COURS 544, EXAMEN FINAL

Luc Maranget

4 décembre 2007

⋆ ⋆ ⋆

Quelques questions de cours

Question 1. Donner un terme de PCF t_1, dont l'exécution termine en appel par nom et ne termine pas en appel par valeur ; puis un terme t_2 dont l'exécution ne termine pas en appel par nom.

Réponse : Pour t_1 on peut prendre :

Let `loop` = **Fix** `x -> x` **In** `1`

Pour t_2 on peut prendre :

Fix `x -> x`

□

Question 2. Donner un terme t_1 typable selon Damas et Milner mais pas selon Hindley (sections 6.2 et 6.1 du poly). Puis un terme t_2 non-typable selon Damas et Milner.

Réponse : Pour t_1 :

Let `id` = **Fun** `x -> x` **In** `id id`

Pour t_2, autant prendre un terme qui déclenche une erreur à l'exécution.

`1 + (`**Fun** `x -> x)`

□

Question 3. On propose les définitions suivantes pour un toplevel PCF.

```
Let b_true = Fun kt -> Fun kf -> kt;;
Let b_false = Fun kt -> Fun kf -> kf;;
Let b_loop = Fix x -> x;;
```

Donner les schémas de types de ces trois définitions selon le système de Damas et Milner. Puis donner une instance commune à ces trois schémas de types. On choisira cette instance de sorte que toutes les instances communes aux trois termes s'en déduisent par substitution de variables de type.

Réponse : Le schéma type principal de `b_true` est $\forall$`X,Y[X -> Y -> X]`, celui de `b_false` est $\forall$`X,Y[X -> Y -> Y]`, et celui de `b_loop` est $\forall$`X[X]`.

L'instance « principale » commune demandée est `X -> X -> X`. □

Question 4. Que peut-on dire de l'exécution d'un terme t tel que $\vdash t$: `X` (le terme t a le type `X` dans l'environnement de typage vide). La réponse sera justifiée par l'emploi d'un théorème du cours.

Réponse : L'exécution du terme t ne termine pas. En effet d'après la correction du typage (exprimée pour la sémantique à petit pas), on a alors $t \to t'$ entraîne $\vdash t'$: `X`, et donc $t \to^* t''$ entraîne $\vdash t''$: `X`. Or, aucune forme normale ne possède le type `X`, puisque les formes normales sont soit les entiers (de type **Nat**) soit les fonctions (de type A `->` B). L'exécution de t ne peut pas non plus produire de terme bloqué, puisque ceux-ci ne sont pas typables. On en déduit donc que l'exécution de t ne termine pas. □

Extension de PCF par les booléens

Question 5. On considère d'abord l'extension de la syntaxe abstraite. Les termes additionnels sont définis par les symboles additionnels suivants.

- Deux constantes, notées **True** et **False**.
- Un symbole à un argument, noté `!`.
- Deux symboles à deux arguments notés `&&` et `||`. Les termes correspondants sont notés de manière infixe (par ex. t_1 `&&` t_2).
- Un symbole à trois arguments, noté **If**. Le terme correspondant est noté **If** t_1 **Then** t_2 **Else** t_3.

Donner une déclaration en Caml du type des termes de PCF étendu, en étendant la définition donnée en annexe.

Réponse : Par exemple

```
type t = ...
| Bool of bool
| Not of t
| And of t | Or of t
| If of t * t * t
```

□

Question 6. Donner la sémantique à grand pas des nouvelles constructions, sous forme de jugements $t \hookrightarrow v$ (section 2.4 du poly). On notera les points suivants :

- Les constantes **True** et **False** sont leur propre valeur.
- Les autres symboles correspondent aux opération usuelles sur les booléens — `!` est la négation, `&&` est le « et », `||` est le « ou », et **If** est la conditionnelle usuelle des langages de programmation.
- On impose que t_1 `&&` t_2 vaut **False** dès que t_1 vaut **False**; et que t_1 `||` t_2 vaut **True** dès que t_1 vaut **True**.

Réponse :

$$\textbf{True} \hookrightarrow \textbf{True} \qquad \textbf{False} \hookrightarrow \textbf{False}$$

$$\frac{t \hookrightarrow \textbf{True}}{\texttt{!}t \hookrightarrow \textbf{False}} \qquad \frac{t \hookrightarrow \textbf{False}}{\texttt{!}t \hookrightarrow \textbf{True}}$$

$$\frac{t_1 \hookrightarrow \textbf{False}}{t_1 \texttt{ \&\& } t_2 \hookrightarrow \textbf{False}} \qquad \frac{t_1 \hookrightarrow \textbf{True} \qquad t_2 \hookrightarrow v}{t_1 \texttt{ \&\& } t_2 \hookrightarrow v}$$

$$\frac{t_1 \hookrightarrow \textbf{True}}{t_1 \texttt{ || } t_2 \hookrightarrow \textbf{True}} \qquad \frac{t_1 \hookrightarrow \textbf{False} \qquad t_2 \hookrightarrow v}{t_1 \texttt{ || } t_2 \hookrightarrow v}$$

$$\frac{t_1 \hookrightarrow \textbf{True} \qquad t_2 \hookrightarrow v}{\textbf{If } t_1 \textbf{ Then } t_2 \textbf{ Else } t_3 \hookrightarrow v} \qquad \frac{t_1 \hookrightarrow \textbf{False} \qquad t_3 \hookrightarrow v}{\textbf{If } t_1 \textbf{ Then } t_2 \textbf{ Else } t_3 \hookrightarrow v}$$

□

Question 7. Exprimer `!`t, t_1 `&&` t_2 et t_1 `||` t_2 uniquement à l'aide du **If**, des constantes **True** et **False**, et de PCF non-étendu. La sémantique donnée à la question précédente doit être respectée, sans que vous ayez à le prouver (cf. question suivante). La réponse à cette question peut servir à simplifier un interpréteur ou un compilateur de PCF étendu.

Réponse : On définit :

$$\begin{aligned} \texttt{!}t &\sim \textbf{If } t \textbf{ Then False Else True} \\ t_1 \texttt{ \&\& } t_2 &\sim \textbf{If } t_1 \textbf{ Then } t_2 \textbf{ Else False} \\ t_1 \texttt{ || } t_2 &\sim \textbf{If } t_1 \textbf{ Then True Else } t_2 \end{aligned}$$

□

Question 8. Prouver que votre traduction de t_1 `&&` t_2 est correcte vis à vis de la sémantique donnée à la question 6. C'est-à-dire, en notant $T(t_1, t_2)$ votre réponse à la question précédente, prouver l'équivalence :

$$t_1 \texttt{ \&\& } t_2 \hookrightarrow v \Leftrightarrow T(t_1, t_2) \hookrightarrow v$$

Réponse : La démonstration la plus sûre consiste à procéder par double implication, en discriminant selon la dernière règle appliquée pour prouver un jugement $t \hookrightarrow v$.

Montrons d'abord que t_1 `&&` $t_2 \hookrightarrow v \Rightarrow T(t_1, t_2) \hookrightarrow v$

- On suppose

$$\frac{t_1 \hookrightarrow \textbf{False}}{t_1 \texttt{ \&\& } t_2 \hookrightarrow \textbf{False}}$$

  On a donc $t_1 \hookrightarrow$ **False**. Alors, par l'axiome **False** $\hookrightarrow$ **False** et la seconde règle du **If**, on a :

$$\frac{t_1 \hookrightarrow \textbf{False} \qquad \textbf{False} \hookrightarrow \textbf{False}}{\textbf{If } t_1 \textbf{ Then } t_2 \textbf{ Else False} \hookrightarrow \textbf{False}}$$

- On suppose

$$\frac{t_1 \hookrightarrow \textbf{True} \qquad t_2 \hookrightarrow v}{t_1 \texttt{ \&\& } t_2 \hookrightarrow v}$$

  On a donc $t_1 \hookrightarrow$ **True** et $t_2 \hookrightarrow v$, l'utilisation directe de ces hypothèses dans la première règle du **If** prouve $T(t_1, t_2) \hookrightarrow v$.

Réciproquement, on suppose $T(t_1, t_2) \hookrightarrow v$. Puisque $T(t_1, t_2)$ est un terme **If**, il n'y a que deux règles possibles selon que l'on a $t_1 \hookrightarrow$ **True** ou $t_1 \hookrightarrow$ **False**.

- Dans le premier cas, on a nécessairement $t_2 \hookrightarrow v$ et donc t_1 `&&` $t_2 \hookrightarrow v$.
- Dans le second, v vaut nécessairement **False**, qui se trouve alors également être la valeur de t_1 `&&` t_2 (première règle du `&&`).

Remarque : Cela peut paraître un peu piéton. Mais il faut effectivement faire très attention dans ce genre de preuve. Suppons par exemple la définition suivante de $T(t_1, t_2)$ (que l'on trouve dans beaucoup de copies).

$$\textbf{If } t_1 \textbf{ Then If } t_2 \textbf{ Then True Else False Else False}$$

Alors l'équivalence est fausse. Contre-exemple : supposer $t_1 \hookrightarrow$ **True** et $t_2 \hookrightarrow 10$, alors d'une part,

$$\frac{t_1 \hookrightarrow \textbf{True} \qquad t_2 \hookrightarrow 10}{t_1 \texttt{ \&\& } t_2 \hookrightarrow 10}$$

Et d'autre part, on *ne peut pas* prouver :

$$\textbf{If } t_1 \textbf{ Then If } t_2 \textbf{ Then True Else False Else False} \hookrightarrow 10$$

Puisque la valeur du terme ci-dessus ne peut être que **True** ou **False**. Or si on procède trop rapidement, on risque de ne pas remarquer ce détail.

Au passage, on peut proposer cette traduction, mais alors il faut considérer une autre sémantique, exprimée par exemple ainsi :

$$\frac{t_1 \hookrightarrow \textbf{False}}{t_1 \texttt{ \&\& } t_2 \hookrightarrow \textbf{False}} \qquad \frac{t_1 \hookrightarrow \textbf{True} \qquad t_2 \hookrightarrow \textbf{True}}{t_1 \texttt{ \&\& } t_2 \hookrightarrow \textbf{True}} \qquad \frac{t_1 \hookrightarrow \textbf{True} \qquad t_2 \hookrightarrow \textbf{False}}{t_1 \texttt{ \&\& } t_2 \hookrightarrow \textbf{False}}$$

Remarque encore : Le correcteur connaît le résultat, ce qu'il veut savoir c'est si vous savez prouver le résultat. Votre but est de l'en convaincre.

De quelles techniques de preuve disposons nous ?

1. Prouver les doubles implications. Sûr, mais lourd, pour alléger un peu on peut se permettre d'être de moins en moins précis.

2. Prouver directement les équivalences, ici c'est pas évident à formuler.

3. Traiter tous les cas possibles d'évaluation de t_1 et t_2, pour constater que t_1 `&&` t_2 et $T(t_1, t_2)$ s'évaluent pareillement. Alors, il faut penser à tous les cas, et ne pas oublier le cas où t_1 ne s'évalue pas en un booléen (*i.e.* la valeur de t_1 n'est pas un booléen, ou t_1 n'a pas de valeur), ainsi que le cas $t_1 \hookrightarrow$ **True** et t_2 n'a pas de valeur.

□

Question 9. Écrire en Caml l'interprétation des nouvelles constructions. On procédera par extension de l'interpréteur (en appel par valeur) donné en annexe. On détaillera les modifications apportées au type `value`, à la fonction `print_value` et à la fonction `interv`.

Réponse : Extension du type des valeurs

```
type value = ... | Bool_v of bool
```

Nouvelle fonction `print_value`

```
let print_value chan v = match v with
  ...
| Bool_v true -> fprintf chan "true"
| Bool_v false -> fprintf chan "false"
```

Nouvel interpréteur

```
let rec interv env t = match t with
  ...
| Bool b -> Bool_v b
| Not t -> Bool_v (not (inter_bool env t))
(* On profite de ce que la sémantique de && et || est celle de Caml *)
| And (t1,t2) ->
    Bool_v (inter_bool env t1 && inter_bool env t2)
| Or (t1,t2) ->
    Bool_v (inter_bool env t1 || inter_bool env t2)
| If (t1,t2,t3) ->
    interv env
      (if inter_bool env t1 then t2 else t3)
```

Avec la définition suivante de la fonction mutuellement récursive `inter_bool` (à la fin)

```
and inter_bool env t = match interv env t with
| Bool_v b -> b
| _        -> raise (Error "bool expected, got something else")
```

Remarque Notre interpréteur ne respecte pas la sémantique de la question 6 sur un point : si la valeur de t_1 est **True** et que la valeur v de t_2 existe mais n'est pas un booléen, alors l'interprétation de t_1 `&&` t_2 échoue — alors que la semantique donne la valeur v. Ce point est mineur (on peut n'interpréter que des termes bien typés) et surtout traité dans une question ultérieure 12.

Un point plus important est bien traité : si la valeur de t_1 est **False**, alors la valeur de t_1 `&&` t_2 est également **False**, quelque soit le comportement de t_2 (valeur booléene, valeur autre, non-terminaison). Ce bon comportement découle directement de la sémantique de `&&` en Caml. Ce point est plus important car il comprend le cas significatif d'un terme t_2 de type **Bool** (Question 11) dont l'évaluation ne termine pas.

Remarquons encore que nous pouvons écrire un interpréteur complètement conforme.

```
| And (t1,t2) ->
    if inter_bool env t1 then
      interv env t2
    else
      Bool_v false
```

Ou même...

```
| And (t1,t2) -> interv env (If (t1,t2,Bool false))
```

□

Question 10. Proposer un schéma de compilation (section 4.4 du poly) pour les nouvelles constructions vers la machine ASEC du cours *non-modifiée*. On pourra considérer, comme c'est l'usage, que la représentation machine du booléen **False** est l'entier zéro.

Réponse : Trois règles de base :

$$|\mathbf{True}| = \text{Ldi } 1 \qquad |\mathbf{False}| = \text{Ldi } 0 \qquad |\mathbf{If}\ t_1\ \mathbf{Then}\ t_2\ \mathbf{Else}\ t_3| = |t_1|;\ \text{Test}(|t_3|, |t_2|)$$

On notera l'inversion des arguments du Test, par rapport à la compilation du **Ifz**.

Il reste à exploiter la question 7. Voici par exemple la compilation de t_1 `&&` t_2.

$$|t_1\ \texttt{\&\&}\ t_2| = |\mathbf{If}\ t_1\ \mathbf{Then}\ t_2\ \mathbf{Else\ False}|$$

□

Question 11. Soit **Bool**, le type des booléens. Donner les règles de typage des nouvelles constructions, dans le style des jugements $E \vdash t : A$ de la section 6.1 du poly. Les règles de typage seront les règles usuelles, qui correspondent par exemple à celles de Java ou de Caml.

Réponse :

$$E \vdash \mathbf{True} : \mathbf{Bool} \qquad\qquad E \vdash \mathbf{False} : \mathbf{Bool}$$

$$\frac{E \vdash t : \mathbf{Bool}}{E \vdash \texttt{!}t : \mathbf{Bool}}$$

$$\frac{E \vdash t_1 : \mathbf{Bool} \qquad E \vdash t_2 : \mathbf{Bool}}{E \vdash t_1\ \texttt{\&\&}\ t_2 : \mathbf{Bool}} \qquad\qquad \frac{E \vdash t_1 : \mathbf{Bool} \qquad E \vdash t_2 : \mathbf{Bool}}{E \vdash t_1\ \texttt{||}\ t_2 : \mathbf{Bool}}$$

$$\frac{E \vdash t_1 : \mathbf{Bool} \qquad E \vdash t_2 : A \qquad E \vdash t_3 : A}{E \vdash \mathbf{If}\ t_1\ \mathbf{Then}\ t_2\ \mathbf{Else}\ t_3 : A}$$

□

Question 12. Soit le terme `(True && 2) + 1`. Ce terme est-il typable selon les règles de la question précédente ? Que se passe-t-il lors de l'exécution avec votre interpréteur ? Avec votre compilateur ?

Réponse : Le terme n'est pas typable car $E \vdash \texttt{2} : \mathbf{Nat}$ (et non pas $E \vdash \texttt{2} : \mathbf{Bool}$).

L'interpréteur échoue lors du calcul de **True** `&&` 2 car il évalue `2` comme `Num_v 2` – plus précisément c'est l'appel `inter_bool (Num 2)` qui va échouer. Le code ASEC en revanche s'exécute sans erreur et calcule la valeur entière 3.

Remarque : Ni l'interpréteur, ni la machine ASEC ne « vérifient les types », ils ne disent rien par ex. du terme 1 + **True**. Tous deux commencent à évaluer le terme, et réagiront lorsqu'une erreur de type empêche l'exécution de se poursuivre. — notons au passage que ces erreurs sont définies sur les valeurs et non sur les termes.

Dans le cas de la machine ASEC, aucune erreur ne se produit, mais le résultat rendu est non-conforme à la sémantique.

□

Question 13. Le domaine de booléens de Scott $\mathcal{B}$ est constitué de trois valeurs $\bot_{\mathcal{B}}$, *true* et *false*, avec $\bot_{\mathcal{B}} \preceq$ *true* et $\bot_{\mathcal{B}} \preceq$ *false*. On interprète le type **Bool** comme étant $\mathcal{B}$. Dans le cadre de PCF explicitement typé, donner la sémantique dénotationnelle du terme suivant :

Fun `(x:Bool)` `->` **Fun** `(y:Bool)` `->` `x && y`

On remarquera qu'une fonction « à deux arguments » est en fait une fonction qui renvoie une fonction.

Réponse : On définit les fonctions suivantes de $\mathcal{B}$ dans $\mathcal{B}$.

- $\bot_{\mathcal{B}\mapsto\mathcal{B}}$, la fonction qui à tout élément de $\mathcal{B}$, associe $\bot_{\mathcal{B}}$.
- F la fonction constante qui à tout élément de $\mathcal{B}$ associe *false*.

- I l'identité sur $\mathcal{B}$.

Alors la dénotation du terme est la fonction de $\mathcal{B}$ dans $\mathcal{B} \mapsto \mathcal{B}$, qui

- À $\bot_{\mathcal{B}}$, associe $\bot_{\mathcal{B}\mapsto\mathcal{B}}$.
- À *false*, associe F.
- À *true*, associe I.

□

Annexes

Syntaxe abstraite des termes de PCF (fichier `ast.mli`)

```
type op = Add | Sub | Mul | Div

type t =
  | Num of int
  | Var of string
  | Op  of op * t * t
  | Ifz of t * t * t
  | Let of string * t * t
  | App of t * t
  | Fun of string * t
  | Fix of string * t
```

Interpréteur par valeur

```
open Ast

type value = Num_v of int | Clo_v of string * Ast.t * env
and env = (string * value) list

open Printf
let print_value chan v = match v with
| Num_v i -> fprintf chan "%i" i
| Clo_v (_,_,_) -> fprintf chan "<fun>"


exception Error of string

let rec interv env t = match t with
| Num i -> Num_v i
| Var x ->
    begin try List.assoc x env
    with Not_found ->
      raise (Error ("variable: "^x^" undefined")) end
| Op (op,t1,t2) ->
    let n1 = inter_int env t1 in
    let n2 = inter_int env t2 in
    Num_v
      (match op with
      | Add -> n1+n2
      | Sub -> let m = n1-n2 in if m < 0 then 0 else m
      | Mul -> n1*n2
      | Div -> n1/n2)
| Ifz (t1,t2,t3) ->
    let v1 = inter_int env t1 in
    interv env (if v1 = 0 then t2 else t3)
| Let (x,t1,t2) ->
    let v1 = interv env t1 in
    interv ((x,v1)::env) t2
| App (t1,t2) ->
```

```
    let x,t_clo,e_clo = inter_clo env t1 in
    let v2 = interv env t2 in
    interv ((x,v2)::e_clo) t_clo
| Fun (x,t) -> Clo_v (x,t,env)
| Fix (f,Fun (x,t)) ->
    let rec clo = Clo_v (x,t,(f,clo)::env) in
    clo
| Fix _ -> raise (Error "Fix allowed on Fun only")

and inter_int env t = match interv env t with
| Num_v i -> i
| Clo_v _ -> raise (Error "Int expected, got Clo")

and inter_clo env t = match interv env t with
| Clo_v (x,t,e) -> (x,t,e)
| Num_v _ -> raise (Error "Clo expected, got Int")
```