; \Gamma \vdash_c T \longrightarrow T'$ ou $B; \Gamma \vdash_c T \equiv T'$ ou $B; \Gamma \vdash_c T <: T'$ alors $B; \Gamma \vdash_c T : *$ et $B; \Gamma \vdash_c T' : *$.
Si $B; \Gamma \vdash_c E :^{\Upsilon} T$ alors $B; \Gamma \vdash_c T : *$.

Démonstration. Simultanément avec le lemme suivant. □

Lemme B.1.34 (préservation de la sorte par conversion) Si $B; \Gamma \vdash_c T \longrightarrow T'$ ou $B; \Gamma \vdash_c T' \longrightarrow T$ ou $B; \Gamma \vdash_c T \equiv T'$ et $B; \Gamma \vdash_c T : K$ alors $B; \Gamma \vdash_c T' : K$.

Démonstration. Simultanément avec le lemme suivant, dont la démonstration ci-dessous prouve dans chaque cas $B; \Gamma \vdash_c T' : *$. Il reste à montrer que si $B; \Gamma \vdash_c T : \wr$ alors $B; \Gamma \vdash_c T' : \wr$. Ceci résulte de l'orthogonalité entre sous-sortage et sortage structurel des types, que l'on vérifie au cas par cas pour chaque règle (tconv.*) et chaque manière de typer contractant et résidu par les règles (tok.*). □

Lemme B.1.35 (préservation du typage par conversion) Si $B; \Gamma \vdash_c E \longrightarrow E'$ ou $B; \Gamma \vdash_c E' \longrightarrow E$ ou $B; \Gamma \vdash_c E \equiv E'$ alors il existe T tel que $B; \Gamma \vdash_c E :^{P} T$; de plus, pour tout T tel que $B; \Gamma \vdash_c E :^{P} T$, on a également $B; \Gamma \vdash_c E' :^{P} T$.

Démonstration. Commençons par remarquer que si $B; \Gamma \vdash_c E :^{P} T$ et $B; \Gamma \vdash_c E' :^{P} T'$, et si de plus $B; \Gamma \vdash_c E \longrightarrow E'$, alors $B; \Gamma \vdash_c E' :^{P} T$. En effet, on a $B; \Gamma \vdash_c S(E) \equiv S(E')$ par (tconv.cong.sing) et (teq.conv), d'où $B; \Gamma \vdash_c S(E') <: S(E)$ par (teq.sym) et (tsub.eq). Or $B; \Gamma \vdash_c E' :^{P} S(E')$ par (et.sing), donc $B; \Gamma \vdash_c E' :^{P} S(E)$ par (et.sub). Comme $B; \Gamma \vdash_c S(E) <: T$ par (tsub.sing), on a $B; \Gamma \vdash_c E' :^{P} T$ par (et.sub). Ce résultat est encore vrai si l'on remplace l'hypothèse $B; \Gamma \vdash_c E \longrightarrow E'$ par $B; \Gamma \vdash_c E' \longrightarrow E$ (il suffit d'omettre l'étape de symétrie). L'hypothèse peut également être $B; \Gamma \vdash_c E \equiv E'$ ou $B; \Gamma \vdash_c E' \equiv E$ (on a encore $B; \Gamma \vdash_c S(E) \equiv S(E')$, par induction sur la dérivation). Ceci prouve la deuxième partie du dernier lemme, qui constitue à proprement parler la préservation du typage par conversion.

Nous démontrons désormais la validité des différentes formes de jugements, par induction sur la dérivation. Noter que dans tous les cas $B; \Gamma \vdash_c \mathsf{ok}$ d'après le Lem. B.1.12 (correction de la couleur et de l'environnement). Nous ne mentionnerons pas systématiquement l'usage de (tok.sub) pour prouver $B; \Gamma \vdash_c T : *$ sachant $B; \Gamma \vdash_c T : \wr$.

Règle (ac.a) : La conclusion est $B; \Gamma \vdash_c a \triangleright E : T$, avec la prémisse $B; \Gamma \vdash_c c_0\,\mathsf{transparent}$ sous la condition $a = E :_{c_0} T \in B$. Par le Cor. B.1.24 (extraction d'un lexique), on a $B; \Gamma \vdash_c E :^{P} T$.

Règle (ac.app) : La conclusion est $B; \Gamma \vdash_c A_1\, E_0 \triangleright E_1\, E_0 : \{x \leftarrow_c E_0\}T_1$, avec les prémisses $B; \Gamma \vdash_c A_1 \triangleright E_1 : \Pi x : T_0.\,{}^{P}T_1$ et $B; \Gamma \vdash_c E_0 :^{P} T_0$. Par induction, on a $B; \Gamma \vdash_c E_1 :^{P} \Pi x : T_0.\,{}^{P}T_1$. Par (et.app), on a $B; \Gamma \vdash_c E_1\, E_0 :^{P} \{x \leftarrow_c E_0\}T_1$.

Règle (ac.proj.1) **:** Même principe que pour (ac.app), avec (et.proj.1) au lieu de (et.app).

Règle (ac.proj.2) **:** Même principe que pour (ac.app), avec (et.proj.2) au lieu de (et.app).

Règle (aconv.cong.app.arg) **:** La conclusion est $B; \Gamma \vdash_c A_1\, E_0 \longrightarrow A_1\, E_0'$, avec les prémisses $B; \Gamma \vdash_c E_0 \longrightarrow E_0'$, $B; \Gamma \vdash_c E_0 :^P T_0$ et $B; \Gamma \vdash_c A_1 \triangleright E_1 : \Pi x : T_0.\, ^P T_1$. La structure de la dérivation de $B; \Gamma \vdash_c A \triangleright E : T$ par (ac.app) impose $E = E_1\, E_0$ et $T = \{x \leftarrow_c E_0\} T_1$. Par induction, on a $B; \Gamma \vdash_c E_0' :^P T_0$. Par (ac.app), on a $B; \Gamma \vdash_c A_1\, E_0' \triangleright E_1\, E_0' : \{x \leftarrow_c E_0'\} T_1$. On a bien $B; \Gamma \vdash_c E_1\, E_0 \longrightarrow E_1\, E_0'$ par (econv.cong.app.arg).

Règle (aconv.cong.app.fun) **:** La conclusion est $B; \Gamma \vdash_c A_1\, E_0 \longrightarrow A_1'\, E_0$, avec les prémisses $B; \Gamma \vdash_c A_1 \longrightarrow A_1'$, $B; \Gamma \vdash_c E_0 :^P T_0$ et $B; \Gamma \vdash_c A_1 \triangleright E_1 : \Pi x : T_0.\, ^P T_1$. La structure de la dérivation de $B; \Gamma \vdash_c A \triangleright E : T$ par (ac.app) impose $E = E_1\, E_0$ et $T = \{x \leftarrow_c E_0\} T_1$. Par induction, on a $B; \Gamma \vdash_c A_1' \triangleright E_1' : \Pi x : T_0.\, ^P T_1'$ avec $B; \Gamma \vdash_c E_1 \longrightarrow E_1'$ et $T_0 \doteq T_0'$ et $T_1 \doteq T_1'$. Par (ac.app), on a $B; \Gamma \vdash_c A_1'\, E_0 \triangleright E_1'\, E_0 : \{x \leftarrow_c E_0\} T_1'$. On a bien $B; \Gamma \vdash_c E_1\, E_0 \longrightarrow E_1'\, E_0$ par (econv.cong.app.fun).

Règle (aconv.cong.proj) **:** La conclusion est $B; \Gamma \vdash_c \pi_i A_0 \longrightarrow \pi_i A_0'$, avec les prémisses $B; \Gamma \vdash_c A_0 \longrightarrow A_0'$ et $B; \Gamma \vdash_c A_0 \triangleright E_0 : \Sigma x : T_1.\, T_2$. La structure de la dérivation de $B; \Gamma \vdash_c A \triangleright E : T$ par (ac.proj.1) ou (ac.proj.2) impose $E = \pi_i\, E_0$ et, suivant la valeur de i, $T = T_1$ ou $T = \{x \leftarrow_c E_1\} T_2$. Par induction, on a $B; \Gamma \vdash_c A_0' \triangleright E_0' : \Sigma x : T_1'.\, T_2'$ avec $B; \Gamma \vdash_c E_0 \longrightarrow E_0'$ et $T_1 \doteq T_1'$ et $T_2 \doteq T_2'$. Par (ac.proj.1) ou (ac.proj.2), on a $B; \Gamma \vdash_c \pi_i A_0' \triangleright \pi_i E_0' : T'$. On a bien $B; \Gamma \vdash_c \pi_i E_0 \longrightarrow \pi_i E_0'$ par (econv.cong.proj), et $T \doteq T'$ que i vale 1 ou 2.

Règle (tconv.cong.abs) **:** La conclusion est $B; \Gamma \vdash_c (\!|A|\!) \longrightarrow (\!|A'|\!)$, avec les prémisses $B; \Gamma \vdash_c A \longrightarrow A'$ et $B; \Gamma \vdash_c A \triangleright E : \text{TYPE}^K$. Par (tok.abs), on a directement $B; \Gamma \vdash_c (\!|A|\!) : K$. Par induction, on a $B; \Gamma \vdash_c A' \triangleright E' : T'$ avec $T' \doteq T$. En vertu de la Rem. B.1.31 (forme de types substitutivement équivalents), $T' = \text{TYPE}^K$. Par (tok.abs), on a $B; \Gamma \vdash_c (\!|A'|\!) : K$.

Règle (tconv.cong.field) **:** La conclusion est $B; \Gamma \vdash_c \mathsf{Typ}\, E_0 \longrightarrow \mathsf{Typ}\, E_0'$, avec les prémisses $B; \Gamma \vdash_c E_0 \longrightarrow E_0'$ et $B; \Gamma \vdash_c E_0 :^P \text{TYPE}^*$. Par induction, on a $B; \Gamma \vdash_c E_0' :^P \text{TYPE}^*$. Par (tok.field), on a $B; \Gamma \vdash_c \mathsf{Typ}\, E_0 : *$ ainsi que $B; \Gamma \vdash_c \mathsf{Typ}\, E_0' : *$.

Règle (tconv.cong.fun.arg) **:** La conclusion est $B; \Gamma \vdash_c \Pi x : T_0.\, ^\gamma T_1 \longrightarrow \Pi x : T_0'.\, ^\gamma T_1$, avec les prémisses $B; \Gamma \vdash_c T_0 \longrightarrow T_0'$ et $B; \Gamma, x :_c T_0 \vdash_{c \cup \{x\}} T_1 : *$. Par le Lem. B.1.21 (affaiblissement du type d'une variable), on a $B; \Gamma, x :_c T_0' \vdash_{c \cup \{x\}} T_1 : *$. Par (tok.fun.*), on a $B; \Gamma \vdash_c \Pi x : T_0.\, T_1 : *$ ainsi que $B; \Gamma \vdash_c \Pi x : T_0'.\, T_1 : *$.

Règle (tconv.cong.fun.ret) **:** La conclusion est $B; \Gamma \vdash_c \Pi x : T_0.\, ^\gamma T_1 \longrightarrow \Pi x : T_0.\, ^\gamma T_1'$, avec les prémisses $B; \Gamma \vdash_c T_0 : *$ et $B; \Gamma, x :_c T_0 \vdash_{c \cup \{x\}} T_1 \longrightarrow T_1'$. Par induction, on a $B; \Gamma, x :_c T_0 \vdash_{c \cup \{x\}} T_1 : *$ et $B; \Gamma, x :_c T_0 \vdash_{c \cup \{x\}} T_1' : *$. Par (tok.fun.P) ou (tok.fun.I) suivant si γ vaut $\mathtt{P}$ ou $\mathtt{I}$, on a $B; \Gamma \vdash_c \Pi x : T_0.\, T_1 : *$ ainsi que $B; \Gamma \vdash_c \Pi x : T_0.\, T_1' : *$.

Règles (tconv.cong.pair.*) **:** Même principe que pour (tconv.cong.fun.*) avec (tok.pair) au lieu de (tok.fun.*).

Règle (tconv.cong.sing) **:** La conclusion est $B; \Gamma \vdash_c S(E) \longrightarrow S(E')$, avec la prémisse $B; \Gamma \vdash_c E \longrightarrow E'$. Par induction, il existe T tel que $B; \Gamma \vdash_c E :^P T$ et $B; \Gamma \vdash_c E' :^P T$. Par (tok.sing) et (tok.sub), on a $B; \Gamma \vdash_c S(E) : *$ ainsi que $B; \Gamma \vdash_c S(E') : *$.

Règle (tconv.field) **:** La conclusion est $B; \Gamma \vdash_c \mathsf{Typ}\, \langle T \rangle \longrightarrow T$, avec la prémisse $B; \Gamma \vdash_c T : *$. Par (et.type) et (tok.field), on a $B; \Gamma \vdash_c \mathsf{Typ}\, \langle T \rangle : *$.

Règle (tconv.unit) **:** La conclusion est $B; \Gamma \vdash_c S(()) \longrightarrow \text{UNIT}$. Par (et.base.unit), (tok.sing) et (tok.sub), on a $B; \Gamma \vdash_c S(()) : *$. Par (tok.base.unit) et (tok.sub), on a $B; \Gamma \vdash_c \text{UNIT} : *$.

Règle (econv.cong.app.*) **:** Même principe que (tconv.cong.fun.*), avec (et.app) au lieu de (tok.fun.*).

Règle (econv.cong.col.e) **:** La conclusion est $B; \Gamma \vdash_c [E_0]_{c_0}^{T_0} \longrightarrow [E_0']_{c_0}^{T_0}$. On a $B; \Gamma \vdash_c [E_0]_{c_0}^{T} :^P T_0$ à partir des prémisses par (et.col), ainsi que $B; \Gamma \vdash_c [E_0']_{c_0}^{T} :^P T_0$ en utilisant l'induction.

Règle (econv.cong.col.t) **:** La conclusion est $B; \Gamma \vdash_c [E_0]^{T_0}_{c_0} \longrightarrow [E_0]^{T'_0}_{c_0}$. On a $B; \Gamma \vdash_c [E_0]^{T}_{c_0} :^P T_0$ à partir des prémisses par (et.col). On a $B; \Gamma \vdash_{c\cap c'} T' : \wr$ par le Lem. B.1.34 (préservation de la sorte par conversion), d'où $B; \Gamma \vdash_c [E_0]^{T'_0}_{c_0} :^P T'_0$ par (et.col).

Règles (econv.cong.dynned.*) **:** Même principe que pour (econv.cong.col.*), avec (et.dynned) au lieu de (et.col).

Règle (econv.cong.field) **:** La conclusion est $B; \Gamma \vdash_c \langle T_0 \rangle \longrightarrow \langle T'_0 \rangle$, avec la prémisse $B; \Gamma \vdash_c T_0 \longrightarrow T'_0$. Par induction et (et.type), on a $B; \Gamma \vdash_c \langle T_0 \rangle :^P \text{TYPE}^*$ ainsi que $B; \Gamma \vdash_c \langle T'_0 \rangle :^P \text{TYPE}^*$.

Règle (econv.cong.fun.arg) **:** Même principe que (tconv.cong.fun.arg), avec (et.fun) au lieu de (tok.fun.*).

Règle (econv.cong.fun.body) **:** Posons $\sigma = \{y \leftarrow_{c\cup\{x\}} E_0\}$ et $\sigma' = \{y \leftarrow_{c\cup\{x\}} E'_0\}$. La conclusion est $B; \Gamma \vdash_c (\lambda x : T_0.\ \sigma E_1) \longrightarrow (\lambda x : T_0.\ \sigma' E_1)$, avec les prémisses $B; \Gamma, x :_c T_0 \vdash_{c\cup\{x\}} E_0 \longrightarrow E'_0$ et $B; \Gamma, x :_c T_0, y :_{c\cup\{x\}} S(E_0) \vdash_{c\cup\{y\}\cup\{x\}} E_1 :^\Upsilon T_1$. Par récurrence, on a $B; \Gamma, x :_c T_0 \vdash_{c\cup\{x\}} E_0 :^P S(E_0)$ et $B; \Gamma, x :_c T_0 \vdash_{c\cup\{x\}} E'_0 :^P S(E_0)$. Par le Th. B.1.27 (préservation du typage par substitution) et (et.fun), on a $B; \Gamma \vdash_c (\lambda x : T_0.\ \sigma E_1) :^P \Pi x : T_0.\ {}^\Upsilon \sigma T_1$, et de même avec E'_0 à la place de E_0.

Règle (econv.cong.fun.seal) **:** La conclusion est $B; \Gamma \vdash_c (\lambda x : T_0.\ E_1 \,!!_{c'}\, T_1) \longrightarrow (\lambda x : T_0.\ E_1 \,!!_{c'}\, T'_1)$. On a $B; \Gamma \vdash_{c'} (\lambda x : T_0.\ E_1 \,!!_{c'}\, T_1) :^P \Pi x : T_0.\ {}^I T_1$ par le Lem. B.1.23 (affaiblissement de la couleur), (et.seal) et (et.fun). Puisque T_1 se convertit en T'_1 sous $(c \cap c') \cup \{x\}$ (donc sous toute couleur plus grande par le Lem. B.1.23 (affaiblissement de la couleur)), le résidu également ce type.

Règle (econv.cong.pair.1) **:** La conclusion est $B; \Gamma \vdash_c (E_1, E_2) \longrightarrow (E'_1, E_2)$, avec les prémisses $B; \Gamma \vdash_c E_1 \longrightarrow E'_1$ et $B; \Gamma \vdash_c E_2 :^P T_2$. Par induction, il existe T_1 tel que $B; \Gamma \vdash_c E_1 :^P T_1$ et $B; \Gamma \vdash_c E'_1 :^P T_1$. Par (et.pair), on a $B; \Gamma \vdash_c (E_1, E_2) :^P T_1 * T_2$ et $B; \Gamma \vdash_c (E'_1, E_2) :^P T_1 * T_2$.

Règle (econv.cong.pair.2) **:** Même principe que pour (econv.cong.pair.1).

Règle (econv.cong.proj) **:** La conclusion est $B; \Gamma \vdash_c \pi_i E_0 \longrightarrow \pi_i E'_0$, avec les prémisses $B; \Gamma \vdash_c E_0 \longrightarrow E'_0$ et $B; \Gamma \vdash_c E_0 :^P \Sigma x : T_1.\ T_2$. Par induction, on a $B; \Gamma \vdash_c E'_0 :^P \Sigma x : T_1.\ T_2$. Par (et.proj.1) ou (et.proj.2s) (suivant la valeur de i), on a un typage de E_0 ainsi que de E'_0.

Règle (econv.app) **:** La conclusion est $B; \Gamma \vdash_c (\lambda x : T_0.\ E_1)\, E_0 \longrightarrow \{x \leftarrow_c E_0\} E_1$, avec les prémisses $B; \Gamma, x :_c T_0 \vdash_{c\cup\{x\}} E_1 :^P T_1$ et $B; \Gamma \vdash_c E_0 :^P T_0$. Par (et.fun) et (et.app), on a $B; \Gamma \vdash_c (\lambda x : T_0.\ E_1)\, E_0 :^P \{x \leftarrow_c E_0\} T_1$. Par le Th. B.1.27 (préservation du typage par substitution), on a $B; \Gamma \vdash_c \{x \leftarrow_c E_0\} E_1 :^P \{x \leftarrow_c E_0\} T_1$.

Règles (econv.col.base.*) **:** La conclusion est $B; \Gamma \vdash_c [V]^T_{c'} \longrightarrow V$ où V est une valeur de base et T est le type de base correspondant. Notons que $B; \Gamma \vdash_{c\cap c'} \mathsf{ok}$. En appliquant la règles (tok.base.*) et (et.base.*) appropriées ainsi que (et.col), on a $B; \Gamma \vdash_{c'} V :^P T$ d'où $B; \Gamma \vdash_c [V]^T_{c'} :^P T$ et $B; \Gamma \vdash_c V :^P T$.

Règle (econv.col.dynned) **:** La conclusion est $B; \Gamma \vdash_c [\mathsf{dynned}\, E_0\, \mathsf{at}\, T_0]^{\text{DYN}}_{c'} \longrightarrow \mathsf{dynned}\, E_0\, \mathsf{at}\, T_0$. On a $B; \Gamma \vdash_{c'} \mathsf{dynned}\, E_0\, \mathsf{at}\, T_0 :^P \text{DYN}$ par (et.dynned), d'où $B; \Gamma \vdash_c [\mathsf{dynned}\, E_0\, \mathsf{at}\, T_0]^{\text{DYN}}_{c'} :^P \text{DYN}$ par (et.col), ainsi que $B; \Gamma \vdash_c \mathsf{dynned}\, E_0\, \mathsf{at}\, T_0 :^P \text{DYN}$ par (et.dynned).

Règle (econv.col.fun.I) **:** La conclusion est $B; \Gamma \vdash_c [\lambda x : T_2.\ E_1]^{\Pi x : T_0.\ {}^I T_1}_{c'} \longrightarrow \lambda x : T_0.\ E_1 \,!!_{c'\cup\{x\}}\, T_1$, avec les prémisses $B; \Gamma \vdash_{c'} T_0 <: T_2$, $B; \Gamma, x :_{c'} T_2 \vdash_{c'\cup\{x\}} E_1 :^I T_1$, $B; \Gamma, x :_{c\cap c'} T_0 \vdash_{(c\cap c')\cup\{x\}} T_1 : \wr$ et $B; \Gamma \vdash_c \mathsf{ok}$. Par (et.fun), on a $B; \Gamma \vdash_{c'} (\lambda x : T_2.\ E_1) :^P \Pi x : T_2.\ {}^I T_1$. Par (tok.fun.I) et (tsub.cong.fun), on a $B; \Gamma \vdash_{c'} \Pi x : T_2.\ {}^I T_1 <: \Pi x : T_0.\ {}^I T_1$, d'où $B; \Gamma \vdash_{c'} (\lambda x : T_2.\ E_1) :^P \Pi x : T_0.\ {}^I T_1$ par (et.sub), ce qui donne le typage du contractant. D'autre part, par le Lem. B.1.21 (affaiblissement du type d'une variable), on a $B; \Gamma, x :_{c'} T_0 \vdash_{c'\cup\{x\}} E_1 :^I T_1$, d'où le typage du résidu par (et.seal).

Règle (econv.col.fun.P) **:** La conclusion est $B; \Gamma \vdash_c [\lambda x : T_2.\ E_1]^{\Pi x : T_0.\ {}^P T_1}_{c'} \longrightarrow \lambda x : T_0.\ [E_1]^{T_1}_{c'\cup\{x\}}$, avec les prémisses $B; \Gamma \vdash_{c'} T_0 <: T_2$, $B; \Gamma, x :_{c'} T_2 \vdash_{c'\cup\{x\}} E_1 :^P T_1$, $B; \Gamma, x :_{c\cap c'} T_0 \vdash_{(c\cap c')\cup\{x\}} T_1 : \wr$ et

$B;\Gamma \vdash_c \mathsf{ok}$. Par (et.fun), on a $B;\Gamma \vdash_{c'} (\lambda x:T_2.\ E_1) :^P \Pi x : T_2.\,^P T_1$. Par (tok.fun.P) et (tsub.cong.fun), on a $B;\Gamma \vdash_{c'} \Pi x : T_2.\,^P T_1 <: \Pi x : T_0.\,^P T_1$, d'où $B;\Gamma \vdash_{c'} (\lambda x:T_2.\ E_1) :^P \Pi x : T_0.\,^P T_1$ par (et.sub). Enfin (et.col) donne le typage du contractant. D'autre part, par le Lem. B.1.21 (affaiblissement du type d'une variable), on a $B;\Gamma, x :_{c'} T_0 \vdash_{c' \cup \{x\}} E_1 :^P T_1$, d'où le typage du résidu par (et.col).

Règle (econv.col.merge) : La conclusion est $B;\Gamma \vdash_c [[E_0]^{T_2}_{c_2}]^{T_1}_{c_1} \longrightarrow [E_0]^{T_1}_{c_1 \cup c_2}$. On a $B;\Gamma \vdash_{c_1} [E_0]^{T_2}_{c_2} :^P T_2$ par (et.col), d'où $B;\Gamma \vdash_{c_1} [E_0]^{T_2}_{c_2} :^P T_1$ par (et.sub), puis $B;\Gamma \vdash_c [[E_0]^{T_2}_{c_2}]^{T_1}_{c_1} :^P T_1$ par (et.col). D'autre part, on a $B;\Gamma \vdash_{c_1 \cup c_2} \mathsf{ok}$ par le Lem. B.1.12 (correction de la couleur et de l'environnement), d'où $B;\Gamma \vdash_{c_1 \cup c_2} E_0 :^P T_2$ ainsi que $B;\Gamma \vdash_{c_1 \cup c_2} T_2 <: T_1$ par le Lem. B.1.23 (affaiblissement de la couleur), puis finalement $B;\Gamma \vdash_c [E_0]^{T_1}_{c_1 \cup c_2} :^P T_1$.

Règle (econv.col.pair) : La conclusion est $B;\Gamma \vdash_c [(E_1, E_2)]^{\Sigma x : T_1.\ T_2}_{c'} \longrightarrow ([E_1]^{T_1}_{c'}, [E_2]^{\{x \leftarrow_{c'} [E_1]^{T_1}_{c'}\} T_2}_{c'})$. On a $B;\Gamma \vdash_c [(E_1, E_2)]^{\Sigma x : T_1.\ T_2}_{c'} :^P \Sigma x : T_1. T_2$ par (et.col).D'autre part, le résidu se type facilement par $T_1 * \{x \leftarrow_{c'} [E_1]^{T_1}_{c'}\} T_2$ avec (et.col) et (et.pair).

Règle (econv.col.sing) : La conclusion est $B;\Gamma \vdash_c [E_0]^{S(E_1)}_{c'} \longrightarrow E_1$, avec les prémisses $B;\Gamma \vdash_{c'} E_0 :^P S(E_1)$, $B;\Gamma \vdash_{c \cap c'} E_1 :^P T_1$ et $B;\Gamma \vdash_c \mathsf{ok}$. On a $B;\Gamma \vdash_c [E_0]^{S(E_1)}_{c'} :^P S(E_1)$ par (et.sing) et (et.col). D'autre part $B;\Gamma \vdash_c E_1 :^P T_1$ par le Lem. B.1.23 (affaiblissement de la couleur).

Règle (econv.proj) : Trivial par (et.pair) et (et.proj.1) ou (et.proj.2s).

Règle (econv.eta.field) : Trivial par (et.type) et (tok.field).

Règle (econv.eta.fun) : La conclusion est $B;\Gamma \vdash_c E_1 \longrightarrow \lambda x : T_0.\ E_1\, x$, avec la prémisse $B;\Gamma \vdash_c E_1 :^P \Pi x : T_0.\,^\gamma T_1$. Par récurrence et inversion de (tok.fun.*), on a $B;\Gamma, x :_c T_0 \vdash_{c \cup \{x\}} T_1 : *$. Par le Lem. B.1.12 (correction de la couleur et de l'environnement), le Lem. B.1.20 (affaiblissement de l'environnement) et le Lem. B.1.23 (affaiblissement de la couleur), on a $B;\Gamma, x :_c T_0 \vdash_{c \cup \{x\}} E_1 :^P \Pi y : T_0.\,^\gamma \{x \leftarrow_{\{y\}} y\} T_1$. Par (et.x), (et.app) et (et.fun), on a $B;\Gamma \vdash_c (\lambda x : T_0.\ E\, x) :^P \Pi x : T_0.\,^\gamma \{x \leftarrow_{c \cup \{x\}} x\} T_1$.

Règle (econv.eta.pair) : Trivial par (et.proj.1), (et.proj.2s) et (et.pair).

Règles (eeq.*) et (teq.*) : Trivial (par récurrence sauf pour la réflexivité).

Règle (tsub.cong.fun) : La conclusion est $B;\Gamma \vdash_c \Pi x : T_0.\,^\gamma T_1 <: \Pi x : T_0'.\,^{\gamma'} T_1'$, et les prémisses sont $B;\Gamma \vdash_c T_0' <: T_0$ et $B;\Gamma, x :_c T_0 \vdash_{c \cup \{x\}} T_1 <: T_1'$. Par récurrence, on obtient $B;\Gamma \vdash_c T_0 : *$, $B;\Gamma \vdash_c T_0' : *$, $B;\Gamma, x :_c T_0 \vdash_{c \cup \{x\}} T_1 : *$ et $B;\Gamma, x :_c T_0 \vdash_{c \cup \{x\}} T_1' : *$. La règle (tok.fun.P) ou (tok.fun.I) permet de conclure que $B;\Gamma \vdash_c T : *$, mais elle demande $B;\Gamma, x :_c T_0' \vdash_{c \cup \{x\}} T_1' : *$ pour produire $B;\Gamma \vdash_c T' : *$.

Règle (tsub.cong.pair) : Appliquer un raisonnement similaire à (tsub.cong.fun), avec la règle (tok.pair) (noter que la variance envers le type T_0 n'importe pas pourvu qu'elle soit cohérente entre les deux prémisses).

Règle (tsub.cong.type) : Appliquer (tok.type).

Règle (tsub.eq) : Trivial par récurrence.

Règle (tsub.sing) : La conclusion est $B;\Gamma \vdash_c S(E) <: T$ et la prémisse est $B;\Gamma \vdash_c E :^P T$. On a $B;\Gamma \vdash_c S(E) : \wr$ par (tok.sing), et $B;\Gamma \vdash_c T : *$ par récurrence.

Règle (tsub.trans) : Trivial par récurrence.

Règle (et.app) : On a $E = E_1\, E_0$, $T = \{x \leftarrow_c E_0\} T_1$, et les prémisses sont $B;\Gamma \vdash_c E_1 :^{\gamma_1} \Pi x : T_0.\,^{\gamma_2} T_1$ et $B;\Gamma \vdash_c E_0 :^P T_0$. Par récurrence, $B;\Gamma \vdash_c \Pi x : T_0.\,^{\gamma_2} T_1 : *$; ce jugement provient forcément de la règle (tok.fun.P) ou (tok.fun.I), dont une prémisse est $B;\Gamma, x :_c T_0 \vdash_{c \cup \{x\}} T_1 : *$. Par le Th. B.1.27 (préservation du typage par substitution), on obtient $B;\Gamma \vdash_c \{x \leftarrow_c E_0\} T_1 : *$.

Règles (et.base.*) : Appliquer la règle (tok.base.*) correspondante.

Règle **(et.col)** **:** Une prémisse est $B; \Gamma \vdash_{c \cap c'} T : \wr$. Par le Lem. B.1.23 (affaiblissement de la couleur), on a $B; \Gamma \vdash_c T : \wr$.

Règles **(et.dyn), (et.dynned)** **:** Appliquer (tok.base.dyn).

Règle **(et.fun)** **:** Utiliser la récurrence et appliquer (tok.fun.*).

Règle **(et.let)** **:** Trivial.

Règle **(et.pair)** **:** Utiliser la récurrence et appliquer (tok.pair).

Règle **(et.proj.1)** **:** Utiliser la récurrence et inverser (tok.pair).

Règle **(et.proj.2)** **:** La conclusion est $B; \Gamma \vdash_c \pi_2 E_0 :^{P} \{x \leftarrow_c E_1\} T_2$, avec les prémisses $B; \Gamma \vdash_c E_0 :^{P} \Sigma x : T_1 . T_2$ et $B; \Gamma \vdash_c E_1 :^{P} S(\pi_1 E_0)$. Par récurrence et inversion de (tok.pair), on a $B; \Gamma, x :_c T_1 \vdash_{c \cup \{x\}} T_2 : *$. On a également $B; \Gamma \vdash_c \pi_1 E_0 :^{P} T_1$ par récurrence. Par le Th. B.1.27 (préservation du typage par substitution), on a $B; \Gamma \vdash_c \{x \leftarrow_c E_1\} T_2 : *$.

Règle **(et.seal)** **:** Trivial.

Règle **(et.sing)** **:** Appliquer la règle (tok.sing).

Règle **(et.sub)** **:** Trivial par récurrence sur l'hypothèse $B; \Gamma \vdash_c T' <: T$.

Règle **(et.type)** **:** Appliquer (tok.type).

Règle **(et.undyn)** **:** Trivial par récurrence.

Règle **(et.x)** **:** On a $E = x$ avec $x :_{c_0} T_0 \in \Gamma$ et la prémisse $B; \Gamma \vdash_c x \,\mathsf{transparent}$. Par le Lem. B.1.15 (transparence point par point) et (vis.env) pour chaque élément de c_0, on a $B; \Gamma \vdash_c c_0 \,\mathsf{transparent}$. Par le Cor. B.1.25 (extraction d'un environnement), on a $B; \Gamma \vdash_c T_0 : *$.

□

B.2 Conversion

Définition B.2.1 (chaîne de conversions) On dit que E est convertible vers E' (sous B, Γ et c), et on note $B; \Gamma \vdash_c E \longrightarrow^* E'$, si et seulement si soit il existe T tel que $B; \Gamma \vdash_c E :^{P} T$, soit il existe $E_0, \ldots, E_k$ tels que $E = E_0$, $E' = E_k$ et pour tout i, $B; \Gamma \vdash_c E_i \longrightarrow E_{i+1}$.
On définit de même la relation $B; \Gamma \vdash_c T \longrightarrow^* T'$, la chaîne de longueur 0 demandant l'hypothèse $B; \Gamma \vdash_c T : *$.
On définit de même la relation $B; \Gamma \vdash_c A \longrightarrow^* A'$, la chaîne de longueur 0 demandant l'hypothèse $B; \Gamma \vdash_c A \triangleright E : T$.

Si $B; \Gamma \vdash_c E \longrightarrow^* E'$, on note également $B; \Gamma \vdash_c E \longrightarrow {}^{k} E'$ si la chaîne est de longueur k ; on note également $B; \Gamma \vdash_c E \longrightarrow^? E'$ si la chaîne est de longueur au plus 1 (c'est-à-dire $E = E'$ ou $B; \Gamma \vdash_c E \longrightarrow E'$), et $B; \Gamma \vdash_c E \longrightarrow^+ E'$ si la chaîne est de longueur au moins 1. On utilise des notations similaires pour préciser la longueur d'une chaîne de conversions de types.

Lemme B.2.2 (validité d'une chaîne de conversions) Si $B; \Gamma \vdash_c E \longrightarrow^* E'$ alors $B; \Gamma \vdash_c E' :^{P} S(E)$ et $B; \Gamma \vdash_c E :^{P} S(E')$.

Démonstration. Par récurrence sur la longueur de la chaîne de conversions, par une application immédiate du Lem. B.1.33 (validité) et grâce à la règle (et.sing). □

B.2.1 Confluence

Lemme B.2.3 (conservation de l'apax sous-jacent) Si $B; \Gamma \vdash_c A \longrightarrow A'$ alors $\mathbf{underl}(A) = \mathbf{underl}(A')$.

Démonstration. Récurrence immédiate au vu des règles (aconv.*). ☐

Lemme B.2.4 (impureté évidente) Si $B; \Gamma \vdash_c E :^{P} T$ alors E n'est pas de l'une des formes $\mathsf{let}\, x = E_1 \,\mathsf{in}\, E_0 : T_0$, $E_0 \,!!_{c_0}\, T_0$ ou $\mathsf{undyn}\, E_0 \,\mathsf{at}\, T_0$.

Démonstration. Par induction sur la démonstration. Si la dernière règle utilisée est (et.sub) avec la prémisse $B; \Gamma \vdash_c E :^{\gamma'} T'$, on a $\gamma' \sqsubseteq P$ donc $\gamma' = P$ et la récurrence donne le résultat souhaité. Si la dernière règle est (et.sing), la prémisse impose également la pureté de E, et l'on conclut pareillement par récurrence. Sinon la dernière règle est structurelle, et l'on vérifie que les constructions énumérées dans l'énoncé requièrent l'annotation d'effets $\mathtt{I}$. ☐

Lemme B.2.5 (conversion sous une substitution) Supposons $B; \Gamma_0 \vdash_{c_0} E_0 \longrightarrow E_0'$.
Si $B; \Gamma_0, x :_{c_0} S(E_0), \Gamma_1 \vdash_c T : K$ alors $B; \Gamma_0, \{x \leftarrow_{c_0} E_0\}\Gamma_1 \vdash_{\{x \leftarrow_{c_0} E_0\}c} \{x \leftarrow_{c_0} E_0\}T \longrightarrow^* \{x \leftarrow_{c_0} E_0'\}T$.
Si $B; \Gamma_0, x :_{c_0} S(E_0), \Gamma_1 \vdash_c E :^{P} T$ alors $B; \Gamma_0, \{x \leftarrow_{c_0} E_0\}\Gamma_1 \vdash_{\{x \leftarrow_{c_0} E_0\}c} \{x \leftarrow_{c_0} E_0\}E \longrightarrow^* \{x \leftarrow_{c_0} E_0'\}E$.
Si $B; \Gamma_0, x :_{c_0} S(E_0), \Gamma_1 \vdash_c A \triangleright E : T$ alors $B; \Gamma_0, \{x \leftarrow_{c_0} E_0\}\Gamma_1 \vdash_{\{x \leftarrow_{c_0} E_0\}c} \{x \leftarrow_{c_0} E_0\}A \longrightarrow^* \{x \leftarrow_{c_0} E_0'\}A$.

Démonstration. Par induction sur la démonstration de la correction de T ou E ou A , en discriminant sur la règle (tok.*) ou (et.*) ou (ac.*) utilisée.

Posons $\sigma = \{x \leftarrow_{c_0} E_0\}$ et $\sigma' = \{x \leftarrow_{c_0} E_0'\}$. Posons également $\Gamma = \Gamma_0, \sigma\Gamma_1$ et $\Gamma' = \Gamma_0, \sigma'\Gamma_1$, ainsi que $c' = \sigma c = \sigma' c$. Notons que d'après le Lem. B.1.35 (préservation du typage par conversion) et (et.sing), on a $B; \Gamma_0 \vdash_{c_0} E_0 :^{P} S(E_0)$ et $B; \Gamma_0 \vdash_{c_0} E_0' :^{P} S(E_0)$. De plus, d'après le Th. B.1.27 (préservation du typage par substitution), on a $B; \Gamma \vdash_{\sigma c} \sigma T : K$ ou $B; \Gamma \vdash_c \sigma A \triangleright \sigma E : \sigma T$ ou $B; \Gamma \vdash_{\sigma c} \sigma E :^{P} \sigma T$.

Nous utiliserons librement le Lem. B.1.35 (préservation du typage par conversion) ainsi que le Lem. B.1.34 (préservation de la sorte par conversion). Nous utiliserons également librement le Lem. B.1.6 (correction de l'environnement).

Règle (tok.abs) : On a $T = (\!|A|\!)$ avec la prémisse $B; \Gamma_0, x :_{c_0} S(E_0), \Gamma_1 \vdash_c A \triangleright E : \text{TYPE}^K$. Par récurrence, on a $B; \Gamma \vdash_c \sigma A \longrightarrow^* \sigma' A$, d'où $B; \Gamma \vdash_c \sigma T \longrightarrow^* \sigma' T$ par (tconv.cong.abs).

Règles (tok.base.*), (tok.type) : Puisque T ne contient pas de variable libre, on a $\sigma T = \sigma' T$, donc $B; \Gamma \vdash_{c'} \sigma T \longrightarrow^* \sigma' T$ puisque l'on a vu que T est correct.

Règle (tok.field) : On a $T = \mathsf{Typ}\, E_1$. Par récurrence, $B; \Gamma \vdash_c \sigma E_1 \longrightarrow^* \sigma' E_1$, d'où $B; \Gamma \vdash_{c'} \sigma T \longrightarrow^* \sigma' T$ par applications répétées de (tconv.cong.field).

Règle (tok.fun.P) : On a $T = \Pi y : T_2.{}^{\gamma} T_1$. Par récurrence, on a $B; \Gamma, y :_{c'} T_2 \vdash_{c' \cup \{y\}} \sigma T_1 \longrightarrow^* \sigma T_1'$ et $B; \Gamma \vdash_{c'} \sigma T_2 \longrightarrow^* \sigma T_2'$. Grâce au Lem. B.1.21 (affaiblissement du type d'une variable) (appliqué autant de fois que la longueur de la chaîne de conversions $\sigma T_2 \longrightarrow^* \sigma T_2'$ le demande), on a $B; \Gamma, y :_{c'} T_2' \vdash_{c' \cup \{y\}} \sigma T_1 \longrightarrow^* \sigma T_1'$. On a finalement $B; \Gamma \vdash_{c'} \sigma T \longrightarrow^* \sigma T'$ par applications répétées de (tconv.cong.fun.arg) puis (tconv.cong.fun.ret).

Règle (tok.fun.I) : Même principe que pour (tok.fun.I).

Règle (tok.pair) : Même principe que pour (tok.fun.P), en utilisant les règles (tconv.cong.pair.*) au lieu de (tconv.cong.fun.*).

Règle (tok.sing) : Même principe que pour (tok.field), en utilisant la règle (tconv.cong.sing) au lieu de (tconv.cong.field).

Règle (tok.sub) : Trivial.

Règles (et.base.*) : Même principe que pour (tok.base.*).

Règle (et.app) : On a $B;\Gamma \vdash_{c'} (\sigma E_1)\,(\sigma E_2) :^{P} \{y{\leftarrow_{c'}}\sigma E_2\}\sigma T_1$ avec les prémisses $B;\Gamma \vdash_{c'} \sigma E_1 :^{P} \Pi y : \sigma T_2.\ ^{P}\sigma T_1$ et $B;\Gamma \vdash_{c'} \sigma E_2 :^{P} \sigma T_2$. Par récurrence, on a $B;\Gamma \vdash_{c'} \sigma E_1 \longrightarrow^{*} \sigma' E_1$ et $B;\Gamma \vdash_{c'} \sigma E_2 \longrightarrow^{*} \sigma' E_2$. On a finalement $B;\Gamma \vdash_{c'} (\sigma E_1)\,(\sigma E_2) \longrightarrow^{*} (\sigma' E_1)\,(\sigma' E_2)$ par applications répétées de (econv.cong.app.arg) et (econv.cong.app.fun) (dans un ordre indifférent).

Règle (et.col) : Même principe que pour (et.app), en notant que la conversion de E_1 ou T_1 n'affecte pas la correction de la prémisse $B;\Gamma \vdash_{c''} E_1 :^{P} T_1$. Les règles contextuelles utilisées sont (econv.cong.col.e) et (econv.cong.col.t).

Règles (et.dyn), (et.let), (et.seal), (et.undyn) : Impossible, l'expression ne peut pas être pure.

Règle (et.dynned) : Même principe que pour (et.app), avec les règles contextuelles (econv.cong.dynned.*).

Règle (et.fun) : Voir la Conj. B.2.6 (conversion sous une substitution dans le corps d'une fonction).

Règle (et.pair) : Même principe que pour (et.app), avec les règles contextuelles (econv.cong.pair.*).

Règles (et.proj.*) : Même principe que pour (tok.field), avec la règle contextuelle (econv.cong.proj).

Règle (et.type) : Même principe que pour (tok.field), avec la règle contextuelle (econv.cong.field).

Règle (et.x) : Si $E = x$, on a $B;\Gamma \vdash_{c'} E_0 \longrightarrow E_0'$ par le Lem. B.1.20 (affaiblissement de l'environnement) et le Lem. B.1.23 (affaiblissement de la couleur). Si E est une variable autre que x, on a $\sigma E = \sigma E'$, et le principe est le même que dans le cas des règles (et.base.*).

Règles (et.seal), (et.sub) : Trivial.

Règle (ac.a) : Même principe que pour (tok.base.*).

Règle (ac.app) : Même principe que pour (et.app), avec les règles contextuelles (aconv.cong.app.*).

Règles (ac.proj.*) : Même principe que pour (et.proj.*), avec la règle contextuelle (aconv.cong.proj).

□

Conjecture B.2.6 (conversion sous une substitution dans le corps d'une fonction) Supposons $B;\Gamma_0 \vdash_{c_0} E_0 \longrightarrow E_0'$.
Si $B;\Gamma_0, x :_{c_0} S(E_0), \Gamma_1 \vdash_c (\lambda y : T_2.\ E_1) :^{P} \Pi y : T_2.\ T_3$ alors
$B;\Gamma_0, \{x{\leftarrow_{c_0}}E_0\}\Gamma_1 \vdash_{\{x\leftarrow_{c_0}E_0\}c} \lambda y : \{x{\leftarrow_{c_0}}E_0\}T_2.\ \{x{\leftarrow_{c_0}}E_0\}E \longrightarrow^{*} \lambda y : \{x{\leftarrow_{c_0}}E_0'\}T_2.\ \{x{\leftarrow_{c_0}}E_0'\}E$.

Justification. Il s'agit essentiellement d'une généralisation de la règle (econv.cong.fun.body). Il n'est cependant pas clair que c'en soit une conséquence admissible : le système de types devrait peut-être comporter une règle plus générale — par exemple autorisant les conversions arbitraires dans toutes les sous-expressions pures d'une expression quelconque. Si le système doit être modifié, la principale contrainte est de respecter la confluence de la conversion ; une autre contrainte en l'état de notre raisonnement est que nous nous sommes arrangé pour que si $E \longrightarrow E'$ alors $\sigma E \longrightarrow \sigma E'$ (cas particulier du Th. B.1.27 (préservation du typage par substitution)), alors que de nombreuses formulations n'admettraient que la conséquence plus faible $\sigma E \equiv \sigma E'$. □

Theorème B.2.7 (confluence locale de la conversion) Si $B;\Gamma \vdash_c E \longrightarrow E'$ et $B;\Gamma \vdash_c E \longrightarrow E''$ alors il existe E''' tel que $B;\Gamma \vdash_c E' \longrightarrow^{*} E'''$ et $B;\Gamma \vdash_c E'' \longrightarrow^{*} E'''$.
Si $B;\Gamma \vdash_c T \longrightarrow T'$ et $B;\Gamma \vdash_c T \longrightarrow T''$ alors il existe T''' tel que $B;\Gamma \vdash_c T' \longrightarrow^{*} T'''$ et $B;\Gamma \vdash_c T'' \longrightarrow^{*} T'''$.

$$\begin{tikzcd}
\aleph \arrow[r] \arrow[d] & \aleph' \arrow[d, dotted, "*"] \\
\aleph'' \arrow[r, dotted, "*"] & \aleph'''
\end{tikzcd}$$

Démonstration. Raisonner par induction sur la somme des tailles des dérivations confrontées. Nous traitons ensemble les règles contextuelles ((tconv.cong.*) et (econv.cong.*)) et les axiomes (autres règles). Nous discriminons sur le couple des règles aux racines respectives des deux dérivations.

Commençons par le cas où les règles appliquées sont les mêmes des deux côtés. On vérifie par examen des axiomes qu'ils sont déterministes (c'est-à-dire qu'un terme donné peut s'instancier d'au plus une manière en tant que contractant d'une règle) : si le contractant $\aleph$ est fixé, le résidu $\aleph' = \aleph''$ est imposé. S'agissant des règles contextuelles, on vérifie que le choix du contractant $\aleph$ impose le contexte ainsi que le sous-terme converti dans ce contexte $\beth$. Ce sous-terme est converti vers deux sous-termes $\beth'$ et $\beth''$. Par induction, $\beth'$ et $\beth''$ confluent en un terme $\beth'''$. Alors $\aleph'$ et $\aleph''$ confluent en le terme $\aleph'''$ formé en plaçant $\beth'''$ dans le contexte considéré : en effet, en vertu du Lem. B.1.34 (préservation de la sorte par conversion) et du Lem. B.1.35 (préservation du typage par conversion), chaque résidu en un nombre quelconque d'étapes de $\beth$ vérifie les mêmes jugements de typage que $\beth$.

Occupons-nous désormais des cas où les règles appliquées sont différentes. Nous nous limitons à une moitié des cas suffisante de par la symétrie du problème. Nous ne listons que les paires de règles qui admettent effectivement un contractant commun (les paires critiques).

Une première classe de cas est celle où l'une des conversions est une êta-expansion, c'est-à-dire une règle (econv.eta.*) appliquée dans un contexte arbitraire. On vérifie que les êta-résidus contiennent toujours le contractant dans un contexte de réduction, et (grâce au Lem. B.1.35 (préservation du typage par conversion)) que les résidus vérifient les mêmes hypothèses de typage que les contractants. Il y a donc dans ce cas confluence forte : les êta-expansions commutent avec toute autre réduction. Dans la suite de cette démonstration, nous supposons qu'aucune des deux conversions n'est une êta-expansion.

Règles (tconv.cong.abs) et (tconv.abs) : On a $B; \Gamma \vdash_c (\!|A|\!) \longrightarrow (\!|A'|\!)$ avec $B; \Gamma \vdash_c A \longrightarrow A'$, et d'autre part $B; \Gamma \vdash_c (\!|A|\!) \longrightarrow \mathsf{Typ}\,E$; de plus $B; \Gamma \vdash_c A \triangleright E : \text{TYPE}^K$ et $B; \Gamma \vdash_c \mathbf{underl}(A)\ \mathsf{transparent}$. Par le Lem. B.1.32 (conversion d'une composante abstraite), il existe E' et T' tels que $B; \Gamma \vdash_c A' \triangleright E' : T'$ et $B; \Gamma \vdash_c E \longrightarrow^* E'$ et $\text{TYPE}^K \triangleq T'$. En vertu de la Rem. B.1.31 (forme de types substitutivement équivalents), $T' = \text{TYPE}^K$. En appliquant autant de fois qu'il le faut la règle (tconv.field), on a $B; \Gamma \vdash_c \mathsf{Typ}\,E \longrightarrow^* \mathsf{Typ}\,E'$. D'autre part, $\mathbf{underl}(A) = \mathbf{underl}(A')$ par le Lem. B.2.3 (conservation de l'apax sous-jacent), donc on peut appliquer (tconv.abs) pour prouver $B; \Gamma \vdash_c (\!|A'|\!) \longrightarrow \mathsf{Typ}\,E'$. Il y a donc confluence en $\mathsf{Typ}\,E'$.

Règles (tconv.cong.field) et (tconv.field) : On a $B; \Gamma \vdash_c \mathsf{Typ}\,\langle T_0 \rangle \longrightarrow \mathsf{Typ}\,E'_0$ avec $B; \Gamma \vdash_c \langle T_0 \rangle \longrightarrow E'_0$, et d'autre part $B; \Gamma \vdash_c \mathsf{Typ}\,\langle T_0 \rangle \longrightarrow T_0$; de plus $B; \Gamma \vdash_c T_0 : *$. Par inversion des règles (econv.*), la conversion $B; \Gamma \vdash_c \langle T_0 \rangle \longrightarrow E'_0$ résulte forcément de la règle (econv.cong.field), et il existe T'_0 tel que $E'_0 = \langle T'_0 \rangle$ et $B; \Gamma \vdash_c T_0 \longrightarrow T'_0$. On a $B; \Gamma \vdash_c \mathsf{Typ}\,\langle T'_0 \rangle \longrightarrow T'_0$ par (tconv.field). Il y a confluence en T'_0.

Règles (tconv.cong.sing) et (tconv.unit) : On a $B; \Gamma \vdash_c \mathsf{S}(()) \longrightarrow \mathsf{S}(E'_0)$ avec $B; \Gamma \vdash_c () \longrightarrow E'_0$, et d'autre part $B; \Gamma \vdash_c \mathsf{S}(()) \longrightarrow \text{UNIT}$. Par inversion des règles (econv.*), $()$ n'est pas convertible, donc cette paire critique n'admet aucune instance.

Règles (econv.cong.app.arg) et (econv.app) : On a $B; \Gamma \vdash_c (\lambda x : T_2.\ E_3)\,E_0 \longrightarrow (\lambda x : T_2.\ E_3)\,E'_0$ avec $B; \Gamma \vdash_c E_0 \longrightarrow E'_0$, et d'autre part $B; \Gamma \vdash_c (\lambda x{:}T_2.E_3)\,E_0 \longrightarrow \{x \leftarrow_c E_0\}E_3$; de plus $B; \Gamma \vdash_c E_0 :^P T_0$ et $B; \Gamma \vdash_c (\lambda x : T_2.\ E_3) :^P \Pi x : T_0.\,{}^P T_1$ et $B; \Gamma, x :_c T_2 \vdash_{c \cup \{x\}} E_3 :^P T_3$.

Par le Lem. B.1.35 (préservation du typage par conversion), on a $B; \Gamma \vdash_c E'_0 :^P T_0$; donc $B; \Gamma \vdash_c (\lambda x{:}T_2.E_3)\,E'_0 \longrightarrow \{x \leftarrow_c E'_0\}E_3$ par (econv.app). D'autre part, $B; \Gamma \vdash_c \{x \leftarrow_c E_0\}E_3 \longrightarrow^* \{x \leftarrow_c E'_0\}E_3$ par le Lem. B.2.5 (conversion sous une substitution). Il y a confluence en $E''' = \{x \leftarrow_c E'_0\}E_3$.

Règles (econv.cong.app.fun) et (econv.app) : On a $B; \Gamma \vdash_c (\lambda x : T_2.\ E_3)\,E_0 \longrightarrow E'_1\,E_0$ avec $B; \Gamma \vdash_c (\lambda x{:}T_2.E_3) \longrightarrow E'_1$, et d'autre part $B; \Gamma \vdash_c (\lambda x{:}T_2.E_3)\,E_0 \longrightarrow \{x \leftarrow_c E_0\}E_3$; de plus $B; \Gamma \vdash_c E_0 :^P T_0$

et $B; \Gamma \vdash_c (\lambda x : T_2.\ E_3) :^P \Pi x : T_0.\ ^P T_1$ et $B; \Gamma, x :_c T_2 \vdash_{c \cup \{x\}} E_3 :^P T_3$. Par inversion des règles (econv.*), la conversion de $(\lambda x : T_2.\ E_3)$ ne peut être obtenue que par l'une des règles (econv.cong.fun.arg), (econv.cong.fun.body) ou (econv.cong.fun.seal).

Règle (econv.cong.fun.arg) : On a $B; \Gamma \vdash_c T_2 \longrightarrow T_2'$. On a $E' \longrightarrow E''$ par (econv.app), sachant que l'hypothèse $B; \Gamma \vdash_c E_0 :^P T_2'$ découle de $B; \Gamma \vdash_c E_0 :^P T_2$ grâce à (eeq.*), (tsub.eq) et (et.sub).

Règle (econv.cong.fun.body) : Il existe E_4 et E_5 tels que $E_3 = \{y \leftarrow_{c \cup \{x\}} E_5\} E_4$, et on a $B; \Gamma \vdash_c \{y \leftarrow_{c \cup \{x\}} E_5\} E_4 \longrightarrow \{y \leftarrow_{c \cup \{x\}} E_5'\} E_4$ avec les prémisses $B; \Gamma, x :_c T_2 \vdash_{c \cup \{x\}} E_5 \longrightarrow E_5'$ et $B; \Gamma, x :_c T_2, y :_{c \cup \{x\}} S(E) \vdash_{c \cup \{y\} \cup \{x\}} E_4 :^{\gamma_4} T_4$. Grâce au Lem. B.2.5 (conversion sous une substitution), on a encore $B; \Gamma, x :_c T_2 \vdash_{c \cup \{x\}} \{y \leftarrow_{c \cup \{x\}} E_5'\} E_4 :^P T_3$, ce qui permet de réduire E' par (econv.app) avec comme résultat $\{x \leftarrow_c E_0\}\{y \leftarrow_{c \cup \{x\}} E_5'\} E_4$.

D'autre part, $E'' = \{x \leftarrow_c E_0\}\{y \leftarrow_{c \cup \{x\}} E_5\} E_4$ se réduit en $\{x \leftarrow_c E_0\}\{y \leftarrow_{c \cup \{x\}} E_5'\} E_4$ par le Lem. B.2.5 (conversion sous une substitution). Il y a donc bien confluence.

Règle (econv.cong.fun.seal) : Impossible en vertu du Lem. B.2.4 (impureté évidente).

Règles (econv.cong.col.*) et (econv.col.*) : On a $E = [E_0]_{c_0}^{T_0}$, et d'une part $B; \Gamma \vdash_{c_0} E_0 \longrightarrow E_0'$ ou $B; \Gamma \vdash_{c \cap c_0} T_0 \longrightarrow T_0'$, d'autre part $B; \Gamma \vdash_c [E_0]_{c_0}^{T_0} \longrightarrow E''$. Le motif du membre de gauche des règles (econv.col.*) restreint les règles appliquables pour convertir E_0 ou T_0. Dans la plupart des cas, cette conversion reste possible dans E'', où l'expression ou le type converti se trouve encore dans un contexte de réduction (éventuellement dans une couleur plus grande, ce qui n'est pas gênant en vertu du Lem. B.1.23 (affaiblissement de la couleur)). De l'autre côté, la conversion de E_0 ou T_0 n'affecte en général pas la possibilité d'appliquer la règle (econv.col.*), ce qui permet de refermer le diagramme. Nous étudions maintenant les paires critiques.

Règles (tconv.cong.fun.ret) et (econv.col.fun.l) : On a $E_0 = \lambda x : T_2.\ E_1$ et $T_0 = \Pi t : T_3.\ T_1$. D'une part $B; \Gamma, x :_{c \cap c_0} T_3 \vdash_{(c \cap c_0) \cup \{x\}} T_1 \longrightarrow T_1'$, et d'autre part $B; \Gamma \vdash_c [\lambda x : T_2.\ E_1]_{c_0}^{\Pi x : T_0.\ T_1} \longrightarrow \lambda x : T_3.\ (E_1 \ !!_{c_0 \cup \{x\}}\ T_1)$. Ce dernier terme se réduit en $\lambda x : T_3.\ (E_1 \ !!_{c_0 \cup \{x\}}\ T_1')$ par (econv.cong.fun.seal). On obtient également ce terme en réduisant E_0' par (econv.col.fun.l).

Règles (econv.col.merge) et (econv.col.merge) : Sachant que $(c_2 \cup c_3) \cup c_1 = c_3 \cup (c_1 \cup c_2)$, il n'y a pas véritablement de paire critique.

Règles (tconv.unit) et (econv.col.sing) : On a $E' = E'' = ()$.

Règles (econv.cong.fun.arg) et (econv.cong.fun.body) : On a $B; \Gamma \vdash_c (\lambda x : T_0.\ E_1) \longrightarrow (\lambda x : T_0'.\ E_1)$ sous les prémisses $B; \Gamma \vdash_c T_0 \longrightarrow T_0'$ et $B; \Gamma, x :_c T_0 \vdash_{c \cup \{x\}} E_1 :^\gamma T_1$, et d'autre part $B; \Gamma \vdash_c (\lambda x : T_0.\ E_1) \longrightarrow (\lambda x : T_0.\ E_1')$ sous des prémisses de la forme $B; \Gamma, x :_c T_0, \Gamma' \vdash_{c \cup \{x\} \cup c_1} J_1$ où J_1 et c_1 sont indépendants de T_0. En vertu du Lem. B.1.21 (affaiblissement du type d'une variable), les prémisses de la conversion par (econv.cong.fun.body) sont encore vérifiées si l'on remplace T_0 par T_0', donc on a $B; \Gamma \vdash_c (\lambda x : T_0'.\ E_1) \longrightarrow (\lambda x : T_0'.\ E_1')$.

Sous réserve de vérifier que $B; \Gamma, x :_c T_0 \vdash_{c \cup \{x\}} E_1' :^\gamma T_1$, on a $B; \Gamma \vdash_c (\lambda x : T_0.\ E_1') \longrightarrow (\lambda x : T_0'.\ E_1')$ par (econv.cong.fun.arg) : il y a confluence en $(\lambda x : T_0'.\ E_1')$. Il nous reste à prouver ce lemme.

On a $E_1 = \{y \leftarrow_{c \cup \{x\}} E_2\} E_3$ et $E_1' = \{y \leftarrow_{c \cup \{x\}} E_2'\} E_3$ avec $B; \Gamma, x :_c T_0 \vdash_{c \cup \{x\}} E_2 \longrightarrow E_2'$ et $B; \Gamma, x :_c T_0, y :_{c \cup \{x\}} S(E_2) \vdash_{c \cup \{y\} \cup \{x\}} E_3 :^\gamma T_3$. Par le Lem. B.1.35 (préservation du typage par conversion), on a $B; \Gamma, x :_c T_0 \vdash_{c \cup \{x\}} E_2' :^P S(E_2)$. Par le Th. B.1.27 (préservation du typage par substitution), on a $B; \Gamma, x :_c T_0 \vdash_{c \cup \{x\}} \{y \leftarrow_{c \cup \{x\}} E_2'\} E_3 :^\gamma T_1$.

Règles (econv.cong.fun.arg) et (econv.cong.fun.seal) : Le raisonnement est similaire au cas précédent ; seul change le lemme de préservation du type du corps.

Règles (econv.cong.fun.body) et (econv.cong.fun.seal) : L'unification des membres de gauche des règles donne $E = \lambda x : T_0.\ ((\sigma E_1)\ !!_{c'}\ (\sigma T_1))$ avec $\sigma = \{y \leftarrow_{c \cup \{x\}} E_2\}$; les réduits sont respectivement $E' = \lambda x : T_0.\ ((\sigma' E_1)\ !!_{c'}\ (\sigma' T_1))$ et $E'' = \lambda x : T_0.\ ((\sigma E_1)\ !!_{c'}\ T_3'')$ avec $\sigma' = \{y \leftarrow_{c \cup \{x\}} E_2'\}$, sous les hypothèses $B; \Gamma \vdash_c E_2 \longrightarrow E_2'$ et $B; \Gamma \vdash_c \sigma T_1 \longrightarrow T_3''$.

Par le Lem. B.2.5 (conversion sous une substitution), on a $B; \Gamma \vdash_c \sigma T_1 \longrightarrow^* \sigma' T_1$. En vertu de la Conj. B.2.8 (confluence de la conversion) ci-dessous, qui est appliqué à une hypothèse dont la démonstration est plus petite, il y a confluence en un type T_3''' depuis les réduits $\sigma' T_1$ et T_3'' de σT_1. En appliquant de part et d'autre la règle (econv.cong.fun.seal) autant de fois qu'il le faut, on en déduit la confluence de E' et E'' en $E''' = \lambda x : T_0.\ ((\sigma E_1)\ !!_{c'}\ T_3''')$.

Règles (econv.cong.proj) et (econv.proj) : On a $B; \Gamma \vdash_c \pi_i (E_1, E_2) \longrightarrow \pi_i E_0'$ avec $B; \Gamma \vdash_c (E_1, E_2) \longrightarrow E_0'$, et d'autre part $B; \Gamma \vdash_c \pi_i (E_1, E_2) \longrightarrow E_i$; de plus $B; \Gamma \vdash_c E_i :^P T_i$. Par inversion des règles (econv.*), la conversion de (E_1, E_2) résulte forcément de l'une des règles (econv.cong.proj.*), et on a $E_0' = E_j'$ avec $B; \Gamma \vdash_c E_j \longrightarrow E_j'$ et $j = 1$ ou $j = 2$. Dans les deux cas, il y a confluence en E_j' si $i = j$ et en E_i sinon.

□

Conjecture B.2.8 (confluence de la conversion) Si $B; \Gamma \vdash_c E \longrightarrow^* E'$ et $B; \Gamma \vdash_c E \longrightarrow^* E''$ alors il existe E''' tel que $B; \Gamma \vdash_c E' \longrightarrow^* E'''$ et $B; \Gamma \vdash_c E'' \longrightarrow^* E'''$.
Si $B; \Gamma \vdash_c T \longrightarrow^* T'$ et $B; \Gamma \vdash_c T \longrightarrow^* T''$ alors il existe T''' tel que $B; \Gamma \vdash_c T' \longrightarrow^* T'''$ et $B; \Gamma \vdash_c T'' \longrightarrow^* T'''$.

$$\begin{tikzcd} \aleph \arrow[r, "*"] \arrow[d, "*"] & \aleph' \arrow[d, dotted, "*"] \\ \aleph'' \arrow[r, dotted, "*"] & \aleph''' \end{tikzcd}$$

Justification. Notre système de réduction est essentiellement composé de trois catégories de règles :
– la règle de bêta-expansion, pour laquelle des méthodes classiques (rédidus finis, conversion parallèle) permettent de passer de la confluence locale à la confluence ;
– un jeu de règles pour lequel la démonstration de la confluence locale ci-dessus montre qu'il y a confluence forte ;
– les règles de poussée des crochets, qui ne font que diminuer la taille totale des termes apparaissant sous un crochet, donc pour lesquelles un argument de résidus finis devrait pouvoir s'appliquer.

Une conjecture plus forte est que la conversion, hors les règles d'êta-expansion (qui conservent la confluence forte), soit fortement normalisante. Cette conjecture est probablement fausse en l'état, mais vraie si l'on stratifie le système en univers bien ordonnés (voir la section V.3.1.5). Dans ce cas, la confluence locale implique la confluence. □

Corollaire B.2.9 (convertibilité par étapes confluentes) Si $B; \Gamma \vdash_c E \equiv E'$ alors il existe E'' tel que $B; \Gamma \vdash_c E \longrightarrow^* E''$ et $B; \Gamma \vdash_c E' \longrightarrow^* E''$.
Si $B; \Gamma \vdash_c T \equiv T'$ alors il existe T'' tel que $B; \Gamma \vdash_c T \longrightarrow^* T''$ et $B; \Gamma \vdash_c T' \longrightarrow^* T''$.

$$\begin{tikzcd} \aleph \arrow[rr, equal, "\equiv"] \arrow[dr, "*"'] & & \aleph' \arrow[dl, "*"] \\ & \aleph'' & \end{tikzcd}$$

Démonstration. Conséquence classique de la confluence des relations de conversion (Conj. B.2.8 (confluence de la conversion)), sachant que les règles (eeq.*) et (teq.*) définissent la convertibilité comme la relation d'équivalence engendrée par la conversion. □

B.2.2 Cohérence de l'équivalence de types

Définition B.2.10 (forme normale de tête faible) Un type T est dit en **forme normale de tête faible** si et seulement si il n'est pas de la forme $\mathsf{Typ}\,\mathsf{E}$ ou $(\!|\mathsf{A}|\!)$ avec A transparente dans la couleur ambiante.

Lemme B.2.11 (têtes de types convertibles) Si $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{T} \longrightarrow \mathsf{T}'$ et si T est en forme normale de tête faible alors l'une des propriétés suivantes est vraie :

- il existe T_1, T_1', T_2 et T_2' tels que $\mathsf{T} = \Sigma x : \mathsf{T}_1.\, \mathsf{T}_2$, $\mathsf{T}' = \Sigma x : \mathsf{T}_1'.\, \mathsf{T}_2'$, $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{T}_1 \longrightarrow^? \mathsf{T}_2$ et $\mathsf{B}; x :_{\mathsf{c}} \mathsf{T}_1 \vdash_{\mathsf{c}\cup\{x\}} \mathsf{T}_1' \longrightarrow^? \mathsf{T}_2'$;
- il existe T_1, T_1', γ, γ', T_2 et T_2' tels que $\mathsf{T} = \Pi x : \mathsf{T}_1.\, {}^{\gamma}\mathsf{T}_2$, $\mathsf{T}' = \Pi x : \mathsf{T}_1'.\, {}^{\gamma'}\mathsf{T}_2'$, $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{T}_1 \longrightarrow^? \mathsf{T}_2$, $\gamma \sqsubseteq \gamma'$ et $\mathsf{B}; x :_{\mathsf{c}} \mathsf{T}_1 \vdash_{\mathsf{c}\cup\{x\}} \mathsf{T}_1' \longrightarrow^? \mathsf{T}_2'$;
- il existe E et E' tels que $\mathsf{T} = \mathsf{S}(\mathsf{E})$, $\mathsf{T}' = \mathsf{S}(\mathsf{E}')$ et $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{E} \longrightarrow \mathsf{E}'$;
- il existe E telle que $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{E} \equiv ()$, et $\mathsf{T} = \mathsf{S}(\mathsf{E})$ et $\mathsf{T}' = \textsc{unit}$.
- il existe A et A' tels que $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{A} \longrightarrow \mathsf{A}'$, et $\mathsf{T} = (\!|\mathsf{A}|\!)$ et $\mathsf{T} = (\!|\mathsf{A}'|\!)$.

Démonstration. Simple discrimination sur la règle (tconv.*) utilisée. On note que la règle (tconv.abs) ne peut révéler qu'une composante dont l'empreinte sous-jacente est transparente, ce qui est exclu par définition des formes normales de tête faibles. □

Lemme B.2.12 (têtes de types équivalents) Si $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{T} \equiv \mathsf{T}'$ et si T et T' sont en forme normale de tête faible alors l'une des propriétés suivantes est vraie :

- il existe T_1, T_1', T_2 et T_2' tels que $\mathsf{T} = \Sigma x : \mathsf{T}_1.\, \mathsf{T}_2$, $\mathsf{T}' = \Sigma x : \mathsf{T}_1'.\, \mathsf{T}_2'$, $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{T}_1 \equiv \mathsf{T}_2$ et $\mathsf{B}; x :_{\mathsf{c}} \mathsf{T}_1 \vdash_{\mathsf{c}\cup\{x\}} \mathsf{T}_1' \equiv \mathsf{T}_2'$;
- il existe T_1, T_1', γ, γ', T_2 et T_2' tels que $\mathsf{T} = \Pi x : \mathsf{T}_1.\, {}^{\gamma}\mathsf{T}_2$, $\mathsf{T}' = \Pi x : \mathsf{T}_1'.\, {}^{\gamma'}\mathsf{T}_2'$, $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{T}_1 \equiv \mathsf{T}_2$, $\gamma \sqsubseteq \gamma'$ et $\mathsf{B}; x :_{\mathsf{c}} \mathsf{T}_1 \vdash_{\mathsf{c}\cup\{x\}} \mathsf{T}_1' \equiv \mathsf{T}_2'$;
- il existe E et E' tels que $\mathsf{T} = \mathsf{S}(\mathsf{E})$, $\mathsf{T}' = \mathsf{S}(\mathsf{E}')$ et $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{E} \equiv \mathsf{E}'$;
- il existe A et A' tels que $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{A} \equiv \mathsf{A}'$, et $\mathsf{T} = (\!|\mathsf{A}|\!)$ et $\mathsf{T} = (\!|\mathsf{A}'|\!)$.
- $\mathsf{T} = \mathsf{T}'$ est un type de base ($\textsc{unit}$, $\textsc{bool}$, $\textsc{int}$, $\textsc{dyn}$) ou $\textsc{type}^{\mathsf{K}}$;
- il existe E telle que $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{E} \equiv ()$, et $\mathsf{T} = \mathsf{S}(\mathsf{E})$ et $\mathsf{T}' = \textsc{unit}$ ou vice versa.

Démonstration. D'après le Cor. B.2.9 (convertibilité par étapes confluentes), il existe T'' tel que $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{T} \longrightarrow^* \mathsf{T}''$ et $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{T}' \longrightarrow^* \mathsf{T}''$. Une récurrence triviale sur la longueur de chaque chaîne de conversions et l'application du Lem. B.2.11 (têtes de types convertibles) montrent que T'' a le même constructeur de tête que T et des sous-termes résultant d'une chaîne de conversions, et de même pour T'' et T' ; donc T et T' ont le même constructeur de tête et des sous-termes convertibles. □

B.2.3 Sous-typage

Définition B.2.13 (sous-typage élémentaire) On dit qu'il y a **sous-typage elémentaire** entre T et T' (ou que T est un sous-type élémentaire de T'), et on note $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{T} \prec: \mathsf{T}'$, si et seulement si $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{T} <: \mathsf{T}'$ par une dérivation dont la règle à la racine n'est ni (tsub.eq), ni (tsub.trans).

Lemme B.2.14 (étapes de sous-typage) On a $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{T} <: \mathsf{T}'$ si et seulement si il existe une suite finie $\mathsf{T}_0, \ldots, \mathsf{T}_k$ telle que $\mathsf{T} = \mathsf{T}_0$, $\mathsf{T}_k = \mathsf{T}'$, et pour tout indice i, on a une **réduction** $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{T}_i \longrightarrow \mathsf{T}_{i+1}$ ou une **expansion** $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{T}_{i+1} \longrightarrow \mathsf{T}_i$ ou un sous-typage élémentaire $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{T}_i \prec: \mathsf{T}_{i+1}$. Pour la réciproque, si $k = 0$, on demande de plus $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{T}_0 : *$.

La suite construite par ce lemme dans le sens direct est appelée la **suite des étapes élémentaires** de la dérivation du jugement de sous-typage $B; \Gamma \vdash_c T <: T'$. Toute autre suite vérifiant ces propriétés est *une* suite d'étapes élémentaires du jugement $B; \Gamma \vdash_c T <: T'$.

Démonstration. Si $B; \Gamma \vdash_c T <: T'$, on construit une suite d'étapes d'équivalence ou de sous-typage élémentaire par induction sur la dérivation de sous-typage.

Règles (tsub.cong.*), (tsub.sing) : On a sous-typage élémentaire.

Règle (tsub.eq) : La prémisse est $B; \Gamma \vdash_c T \equiv T'$. Une récurrence facile sur les règles (teq.*) permet de construire une suite de conversions et de conversions réciproques.

Règle (tsub.trans) : Mettre bout à bout les suites associées aux prémisses.

Si $B; \Gamma \vdash_c T \longrightarrow T'$ ou $B; \Gamma \vdash_c T' \longrightarrow T$ alors $B; \Gamma \vdash_c T <: T'$ par (teq.conv), éventuellement (teq.refl), et (tsub.eq). Si $B; \Gamma \vdash_c T \prec: T'$ alors évidemment $B; \Gamma \vdash_c T <: T'$. On obtient la réciproque du présent lemme pour une suite quelconque de longueur au moins 1 en concaténant les étapes à l'aide de (tsub.trans). Enfin, si $B; \Gamma \vdash_c T_0 : *$, on a $B; \Gamma \vdash_c T_0 <: T_0$ par (teq.refl) et (tsub.eq). □

Lemme B.2.15 (étapes de sous-typage avec confluence) Si $B; \Gamma \vdash_c T <: T'$ alors il existe une suite d'étapes élémentaires du jugement $B; \Gamma \vdash_c T <: T'$ dans laquelle une expansion n'est jamais suivie immédiatement d'une réduction.

Démonstration. Considérons la suite des étapes de la dérivation initiale donnée par le Lem. B.2.14 (étapes de sous-typage). Pour toute sous-suite $T_i, \ldots, T_j$ avec $i < j$ ne contenant pas d'étape de sous-typage, on a $T_i \equiv T_j$, et on obtient une nouvelle suite d'étapes du jugement initial en remplaçant cette sous-suite par la suite $T_i, \ldots, T'', \ldots, T_j$ donnée par la confluence de T_i et T_j en T'' résultant du Cor. B.2.9 (convertibilité par étapes confluentes). Si l'on applique (en parallèle) cette transformation à toutes les sous-suites maximales ne contenant pas d'étapes de sous-typage, on obtient une suite d'étapes vérifiant la propriété souhaitée. □

Lemme B.2.16 (têtes de types comparables) Si $B; \Gamma \vdash_c T <: T'$ et si T et T' sont en forme normale de tête faible alors l'une des propriétés suivantes est vraie :
- il existe T_1, T_1', T_2 et T_2' tels que $T = \Sigma x : T_1.\, T_2$, $T' = \Sigma x : T_1'.\, T_2'$, $B; \Gamma \vdash_c T_1 <: T_1'$, $B; x :_c T_1 \vdash_c T_2 <: T_2'$ et $B; x :_c T_1' \vdash_{c \cup \{x\}} T_2' : *$;
- il existe T_0, T_0', γ, γ', T_1 et T_1' tels que $T = \Pi x : T_0.\, {}^{\gamma}T_1$, $T' = \Pi x : T_0'.\, {}^{\gamma'}T_1'$, $B; \Gamma \vdash_c T_0' <: T_0$, $\gamma \sqsubseteq \gamma'$, $B; x :_c T_0' \vdash_c T_1 <: T_1'$ et $B; x :_c T_0 \vdash_{c \cup \{x\}} T_1 : *$;
- il existe K et K' tels que $T = \mathrm{TYPE}^K$, $T' = \mathrm{TYPE}^{K'}$ et $K \leqslant K'$;
- il existe A et A' tels que $B; \Gamma \vdash_c A \equiv A'$, et $T = (\!|A|\!)$ et $T = (\!|A'|\!)$.
- $T = T'$ est un type de base (UNIT, BOOL, INT ou DYN) ;
- $T = \mathrm{UNIT}$ et $T' = S(E)$ avec $B; \Gamma \vdash_c E \equiv ()$ ou vice versa ;
- il existe E telle que $T = S(E)$.

Démonstration. Considérons la suite des étapes du jugement $B; \Gamma \vdash_c T <: T'$ donnée par le Lem. B.2.15 (étapes de sous-typage avec confluence). Soit n le nombre d'étapes qui sont des sous-typages élémentaires ; la suite d'étapes peut se décomposer ainsi :

$$
\begin{tikzcd}
T_{i_0} \arrow[dr, "* *"] & & T_{k_0} \arrow[dl] \arrow[r, dash, "\prec:"] & T_{i_1} \arrow[dr, "* *"] & & T_{k_1} \arrow[dl] \arrow[r, dash, "\prec:"] & \cdots \arrow[r, dash, "\prec:"] & T_{i_n} \arrow[dr, "* *"] & & T_{k_n} \arrow[dl] \\
& T_{j_0} & & & T_{j_1} & & & & T_{j_n} &
\end{tikzcd}
$$

(On a $i_0 = 0$, $T_0 = T$ et $T_{j_n} = T'$.) Notons que pour tout m on a $T_{i_m} \equiv T_{k_m}$.

La forme des règles (tsub.*) montre immédiatement que pour tout m, le type T_{k_m} est en forme normale de tête faible. C'est également le cas pour T_{i_m}, sauf si $m \geqslant 1$ et que l'étape menant à T_{i_m} est un sous-typage par (tsub.sing).

Supposons dans un premier temps qu'aucune des étapes de sous-typage ne provient de la règle (tsub.sing) appliquée à la racine. Alors les types T_{i_m} sont tous en forme normale de tête, donc on peut appliquer le Lem. B.2.12 (têtes de types équivalents) à chaque équivalence. Ce lemme indique en particulier que pour tout m, les types T_{i_m} et T_{k_m} ont le même constructeur en tête. De plus, l'examen des règles (tsub.*) montre, après une récurrence triviale sur m, que les types T_{i_m} et T_{k_m} ont tous le même constructeur en tête (ou alors l'un est UNIT et l'autre $S(E)$ avec $E \equiv ()$), et que les étapes de sous-typage utilisent la règle (tsub.cong.*) appropriée pour ce constructeur. Pour chaque constructeur, on vérifie que la conclusion du Lem. B.2.12 (têtes de types équivalents), combinée aux règles (tsub.eq) et (tsub.trans), ainsi qu'au Lem. B.1.33 (validité) et au Lem. B.1.21 (affaiblissement du type d'une variable) pour les types dépendants, donne le cas correspondant du présent lemme.

S'il l'une des étapes de sous-typage provient de la règle (tsub.sing), c'est notamment le cas de $T_{k_0} \prec: T_{i_1}$: en effet, dans le cas contraire, le raisonnement précédent impose la préservation d'un constructeur autre que $S(\_\_)$ en tête des types. Au vu du Lem. B.2.12 (têtes de types équivalents), on a forcément $T = S(E)$. $\square$

B.2.4 Analyse du typage d'une expression

Remarque B.2.17 (règles structurelles du typage des expressions) En dehors de (et.sing) et (et.sub), les règles (et.*) sont structurelles : si une dérivation d'un jugement $B; \Gamma \vdash_c E :^{\gamma} T$ ne se termine pas par (et.sing) ni (et.sub), sa règle à la racine est déterminée par le constructeur de tête de E.

Lemme B.2.18 (analyse du dernier type d'une expression) Si $B; \Gamma \vdash_c E :^{\gamma} T$ alors il existe T' et γ' tels que les propriétés suivantes soient toutes vérifiées :
- $B; \Gamma \vdash_c E :^{\gamma'} T'$ par une sous-dérivation de la dérivation originale se terminant par une règle structurelle ;
- $\gamma' \sqsubseteq \gamma$;
- soit $\gamma' = P$ et $B; \Gamma \vdash_c S(E) \equiv T$, soit $B; \Gamma \vdash_c T' <: T$.

Ce lemme transcrit l'effet des règles non structurelles près de la racine de la dérivation. Le premier cas de la dernière propriété correspond au passage par une instance de la règle (et.sing) qui n'est pas compensée par un sous-typage subséquent.

Démonstration. On remarque que si un jugement de typage est dérivable, il en existe une dérivation dans laquelle une prémisse de (et.sub) n'est jamais obtenue par la règle (tsub.trans). En effet, si un tel agencement se produit, on peut supprimer la règle (tsub.trans) en question en la remplaçant par une application supplémentaire de (et.sub). Dans la suite de la présente démonstration, nous supposons la dérivation de typage sous cette forme, pour la partie de la dérivation à laquelle on accède à partir de la racine sans franchir une règle (et.*) structurelle.

Raisonner par induction sur la taille de la dérivation ainsi obtenue pour $B; \Gamma \vdash_c E :^{\gamma} T$.

Si la dernière règle est une règle (et.*) structurelle, on prend $T' = T$ et $\gamma' = \gamma$; on a $B; \Gamma \vdash_c T <: T$ par le Lem. B.1.33 (validité) et les règles (teq.refl) et (tsub.eq).

Sinon, la dernière règle est (et.sub) et (et.sing). Dans les deux cas, une prémisse est de la forme $B; \Gamma \vdash_c E :^{\gamma_1} T_1$. Par induction, il existe T' et γ' tels que $B; \Gamma \vdash_c E :^{\gamma'} T'$ par une sous-dérivation se terminant sur règle structurelle, $\gamma' \sqsubseteq \gamma_1$, et soit $\gamma' = P$ et $S(E) \equiv T_1$, soit $B; \Gamma \vdash_c T' <: T_1$. Notons que dans tous les cas, $\gamma_1 \sqsubseteq \gamma$, donc $\gamma' \sqsubseteq \gamma$ par transitivité. Pour montrer que T' et γ' conviennent, il reste seulement à démontrer une alternative de la dernière propriété ; nous étudions chacun des 4 cas séparément.

Règle (et.sub), si $\gamma' = \mathsf{P}$ et $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{S}(\mathsf{E}) \equiv \mathsf{T}_1$: On a $\gamma' = \mathsf{P}$ car $\gamma' \sqsubseteq \gamma_1$. D'après le Lem. B.2.12 (têtes de types équivalents), soit $\mathsf{T}_1 = \mathsf{S}(\mathsf{E}_1)$ avec $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{E} \equiv \mathsf{E}_1$, soit $\mathsf{T}_1 = \textsc{unit}$ et $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{E} \equiv ()$, soit $\mathsf{T}_1 = \mathsf{Typ}\,\mathsf{E}_1$. Une prémisse de la règle (et.sub) est $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{T}_1 <: \mathsf{T}$. Discriminons suivant la règle (tsub.*) utilisée pour prouver cette prémisse.

Règles (tsub.cong.*) : Incompatibles avec les formes possibles pour T_1.

Règle (tsub.eq) : On a $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{T}_1 \equiv \mathsf{T}'$, d'où $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{S}(\mathsf{E}) \equiv \mathsf{T}'$ par (teq.trans). Les conditions de la première alternative sont remplies.

Règle (tsub.trans) : Exclue au début de la présente démonstration.

Règle (tsub.sing) : On a $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{E} :^{\mathsf{P}} \mathsf{T}$ par une sous-dérivation de la dérivation initiale. L'application de l'hypothèse de récurrence à cette sous-dérivation produit un type T'' et un effet γ'', qui remplissent les conditions souhaitées.

Règle (et.sub), si $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{T}' <: \mathsf{T}_1$: On a $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{T}' <: \mathsf{T}$ par (tsub.trans). Les conditions de la seconde alternative sont remplies.

Règle (et.sing), si $\gamma' = \mathsf{P}$ et $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{S}(\mathsf{E}) \equiv \mathsf{T}_1$: On a $\mathsf{T} = \mathsf{S}(\mathsf{E})$. Par (tok.sing) et (teq.refl), on a $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{S}(\mathsf{E}) \equiv \mathsf{S}(\mathsf{E})$. Les conditions de la première alternative sont remplies.

Règle (et.sing), si $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{T}' <: \mathsf{T}_1$: On a $\gamma_1 = \mathsf{P}$, d'où $\gamma' = \mathsf{P}$. Comme dans le cas précédent, on a $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{S}(\mathsf{E}) \equiv \mathsf{S}(\mathsf{E})$, et les conditions de la première alternative sont remplies.

□

Lemme B.2.19 (inversion du typage d'un champ type) Si $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \langle \mathsf{T} \rangle :^{\gamma} \mathsf{T}'$ alors $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{T} : *$. Si $\mathsf{T}' = \textsc{type}^{\wr}$, on a même $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{T} : \wr$.

Démonstration. Grâce au Lem. B.2.18 (analyse du dernier type d'une expression), on obtient une dérivation de typage de $\langle \mathsf{T} \rangle$ qui se termine par la règle structurelle (et.type) ; sa prémisse est $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{T} : \mathsf{K}$. On en déduit $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{T} : *$ par (tok.sub).

La conclusion de la règle (et.type) est $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \langle \mathsf{T} \rangle :^{\mathsf{P}} \textsc{type}^{\mathsf{K}}$, et on a $\textsc{type}^{\mathsf{K}} <: \mathsf{T}'$. Par le Lem. B.2.16 (têtes de types comparables), on a $\mathsf{T}' = \textsc{type}^{\mathsf{K}'}$ avec $\mathsf{K} \leqslant \mathsf{K}'$. Si $\mathsf{T}' = \textsc{type}^{\wr}$, alors $\mathsf{K} = \wr$. □

Lemme B.2.20 (inversion du typage d'une paire) Si $\mathsf{B};\Gamma \vdash_{\mathsf{c}} (\mathsf{E}_1, \mathsf{E}_2) :^{\gamma} \Sigma x : \mathsf{T}_1.\, \mathsf{T}_2$ et il existe T_1' et T_2' tels que $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{E}_1 :^{\gamma} \mathsf{T}_1'$ et $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{E}_2 :^{\gamma} \mathsf{T}_2'$ et $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{T}_1' <: \mathsf{T}_1$ et $\mathsf{B};\Gamma, x :_{\mathsf{c}} \mathsf{T}_1' \vdash_{\mathsf{c} \cup \{x\}} \mathsf{T}_2' <: \mathsf{T}_2$.

Démonstration. D'après le Lem. B.2.18 (analyse du dernier type d'une expression), il existe T' et γ' tels que $\mathsf{B};\Gamma \vdash_{\mathsf{c}} (\mathsf{E}_1, \mathsf{E}_2) :^{\gamma'} \mathsf{T}'$ par une règle structurelle, $\gamma' \sqsubseteq \gamma$, et soit $\gamma' = \mathsf{P}$ et $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{S}((\mathsf{E}_1, \mathsf{E}_2)) \equiv \Sigma x : \mathsf{T}_1.\, \mathsf{T}_2$, soit $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{T}' <: \Sigma x : \mathsf{T}_1.\, \mathsf{T}_2$. Le cas $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{S}((\mathsf{E}_1, \mathsf{E}_2)) \equiv \mathsf{T}$ est exclu par le Lem. B.2.12 (têtes de types équivalents), donc on a $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{T}' <: \Sigma x : \mathsf{T}_1.\, \mathsf{T}_2$. La règle structurelle est forcément (et.pair), donc il existe T_1' et T_2' tels que $\mathsf{T}' = \mathsf{T}_1' * \mathsf{T}_2'$, et les prémisses sont $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{E}_1 :^{\gamma'} \mathsf{T}_1'$ et $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{E}_2 :^{\gamma'} \mathsf{T}_2'$. D'après le Lem. B.2.16 (têtes de types comparables), de $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{T}_1' * \mathsf{T}_2' <: \Sigma x : \mathsf{T}_1.\, \mathsf{T}_2$, on déduit que $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{T}_1' <: \mathsf{T}_1$ et $\mathsf{B}; x :_{\mathsf{c}} \mathsf{T}_1' \vdash_{\mathsf{c} \cup \{x\}} \mathsf{T}_2' <: \mathsf{T}_2$. □

Lemme B.2.21 (inversion du typage d'une fonction) Si $\mathsf{B};\Gamma \vdash_{\mathsf{c}} (\lambda x : \mathsf{T}_0.\ \mathsf{E}_1) :^{\gamma} \Pi x : \mathsf{T}_2.\, {}^{\gamma_1}\mathsf{T}_1$ alors $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{T}_2 <: \mathsf{T}_0$ et $\mathsf{B};\Gamma, x :_{\mathsf{c}} \mathsf{T}_0 \vdash_{\mathsf{c} \cup \{x\}} \mathsf{E}_1 :^{\gamma_1} \mathsf{T}_1$.

Démonstration. D'après le Lem. B.2.18 (analyse du dernier type d'une expression), il existe T' et γ' tels que $\mathsf{B};\Gamma \vdash_{\mathsf{c}} (\lambda \mathsf{x} : \mathsf{T}_0.\ \mathsf{E}_1) :^{\gamma'} \mathsf{T}'$ par une règle structurelle, $\gamma' \sqsubseteq \gamma$, et soit $\gamma' = \mathtt{P}$ et $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{S}(\lambda \mathsf{x} : \mathsf{T}_0.\ \mathsf{E}_1) \equiv \Pi \mathsf{x} : \mathsf{T}_2.\ ^{\gamma_1}\mathsf{T}_1$, soit $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{T}' <: \Pi \mathsf{x} : \mathsf{T}_2.\ ^{\gamma_1}\mathsf{T}_1$. Le cas $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{S}(\lambda \mathsf{x} : \mathsf{T}_0.\ \mathsf{E}_1) \equiv \mathsf{T}$ est exclu par le Lem. B.2.12 (têtes de types équivalents), donc on a $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{T}' <: \Pi \mathsf{x} : \mathsf{T}_2.\ ^{\gamma_1}\mathsf{T}_1$. La règle structurelle est forcément (et.fun), donc il existe γ_1' et T_1' tels que $\mathsf{T}' = \Pi \mathsf{x} : \mathsf{T}_0.\ ^{\gamma_1'}\mathsf{T}_1'$, et la prémisse est $\mathsf{B};\Gamma, \mathsf{x} :_{\mathsf{c}} \mathsf{T}_0 \vdash_{\mathsf{c}} \mathsf{E}_1 :^{\gamma_1'} \mathsf{T}_1'$. D'après le Lem. B.2.16 (têtes de types comparables), de $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \Pi \mathsf{x} : \mathsf{T}_0.\ ^{\gamma_1'}\mathsf{T}_1' <: \Pi \mathsf{x} : \mathsf{T}_2.\ ^{\gamma_1}\mathsf{T}_1$, on déduit que $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{T}_2 <: \mathsf{T}_0$, $\gamma_1' \sqsubseteq \gamma_1$ et $\mathsf{B}; \mathsf{x} :_{\mathsf{c}} \mathsf{T}_0 \vdash_{\mathsf{c} \cup \{\mathsf{x}\}} \mathsf{T}_1' <: \mathsf{T}_1$. Par (et.sub), on a $\mathsf{B};\Gamma, \mathsf{x} :_{\mathsf{c}} \mathsf{T}_0 \vdash_{\mathsf{c} \cup \{\mathsf{x}\}} \mathsf{E}_1 :^{\gamma} \mathsf{T}_1$. □

Lemme B.2.22 (inversion du typage d'un dynamique) Si $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{dyn}\,\mathsf{E}_0\,\mathsf{at}\,\mathsf{T}_0 :^{\gamma} \textsc{dyn}$ alors $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{E}_0 :^{\gamma} \mathsf{T}_0$.

Démonstration. D'après le Lem. B.2.18 (analyse du dernier type d'une expression), il existe T' et γ' tels que $\mathsf{B};\Gamma \vdash_{\mathsf{c}} (\mathsf{dyn}\,\mathsf{E}_0\,\mathsf{at}\,\mathsf{T}_0) :^{\gamma'} \mathsf{T}'$ par une règle structurelle et $\gamma' \sqsubseteq \gamma$. La règle structurelle est forcément (et.dyn) ou (et.dynned), et dans les deux cas une prémisse est $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{E}_0 :^{\gamma'} \mathsf{T}_0$. Par (et.sub), on a $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{E}_0 :^{\gamma} \mathsf{T}_0$. □

Lemme B.2.23 (inversion du typage d'un crochet coloré) Si $\mathsf{B};\Gamma \vdash_{\mathsf{c}} [\mathsf{E}']_{\mathsf{c}'}^{\mathsf{T}'} :^{\gamma} \mathsf{T}$ alors
- soit $\mathsf{B};\Gamma \vdash_{\mathsf{c}'} \mathsf{E}' :^{\mathtt{P}} \mathsf{T}'$ et $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{S}([\mathsf{E}']_{\mathsf{c}'}^{\mathsf{T}'}) \equiv \mathsf{T}$;
- soit $\mathsf{B};\Gamma \vdash_{\mathsf{c}'} \mathsf{E}' :^{\gamma} \mathsf{T}'$ et $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{T}' <: \mathsf{T}$.

Démonstration. D'après le Lem. B.2.18 (analyse du dernier type d'une expression), il existe T'' et γ' tels que $\mathsf{B};\Gamma \vdash_{\mathsf{c}} ([\mathsf{E}']_{\mathsf{c}'}^{\mathsf{T}'}) :^{\gamma'} \mathsf{T}'$ par une règle structurelle, $\gamma' \sqsubseteq \gamma$, et soit $\gamma' = \mathtt{P}$ et $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{S}([\mathsf{E}']_{\mathsf{c}'}^{\mathsf{T}'}) \equiv \mathsf{T}$, soit $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{T}'' <: \mathsf{T}$. La règle structurelle est forcément (et.col), donc $\mathsf{T}'' = \mathsf{T}'$ et une prémisse est $\mathsf{B};\Gamma \vdash_{\mathsf{c}'} \mathsf{E}' :^{\gamma'} \mathsf{T}'$. Les deux cas du Lem. B.2.18 (analyse du dernier type d'une expression) donnent les deux cas du présent lemme. □

B.2.5 Typage des valeurs

Lemme B.2.24 (pureté des valeurs) Si $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{V} :^{\gamma} \mathsf{T}$ et V est une quasi-valeur alors $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{V} :^{\mathtt{P}} \mathsf{T}$.

Démonstration. Par induction sur le jugement de typage. Si la dernière règle est (et.sub) ou (et.sing), le résultat est trivial par induction (sachant pour (et.sub) que $\mathtt{P}$ est l'annotation d'effet minimale). Si la dernière règle est structurelle, on vérifie que toutes les règles de typage possibles (règles correspondant aux constructeurs, et (et.col)) ont l'annotation d'effet $\mathtt{P}$ dans leur conclusion si elle est $\mathtt{P}$ dans les prémisses, ce qui donne le résultat par induction. □

Lemme B.2.25 (conversion d'une forme normale de tête faible) Si E a un constructeur en tête (si $\mathsf{E} = [\mathsf{E}_0]_{\mathsf{c}_0}^{\mathsf{T}_0}$, nous supposons que T_0 est un type abstrait opaque dans la couleur ambiante c) et que $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{E} \longrightarrow \mathsf{E}'$ alors E' a un constructeur en tête qui est soit le même, soit éventuellement celui imposé par une êta-expansion en tête si la conversion $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{E} \longrightarrow \mathsf{E}'$ est directement dérivée par une règle (econv.eta.*).

Démonstration. Examen direct des règles (econv.*). □

En fait, le constructeur en tête reste toujours le même, mais la démonstration dans le cas des êta-expansions repose sur un lemme de cohérence du typage dont nous ne disposons pas encore.

Lemme B.2.26 (seule l'unité est l'unité) Si $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{V}^{\mathsf{c}} \equiv ()$ alors $\mathsf{V}^{\mathsf{c}} = ()$.

Démonstration. L'observation des règles $\mathsf{(econv.*)}$ montre que $()$ est une forme normale pour la conversion. Par le Cor. B.2.9 (convertibilité par étapes confluentes), on a donc $B; \Gamma \vdash_c V^c \longrightarrow^* ()$. Une valeur a un constructeur en tête au sens du Lem. B.2.25 (conversion d'une forme normale de tête faible), donc (par une récurrence triviale) ses réduits successifs ont ce même constructeur en tête, sauf à ce qu'une êta-expansion impose un constructeur différent. Or le constructeur $()$ ne peut pas être imposé par une êta-conversion, il provient donc de V^c, ce qui signifie que $V^c = ()$. □

Lemme B.2.27 (forme des valeurs) Si $B; \Gamma \vdash_c V^c :^\gamma T$ alors :

- si $T = \textsc{unit}$, $T = \textsc{bool}$ ou $T = \textsc{int}$ alors V^c est une constante de base du type correspondant.
- si $T = \textsc{type}^K$ alors il existe T_0 tel que $V^c = \langle T_0 \rangle$ et $B; \Gamma \vdash_c T_0 : K$;
- si $T = \Sigma x : T_1.\ T_2$ alors il existe V_1^c et V_2^c tels que $V^c = (V_1^c, V_2^c)$, $B; \Gamma \vdash_c V_1^c :^P T_1$ et $B; \Gamma \vdash_c V_2^c :^P \{x \leftarrow_c V_1^c\} T_2$;
- si $T = \Pi x : T_0.\ {}^{\gamma'} T_1$ alors il existe T_2, T_3 et E_1 tels que $V^c = \lambda x : T_2.\ E_1$, $B; \Gamma \vdash_c T_0 <: T_2$, $B; \Gamma, x :_c T_0 \vdash_c T_3 <: T_1$ et $B; \Gamma, x :_c T_2 \vdash_{c \cup \{x\}} E_1 :^{\gamma'} T_3$;
- si $T = \textsc{dyn}$ alors il existe V_0^c et T_0 tels que $V^c = \mathsf{dynned}\ V_0^c\ \mathsf{at}\ T_0$, et $B; \Gamma \vdash_c V_0^c :^P T_0$;
- si $T = (\!|A|\!)$ avec A abstraite dans c alors il existe c', $V^{c'}$ et A' tels que $V^c = [V^{c'}]_{c'}^{(\!|A'|\!)}$, et A' est transparente dans c' mais pas dans c.

Démonstration. D'après le Lem. B.2.24 (pureté des valeurs), on peut supposer sans perte de généralité $\gamma = P$.

D'après le Lem. B.2.18 (analyse du dernier type d'une expression), il existe T' tel que $B; \Gamma \vdash_c V^c :^P T'$ par une règle structurelle, et soit $B; \Gamma \vdash_c S(V^c) \equiv T$, soit $B; \Gamma \vdash_c T' <: T$.

Traitons d'abord le cas où $B; \Gamma \vdash_c S(V^c) \equiv T$. Au vu du Lem. B.2.12 (têtes de types équivalents), T n'étant dans aucun des cas du présent lemme un singleton, il ne peut être équivalent à un singleton que si $T = \textsc{unit}$, auquel cas $B; \Gamma \vdash_c V^c \equiv ()$. Par le Lem. B.2.26 (seule l'unité est l'unité), $V^c = ()$.

Nous supposons désormais que $B; \Gamma \vdash_c T' <: T$. Le fait que V^c est une valeur limite les règles structurelles pour la typer. Nous allons examiner les règles possibles.

Règles $\mathsf{(et.base.*)}$: V^c est une constante et T' est le type de base correspondant D'après le Lem. B.2.16 (têtes de types comparables), $T = T'$.

Règle $\mathsf{(et.col)}$: $V^c = [V^{c'}]_{c'}^{(\!|A'|\!)}$ et $T' = (\!|A'|\!)$ Des prémisses sont $B; \Gamma \vdash_{c'} V^{c'} :^P (\!|A'|\!)$ et $B; \Gamma \vdash_{c \cap c'} (\!|A'|\!) : \wr$. Comme V^c est une valeur, A' est transparente dans c' mais opaque dans c donc également dans $c \cap c'$. Par le Lem. B.2.16 (têtes de types comparables), on a $T = (\!|A|\!)$ et $B; \Gamma \vdash_c (\!|A|\!) \equiv (\!|A'|\!)$.

Règle $\mathsf{(et.dynned)}$: $V^c = \mathsf{dynned}\ V_0^c\ \mathsf{at}\ T_0$ et $T' = \textsc{dyn}$ D'après le Lem. B.2.16 (têtes de types comparables), $T = \textsc{dyn}$. La prémisse est $B; \Gamma \vdash_c V_0^c :^P T_0$.

Règle $\mathsf{(et.fun)}$: $V^c = \lambda x : T_2.\ E_1$ et $T' = \Pi x : T_2.\ {}^{\gamma''} T_3$ D'après le Lem. B.2.16 (têtes de types comparables), il existe T_0, γ' et T_1 tels que $T = \Pi x : T_0.\ {}^{\gamma'} T_1$, $B; \Gamma \vdash_c T_0 <: T_2$, $\gamma'' \sqsubseteq \gamma'$ et $B; \Gamma, x :_c T_0 \vdash_c T_3 <: T_1$. La prémisse de $\mathsf{(et.fun)}$ est $B; \Gamma, x :_c T_2 \vdash_{c \cup \{x\}} E_1 :^{\gamma''} T_3$. Par Lem. B.1.33 (validité), $\mathsf{(teq.refl)}$, $\mathsf{(tsub.eq)}$ et $\mathsf{(et.sub)}$, on a $B; \Gamma, x :_c T_2 \vdash_{c \cup \{x\}} E_1 :^{\gamma'} T_3$.

Règle $\mathsf{(et.pair)}$: $V^c = (V_1, V_2)$ et $T' = T_1' * T_2'$ D'après le Lem. B.2.16 (têtes de types comparables), il existe T_1 et T_2 tels que $T = \Sigma x : T_1.\ T_2$, $B; \Gamma \vdash_c T_1' <: T_1$ et $B; \Gamma, x :_c T_1' \vdash_{c \cup \{x\}} T_2' <: T_2$. Les prémisses de $\mathsf{(et.pair)}$ sont $B; \Gamma \vdash_c V_1 :^P T_1'$ et $B; \Gamma \vdash_c V_2 :^P T_2'$. Par $\mathsf{(et.sub)}$, on a $B; \Gamma \vdash_c V_1 :^P T_1$. Par le Th. B.1.27 (préservation du typage par substitution), on a $B; \Gamma \vdash_c T_2' <: \{x \leftarrow_c V_1\} T_2$, d'où $B; \Gamma \vdash_c V_2 :^P T_2$ par $\mathsf{(et.sub)}$.

Règle $\mathsf{(et.type)}$: $V^c = \langle T_0 \rangle$ et $T' = \textsc{type}^{K'}$ D'après le Lem. B.2.16 (têtes de types comparables), $T = \textsc{type}^K$ avec $K' \leqslant K$. La prémisse est $B; \Gamma \vdash_c T_0 : K'$. On a $B; \Gamma \vdash_c T_0 : K$ par $\mathsf{(tok.sub)}$.

□

B.3 Sûreté

B.3.1 Opérations sur les types

Lemme B.3.1 (révélation d'un type abstrait) Si $B; \Gamma \vdash_c A \triangleright E : T$ alors $E = \mathbf{reveal}^B(A)$.
Si $B; \Gamma \vdash_c (\!|A|\!) : K$ alors il existe K' tel que $K' \leqslant K$ et $B; \Gamma \vdash_c A \triangleright \mathbf{reveal}^B(A) : \textsc{type}^{K'}$.

Démonstration. La première affirmation procède d'une récurrence triviale sur les règles (ac.*), la structure de l'expression suivant la définition de $\mathbf{reveal}^B(A)$.

La seconde affirmation est un corollaire de la première, compte tenu du fait que $B; \Gamma \vdash_c (\!|A|\!) : K$ est dérivé par la règle (tok.abs) suivie éventuellement d'applications de la règle (tok.sub) qui ne font qu'augmenter la sorte. □

Lemme B.3.2 (concrétisation) Supposons $c' \subseteq c$. Si $B; \Gamma \vdash_c E :^P T$ alors $B; \Gamma \vdash_c E \equiv \mathbf{conc}^B_{c'}(E)$.
Si $B; \Gamma \vdash_c T : K$ alors $B; \Gamma \vdash_c T \equiv \mathbf{conc}^B_{c'}(T)$.

Démonstration. Par induction sur la dérivation de correction de E ou T. Dans la plupart des cas, E est assemblée à partir de parties pour lesquelles l'hypothèse de récurrence plus l'application de règles de contexte donne le résultat souhaité.

Le cas non trivial est celui d'un type abstrait $(\!|A|\!)$ transparent, typé par la règle (tok.abs). On a alors $A \triangleright \mathbf{reveal}^B(A) : \textsc{type}^{\wr}$ par le Lem. B.3.1 (révélation d'un type abstrait). Comme $(\!|A|\!)$ est transparent, on a $B; \Gamma \vdash_c (\!|A|\!) \longrightarrow \mathsf{Typ}\,\mathbf{reveal}^B(A)$ par (tconv.abs), d'où le résultat souhaité $B; \Gamma \vdash_c (\!|A|\!) \equiv \mathsf{Typ}\,\mathbf{reveal}^B(A)$ par (teq.conv). □

Conjecture B.3.3 (renforcement dépendant) Supposons $B; \Gamma \vdash_c A \triangleright E : \Sigma x : T_1.\, T_2$.
Si $B; \Gamma, x :_c S(\pi_1 E) \vdash_{\{x\}} \mathbf{self}^{T_2}(\pi_2 A) : \wr$ alors $B; \Gamma, x :_c \mathbf{self}^{T_1}(\pi_1 A) \vdash_{\{x\}} \mathbf{self}^{T_2}(\pi_2 A) : \wr$.
Si $B; \Gamma, x :_c S(\pi_1 E) \vdash_{\{x\}} \mathbf{self}^{T_2}(\pi_2 A) <: T_2$ alors $B; \Gamma, x :_c \mathbf{self}^{T_1}(\pi_1 A) \vdash_{\{x\}} \mathbf{self}^{T_2}(\pi_2 A) <: T_2$.

Justification. Les affirmations sont triviales dans le cas particulier où $\mathbf{self}^{T_1}(\pi_1 A) = S(E_1)$ avec $B; \Gamma \vdash_c E_1 \equiv \pi_1 E$ — ce qui est toujours le cas si T_1 indique un champ type (soit concret sous la forme d'un singleton caractérisant son contenu $\pi_1 E$, soit abstrait auquel cas on a carrément $\mathbf{self}^{T_1}(\pi_1 A) = S(E_1)$).

Les affirmations sont également faciles à démontrer dans le cas particulier d'un produit ordinaire (c'est-à-dire que $x \notin \mathbf{fv}\, T_2$). Il suffit d'utiliser le Th. B.1.27 (préservation du typage par substitution) et le Lem. B.1.20 (affaiblissement de l'environnement) pour supprimer puis rétablir l'hypothèse sur x.

Dans le cas général, il s'agit d'un renforcement parallèle de la conclusion et de l'hypothèse sur x. Cette conjecture demande une certaine compatibilité entre la définition du renforcement et les capacités du système de types. Le renforcement n'affecte que les champs types, à l'exclusion des valeurs des types de base. La propriété voulue repose sur le fait que la valeur d'un champ type ne peut influencer le typage d'un champ valeur ; par exemple, il n'existe pas de fonction non constante de $\textsc{type}^K$ vers $\textsc{bool}$. □

Lemme B.3.4 (propriétés du renforcement) Si $B; \Gamma \vdash_c A \triangleright E : T$ alors $B; \Gamma \vdash_c \mathbf{self}^T(A) : \wr$ et $B; \Gamma \vdash_c \mathbf{self}^T(A) <: T$. Si de plus $B; \Gamma \vdash_c \mathbf{underl}(A)\ \mathsf{transparent}$ alors $B; \Gamma \vdash_c E :^P \mathbf{self}^T(A)$.

Démonstration. Posons $a = \mathbf{underl}(A)$. On a $B; \Gamma \vdash_c E :^P T$ et $B; \Gamma \vdash_c T : K$ par le Lem. B.1.33 (validité), et $B; \Gamma \vdash_c \mathsf{ok}$ par le Lem. B.1.6 (correction de l'environnement). Nous raisonnons par induction sur la structure de T, et à T fixé sur la dérivation de $B; \Gamma \vdash_c T : K$.

Règles (tok.abs), (tok.base.*), (tok.fun.I), (tok.sing) **:**
Dans tous ces cas, on a $\textbf{self}^{\,\mathsf{T}}(A) = \mathsf{T}$. On a $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{T} : \wr$ en vérifiant dans chaque cas que la règle (tok.*) invoquée donne ce résultat, $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{T} <: \mathsf{T}$ par (teq.refl) et (tsub.eq), et enfin $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{E} :^{\mathsf{P}} \mathsf{T}$ a déjà été vu.

Règle (tok.field) — $\mathsf{T} = \mathsf{Typ}\,\mathsf{E}_1$ **:**
Si $\mathsf{E}_1 = \langle \mathsf{T}_1 \rangle$, alors $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{T} \longrightarrow \mathsf{T}_1$ par (tconv.field). La récurrence permet de conclure grâce aux règles (teq.*), (tsub.eq) et (et.sub).

Si E_1 est irréductible, alors $\textbf{self}^{\,\mathsf{T}}(A) = \mathsf{T}$ comme dans le cas précédent.

Règle (tok.fun.P) — $\mathsf{T} = \Pi x : \mathsf{T}_0.\,{}^{\mathsf{P}}\mathsf{T}_1$ **:**
Les prémisses sont $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{T}_0 : \mathsf{K}_0$ et $\mathsf{B};\Gamma, x :_{\mathsf{c}} \mathsf{T}_0 \vdash_{\mathsf{c}\cup\{x\}} \mathsf{T}_1 : \mathsf{K}_1$; on a $\textbf{self}^{\,\mathsf{T}}(A) = \Pi x : \mathsf{T}_0.\,{}^{\mathsf{P}}\textbf{self}^{\,\mathsf{T}_1}(A\,x)$.

Par le Lem. B.1.12 (correction de la couleur et de l'environnement), le Lem. B.1.17 (dénotation de la transparence) (et.x), on a $\mathsf{B};\Gamma, x :_{\mathsf{c}} \mathsf{T}_0 \vdash_{\{x\}} x :^{\mathsf{P}} \mathsf{T}_0$. Par le Lem. B.1.6 (correction de l'environnement) et le Lem. B.1.20 (affaiblissement de l'environnement), on a $\mathsf{B};\Gamma, x :_{\mathsf{c}} \mathsf{T}_0 \vdash_{\mathsf{c}} A \triangleright \mathsf{E} : \Pi x : \mathsf{T}_0.\,{}^{\mathsf{P}}\mathsf{T}_1$. Par le Lem. B.1.17 (dénotation de la transparence) et le Lem. B.1.23 (affaiblissement de la couleur), on a $\mathsf{B};\Gamma, x :_{\mathsf{c}} \mathsf{T}_0 \vdash_{\{x\}} A \triangleright \mathsf{E} : \Pi x : \mathsf{T}_0.\,{}^{\mathsf{P}}\mathsf{T}_1$. On a donc $\mathsf{B};\Gamma, x :_{\mathsf{c}} \mathsf{T}_0 \vdash_{\{x\}} A\,x \triangleright \mathsf{E}\,x : \mathsf{T}_1$ par (ac.app), puis $\mathsf{B};\Gamma, x :_{\mathsf{c}} \mathsf{T}_0 \vdash_{\mathsf{c}\cup\{x\}} A\,x \triangleright \mathsf{E}\,x : \mathsf{T}_1$ par le Lem. B.1.23 (affaiblissement de la couleur).

Par induction, on obtient $\mathsf{B};\Gamma, x :_{\mathsf{c}} \mathsf{T}_0 \vdash_{\mathsf{c}\cup\{x\}} \textbf{self}^{\,\mathsf{T}}(A\,x) : \wr$ et $\mathsf{B};\Gamma, x :_{\mathsf{c}} \mathsf{T}_0 \vdash_{\mathsf{c}\cup\{x\}} \textbf{self}^{\,\mathsf{T}}(A\,x) <: \mathsf{T}$. On en déduit $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \textbf{self}^{\,\mathsf{T}}(A) : \wr$ par (tok.fun.P). De plus, on a $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{T}_0 <: \mathsf{T}_0$ par (teq.refl) et (tsub.eq), d'où $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \textbf{self}^{\,\mathsf{T}}(A) <: \mathsf{T}$ par (tsub.cong.fun).

Si de plus $\mathsf{B};\Gamma \vdash_{\mathsf{c}} a\,\mathsf{transparent}$ alors $\mathsf{B};\Gamma, x :_{\mathsf{c}} \mathsf{T}_0 \vdash_{\mathsf{c}\cup\{x\}} a\,\mathsf{transparent}$ par le Lem. B.1.20 (affaiblissement de l'environnement) et le Lem. B.1.23 (affaiblissement de la couleur), donc l'induction donne également $\mathsf{B};\Gamma, x :_{\mathsf{c}} \mathsf{T}_0 \vdash_{\mathsf{c}\cup\{x\}} \mathsf{E}\,x :^{\mathsf{P}} \textbf{self}^{\,\mathsf{T}}(A\,x)$. Par (et.fun), on a $\mathsf{B};\Gamma \vdash_{\mathsf{c}} (\lambda x : \mathsf{T}_0.\ \mathsf{E}\,x) :^{\mathsf{P}} \textbf{self}^{\,\mathsf{T}}(A)$. Sachant que $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{E} \longrightarrow (\lambda x : \mathsf{T}_0.\ \mathsf{E}\,x)$ par (econv.eta.fun), on a $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{E} :^{\mathsf{P}} \textbf{self}^{\,\mathsf{T}}(A)$ par le Lem. B.1.35 (préservation du typage par conversion).

Règle (tok.pair) — $\mathsf{T} = \Sigma x : \mathsf{T}_1.\,\mathsf{T}_2$ **:**
Les prémisses sont $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{T}_1 : \mathsf{K}_1$ et $\mathsf{B};\Gamma, x :_{\mathsf{c}} \mathsf{T}_1 \vdash_{\mathsf{c}\cup\{x\}} \mathsf{T}_2 : \mathsf{K}_2$; on a $\textbf{self}^{\,\mathsf{T}}(A) = \Sigma x : \textbf{self}^{\,\mathsf{T}_1}(\pi_1 A).\,\textbf{self}^{\,\mathsf{T}_2}(\pi_2 A)$.

Par (ac.proj.1), on a $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \pi_1 A \triangleright \pi_1 \mathsf{E} : \mathsf{T}_1$. Par récurrence, on a $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \textbf{self}^{\,\mathsf{T}_1}(\pi_1 A) : \wr$ et $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \textbf{self}^{\,\mathsf{T}_1}(\pi_1 A) <: \mathsf{T}_1$, ainsi que $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \pi_1 \mathsf{E} :^{\mathsf{P}} \textbf{self}^{\,\mathsf{T}_1}(\pi_1 A)$ si a est transparent.

Par (et.proj.1), on a $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \pi_1 \mathsf{E} :^{\mathsf{P}} \mathsf{T}_1$. Par (tsub.sing), on a $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \mathsf{S}(\pi_1 \mathsf{E}) <: \mathsf{T}_1$. Par (tok.sing), (envok.x), le Lem. B.1.23 (affaiblissement de la couleur) et (envok.c.x), on a $\mathsf{B};\Gamma, x :_{\mathsf{c}} \mathsf{S}(\pi_1 \mathsf{E}) \vdash_{\mathsf{c}\cup\{x\}} \mathsf{ok}$. Par le Lem. B.1.20 (affaiblissement de l'environnement), le Lem. B.1.23 (affaiblissement de la couleur) et le Cor. B.1.26 (nommage d'une couleur), on a $\mathsf{B};\Gamma, x :_{\mathsf{c}} \mathsf{S}(\pi_1 \mathsf{E}) \vdash_{\{x\}} A \triangleright \mathsf{E} : \Sigma x : \mathsf{T}_1.\,\mathsf{T}_2$, ainsi que $\mathsf{B};\Gamma, x :_{\mathsf{c}} \mathsf{S}(\pi_1 \mathsf{E}) \vdash_{\{x\}} \textbf{underl}(A)\,\mathsf{transparent}$ si $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \textbf{underl}(A)\,\mathsf{transparent}$. Par (et.x), on a $\mathsf{B};\Gamma, x :_{\mathsf{c}} \mathsf{S}(\pi_1 \mathsf{E}) \vdash_{\{x\}} x :^{\mathsf{P}} \mathsf{S}(\pi_1 \mathsf{E})$. Par (ac.proj.2), on a $\mathsf{B};\Gamma, x :_{\mathsf{c}} \mathsf{S}(\pi_1 \mathsf{E}) \vdash_{\{x\}} \pi_2 A \triangleright \pi_2 \mathsf{E} : \mathsf{T}_2$ (sachant que $\{x \leftarrow_{\{x\}} x\}\mathsf{T}_2 = \mathsf{T}_2$).

Par récurrence, on obtient $\mathsf{B};\Gamma, x :_{\mathsf{c}} \mathsf{S}(\pi_1 \mathsf{E}) \vdash_{\{x\}} \textbf{self}^{\,\mathsf{T}_2}(\pi_2 A) : \wr$ et $\mathsf{B};\Gamma, x :_{\mathsf{c}} \mathsf{S}(\pi_1 \mathsf{E}) \vdash_{\{x\}} \textbf{self}^{\,\mathsf{T}_2}(\pi_2 A) <: \mathsf{T}_2$, ainsi que $\mathsf{B};\Gamma, x :_{\mathsf{c}} \mathsf{S}(\pi_1 \mathsf{E}) \vdash_{\{x\}} \pi_2 \mathsf{E} :^{\mathsf{P}} \textbf{self}^{\,\mathsf{T}_2}(\pi_2 A)$ si $\textbf{underl}(A)$ est transparent.

Par la Conj. B.3.3 (renforcement dépendant), on a $\mathsf{B};\Gamma, x :_{\mathsf{c}} \textbf{self}^{\,\mathsf{T}_1}(\pi_1 A) \vdash_{\{x\}} \textbf{self}^{\,\mathsf{T}_2}(\pi_2 A) : \wr$ et $\mathsf{B};\Gamma, x :_{\mathsf{c}} \textbf{self}^{\,\mathsf{T}_1}(\pi_1 A) \vdash_{\{x\}} \textbf{self}^{\,\mathsf{T}_2}(\pi_2 A) <: \mathsf{T}_2$, ainsi que $\mathsf{B};\Gamma, x :_{\mathsf{c}} \textbf{self}^{\,\mathsf{T}_1}(\pi_1 A) \vdash_{\{x\}} \pi_2 \mathsf{E} :^{\mathsf{P}} \textbf{self}^{\,\mathsf{T}_2}(\pi_2 A)$ si $\textbf{underl}(A)$ est transparent.

Par (tok.pair), on a $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \Sigma x : \textbf{self}^{\,\mathsf{T}_1}(\pi_1 A).\,\textbf{self}^{\,\mathsf{T}_2}(\pi_2 A) : \wr$, c'est-à-dire $\mathsf{B};\Gamma \vdash_{\mathsf{c}} \textbf{self}^{\,\mathsf{T}}(A) : \wr$.

Par (tsub.cong.pair), on a $B; \Gamma \vdash_c \Sigma x : \textbf{self}^{T_1}(\pi_1 A).\ \textbf{self}^{T_2}(\pi_2 A) <: \Sigma x : T_1.\ T_2$, c'est-à-dire $B; \Gamma \vdash_c \textbf{self}^{T}(A) <: T$.

Si $\textbf{underl}(A)$ est transparent, on a $B; \Gamma \vdash_c S(\pi_1 E) <: T_1'$ ainsi que $B; \Gamma, x :_c S(\pi_1 E) \vdash_{c \cup \{x\}} S(\pi_2 E) <: \textbf{self}^{T_2}(\pi_2 A)$ par (tsub.sing) (et par affaiblissemnt de la couleur $\{x\}$ et $c \cup \{x\}$ dans le deuxième cas). On a de plus $B; \Gamma, x :_c \textbf{self}^{T_1}(\pi_1 A) \vdash_{\{x\}} \textbf{self}^{T_2}(\pi_2 A) : *$ par (tok.sub). Par (tsub.cong.pair), on obtient $B; \Gamma \vdash_c S(\pi_1 E) * S(\pi_2 E) <: \textbf{self}^{T}(A)$. Par le Cor. B.1.24 (extraction d'un lexique), on a $B; \Gamma \vdash_c E :^P \Sigma x : T_1.\ T_2$. Par (et.proj.*) et (et.sing), on a $B; \Gamma \vdash_c \pi_i E :^P S(\pi_i E)$ pour $i \in \{1, 2\}$, d'où $B; \Gamma \vdash_c (\pi_1 E, \pi_2 E) :^P \textbf{self}^{T}(A)$ par (et.pair) et (et.sub). Par (econv.eta.pair) on a $B; \Gamma \vdash_c E \longrightarrow (\pi_1 E, \pi_2 E)$. Par le Lem. B.1.35 (préservation du typage par conversion), on obtient enfin $B; \Gamma \vdash_c E :^P \textbf{self}^{T}(A)$.

Règle (tok.type) — $T = \text{TYPE}^{K_1}$ **:**

Alors $\textbf{self}^{T}(A) = S(\langle (\!| A |\!) \rangle)$. On a $B; \Gamma \vdash_c \langle (\!| A |\!) \rangle :^P \text{TYPE}^{\wr}$ par (tok.abs) et (et.type). Alors $B; \Gamma \vdash_c S(\langle (\!| A |\!) \rangle) : \wr$ par (tok.sing) ; de plus $B; \Gamma \vdash_c S(\langle (\!| A |\!) \rangle) <: \text{TYPE}^{\wr}$ par (tsub.sing), d'où $B; \Gamma \vdash_c S(\langle (\!| A |\!) \rangle) <: \text{TYPE}^{K_1}$ par (tsub.cong.type) et (tsub.trans).

Si de plus a est transparent alors $B; \Gamma \vdash_c \langle (\!| A |\!) \rangle \longrightarrow \langle \mathsf{Typ}\ E \rangle$ par (tconv.abs) et (econv.cong.field). On a d'autre part $B; \Gamma \vdash_c E \longrightarrow \langle \mathsf{Typ}\ E \rangle$ par (econv.eta.field), d'où $B; \Gamma \vdash_c E \equiv \langle (\!| A |\!) \rangle$ par (eeq.*) puis $B; \Gamma \vdash_c E :^P S(\langle (\!| A |\!) \rangle)$ par (tsub.eq), (et.sing) et (et.sub).

Règle (tok.sub) :

Trivial par récurrence.

□

B.3.2 Préservation du typage par réduction

Lemme B.3.5 (réduction des expressions pures) Si $B; \mathsf{nil} \vdash_c E :^P T$ et $B \vdash E \longrightarrow_c B \vdash E'$ alors $B; \mathsf{nil} \vdash_c E \equiv E'$.

Démonstration. Par induction sur la dérivation du jugement de typage ; celle-ci étant fixée, par induction sur la dérivation du jugement de réduction. Notons que $B; \mathsf{nil} \vdash_c T : *$ par le Lem. B.1.33 (validité), et $B; \mathsf{nil} \vdash_c \mathsf{ok}$ par le Lem. B.1.12 (correction de la couleur et de l'environnement).

Dans la plupart des cas, on démontre que $B; \mathsf{nil} \vdash_c E \longrightarrow E'$, ce qui suffit (appliquer (eeq.conv)).

Si le jugement de typage est obtenu par (et.sub) ou (et.sing), le résultat est trivial par induction ; pour (et.sub), il convient de remarquer que si $\gamma \sqsubseteq P$ alors $\gamma = P$. Sinon, en vertu de la Rem. B.2.17 (règles structurelles du typage des expressions), le jugement de typage est obtenu par une règle structurelle ; dans ce cas, nous discriminons suivant la règle de réduction utilisée. Les règles (econv.*) ayant été modelées sur les règles (ered.*), il s'agit essentiellement de vérifier dans chaque cas que les hypothèses de la règle de conversion correspondante sont vérifiées.

Règle (ered.app) : On a $E = (\lambda x{:}T_0.\ E_1)\ E_0$, et le jugement de typage utilise (et.app) avec les prémisses $B; \mathsf{nil} \vdash_c (\lambda x : T_0.\ E_1) :^P \Pi x : T_2.\ ^P T_1$ et $B; \mathsf{nil} \vdash_c E_0 :^P T_2$. D'après le Lem. B.2.21 (inversion du typage d'une fonction), on a $B; \mathsf{nil} \vdash_c T_2 <: T_0$ et $B; x :_c T_0 \vdash_{c \cup \{x\}} E_1 :^P T_2$. D'après (et.sub), $B; \mathsf{nil} \vdash_c E_0 :^P T_0$. La règle (econv.app) donne le résultat souhaité.

Règle (ered.colTyp) : On a $E = [V_1]_{c_1}^{\mathsf{Typ}\,\langle T_1 \rangle}$, et le jugement de typage utilise (et.col) avec les prémisses $B; \mathsf{nil} \vdash_{c_1} V_1 :^P \mathsf{Typ}\,\langle T_1 \rangle$ et $B; \mathsf{nil} \vdash_{c \cap c_1} \mathsf{Typ}\,\langle T_1 \rangle : \wr$. Par (tconv.field), on a $B; \mathsf{nil} \vdash_{c \cap c_1} \mathsf{Typ}\,\langle T_1 \rangle \longrightarrow T_1$. Par le Lem. B.1.13 (correction d'une sous-couleur), $B; \Gamma \vdash_{c \cap c_1} \mathsf{ok}$. Par le Lem. B.1.23 (affaiblissement de la couleur), $B; \Gamma \vdash_{c \cap c_1} \mathsf{Typ}\,\langle T_1 \rangle \longrightarrow T_1$. Finalement $B; \mathsf{nil} \vdash_c E \longrightarrow E'$ par (econv.cong.col.t).

Règle (ered.colAbs) : On a $E = [V_1]_{c_1}^{(\!|A|\!)}$, et le jugement de typage utilise (et.col) avec les prémisses $B; \mathsf{nil} \vdash_{c_1} V_1 :^{P} (\!|A|\!)$ et $B; \mathsf{nil} \vdash_{c \cap c_1} (\!|A|\!) : \wr$. Par le Lem. B.3.1 (révélation d'un type abstrait), on a $B; \mathsf{nil} \vdash_{c \cap c_1} A \triangleright \mathbf{reveal}^{B}(A) : \mathrm{TYPE}^{\wr}$.

On sait de plus que $\mathbf{underl}(A) \in c \cap c_1$. Par le Lem. B.1.17 (dénotation de la transparence), on a $B; \mathsf{nil} \vdash_{c \cap c_1} \mathbf{underl}(A)\,\mathsf{transparent}$, ce qui permet d'appliquer (tconv.abs) pour obtenir $B; \mathsf{nil} \vdash_{c \cap c_1} (\!|A|\!) \longrightarrow \mathsf{Typ}\,\mathbf{reveal}^{B}(A)$. Par (econv.cong.col.t), on a $B; \mathsf{nil} \vdash_{c} E \longrightarrow [V_1]_{c_1}^{\mathsf{Typ}\,\mathbf{reveal}^{B}(A)}$.

Règle (ered.let) : Impossible : le radical est typé par (et.let) et ne peut pas être pur.

Règle (ered.proj) : On a $E = \pi_i(V_1, V_2)$, et le jugement de typage utilise (et.proj.1) ou (et.proj.2) avec la prémisse $B; \mathsf{nil} \vdash_{c} (V_1, V_2) :^{P} \Sigma x : T_1.\, T_2$. D'après le Lem. B.2.20 (inversion du typage d'une paire) et (et.sub), on a $B; \mathsf{nil} \vdash_{c} V_1 :^{P} T_1$ et $B; \mathsf{nil} \vdash_{c} V_2 :^{P} T_3$ pour un certain T_3. La règle (econv.proj) donne le résultat souhaité.

Règle (ered.seal) : Impossible : le radical est typé par (et.seal) et ne peut pas être pur.

Règle (ered.undyn) : Impossible : le radical est typé par (et.undyn) et ne peut pas être pur.

Règles (ered.col.*) : On a $E = [E_0]_{c_0}^{T}$, et le jugement de typage utilise (et.col) avec les prémisses $B; \mathsf{nil} \vdash_{c_0} E_0 :^{P} T$ et $B; \mathsf{nil} \vdash_{c \cap c_0} T : \wr$. La suite de l'analyse dépend de la règle de réduction.

Règles (ered.col.base.*) : La règle (econv.col.base.*) correspondante donne le résultat souhaité.

Règle (ered.col.merge) : On a $E_0 = [V_1]_{c_1}^{\mathsf{Typ}\,\langle \mathsf{Typ}\,A_1 \rangle}$ et $T = \mathsf{Typ}\,\langle \mathsf{Typ}\,A_0 \rangle$, avec A_0 et A_1 toutes deux opaques dans c_0, et A_1 transparente dans c_1. D'après le Lem. B.2.23 (inversion du typage d'un crochet coloré), on a $B; \mathsf{nil} \vdash_{c_1} V_1 :^{P} \mathsf{Typ}\,\langle \mathsf{Typ}\,A_1 \rangle$ et $B; \mathsf{nil} \vdash_{c_0} \mathsf{Typ}\,\langle \mathsf{Typ}\,A_1 \rangle <: \mathsf{Typ}\,\langle \mathsf{Typ}\,A_0 \rangle$. On a également $B; \mathsf{nil} \vdash_{c_1 \cap c_0} A_1 :^{P} \mathrm{TYPE}^{\wr}$ par le Lem. B.2.19 (inversion du typage d'un champ type), ainsi que $B; \mathsf{nil} \vdash_{c \cap c_1} A_0 :^{P} \mathrm{TYPE}^{\wr}$ par un raisonnement analogue. La règle (econv.col.merge) donne le résultat souhaité.

Règle (ered.col.dynned) : On a $E_0 = \mathsf{dynned}\,E_1\,\mathsf{at}\,T_1$ et $T = \mathrm{DYN}$. D'après le Lem. B.2.22 (inversion du typage d'un dynamique), $B; \mathsf{nil} \vdash_{c_0} E_1 :^{P} T_1$. La règle (econv.col.dynned) donne le résultat souhaité.

Règles (ered.col.fun.I), (ered.col.fun.P) : On a $E_0 = \lambda x' : T_2.\, E_1$ et $T = \Pi y : T_0.\, {}^{\gamma_1} T_1$. D'après le Lem. B.2.21 (inversion du typage d'une fonction), $B; x' :_{c_0} T_2 \vdash_{c_0 \cup \{x\}} E_1 :^{P} \{y \leftarrow_{\{x'\}} x'\} T_1$ et $B; \mathsf{nil} \vdash_{c_0} T_0 <: T_2$. De plus, $B; y :_{c} T_0 \vdash_{c \cup \{x\}} T_1 : \wr$ par inversion de (tok.fun.*) dans $B; \mathsf{nil} \vdash_{c} T : \wr$. La règle (econv.col.fun.P) (si $\gamma_1 = \mathtt{P}$) ou (econv.col.fun.I) (si $\gamma_1 = \mathtt{I}$) donne le résultat souhaité.

Règle (ered.col.pair) : On a $E_0 = (V_1, V_2)$ et $T = \Sigma x : T_1.\, T_2$. En inversant (tok.pair), on a $B; \Gamma \vdash_{c} T_1 : \wr$, $B; \Gamma \vdash_{c_0} T_1 : \wr$, $B; \Gamma, x :_{c} T_1 \vdash_{c \cup \{x\}} T_2 : \wr$ et $B; \Gamma, x :_{c_0} T_1 \vdash_{c_0 \cup \{x\}} T_2 : \wr$. D'après le Lem. B.2.20 (inversion du typage d'une paire), on a $B; \mathsf{nil} \vdash_{c_0} V_1 :^{P} T_1'$ et $B; \mathsf{nil} \vdash_{c_0} V_2 :^{P} T_2'$ avec $B; x :_{c_0} T_1' \vdash_{c_0 \cup \{x\}} T_2' <: T_2$. Par le Lem. B.1.6 (correction de l'environnement), le Lem. B.1.20 (affaiblissement de l'environnement) et le Lem. B.1.23 (affaiblissement de la couleur), on a $B; x :_{c_0} T_1' \vdash_{c_0 \cup \{x\}} V_2 :^{P} T_2'$, d'où $B; x :_{c_0} T_1' \vdash_{c_0 \cup \{x\}} V_2 :^{P} T_2$ par (et.sub). Par le Lem. B.1.33 (validité), on a $B; x :_{c_0} T_1' \vdash_{c_0 \cup \{x\}} T_2 : \wr$. La règle (econv.col.pair) donne le résultat souhaité.

Règle (ered.col.sing) : On a $T = S(E')$ et $B; \mathsf{nil} \vdash_{c_0} E_0 :^{P} S(E')$. Par inversion de (tok.sing), il existe un type T' tel que $B; \mathsf{nil} \vdash_{c \cap c_0} E' :^{P} T'$. La règle (econv.col.sing) donne le résultat souhaité.

Règle (ered.context) : La prémisse est $E_0 \longrightarrow_{c'} E_0'$, avec $c' = c$ sauf pour les contextes faisant intervenir des crochets colorés. Discrimine sur la forme du contexte.

Contexte $E_1 \_\_$ **:** Le jugement de typage utilise (et.app) avec les prémisses $B; \mathsf{nil} \vdash_c E_1 :^P \Pi x : T_0.\,^P T_1$ et $B; \mathsf{nil} \vdash_c E_0 :^P T_0$. Par récurrence, on obtient $B; \mathsf{nil} \vdash_c E_0 \longrightarrow E_0'$. La règle (econv.cong.app.arg) donne le résultat souhaité.

Contexte $\_\_ V_2$ **:** Même principe, avec la règle (econv.cong.app.fun).

Contexte $(\_\_, E_2)$ **:** Même principe, avec la règle de typage (et.pair) et la règle de conversion (econv.cong.pair.1).

Contexte $(V_1, \_\_)$ **:** Même principe, avec la règle de typage (et.pair) et la règle de conversion (econv.cong.pair.2).

Contexte $\pi_i \_\_$ **:** Même principe, avec la règle de typage (et.proj.1) ou (et.proj.2) et la règle de conversion (econv.cong.proj).

Contexte $\mathsf{let}\, x = \_\_ \,\mathsf{in}\, E_0 : T_0$ **:** Impossible : le radical est typé par (et.let) et ne peut pas être pur.

Contexte $\_\_ \,!!_{c_0}\, T_0$ **:** Impossible : le radical est typé par (et.seal) et ne peut pas être pur.

Contexte $[\_\_]_{c_1}^{T_0}$ **:** Même principe que le contexte $E_1 \_\_$, avec la règle de typage (et.col) et la règle de conversion (econv.cong.col.e).

Contexte $[V^{c_1}]_{c_1}^{\mathsf{Typ}\,\_\_}$ **:** La prémisse de la règle de réduction est $E_0 \longrightarrow_{c \cap c_1} E_0'$, et le jugement de typage utilise (et.col) avec les prémisses $B; \mathsf{nil} \vdash_{c_1} V^{c_1} :^P \mathsf{Typ}\, E_0$ et $B; \mathsf{nil} \vdash_{c \cap c_1} \mathsf{Typ}\, E_0 : \wr$. Par récurrence, on obtient $B; \mathsf{nil} \vdash_{c \cap c_1} E_0 \longrightarrow E_0'$. La règle (econv.cong.col.t) donne le résultat souhaité.

Contexte $\mathsf{dyn} \_\_ \,\mathsf{at}\, T_0$ **:** Impossible : le radical est typé par (et.dyn) et ne peut pas être pur.

Contexte $\mathsf{dynned} \_\_ \,\mathsf{at}\, T_0$ **:** Même principe que le contexte $E_1 \_\_$, avec la règle de typage (et.dynned) et la règle de conversion (econv.cong.dynned.e).

Contexte $\mathsf{undyn} \_\_ \,\mathsf{at}\, T_0 \,\mathsf{else}\, E_1$ **:** Impossible : le radical est typé par (et.undyn) et ne peut pas être pur.

□

Theorème B.3.6 (préservation du typage) Si $B; \mathsf{nil} \vdash_c E :^\gamma T$ et $B \vdash E \longrightarrow_c B' \vdash E'$ alors $B'; \mathsf{nil} \vdash_c E' :^\gamma T$.

Démonstration. Par induction sur la dérivation du jugement de typage ; celle-ci étant fixée, par induction sur la dérivation du jugement de réduction. Notons que $B; \mathsf{nil} \vdash_c T : *$ par le Lem. B.1.33 (validité), et $B; \mathsf{nil} \vdash_c \mathsf{ok}$ par le Lem. B.1.12 (correction de la couleur et de l'environnement).

De plus, on a $B' = B$ sauf dans le cas de la règle (et.seal) éventuellement appliquée dans un contexte ; dans ce cas le contexte doit autoriser une expression impure et B est un préfixe de B'.

Si le jugement de typage est obtenu par (et.sub), le résultat est trivial par induction.

Si $\gamma = P$, d'après le Lem. B.3.5 (réduction des expressions pures), on a $B; \mathsf{nil} \vdash_c E \longrightarrow E'$; d'après le Lem. B.1.33 (validité), $B; \mathsf{nil} \vdash_c E' :^P S(E')$; d'autre part, $B; \mathsf{nil} \vdash_c S(E') <: S(E)$ par (tconv.cong.sing), (teq.conv), (teq.sym) et (tsub.eq). On a donc $B; \mathsf{nil} \vdash_c E' :^P S(E)$ par (et.sub).

Nous supposons désormais que $\gamma \neq P$ et que la dernière règle n'est pas (et.sub). En vertu de la Rem. B.2.17 (règles structurelles du typage des expressions), le jugement de typage est obtenu par une règle structurelle. Nous discriminons suivant la règle de réduction utilisée (et dans le cas de (ered.context), suivant la forme du contexte d'évaluation).

Règle (ered.app) : On a $E = (\lambda x : T_0.\ E_1)\, E_0$, $E' = \{x \leftarrow_c E_0\} E_1$, et le jugement de typage utilise (et.app) avec les prémisses $B; \mathsf{nil} \vdash_c E_1 :^{\gamma_0} \Pi x : T_2.\,^{\gamma_1} T_1$ et $B; \mathsf{nil} \vdash_c E_0 :^P T_2$ et la conclusion $B; \mathsf{nil} \vdash_c E :^{\gamma_0 \sqcup \gamma_1} \{x \leftarrow_c E_0\} T_1$. D'après le Lem. B.2.21 (inversion du typage d'une fonction), on a $B;\mathsf{nil} \vdash_c T_2 <: T_0$ et $B; x :_c T_0 \vdash_{c\cup\{x\}} E_1 :^{\gamma_1} T_2$. D'après (et.sub), $B;\mathsf{nil} \vdash_c E_0 :^{P} T_0$. Par le Th. B.1.27 (préservation du typage par substitution), on a $B;\mathsf{nil} \vdash_c \{x{\leftarrow_c}E_0\}E_1 :^{\gamma_1} \{x{\leftarrow_c}E_0\}T_1$, d'où $B;\mathsf{nil} \vdash_c \{x{\leftarrow_c}E_0\}E_1 :^{\gamma_0\sqcup\gamma_1} \{x{\leftarrow_c}E_0\}T_1$ par (et.sub).

Règles (ered.colTyp), (ered.col.*) : Le radical est une quasi-valeur, donc peut être typé pur en vertu du Lem. B.2.24 (pureté des valeurs). On a donc $B';\mathsf{nil} \vdash_c E' :^{P} T$, d'où $B';\mathsf{nil} \vdash_c E' :^{\gamma} T$ par (et.sub).

Règle (ered.dyn) : On a $E = \mathsf{dyn}\, V_0\, \mathsf{at}\, T_0$, et le jugement de typage utilise (et.dyn) avec la prémisse $B;\mathsf{nil} \vdash_c V_0 :^{\gamma} T_0$. On a même $B;\mathsf{nil} \vdash_c V_0 :^{P} T_0$ en vertu du Lem. B.2.24 (pureté des valeurs). Par le Lem. B.1.33 (validité) et le Lem. B.3.2 (concrétisation), on a $B;\mathsf{nil} \vdash_c T_0 \equiv \mathbf{conc}_c^B(T_0)$. Alors, grâce aux règles (tsub.eq) et (et.sub), on a $B;\mathsf{nil} \vdash_c V_0 :^{P} \mathbf{conc}_c^B(T_0)$. Par (et.col), on obtient $B;\mathsf{nil} \vdash_\bullet [V_0]_c^{\mathbf{conc}_c^B(T_0)} :^{P} \mathbf{conc}_c^B(T_0)$. Par (et.dynned), on a finalement $B;\mathsf{nil} \vdash_c \mathsf{dynned}\, [V_0]_c^{\mathbf{conc}_c^B(T_0)}\, \mathsf{at}\, \mathbf{conc}_c^B(T_0) :^{P} \text{DYN}$.

Règle (ered.let) : On a $E = \mathsf{let}\, x = V\, \mathsf{in}\, E_1 : T_1$, $E' = \{x{\leftarrow_c}V\}E_1$, $\gamma = \mathtt{I}$, et le jugement de typage utilise (et.let) avec les prémisses $B;\mathsf{nil} \vdash_c V :^{\gamma'} T_0$ et $B; x :_c T_0 \vdash_{c\cup\{x\}} E_1 :^{\gamma'} T_1$ et $B;\mathsf{nil} \vdash_c T_1 : *$. Par le Th. B.1.27 (préservation du typage par substitution), on a $B;\mathsf{nil} \vdash_c \{x{\leftarrow_c}V\}E_1 :^{\gamma'} \{x{\leftarrow_c}V\}T_1$, d'où $B;\mathsf{nil} \vdash_c \{x{\leftarrow_c}V\}E_1 :^{\gamma} T_1$ par (et.sub) et sachant que $\{x{\leftarrow_c}V\}T_1 = T_1$ par le Lem. B.1.1 (les variables sont liées).

Règle (ered.proj) : On a $E = \pi_i(V_1, V_2)$, $E' = V_i$, et le jugement de typage utilise (et.proj.1) ou (et.proj.2) avec la prémisse $B;\mathsf{nil} \vdash_c (V_1, V_2) :^{\gamma} \Sigma x : T_1.\, T_2$. D'après le Lem. B.2.20 (inversion du typage d'une paire), on a $B;\mathsf{nil} \vdash_c V_1 :^{\gamma} T_1$ et $B;\mathsf{nil} \vdash_c V_2 :^{\gamma} T_3$ et $B;\mathsf{nil}, x :_c T_1 \vdash_{c\cup\{x\}} T_3 <: T_2$ pour un certain T_3.

Si $i = 1$, alors $T = T_1$ et $E' = V_1$, et on a le résultat souhaité. Sinon, $i = 2$, $T = \{x{\leftarrow_c}V_1\}T_2$ et $E' = V_2$. D'après le Lem. B.2.24 (pureté des valeurs), on peut prendre $\gamma = \mathtt{P}$, donc on a $B;\mathsf{nil} \vdash_c V_1 :^{P} T_1$. Par le Lem. B.1.20 (affaiblissement de l'environnement), $B;\mathsf{nil}, x :_c T_1 \vdash_{c\cup\{x\}} V_2 :^{\gamma} T_3$, d'où $B;\mathsf{nil}, x :_c T_1 \vdash_{c\cup\{x\}} V_2 :^{\gamma} T_2$. Par le Th. B.1.27 (préservation du typage par substitution), on obtient comme souhaité $B;\mathsf{nil} \vdash_c \{x{\leftarrow_c}V_1\}V_2 :^{\gamma} \{x{\leftarrow_c}V_1\}T_2$.

Règle (ered.seal) : On a $E = V\ !!_{c'}\ T$ et le jugement de typage utilise (et.seal) avec la prémisse $B;\mathsf{nil} \vdash_{c\cup c'} V :^{I} T$. Par le Lem. B.2.24 (pureté des valeurs), $B;\mathsf{nil} \vdash_{c\cup c'} V :^{P} T$.

On a $B' = B, a = V :_{c\cup c'} T$. Posons $c'' = c \cup c' \cup \{a\}$. On a $B';\mathsf{nil} \vdash_{c''} \mathsf{ok}$ par (envok.c.a). Par le Lem. B.1.15 (transparence point par point), on a $B';\mathsf{nil} \vdash_{c''} c \cup c'\, \mathsf{transparent}$ et $B';\mathsf{nil} \vdash_{c''} c''\, \mathsf{transparent}$.

Par (ac.a), on a $B';\mathsf{nil} \vdash_{c\cup c'} a \rhd V : T$. Par le Lem. B.3.4 (propriétés du renforcement), on a $B';\mathsf{nil} \vdash_{c\cup c'} \mathbf{self}^{\,T}(a) : \wr$ ainsi que $B';\mathsf{nil} \vdash_c \mathbf{self}^{\,T}(a) <: T$.

Par le Cor. B.1.24 (extraction d'un lexique), on a $B';\mathsf{nil} \vdash_{c''} V :^{P} T$. Par (ac.a), on a $B';\mathsf{nil} \vdash_{c''} a \rhd V : T$. Par le Lem. B.3.4 (propriétés du renforcement), on a $B';\mathsf{nil} \vdash_{c''} V :^{P} \mathbf{self}^{\,T}(a)$.

Par (et.col), on a $B;\Gamma \vdash_c [V]_{c''}^{\mathbf{self}^{\,T}(a)} :^{P} \mathbf{self}^{\,T}(a)$. Par (et.sub), on a $B;\Gamma \vdash_c [V]_{c''}^{\mathbf{self}^{\,T}(a)} :^{P} T$.

Règle (ered.undyn) : On a $E = \mathsf{undyn}\,(\mathsf{dyn}\, V\, \mathsf{at}\, T_0)\, \mathsf{at}\, T\, \mathsf{else}\, E_1$ typée par la règle (et.undyn) avec les prémisses $B;\mathsf{nil} \vdash_c \mathsf{dyn}\, V\, \mathsf{at}\, T_0 :^{I} \text{DYN}$ et $B;\mathsf{nil} \vdash_c E_1 :^{I} T$. Deux cas se présentent, suivant si la vérification de typage réussit ou non. Si $B;\mathsf{nil} \vdash_c T_0 <: T$, alors $E' = V$. Par le Lem. B.2.22 (inversion du typage d'un dynamique), $B;\mathsf{nil} \vdash_c V :^{I} T_0$, d'où $B;\mathsf{nil} \vdash_c V :^{I} T$ par (et.sub). Sinon $E' = E_1$, et on a bien $B;\mathsf{nil} \vdash_c E_1 :^{I} T$.

Contexte $E_1\,\_\_$: La prémisse de la règle de réduction est $E_0 \longrightarrow_c E_0'$, et le jugement de typage utilise (et.app) avec les prémisses $B;\mathsf{nil} \vdash_c E_1 :^{\gamma_0} \Pi x : T_0.\ ^{\gamma_1}T_1$ et $B;\mathsf{nil} \vdash_c E_0 :^{P} T_0$. Par récurrence, on obtient $B;\mathsf{nil} \vdash_c E_0' :^{P} T_0$. Une nouvelle application de (et.app) avec E_0' à la place de E_0 donne le résultat souhaité.

Contexte $\_\_ \mathsf{V}_2$ **:** Même principe, en changeant E_1 en E_1'. On applique le Lem. B.1.2 (correction du lexique) et le Lem. B.1.19 (affaiblissement du lexique) pour passer du lexique B à B' qui l'étend.

Contexte $(\_\_, \mathsf{E}_2)$ **:** Même principe, avec la règle de typage (et.pair).

Contexte $(\mathsf{V}_1, \_\_)$ **:** Même principe, avec la règle de typage (et.pair).

Contexte $\pi_i \_\_$ **:** Même principe, avec la règle de typage (et.proj.1) ou (et.proj.2).

Contexte $\mathsf{let}\ x = \_\_ \ \mathsf{in}\ \mathsf{E}_0 : \mathsf{T}_0$ **:** Même principe, avec la règle de typage (et.let).

Contexte $\_\_ \ !!_{c_0}\ \mathsf{T}_0$ **:** Même principe, avec la règle de typage (et.seal).

Contexte $[\_\_]_{c_1}^{\mathsf{T}_0}$ **:** Même principe, avec la règle de typage (et.col).

Contexte $[\mathsf{V}^{c_1}]_{c_1}^{\mathsf{Typ}\,\_\_}$ **:** Le radical est une quasi-valeur, donc peut être typé pur en vertu du Lem. B.2.24 (pureté des valeurs). On a donc $\mathsf{B}'; \mathsf{nil} \vdash_c \mathsf{E}' :^{\mathsf{P}} \mathsf{T}$, d'où $\mathsf{B}'; \mathsf{nil} \vdash_c \mathsf{E}' :^{\gamma} \mathsf{T}$ par (et.sub).

Contexte $\mathsf{dyn} \_\_ \ \mathsf{at}\ \mathsf{T}_0$ **:** Même principe que le contexte $\mathsf{E}_1 \_\_$, avec la règle de typage (et.dyn).

Contexte $\mathsf{dynned} \_\_ \ \mathsf{at}\ \mathsf{T}_0$ **:** Même principe que le contexte $\mathsf{E}_1 \_\_$, avec la règle de typage (et.dynned).

Contexte $\mathsf{undyn} \_\_ \ \mathsf{at}\ \mathsf{T}_0\ \mathsf{else}\ \mathsf{E}_1$ **:** Même principe, avec la règle de typage (et.undyn).

□

B.3.3 Progrès

Theorème B.3.7 (progrès) Si $\mathsf{B}; \mathsf{nil} \vdash_c \mathsf{E} :^{\gamma} \mathsf{T}$ alors E est une valeur ou E est réductible.

Theorème B.3.8 (déterminisme) Si $\mathsf{B}; \mathsf{nil} \vdash_c \mathsf{E} :^{\gamma} \mathsf{T}$ alors E est une valeur ou bien E se réduit d'exactement une manière.

On appelle **comportement** d'une expression E (sous un lexique B et dans une couleur c) soit le fait que E est une valeur (sous B dans c), soit une dérivation de $\mathsf{B} \vdash \mathsf{E} \longrightarrow_c \mathsf{B}' \vdash \mathsf{E}'$ pour un B' et un E' quelconques. Le théorème de progrès affirme que toute expression bien typée admet au moins un comportement ; le théorème de déterminisme affirme l'unicité du comportement.

Démonstration. Par induction sur la dérivation du jugement de typage. Le cas des règles (et.sub) et (et.sing) est trivial ; d'après la Rem. B.2.17 (règles structurelles du typage des expressions), il reste donc à examiner les règles structurelles, ce qui revient à examiner le constructeur de tête de E.

Si $\mathsf{E} = \mathsf{E}_1\, \mathsf{E}_0$ — **règle (et.app) :** Les comportements candidats sont donnés par la règle (ered.app) ainsi que les contextes $\mathsf{E}_1 \_\_$ et $\_\_ \mathsf{E}_0$. Les prémisses de (et.app) sont $\mathsf{B}; \mathsf{nil} \vdash_c \mathsf{E}_1 :^{\gamma_0} \Pi x : \mathsf{T}_0.^{\gamma_1} \mathsf{T}_1$ et $\mathsf{B}; \mathsf{nil} \vdash_c \mathsf{E}_0 :^{\mathsf{P}} \mathsf{T}_0$. Par récurrence, E_0 et E_1 ont un unique comportement. Si E_0 est réductible, ce n'est pas une valeur, et le contexte $\mathsf{E}_1 \_\_$ est seul convenable. Sinon, E_0 est une valeur. Si E_1 est réductible, ce n'est pas une valeur, et le contexte $\mathsf{E}_0 \_\_$ est seul convenable. Il reste le cas où E_0 et E_1 sont tous les deux des valeurs ; comme ils sont irréductibles, le seul comportement possible est (ered.app). D'après le Lem. B.2.27 (forme des valeurs), E_1 est de la forme $\lambda x{:}\mathsf{T}_2.\ \mathsf{E}_2$, donc la règle (ered.app) s'applique.

Si $\mathsf{E} = ()$ **ou** $\mathsf{E} = bv$ **ou** $\mathsf{E} = \underline{n}$ — **règles (et.base.*) :** E est une valeur, et aucune règle de réduction ne s'applique.

Si $\mathsf{E} = [\mathsf{E}_0]_{c_0}^{\mathsf{T}}$ — **règle (et.col) :** Les comportements candidats sont les contextes $[\_\_]_{c_0}^{\mathsf{T}}$ et $[\mathsf{E}_0]_{c_0}^{\mathsf{Typ}\,\_\_}$, les règles (ered.col.*) et (ered.colTyp), et la valorité. Les prémisses de (et.col) sont $\mathsf{B}; \Gamma \vdash_{c_0} \mathsf{E}_0 :^{\gamma} \mathsf{T}$, $\mathsf{B}; \Gamma \vdash_{c \cap c_0} \mathsf{T} : \wr$ et $\mathsf{B}; \Gamma \vdash_c \mathsf{ok}$. Par récurrence, E_0 a un unique comportement.

Si E_0 est réductible, un comportement de E est la réduction de E_0 dans le contexte $[\_\_]^{\mathsf{T}}_{\mathsf{c}_0}$. Pour prouver que c'est le seul, il faut vérifier qu'aucun des autres candidats ne convient ; il est clair pour tous les autres candidats que E_0 doit être une valeur (dans le cas de (ered.col.merge), $\mathsf{E}_0 = [\mathsf{V}^{\mathsf{c}_2}]^{(\!|\mathsf{A}_2|\!)}_{\mathsf{c}_2}$ où A_2 est bien transparente dans c_2 et opaque dans la couleur c_1 dans laquelle est plongé le crochet).

Nous traitons maintenant le cas où E_0 est une valeur. Dans ce cas, la discrimination se fait sur la forme de T. Les candidats possibles sont les règles (ered.col.*), (ered.colAbs) et (ered.colTyp), le contexte $[\mathsf{E}_0]^{\mathsf{Typ}\,\_\_}_{\mathsf{c}_0}$, et la valorité ; la forme de T force la règle de réduction à appliquer, sauf lorsque T est un type abstrait A auquel cas le caractère transparent ou opaque de A dans c_0 et dans c force l'un des comportements incompatibles que sont la réduction par (ered.colAbs) (si $\mathbf{underl}(\mathsf{A}) \in \mathsf{c}_0 \cap \mathsf{c}$), la réduction par (ered.col.merge) (si $\mathbf{underl}(\mathsf{A}) \notin \mathsf{c}_0$) et la valorité (si $\mathbf{underl}(\mathsf{A}) \in \mathsf{c}_0 \setminus \mathsf{c}$). Il reste à vérifier que le comportement est bien possible, ce qui demande de déterminer la forme de la valeur E_0. On applique pour cela le Lem. B.2.27 (forme des valeurs).

Si $\mathsf{T} = \text{UNIT}$: On peut appliquer (ered.col.base.unit).

Si $\mathsf{T} = \text{BOOL}$: On peut appliquer (ered.col.base.bool).

Si $\mathsf{T} = \text{INT}$: On peut appliquer (ered.col.base.int).

Si $\mathsf{T} = \text{TYPE}^{\mathsf{K}}$: Impossible car T est monomorphe.

Si $\mathsf{T} = \mathsf{Typ}\,\mathsf{E}_1$: Par inversion de la règle (tok.field), on a $\mathsf{B}; \mathsf{nil} \vdash_{\mathsf{c}_0 \cap \mathsf{c}} \mathsf{E}_1 :^{\mathsf{P}} \text{TYPE}^*$. Par récurrence, E_1 a un unique comportement. Si E_1 est réductible, E se réduit via le contexte $[\mathsf{E}_0]^{\mathsf{Typ}\,\_\_}_{\mathsf{c}_0}$. Si E_1 est une valeur, elle est de la forme $\langle \mathsf{T}_1 \rangle$, et E se réduit par (ered.colTyp).

Si $\mathsf{T} = \Sigma \mathsf{x} : \mathsf{T}_1.\, \mathsf{T}_2$: On peut appliquer (ered.col.pair).

Si $\mathsf{T} = \Pi \mathsf{x} : \mathsf{T}_0.\, {}^{\mathsf{P}}\mathsf{T}_1$: On peut appliquer (ered.col.fun.P).

Si $\mathsf{T} = \Pi \mathsf{x} : \mathsf{T}_0.\, {}^{\mathsf{I}}\mathsf{T}_1$: On peut appliquer (ered.col.fun.I).

Si $\mathsf{T} = \mathsf{S}(\mathsf{E}_1)$: On peut appliquer (ered.col.sing), sans condition sur la forme de E_0.

Si $\mathsf{T} = \mathsf{A}$: Par inversion de la règle (tok.abs), on a $\mathsf{B}; \mathsf{nil} \vdash_{\mathsf{c}_0 \cap \mathsf{c}} \mathsf{A} \triangleright \mathsf{E}_1 : \text{TYPE}^{\mathsf{l}}$. Si $\mathbf{underl}(\mathsf{A}) \in \mathsf{c} \cap \mathsf{c}_0$, $[\mathsf{E}_0]^{(\!|\mathsf{A}|\!)}_{\mathsf{c}_0}$ se réduit par (ered.colAbs). Si $\mathbf{underl}(\mathsf{A}) \in \mathsf{c}_0 \setminus \mathsf{c}$, $[\mathsf{E}_0]^{(\!|\mathsf{A}|\!)}_{\mathsf{c}_0}$ est une valeur. Il reste à traiter le cas où $\mathbf{underl}(\mathsf{A}) \notin \mathsf{c}_0$.

Par le Lem. B.2.27 (forme des valeurs) appliqué à E_0, il existe $\mathsf{V}^{\mathsf{c}_2}$, c_2 et A_2 tels que $\mathsf{E}_0 = [\mathsf{V}^{\mathsf{c}_2}]^{(\!|\mathsf{A}_2|\!)}_{\mathsf{c}_2}$. Comme E_0 est une valeur, $\mathbf{underl}(\mathsf{A}_2) \in \mathsf{c}_2 \setminus \mathsf{c}_0$. Les hypothèses de la règle (ered.col.merge) sont réunies.

Si $\mathsf{T} = \text{DYN}$: On peut appliquer (ered.col.dynned).

Si $\mathsf{E} = \mathsf{dyn}\,\mathsf{E}_0\,\mathsf{at}\,\mathsf{T}_0$ — règle (et.dyn) : Les comportements candidats sont la règle (ered.dyn) et le contexte $\mathsf{dyn}\,\_\_\,\mathsf{at}\,\mathsf{T}_0$. La prémisse de (et.dyn) est $\mathsf{B}; \mathsf{nil} \vdash_{\mathsf{c}} \mathsf{E}_0 :^{\gamma} \mathsf{T}_0$. Par récurrence, E_0 a un unique comportement. Si c'est une réduction, alors E a pour unique comportement la réduction de E_0 dans le contexte $\mathsf{dyn}\,\_\_\,\mathsf{at}\,\mathsf{T}_0$. Si E_0 est une valeur, E a pour unique comportement la réduction par (ered.dyn).

Si $\mathsf{E} = \mathsf{dynned}\,\mathsf{E}_0\,\mathsf{at}\,\mathsf{T}_0$ — règle (et.dynned) : Les comportements candidats sont la valorité et le contexte $\mathsf{dyn}\,\_\_\,\mathsf{at}\,\mathsf{T}_0$. La prémisse de (et.dyn) est $\mathsf{B}; \mathsf{nil} \vdash_{\mathsf{c}} \mathsf{E}_0 :^{\gamma} \mathsf{T}_0$. Par récurrence, E_0 a un unique comportement. Si c'est une réduction, alors E a pour unique comportement la réduction de E_0 dans le contexte $\mathsf{dyn}\,\_\_\,\mathsf{at}\,\mathsf{T}_0$. Si E_0 est une valeur, E a pour unique comportement la valorité.

Si $\mathsf{E} = \lambda \mathsf{x} : \mathsf{T}_0.\ \mathsf{E}_1$ — règle (et.fun) : E est une valeur, et aucune règle de réduction ne s'applique.

Si $\mathsf{E} = \mathsf{let}\, x = \mathsf{E}_0 \,\mathsf{in}\, \mathsf{E}_1 : \mathsf{T}_1$ — **règle** (**et.let**) **:** Les comportements candidats sont la règle (ered.let) et le contexte $\mathsf{let}\, x = \_\_ \,\mathsf{in}\, \mathsf{E}_1 : \mathsf{T}_1$. Une prémisse de (et.let) est $\mathsf{B}; \Gamma \vdash_{\mathsf{c}} \mathsf{E}_0 :^{\mathsf{I}} \mathsf{T}_0$. Si c'est une réduction, alors E a pour unique comportement la réduction de E_0 par le contexte $\mathsf{let}\, x = \_\_ \,\mathsf{in}\, \mathsf{E}_1 : \mathsf{T}_1$. Si E_0 est une valeur, E a pour unique comportement la réduction par (ered.let).

Si $\mathsf{E} = (\mathsf{E}_1, \mathsf{E}_2)$ — **règle** (**et.pair**) **:** Les comportements candidats sont la valorité et les contextes $(\mathsf{E}_1, \_\_)$ et $(\_\_, \mathsf{E}_2)$. Les prémisses de (et.pair) sont $\mathsf{B}; \Gamma \vdash_{\mathsf{c}} \mathsf{E}_1 :^{\gamma} \mathsf{T}_1$ et $\mathsf{B}; \Gamma \vdash_{\mathsf{c}} \mathsf{E}_2 :^{\gamma} \mathsf{T}_2$. Par récurrence, E_1 et E_2 ont un unique comportement. Si E_1 est réductible, l'unique comportement de E est la réduction de E_1 dans le contexte $(\_\_, \mathsf{E}_2)$. Sinon, E_2 est une valeur. Si E_2 est réductible, l'unique comportement de E est la réduction de E_2 dans le contexte $(\mathsf{E}_1, \_\_)$. Sinon, E_2 est une valeur, et l'unique comportement de E est la valorité.

Si $\mathsf{E} = \pi_{\mathsf{i}}\, \mathsf{E}_0$ — **règles** (**et.proj.***) **:** Les comportements candidats sont la règle (ered.proj) et le contexte $\pi_{\mathsf{i}} \_\_$. Une prémisse de (et.proj.*) est $\mathsf{B}; \Gamma \vdash_{\mathsf{c}} \mathsf{E}_0 :^{\gamma} \Sigma x : \mathsf{T}_1.\, \mathsf{T}_2$. Par récurrence, E_0 a un unique comportement. Si E_0 est réductible, l'unique comportement de E est la réduction de E_0 dans le contexte $\pi_{\mathsf{i}} \_\_$. Sinon E_0 est une valeur. D'après le Lem. B.2.27 (forme des valeurs), E_0 est soit de la forme $(\mathsf{E}_1, \mathsf{E}_2)$, soit une composante abstraite. Si E_0 est une paire, l'unique comportement de E est alors la réduction par (ered.proj). Si E_0 est une composante abstraite, c'est aussi le cas de E.

Si $\mathsf{E} = \mathsf{E}_0 \,!!_{\mathsf{c}_0}\, \mathsf{T}$ — **règle** (**et.seal**) **:** Les comportements candidats sont la règle (ered.seal) et le contexte $\_\_ \,!!_{\mathsf{c}_0}\, \mathsf{T}$. Une prémisse de (et.seal) est $\mathsf{B}; \Gamma \vdash_{\mathsf{c} \cup \mathsf{c}_0} \mathsf{E}_0 :^{\mathsf{I}} \mathsf{T}$. Par récurrence, E_0 a un unique comportement. Si E_0 est réductible, l'unique comportement de E est la réduction de E_0 dans le contexte $\_\_ \,!!_{\mathsf{c}_0}\, \mathsf{T}$. Sinon E_0 est une valeur, et l'unique comportement de E est la réduction par (ered.seal).

Si $\mathsf{E} = \langle \mathsf{T}_0 \rangle$ — **règle** (**et.type**) **:** E est une valeur, et aucune règle de réduction ne s'applique.

Si $\mathsf{E} = \mathsf{undyn}\, \mathsf{E}_0 \,\mathsf{at}\, \mathsf{T} \,\mathsf{else}\, \mathsf{E}_1$ — **règle** (**et.undyn**) **:** Les comportements candidats sont la règle (ered.undyn) et le contexte $\mathsf{undyn} \_\_ \mathsf{at}\, \mathsf{T} \,\mathsf{else}\, \mathsf{E}_1$. Une prémisse de (et.undyn) est $\mathsf{B}; \Gamma \vdash_{\mathsf{c}} \mathsf{E}_0 :^{\mathsf{I}} \mathsf{T}$. Par récurrence, E_0 a un unique comportement. Si E_0 est réductible, l'unique comportement de E est la réduction de E_0 dans le contexte $\mathsf{undyn} \_\_ \mathsf{at}\, \mathsf{T} \,\mathsf{else}\, \mathsf{E}_1$. Sinon E_0 est une valeur. Par le Lem. B.2.27 (forme des valeurs), E_0 est de la forme $\mathsf{dyn}\, \mathsf{E}_2 \,\mathsf{at}\, \mathsf{T}_2$, et l'unique comportement de E est la réduction par (ered.undyn).

Si $\mathsf{E} = x$ — **règle** (**et.x**) **:** Impossible d'après le Lem. B.1.1 (les variables sont liées).

□

Bibliographie

[ACPP91] Martin Abadi, Luca Cardelli, Benjamin Pierce et Gordon Plotkin. Dynamic typing in a statically typed language. *ACM Trans. Program. Lang. Syst.*, 13(2):237–268, 1991.

[ACPR94] Martin Abadi, Luca Cardelli, Benjamin Pierce et Didier Rémy. Dynamic Typing in Polymorphic Languages. Research Report RR-120, DEC SRC, 1994.

[AM91] Andrew Appel et David B. MacQueen. Standard ML of New Jersey. Dans J. Maluszynski et M. Wirsing (réds.), *Programming Language Implementation and Logic Programming, Proceedings of the 3rd Intn'l Symposium*, t. 528 de *LNCS*, p. 1–13. Springer Verlag, 1991.

[AM94] Andrew W. Appel et David B. MacQueen. Separate Compilation for Standard ML. Dans *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, p. 13–23. ACM Press, New York, NY, USA, 1994.

[Asp95] David Aspinall. Subtyping with Singleton Types. Dans *CSL '94: Selected Papers from the 8th International Workshop on Computer Science Logic*, p. 1–15. Springer-Verlag, London, UK, 1995.

[AZ02] Davide Ancona et Elena Zucca. A calculus of module systems. *J. Funct. Program.*, 12(2):91–132, 2002.

[BC68] T. O. Barnett et L. Constantine (réds.). *Modular Programming: Proceedings of a National Symposium*. Information and Systems Press, Cambridge, Mass., 1968.

[Ben] Jim Bender. Mini-Bibliography on Modules for Functional Programming Languages. En ligne à `http://readscheme.org/modules/`.

[BS02] Arthur I. Baars et S. Doaitse Swierstra. Typing dynamic typing. Dans *ICFP '02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, p. 157–166. ACM Press, New York, NY, USA, 2002.

[BSSS06] John Billings, Peter Sewell, Mark Shinwell et Rok Strniša. Type-Safe Distributed Programming for OCaml. Dans *ML '06: Proceedings of the 2006 workshop on ML*. ACM Press, New York, NY, USA, 2006.

[Car88] Luca Cardelli. Phase Distinctions in Type Theory, 1988. Manuscrit.

[CDG+92] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow et Greg Nelson. Modula-3 Language Definition. *SIGPLAN Notices*, 8(27):15–42, août 1992.

[CHC+98] Karl Crary, Robert Harper, Perry Cheng, Leaf Petersen et Chris Stone. Transparent and Opaque Interpretations of Datatypes. Rap. tech. CS-98-177, Carnegie Mellon University, 1998.

[CL90] Luca Cardelli et Xavier Leroy. Abstract types and the dot notation. Dans M. Broy et C. B. Jones (réds.), *Proceedings IFIP TC2 working conference on programming concepts and methods*, p. 479–504. North-Holland, 1990.

[CM88] L. Cardelli et D. MacQueen. Persistence and type abstraction. Dans Malcolm P. Atkinson, Peter Buneman et Ronald Morrison (réds.), *Data Types and Persistence. Edited Papers from the Proceedings of the First Workshop on Persistent Objects, Appin, Scotland, August 1985*, Topics in Information Systems, p. 31–41. Springer, 1988. (Version révisée).

[Coo90] William R. Cook. Object-oriented programming versus abstract data types. Dans *Proc. of the REX Workshop/School on the Foundations of Object-Oriented Languages (FOOL)*, t. 173 de *Lecture Notes in Computer Science*, p. 151–178. Springer-Verlag, 1990.

[Coq] The Coq Development Team. *The Coq Proof Assistant Reference Manual*.

[Coq86] Thierry Coquand. An Analysis of Girard's Paradox (LICS 1986). Dans *LICS '86: Proceedings of the 1986 IEEE Symposium on Logic in Computer Science*, p. 227–236. juin 1986.

[Cou97a] Judicaël Courant. An applicative module calculus. Dans *Theory and Practice of Software Development 97*, Lecture Notes in Computer Science, p. 622–636. Springer-Verlag, avr. 1997.

[Cou97b] Judicaël Courant. A Module Calculus for Pure Type Systems. Dans *Typed Lambda Calculi and Applications 97*, Lecture Notes in Computer Science, p. 112 – 128. Springer-Verlag, 1997.

[Cou98] Judicaël Courant. *Un calcul de modules pour les systèmes de types purs*. Thèse de doctorat, Ecole Normale Supérieure de Lyon, 1998.

[CW85] Luca Cardelli et Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17(4):471–523, 1985.

[DCH02] Derek Dreyer, Karl Crary et Robert Harper. A Type System for Higher-Order Modules. Rap. tech. CS-02-122R, Carnegie Mellon University, 2002. Version étoffée de [DCH03].

[DCH03] Derek Dreyer, Karl Crary et Robert Harper. A type system for higher-order modules. Dans *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, p. 236–249. ACM Press, New York, NY, USA, 2003.

[Den72] Jack B. Dennis. Modularity. Computation Structures Group Memo 70, MIT (project MAC), juin 1972. Notes prepared for an advanced course on software engineering, Technical University of Munich, February 1972.

[DL06] Pierre-Malo Deniélou et James J. Leifer. Abstraction preservation and subtyping in distributed languages. Dans *ICFP '06: Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming*, p. 286–297. ACM Press, New York, NY, USA, 2006.

[Dre02] Derek Dreyer. Moscow ML's higher-order modules are unsound, déc. 2002. Message on the TYPES forum. En ligne à `http://www.seas.upenn.edu/~sweirich/types/archive/1999-2003/msg01136.html`.

[Dre05] Derek Dreyer. *Understanding and Evolving the ML Module System*. Thèse de doctorat, Carnegie Mellon University, 2005.

[FLFS07] Cédric Fournet, Fabrice Le Fessant et Alan Schmitt. *The JoCaml language (beta release)*, 2007.

[FLMR97] Cédric Fournet, Cosimo Laneve, Luc Maranget et Didier Rémy. Implicit Typing à la ML for the join-calculus. Dans *CONCUR '97: Proceedings of the 1997 8th International Conference on Concurrency Theory*, p. 196–212. Springer-Verlag, juil. 1997.

[Fur02] Jun Furuse. *Extensional polymorphism: Theory and Application.* Thèse de doctorat, Université Denis Diderot (Paris 7), 2002.

[FW00] Jun Furuse et Pierre Weis. Entrées/Sorties de valeurs en Caml. Dans *Actes des Journées Francophones des Langages Applicatifs*. 2000.

[Gar04] Jacques Garrigue. Relaxing the value restriction. Dans *FLOPS '04: Proceedings of the 7th International Symposium on Functional and Logic Programming*, t. 2998 de *Lecture Notes in Computer Science*, p. 196–213. Springer-Verlag, avr. 2004.

[GHW81] John Guttag, James Horning et John Williams. FP with data abstraction and strong typing. Dans *FPCA '81: Proceedings of the 1981 conference on Functional programming languages and computer architecture*, p. 11–24. ACM Press, New York, NY, USA, 1981.

[GMM+78] M. Gordon, R. Milner, L. Morris, M. Newey et C. Wadsworth. A Metalanguage for interactive proof in LCF. Dans *POPL '78: Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, p. 119–130. ACM Press, New York, NY, USA, 1978.

[GMZ00] Dan Grossman, Greg Morrisett et Steve Zdancewic. Syntactic type abstraction. *ACM Trans. Program. Lang. Syst.*, 22(6):1037–1080, 2000.

[Gog05] Healfdene Goguen. A syntactic approach to eta equality in type theory. Dans *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, p. 75–84. 2005.

[GP98] Giorgio Ghelli et Benjamin Pierce. Bounded existentials and minimal typing. *Theor. Comput. Sci.*, 193(1–2):75–96, 1998. Circulated in manuscript form since 1992.

[Gut77] John Guttag. Abstract data types and the development of data structures. *Commun. ACM*, 20(6):396–404, 1977.

[HL94] Robert Harper et Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. Dans *POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, p. 123–137. ACM Press, New York, NY, USA, 1994.

[HM93] Robert Harper et John C. Mitchell. On the type structure of Standard ML. *ACM Trans. Program. Lang. Syst.*, 15(2):211–252, 1993.

[HMC06] Grégoire Henry, Michel Mauny et Emmanuel Chailloux. Typer la désérialisation sans sérialiser les types. Dans *Actes des Journées Francophones des Langages Applicatifs*. 2006.

[HMM90] Robert Harper, John C. Mitchell et Eugenio Moggi. Higher-order modules and the phase distinction. Dans *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, p. 341–354. ACM Press, New York, NY, USA, 1990.

[HMN05] Fritz Henglein, Henning Makholm et Henning Niss. *Effect Types and Region-Based Memory Management*, chap. 3, p. 87–140. Dans Pierce [Pie05], 2005.

[HP05] Robert Harper et Benjamin C. Pierce. *Design Issues in Advanced Module Systems*, chap. 8, p. 293–345. Dans Pierce [Pie05], 2005.

[IBFW83] Jean D. Ichbiah, John G. P. Barnes, Robert J. Firth et Mike Woodger. *Rationale for the Design of the Ada Programming Language*, 1983.

[Jac06] Michael Jackson. The Role of Structure in Dependable Systems: A Software Engineering Perspective. Dans D. Besnard, C. Gacek et C. B. Jones (réds.), *Structure for Dependability: Computer-Based Systems from an Interdisciplinary Perspective*, chap. 2, p. 16–45. Springer, 2006.

[Jon96] Mark P. Jones. Using parameterized signatures to express modular structure. Dans *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, p. 68–78. ACM Press, New York, NY, USA, 1996.

[Kay93] Alan C. Kay. The early history of Smalltalk. Dans *HOPL-II: The second ACM SIGPLAN conference on History of programming languages*, p. 511–598. ACM Press, New York, NY, USA, 1993.

[Klo80] Jan Willem Klop. *Combinatory reduction systems*. Thèse de doctorat, Mathematisch Centrum, Amsterdam, 1980.

[L^+] Xavier Leroy *et al. The Objective Caml system*.

[Ler] Xavier Leroy. Communication privée.

[Ler93] Xavier Leroy. Polymorphism by name for references and continuations. Dans *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, p. 220–231. ACM Press, New York, NY, USA, 1993.

[Ler94] Xavier Leroy. Manifest types, modules, and separate compilation. Dans *POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, p. 109–122. ACM Press, New York, NY, USA, 1994.

[Ler95] Xavier Leroy. Applicative functors and fully transparent higher-order modules. Dans *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, p. 142–153. ACM Press, New York, NY, USA, 1995.

[Ler96] Xavier Leroy. A syntactic theory of type generativity and sharing. *J. Funct. Program.*, 6(5):667–698, 1996.

[Ler00] Xavier Leroy. A modular module system. *J. Funct. Program.*, 10(3):269–303, 2000.

[LG88] J. M. Lucassen et D. K. Gifford. Polymorphic effect systems. Dans *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, p. 47–57. ACM Press, New York, NY, USA, 1988.

[Lil97] Mark Lillibridge. *Translucent Sums: A Foundation for Higher-Order Module Systems*. Thèse de doctorat, Carnegie Mellon University, mai 1997.

[Lis93] Barbara Liskov. A history of CLU. Dans *HOPL-II: The second ACM SIGPLAN conference on History of programming languages*, p. 133–147. ACM Press, New York, NY, USA, 1993.

[LM93] Xavier Leroy et Michel Mauny. Dynamics in ML. *J. Funct. Program.*, 3(4):431–463, 1993.

[LPSW03a] James J. Leifer, Gilles Peskine, Peter Sewell et Keith Wansbrough. Global abstraction-safe marshalling with hash types. Dans *ICFP '03: Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, p. 87–98. ACM Press, New York, NY, USA, 2003.

[LPSW03b] James J. Leifer, Gilles Peskine, Peter Sewell et Keith Wansbrough. Global abstraction-safe marshalling with hash types. Rapport de recherche RR-4851, INRIA Rocquencourt, juin 2003.

[LSAS77] Barbara Liskov, Alan Snyder, Russell Atkinson et Craig Schaffert. Abstraction mechanisms in CLU. *Commun. ACM*, 20(8):564–576, 1977.

[LZ73] Barbara Liskov et Stephen Zilles. An Approach to Abstraction. Computation Structures Group Memo 88, MIT (project MAC), sept. 1973.

[Mac84] David MacQueen. Modules for Standard ML. Dans *LFP '84: Proceedings of the 1984 ACM Symposium on LISP and functional programming*, p. 198–207. ACM Press, New York, NY, USA, 1984.

[Mac86] David B. MacQueen. Using dependent types to express modular structure. Dans *POPL '86: Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, p. 277–286. ACM Press, New York, NY, USA, 1986.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, août 1978.

[Mit86] John C. Mitchell. Representation independence and data abstraction. Dans *POPL '86: Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, p. 263–276. ACM Press, New York, NY, USA, 1986.

[MM01] Louis Mandel et Luc Maranget. *The JoCaml language*, 2001.

[Mor73a] James H. Morris, Jr. Protection in programming languages. *Commun. ACM*, 16(1):15–21, 1973.

[Mor73b] James H. Morris, Jr. Types are not sets. Dans *POPL '73: Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, p. 120–124. ACM Press, New York, NY, USA, 1973.

[MP88] John C. Mitchell et Gordon D. Plotkin. Abstract types have existential type. *ACM Trans. Program. Lang. Syst.*, 10(3):470–502, 1988.

[MS01] Microsoft Corporation. *.NET Framework Developer's Guide: Serializing Objects*, 2001.

[MTH90] Robin Milner, Mads Tofte et Robert Harper. *The Definition of Standard ML.* MIT Press, Cambridge, Massachusetts, 1990.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper et David MacQueen. *The Definition of Standard ML (Revised edition).* MIT Press, Cambridge, Massachusetts, 1997.

[ND78] Kristen Nygaard et Ole-Johan Dahl. The development of the SIMULA languages. Dans *HOPL-1: The first ACM SIGPLAN conference on History of programming languages*, p. 245–272. ACM Press, New York, NY, USA, 1978.

[OTCP90] Atsushi Ohori, Ivan Tabkha, Richard Connor et Paul Philbrow. Persistence and Type Abstraction Revisited. Dans *Implementing Persistent Object Bases, Principles and Practice, Proceedings of the Fourth International Workshop on Persistent Objects, 23-27 September 1990, Martha's Vineyard, MA, USA*, p. 141–153. Morgan Kaufmann, 1990.

[Pie05] Benjamin C. Pierce (réd.). *Advanced Topics in Types and Programming Languages.* MIT Press, 2005.

[PS00] Benjamin C. Pierce et Eijiro Sumii. Relating Cryptography and Polymorphism, juil. 2000. Manuscrit.

[PSL] Programming System Lab, Saarland University. *The Alice ML Language.*

[RFG05] Norman Ramsey, Kathleen Fisher et Paul Govereau. An Expressive Language of Signatures. Dans *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming.* ACM Press, New York, NY, USA, 2005.

[Ros] Andreas Rossberg. SML vs. Ocaml. En ligne à `http://www.ps.uni-sb.de/~rossberg/SMLvsOcaml.html`.

[Ros03] Andreas Rossberg. Generativity and dynamic opacity for abstract types. Dans *PPDP '03: Proceedings of the 5th ACM SIGPLAN international conference on Principles and practice of declaritive programming*, p. 241–252. ACM Press, New York, NY, USA, 2003.

[Ros07] Andreas Rossberg. *Typed Open Programming — A higher-order, typed approach to dynamic modularity and distribution.* Thèse de doctorat, Universität des Saarlandes, jan. 2007.

[RRS] Sergei Romanenko, Claudio Russo et Peter Sestoft. *Moscow ML Language Overview.*

[Rus98] Claudio Russo. *Types For Modules.* Thèse de doctorat, University of Edinburgh, 1998.

[Rus99] Claudio Russo. First-Class Structures for Standard ML. Dans *ICFP '99: Proceedings of the fourth ACM SIGPLAN international conference on Functional programming.* ACM Press, New York, NY, USA, 1999.

[Sew01] Peter Sewell. Modules, abstract types, and distributed versioning. Dans *POPL '01: Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, p. 236–247. ACM Press, New York, NY, USA, 2001.

[Sha99] Zhong Shao. Transparent modules with fully syntatic signatures. Dans *ICFP '99: Proceedings of the fourth ACM SIGPLAN international conference on Functional programming*, p. 220–232. ACM Press, New York, NY, USA, 1999.

[SHB+05] Gareth Stoyle, Michael Hicks, Gavin Bierman, Peter Sewell et Iulian Neamtiu. Mutatis mutandis: safe and predictable dynamic software updating. Dans *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, p. 183–194. ACM Press, New York, NY, USA, 2005.

[SLW+04] Peter Sewell, James J. Leifer, Keith Wansbrough, Mair Allen-Williams, Francesco Zappa Nardelli, Pierre Habouzit et Viktor Vafeiadis. Acute: high-level programming language design for distributed computation — Design rationale and language definition. Rap. tech. TR-605, Computer Laboratory, University of Cambridge, Oct 2004.

[SLW+05] Peter Sewell, James J. Leifer, Keith Wansbrough, Francesco Zappa Nardelli, Mair Allen-Williams, Pierre Habouzit et Viktor Vafeiadis. Acute: high-level programming language design for distributed computation. Dans *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, p. 15–26. ACM Press, New York, NY, USA, 2005.

[SP04] Eijiro Sumii et Benjamin C. Pierce. A bisimulation for dynamic sealing. Dans *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, p. 161–172. ACM Press, New York, NY, USA, 2004.

[Sun] Sun Microsystems, Inc. *Java API Specifications.*

[Sun06] Sun Microsystems, Inc. *Java SE 6 Object Serialization Specification*, 2006.

[SWL77] Mary Shaw, William A. Wulf et Ralph L. London. Abstraction and verification in Alphard: defining and specifying iteration and generators. *Commun. ACM*, 20(8):553–564, 1977.

[USG83] *Ada '83 Language Reference Manual*, 1983.

[W+89] Pierre Weis *et al.* The CAML Reference Manual. Rap. tech. RT-0121, Projet Formel, INRIA-ENS, 1989.

[Wad89] Philip Wadler. Theorems for free! Dans *FPCA '89: Proceedings of the fourth international conference on Functional programming languages and computer architecture*, p. 347–359. ACM Press, New York, NY, USA, 1989.

[Wil80] Maurice V. Wilkes. Early Programming Developments in Cambridge. Dans N. Metropolis, J. Howlett et G.-C. Rota (réds.), *A History of Computing in the Twentieth Century*, p. 497–501. Academic Press, 1980.

[WLS76] William A. Wulf, Ralph L. London et Mary Shaw. Abstraction and verification in Alphard: introduction to language and methodology. Rap. tech., University of Southern California and Carnegie Mellon University, 1976.

[Wri95] Andrew K. Wright. Simple imperative polymorphism. *Lisp Symb. Comput.*, 8(4):343–355, 1995.

[XCC03] Hongwei Xi, Chiyan Chen et Gang Chen. Guarded recursive datatype constructors. Dans *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, p. 224–235. ACM Press, New York, NY, USA, 2003.

[XP99] Hongwei Xi et Frank Pfenning. Dependent types in practical programming. Dans *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, p. 214–227. ACM Press, New York, NY, USA, 1999.

[ZGM99] Steve Zdancewic, Dan Grossman et Greg Morrisett. Principals in programming languages: a syntactic proof technique. Dans *ICFP '99: Proceedings of the fourth ACM SIGPLAN international conference on Functional programming*, p. 197–207. ACM Press, New York, NY, USA, 1999.

Index

abstraction, **20**, 23
  sorte —, voir sorte
abstrait, voir type
affaiblissement
  de couleur, **166**, 168, 169
agrégat, voir structure
alpha-conversion, **123**, 169
Alphard, 22
annotation de type, 23
apax, 87, **160**, 192, 198
  sous-jacent, 163, **176**
  transparent, 182
appartenance au singleton, **135**
applicatif, voir foncteur
ascription, **157**
authenticité, 22, 24, 25, 28
auto-typage, **37**, 107, 137

bac à sable, 170
bibliothèque, 19, 30
Burali-Forti
  paradoxe, 199

calcul des constructions inductives, 202
champ, **30**, 119
chemin de type, **38**
classe, 21
classe de types, 44
clos (terme), **123**
CLU, 21
cohérence des imports indirects, 129
communication, 101, 111, 190
comparable (module), **148**
compilation séparée, 30, 31, 36
complètement spécifié, voir monomorphe
comportement, **246**
composante, **163**, 176
composante abstraite, 178
composante valeur, **177**
concret, voir type
concrétisation, **167**, 171, 190
confidentialité, 22, 25, 28, 57
conflit de variables, 104
contexte d'évaluation, 109, **127**
contrainte de partage, voir partage de types
conversion (réécriture), 136
  des expressions, **139**
  des types, **138**
conversion (transtypage), 22, 25, 96
convertibilité, 138
couleur, **97**, 162
  ambiante, **99**
  de variable, 168
  primaire, **166**, **176**
couleur ambiante, **165**
couleur vide, **166**
crochet
  additif, 175
crochet coloré, **97**, **162**, 165, 175
  absolu, **170**
  additif, **170**
  effacement, **114**
  poussée, 99
  universel, 170
crochets colorés
  poussée, 165, 172, 179

décidabilité, 114, 194, 201
delta-règle, 202
dépendance
  d'un apax, **167**, 181
désérialisation, **44**, **51**, voir sérialisation
distribué, voir réparti
domaine, **124**
dynamique (typage), 45, 184
dynamique (valeur), **45**, **101**, **185**
  universel, **187**

effacement, voir crochet coloré
effet, 149

système d'effets, **146**, 200
égalité polymorphe, 43
empreinte, 53, **62**, 68, 73, 95
singularisée, **86**, 162, 192
structurelle, 86, 193
empreinte singularisée, voir apax
encapsulation, 21
entité substituable, **103**
environnement, **103**, **124**, 165
de compilation, **114**
équitypable, **156**, 158
équivalence
convertibilité, voir convertibilité
de modules, 144
de types, 22, 105, 185
espace de noms, 30
estampillage, voir type génératif
estampille, 87
êta-expansion, 164
évitement, **42**
existentiel
paquet —, voir paquet existentiel
type —, voir type
expansion (convertibilité), **234**
extensionnalité, 140

famille
de modules, 30, 121
fermeture transparente, **219**
foncteur, **30**, 75, **76**, **121**
applicatif, **39**, **79**, **88**, **153**
de premier ordre, 31
d'ordre supérieur, 37
génératif, **39**, **88**, **153**
transparent, **153**
fonction
polymorphe, **173**
forme normale de tête faible, **234**

génératif, voir foncteur
générique, voir programmation
grappe, **21**

horodatage, 87

identité
foncteur, 130
identité de module, 160
imbriqué, voir module
implémentation (d'une abstraction), 58, 162
imprédicatif, **199**
impur, 149
expression, **147**
indépendance, 21
inférence de types, **194**
initialisation, 32
interface, 23
invariant, 23, 54, 55, 58
invariants, **55**

jugement
local, **124**
jugement coloré, **103**
jugement de typage, **103**

`let`, voir liaison locale
lexique, **161**, 165, 198
liaison locale, **121**, 152
local, voir module

ML, 23, 29
modularité, **20**
module, 20, 21, 29, 54, **76**, 123
de première classe, 32, 72
de seconde classe, 72
imbriqué, 32
local, 32
récursif, 32
monade, 144
monomorphe, 168, **171**

nom, 59
noyau, 30, 198

objet, 21
occurence liée, **123**
opacité, 198
opaque, **166**, voir signature
scellé, 25
ouverture, voir type existentiel

paquet existentiel, 42, **141**
paramétricité, **91**, 171, 198, **201**
partage de types, 36, **129**
partiellement spécifié, voir polymorphe
polymorphe, **171**, voir fonction
paramètre type —, **174**, 201
valeur —, voir valeur
polymorphisme *ad hoc*, **46**

poussée, voir crochet coloré
première classe, voir module
problème de l'évitement, 126
produit dépendant, voir type
programmation générique, **46**, 49, 174, 185
projection, **119**
pur, 85, 149
  expression, **135**, **146**, **147**
  module, 148

quasi-valeur, **101**, **177**

racine flexible, **142**, 145
récursif, voir module
réduction, 126
réduction (convertibilité), **234**
référentiellement transparent, voir pur
règle VALUE, 37
réification, 48, **67**
renforcement, 37, **162**, 172, voir auto-typage
réparti
  système, 15, 51
représentation, 25, **55**, 57
restriction aux valeurs, 87, 204
révélation, **163**, 176

scellage, 22, 25, 54, **55**, **96**, **142**, 150, 176
  coloré, **175**
  d'un foncteur, 81
  dynamique, 143, **154**, 157, 192, 198
  faible, **153**, 157
  fort, **153**, 157
  impur, 157
  inséparable, 157
  minimal, **157**
  séparable, 157
  statique, **154**, 157, 192, 200
seconde classe, voir module
séparable, **148**, 204
séparation des phases, **72**, 148, 184, 204
sérialisation, **15**, **44**, **51**, 189, 204
  de module, 72
serveur de noms, 52, 205
signature, 30, 54, **119**, 123
  abstraite, 142
  famille, 33
  opaque, **55**, **119**
  principale, 36, 129
  singleton, voir singleton
  transparente, **55**, **119**
SIMULA 67, 21
singleton, 147, 153, voir sorte
  de valeur, 131
  d'ordre supérieur, 132
  signature, **120**
  sorte —, voir sorte
  type, **128**
Smalltalk, 21
somme
  dépendante, voir type paire dépendant
  faible, **35**
  forte, **35**, 37
  translucente, **36**
sorte, **100**, **120**, **171**, 176, 199
  abstraction, 198
  singleton, **100**, 198
sous-effectuation, **146**
sous-empreinte, **92**
sous-signaturage, 120, **128**
sous-typage, **128**
  élémentaire, **234**
  en largeur, **202**
  en profondeur, **202**
spécification, 21, 23
strate, **198**
structure, **30**, **54**, **76**, **118**
  élémentaire, **119**
  vide, **119**
substitution, **123**, 169
  de compilation, **114**
suite des étapes élémentaires, **235**
sûr, 194
surcharge, **46**

tampon, 36, 160
translucent
  somme —e, voir somme
transparence, 182
transparent, 162, 165, **166**, 176
  foncteur —, voir foncteur
  scellé, 25
  signature —e, voir signature
  variable —, 169
type
  abstrait, 16, 21, **22**, 23, 28, **36**, 48, 49, 52, 54, **55**, **119**, 143
  (de données) algébrique, 28

composant, **163**
concret, 54, **55**, **119**, 140
conversion, 162
de fonction dépendant, voir type produit dépendant
d'empreinte, **63**
équivalents, voir équivalence de types
existentiel, 35, 48, **142**, 144
ouverture, 35
fantôme, 26
flèche, 123
fonction, 123
génératif, 26
implémentation, **101**, voir type représentation
linéaire, 144
manifeste, **36**
paire dépendant, 35, **119**, 121
paire ordinaire, **122**
prédéfini, 22
produit, 123
produit dépendant, **31**, 35, 119, **121**
singleton, voir singleton
somme dépendant, 119
universel, 190
type abstrait, 175
type concret, 36

univers, **199**
universel, 190, voir type
crochet —, voir crochet coloré
dynamique —, voir dynamique (valeur)
terme —, 168, **187**

valeur, 98, **126**
polymorphe, **174**, 201
VALUE (règle —), voir auto-typage
variable
transparente, 182
variable libre, **123**
vérification, 24
de typage, 53
dynamique, voir dynamique (typage)
version, 70, 92