Acute

High-level programming language design for distributed computation

Design rationale and language definition

12th October 2004

Peter Sewell[*] James J. Leifer[†] Keith Wansbrough[*] Mair Allen-Williams[*]
Francesco Zappa Nardelli[†] Pierre Habouzit[†] Viktor Vafeiadis[*]

[*]University of Cambridge [†]INRIA Rocquencourt

`http://www.cl.cam.ac.uk/users/pes20/acute`

N° 5329

October 2004

THÈME 1

ISSN 0249-6399

Acute

High-level programming language design for distributed computation

Design rationale and language definition

12th October 2004

Peter Sewell[*] James J. Leifer[†] Keith Wansbrough[*] Mair Allen-Williams[*]
Francesco Zappa Nardelli[†] Pierre Habouzit[†] Viktor Vafeiadis[*]

[*]University of Cambridge [†]INRIA Rocquencourt

`http://www.cl.cam.ac.uk/users/pes20/acute`

Thème 1 — Réseaux et systèmes
Projet Moscova

Rapport de recherche n° 5329 — October 2004 — 193 pages

Abstract: This paper studies key issues for distributed programming in high-level languages. We discuss the design space and describe an experimental language, Acute, which we have defined and implemented.

Acute extends an OCaml core to support distributed development, deployment, and execution, allowing type-safe interaction between separately-built programs. It is expressive enough to enable a wide variety of distributed infrastructure layers to be written as simple library code above the byte-string network and persistent store APIs, disentangling the language runtime from communication.

This requires a synthesis of novel and existing features: (1) type-safe marshalling of values between programs; (2) dynamic loading and controlled rebinding to local resources; (3) modules and abstract types with abstraction boundaries that are respected by interaction; (4) global names, generated either freshly or based on module hashes: at the type level, as runtime names for abstract types; and at the term level, as channel names and other interaction handles; (5) versions and version constraints, integrated with type identity; (6) local concurrency and thread thunkification; and (7) second-order polymorphism with a `namecase` construct. We deal with the interplay among these features and the core, and develop a semantic definition that tracks abstraction boundaries, global names, and hashes throughout compilation and execution, but which still admits an efficient implementation strategy.

Key-words: programming languages, distributed programming, marshalling, serialisation, abstract types, modules, rebinding, version control, type theory, ML

Acute

Langage de programmation de haut niveau pour la programmation des systèmes distribués

Motivations et définition du langage

Résumé : Cet article étudie les problèmes clés posés par la conception et l'implémentation de langages pour la programmation distribuée. Nous explorons l'espace de conception et proposons un langage, Acute, que nous avons défini et implémenté.

Acute étend le noyau de OCaml afin de faciliter le développement, le déploiement et l'exécution des programmes répartis. Il garantit la préservation des types et des abstractions, même lors des interactions entre programmes développés séparément. Acute ne se focalise pas sur le média de transport ou de stockage (TCP, UDP, fichiers, ...) : il est suffisamment expressif pour permettre l'implémentation aisée, sous forme de bibliothèques de communication, de multiples schémas d'interaction.

Cela requiert une synthèse de fonctionnalités nouvelles et existantes : (1) sérialisation sûre des valeurs ; (2) liaison dynamique des valeurs transmises aux ressources locales ; (3) système de modules et types abstraits ; (4) espace de nommage global ; les noms sont soit construits frais à l'exécution, soit basés sur des « hashes » de modules ; ils sont utilisés à la fois pour les types abstraits, et comme clés de multiplexage pour les interactions ; (5) gestion de versions de modules et de contraintes sur ces versions ; (6) concurrence locale et « thunkification » des threads ; (7) polymorphisme du deuxième ordre.

Nous intégrons ces fonctionnalités par dessus le noyau OCaml. Nous avons développé une sémantique qui préserve les abstractions, les noms globaux, et les « hashes » à travers les processus de compilation et d'exécution, et qui, malgré tout, se prête à une implémentation efficace.

Mots-clés : langages de programmation, programmation distribuée, marshalling, sérialisation, types abstraits, modules, théorie des types, ML

Contents

Typing judgements

List of Figures

1 Introduction

Distributed computation is now pervasive, with execution, software development, and deployment spread over large networks, long timescales, and multiple administrative domains. Because of this, many distributed systems cannot be deployed or updated atomically. They are not composed of multiple instances of a single program version, but instead of many versions of many programs that need to interoperate, perhaps sharing some libraries but not others. Moreover, the intrinsic concurrency and nondeterminism of distributed systems, and the complexity of the underlying network layers, makes them particularly hard to understand and debug.

Existing programming languages, such as ML, Haskell, Java and C$^\sharp$, provide good support for local computation, with rich type structures and (mostly) static guarantees of type safety. When it comes to distributed computation, however, they fall short, with little support for its many system-development challenges.

This paper addresses the design of distributed languages. Our focus is on the higher-order, typed, call-by-value programming of the ML tradition: we concentrate on what must be added to ML-like languages to support typed distributed programming. We discuss the design space and describe a programming language, Acute, which we have defined and implemented.

Acute extends an OCaml core with a synthesis of several novel and existing features, broadly addressing naming and identity in the distributed setting. It is not a proposal for a full-scale language, but rather a testbed for experimentation. Our extensions are lightweight changes to ML, but suffice to enable sophisticated distributed infrastructure, e.g. substantial parts of JoCaml [JoC] or Nomadic Pict [SWP99], to be programmed as simple libraries (and its support for interaction between programs goes well beyond these). We demonstrate this with an example typed communication library, written in Acute above the byte-string TCP Sockets API, which requires and uses most of the new features.

The paper is divided into four parts. Part I is devoted to an informal presentation of the main design points from the programmer's point of view, omitting details of the semantics. It is supported by a full definition of Acute in Part III, with the main points of the semantics explained in Part II (including the compiled code and marshalled values from the Part I examples), and by an implementation. The definition covers syntax, typing, compilation, and operational semantics. The implementation is a prototype, efficient enough to run moderate examples while remaining close to the semantics. Part IV gives the Acute code for the communication infrastructure example of §11. Part V gives a brief description of the implementation together with the current command-line options, concrete syntax and standard libraries. The definition and implementation have both been essential: the synthesis of the various features has involved many semantic subtleties. The definition is too large (on the scale of the ML definition rather than an idealised λ-calculus) to make proofs of the properties feasible with the available resources and tools. To increase confidence in both semantics and implementation, therefore, our implementation can optionally type-check the entire configuration after each reduction step.

Design rationale Part I is structured as follows, with §2–10 discussing the main design points, §11 demonstrating that Acute does indeed support typeful distributed programs with an example distributed communication infrastructure library, and §12 and §13 describing related and future work and concluding. An appendix summarises most of the Acute syntax.

§2 and §3 set the scene: we discuss the right level of abstraction for a general-purpose distributed language, arguing that it should not have a commitment to any particular form of communication. We then recall the design choices for simple type-safe marshalling, for trusted and untrusted interaction.

§4: We introduce *dynamic linking* and *rebinding* to local resources in the setting of a language with an ML-like second-class module system. There are many questions here: of how to specify which resources should be shipped with a marshalled value and which dynamically rebound; what evaluation strategy to use; when rebinding takes effect; and what to rebind to. In this Section our aim is to expose the design choices rather than identify definitive solutions. It is a necessary preliminary to our work in §§5–11. For Acute we make interim choices, reasonably simple and sufficient to bring out the typing and versioning issues involved in rebinding, which here is at the granularity of module identifiers. A running Acute program consists roughly of a sequence of `module` definitions (of ML structures), `imports` of modules with specified signatures, which may or may not be linked, and `marks` which indicate where rebinding can take effect; together with running processes and a shared store.

§5: Type-safe marshalling demands a notion of *type identity* that makes sense across multiple versions of differing programs. For concrete types this is conceptually straightforward, but with abstract types more care is necessary. We generate globally-meaningful *type names* either by *hashing* module definitions, taking their dependencies into account; *freshly at compile-time*; or *freshly at run-time*. The first two enable different builds or different programs to share abstract type names, by sharing their module source code or object code respectively; the last is needed to protect the invariants of modules with effect-full initialisation.

§6: Globally-meaningful *expression-level names* are needed for type-safe interaction, e.g. for communication channel names or RPC handles. They can also be constructed as hashes or created fresh at compile time or run time; we show how these support several important idioms. The polytypic `support` and `swap` operations of Shinwell, Pitts and Gabbay's FreshOCaml [SPG03] are included to support swizzling of local names during communication.

§7: In a single-program development process one ensures the executable is built from a coherent set of versions of its modules by controlling static linking — often by building from a single source tree. With dynamic linking and rebinding more support is required: we add *versions* and *version constraints* to `modules` and `imports` respectively. Allowing these to refer to module names gives flexibility over whether code consumers or producers have control.

§8: There is subtle interplay between versions, modules, imports, and type identity, requiring additional structure in `modules` and `imports`. A mechanism for looking through abstraction boundaries is also needed for some version-change scenarios.

§9: Local concurrency is important for distributed programming. Acute provides a minimal level of support, with threads, mutexes and condition variables. Local messaging libraries can be coded up using these, though in a production implementation they might be built-in for performance. We also provide *thunkification*, allowing a collection of threads (and mutexes and condition variables) to be captured as a thunk that can then be marshalled and communicated (or stored); this enables various constructs for mobility to be coded up.

Part I is an extended version of [SLW$^+$]. The main changes are:

- addition of §4.7 on the relationship between modules and the filesystem;
- addition of §4.8 on module initialisation;
- addition of §4.9 on marshalling references;
- addition of §6.2–§6.4 on naming: name ties, polytypic name operations, and the implementation of names;
- extension of §7 on versioning;
- extension of §8.2 on breaking abstractions and `with!`;
- addition of §8.5 on marshalling inside abstraction boundaries;
- extension of §9 on concurrency, with §9.1–9.11 covering the choices for threads and `thunkify` in more detail, discussing several interactions between language features; and
- addition of §10 on polymorphism and `namecase`.

Semantics and Implementation The definition of compilation describes how global type- and expression-level names are constructed. Unusually, the semantics preserves the module structure throughout computation, instead of substituting it away; this is needed to express rebinding. Abstraction boundaries are also preserved, with a generalisation of the *coloured brackets* of Grossman et al [GMZ00] to the entire Acute language (except, to date, the System F constructs). This is technically delicate (and not needed for implementations, which can erase all brackets) but provides useful clarity in a setting where abstraction boundaries may be complex, with abstract types shared between programs.

The semantics preserves also the internal structure of hashes and type data associated with freshly-created names. This too can be erased in implementations, which can implement hashes and fresh names with literal bit-strings (e.g. 160-bit SHA1 hashes and pseudo-random numbers), but is needed to state type preservation and progress properties. The abstraction-preserving semantics makes these rather stronger than usual.

The Acute implementation is written in FreshOCaml, as a meta-experiment in using the Fresh features for a medium-scale program (some 25 000 lines). It is a prototype: designed to be efficient enough to run moderate examples while remaining rather close to the semantics. The runtime interprets an intermediate language which is essentially the abstract syntax extended with closures.

Syntax For concreteness we summarise the most interesting constructs of Acute for types, expressions, and definitions. The full grammar is given in the Definition and summarised in an appendix. The highlighted forms do not occur in source programs. Here h is a module name, hash- or freshly-generated; n is a freshly-generated name, and $[e]^{T}_{eqs}$ is a coloured bracket. The other constructs are explained later.

$$T ::= \ldots \mid T\ \mathsf{name} \mid \mathsf{thread} \mid h.\mathrm{t} \mid \mathrm{n}$$

$$\begin{array}{l} e ::= \ldots \mid \mathbf{marshal}\ e_1\ e_2\ :\ T \mid \mathbf{unmarshal}\ \ e\ \mathbf{as}\ T \mid \\ \quad \mathbf{fresh}_T \mid \mathbf{cfresh}_T \mid \mathbf{hash}(\mathrm{M}_M.\mathrm{x})_T \mid \mathbf{hash}(T, e_2)_{T'} \mid \mathbf{hash}(T, e_2, e_1)_{T'} \mid \\ \quad \mathbf{swap}\ \ e_1\ \ \mathbf{and}\ \ e_2\ \ \mathbf{in}\ \ e_3 \mid \mathbf{support}_T\, e \mid \mathbf{thunkify} \mid [e]^{T}_{eqs} \end{array}$$

$$\begin{array}{l} \mathit{sourcedefinition} ::= \\ \quad \mathbf{module}\ \mathit{mode}\ \mathrm{M}_M : \mathit{Sig}\ \ \mathbf{version}\ \ \mathit{vne} = \mathit{Str}\ \mathit{withspec} \mid \\ \quad \mathbf{import}\ \mathit{mode}\ \mathrm{M}_M : \mathit{Sig}\ \ \mathbf{version}\ \ \mathit{vce}\ \mathit{likespec}\ \ \mathbf{by}\ \ \mathit{resolvespec} = \mathit{Mo} \mid \\ \quad \mathbf{mark}\ \ \mathrm{MK} \end{array}$$

These are added to a substantial fragment of ML. The core language of Acute consists of normal ML types and expressions: booleans, integers, strings, tuples, lists, options, recursive functions, pattern matching, references, exceptions, and invocations of OS primitives in standard libraries. It does not have standard ML-style polymorphism, as our distributed infrastructure examples need first-class existentials (e.g. to code up polymorphic channels) and first-class universals (for marshalling polymorphic functions). We therefore have explicit System F style polymorphism, and for the time being the implementation does some ad-hoc partial inference. The full grammar of types is

$$\begin{array}{lcl} T & ::= & \mathsf{int} \mid \mathsf{bool} \mid \mathsf{string} \mid \mathsf{unit} \mid \mathsf{char} \mid \mathsf{void} \mid T_1 * .. * T_n \mid T_1 + .. + T_n \mid T \to T' \mid T\ \ \mathsf{list} \mid T\ \ \mathsf{option} \mid T\ \ \mathsf{ref} \mid \mathsf{exn} \mid \\ & & \mathrm{M}_M.\mathrm{t} \mid t \mid \forall\ t.T \mid \exists\ t.T \mid T\ \ \mathsf{name} \mid T\ \ \mathsf{tie} \mid \mathsf{thread} \mid \mathsf{mutex} \mid \mathsf{cvar} \mid \mathsf{thunkifymode} \mid \mathsf{thunkkey} \mid \mathsf{thunklet} \mid h.\mathrm{t} \mid \mathrm{n} \end{array}$$

The module language includes top-level declarations of structures, containing expression fields and type fields, with both abstract and manifest types in signatures. Module initialisation can involve arbitrary computation.

We omit some other standard features, simply to keep the language small: user-defined type operators, constructors, and exceptions; substructures; and functors (we believe that adding first-order applicative functors would be straightforward; going beyond that would be more interesting). Some more substantial extensions are discussed in the Conclusion. To avoid syntax debate we fix on one in use, that of OCaml.

Contribution Our contribution is threefold: discussion of the design space and identification of features needed for high-level typed distributed programming, the synthesis of those features into a usable experimental language, and their detailed semantic design. We build on our previous work on global type names and dynamic rebinding [Sew01, LPSW03, BHS$^+$03] which developed some of these ideas for small calculi. The main technical innovations here are: a uniform treatment of names created by hash, fresh, or compile-time fresh, both for type names and (covering the main usage scenarios) for expression names, dealing with module initialisation and dependent-record modules; explicit versions and version constraints, with their delicate interplay with imports and type equality; module-level dynamic linking and rebinding; support for thunkification; and an abstraction-preserving semantics for all the above.

Part I
Design Rationale

2 Distributed abstractions: language vs libraries

A fundamental question for a distributed language is what communication should be built in to the language runtime and what should be left to libraries. The runtime must be widely deployed and so is not easily changed, whereas additional libraries can easily be added locally. In contrast to some previous languages (e.g. Facile [TLK96], Obliq [Car95], and JoCaml [JoC]), we believe that *a general-purpose distributed programming language should not have a built-in commitment to any particular means of interaction.*

The reason for this is essentially the complexity of the distributed environment: system designers must deal with partial failure, attack, and mobility — of code, of devices, and of running computations. This complexity demands a great variety of communication and persistent store abstractions, with varying performance, security, and robustness properties. At one extreme there are systems with tightly-coupled computation over a reliable network in a single trust domain. Here it might be appropriate to use a distributed shared memory abstraction, implemented above TCP. At another extreme, interaction may be intrinsically asynchronous between mutually-untrusting runtimes, e.g. with cryptographic certificates communicated via portable persistent storage devices (smartcards or memory sticks), between machines that have no network connection. In between, there are systems that require asynchronous messaging or RMI but, depending on the network firewall structure, tunnel this over a variety of network protocols.

To attempt to build in direct support for all the required abstractions, in a single general-purpose language, would be a never-ending task. Rather, the language should be at a level of abstraction that makes distribution and communication explicit, allowing distributed abstractions to be expressed as libraries.

Acute has constructs `marshal` and `unmarshal` to convert arbitrary values to and from byte strings; they can be used above any byte-oriented persistent storage or communication APIs.

This leaves the questions of (a) how these should behave, especially for values of functional or abstract types, and (b) what other local expressiveness is required, especially in the type system, to make it possible to code the many required libraries. The rest of the paper is devoted to these.

3 Basic type-safe distributed interaction

In this section we establish our basic conventions and assumptions, beginning with the simplest possible examples of type-safe marshalling. We first consider one program that sends the result of marshalling `5` on a fixed channel:

```
IO.send( marshal "StdLib" 5 : int )
```

(ignore the `"StdLib"` for now) and another that receives it, adds `3` and prints the result:

```
IO.print_int(3+(unmarshal(IO.receive()) as int))
```

Compiling the two programs and then executing them in parallel results in the second printing `8`. This and subsequent examples are executable Acute code. For brevity they use a simple address-less `IO` library, providing communication primitives `send:string->unit` and `receive:unit->string`. (There are two implementations of `IO`, one uses TCP via the Acute sockets API, with the loopback interface and a fixed port; the other writes and reads strings from a file with a fixed name.) Below we write the parallel execution of the two separately-built programs `p1` and `p2` separated by a —.

For safety, a type check is obviously needed at run-time in the second program, to ensure that the type of the marshalled value is compatible with the type at which it will be used. For example, the second program here

```
IO.send( marshal "StdLib" "five" : string )
—
```

```
IO.print_int(3+(unmarshal(IO.receive()) as int))
```

should raise an exception. Allowing interaction via an untyped medium inevitably means that some dynamic errors are possible, but they should be restricted to clearly-identifiable program points, and detected as early as possible. This error can be detected at unmarshal-time, rather than when the received value is used as an argument to `+`, so we should do that type check at unmarshal-time. (In some scenarios one may be able to exclude such errors at compile-time, e.g. when communicating on a typed channel; we return to this in §6.)

The `unmarshal` dynamic check might be of two strengths. We can:

(a) include with the marshalled value an explicit representation of the type at which it was marshalled, and check at unmarshal-time that that type is equal to the type expected by the `unmarshal` — in the examples above, `int=int` and `string=int` respectively; or

(b) additionally check that the marshalled value is a well-formed representation of something of that type.

In a trusted setting, where one can assume that the string was created by marshalling in a well-behaved runtime (which might be assured by network locality or by cryptographically-protected interaction with trusted partners), option (a) suffices for safety.

If, however, the string might have been created or modified by an attacker, then we should choose (b), to protect the integrity of the local runtime. This option is not always available, however: when we consider marshalled values of an abstract type, it may not be possible to check at unmarshal-time that the intended invariants of the type are satisfied. They may have never been expressed explicitly, or be truly global properties. In this case one should marshal only values of concrete types.[1]

A full language should provide both, but in Acute we focus on the trusted case, with option (a), and the problems of distributed typing, naming, and rebinding it raises. Techniques for the untrusted case, including XML support and proof-carrying code, are also necessary but are largely orthogonal.

We do not discuss the design of the concrete wire format for marshalled values — the Acute semantics presupposes just a partial raw_unmarshal function from strings to abstract syntax of configurations, including definitions and store fragments; the prototype implementation simply uses canonical pretty-prints of abstract syntax. A production language would need an efficient and standardised internal wire format, and for some purposes (and for simple types) a canonical XML representation would be useful for interoperation. In the untrusted case XML is now widely used and good language support for (b) is clearly important.

4 Dynamic linking and rebinding to local resources

4.1 References to local resources

Marshalling closed values, such as the `5` and `"five"` above, is conceptually straightforward. The design space becomes more interesting when we consider marshalling a value that refers to some local resources. For example, the source code of a function (it may be useful to think of a large plug-in software component) might mention identifiers for:

(1) ubiquitous standard library calls, e.g., `print_int`;
(2) application-specific library calls with location-dependent semantics, e.g., routing functions;
(3) application code that is not location-dependent but is known to be present at all relevant sites; and
(4) other let-bound application values.

In (1–3) the function should be *rebound* to the local resource where and when it is unmarshalled, whereas in (4) the definitions of resources must be copied and sent along before their usages can be evaluated.

There is another possibility: a local resource could be converted into a *distributed reference* when the function is marshalled, and usages of it indirected via further network communication. In some scenarios this may be desirable, but in others it is not, where one cannot pay the performance cost for those future invocations, or cannot depend

[1]One could imagine an intermediate point, checking the representation type but ignoring the invariants, but the possibility of breaking key invariants is in general as serious as the possibility of breaking the local runtime.

on future reliable communication (and do not want to make each invocation of the resource separately subject to communication failures). Most sharply, where the function is marshalled to persistent store, and unmarshalled after the original process has terminated, distributed references are nonsensical. Following the design rationale of §2, we do not support distributed references directly, aiming rather to ensure our language is expressive enough to allow libraries of 'remotable' resources to be written above our lower-level marshalling primitives.

4.2 What to ship and what to rebind

Which definitions fall into (2) (to be rebound) and (4) (to be shipped) must be specified by the programmer; there is usually no way for an implementation to infer the correct behaviour. How this should be expressed in the language is explored below.

On the other hand, tracking which definitions need not be shipped (3) because they are present at the receiver can be amenable to automation in some scenarios: in the case where we have good connectivity, and are communicating one-to-one rather than via multicast, the two parties can exchange fingerprints of what is required/present. If there is a repeated interchange of messages, the parties may even cache this data from one to another. We believe a good language should make it possible to encode such algorithms, but again, the variety of choices of desirable distributed behaviour leads us to believe that none should be built in. Encoding them requires some reflectivity — to inspect the set of resources required by a value, and calculate the subset of those that are not already present at the receiver. In this paper we do not go into this further, and such *negotiation* protocols are not expressible in Acute at present.

Instead, we adapt the mechanism proposed in [BHS$^+$03] (from a lambda-calculus setting to an ML-style module language) to indicate which resources should be rebound and which shipped for any marshal operation. An Acute program consists roughly of a sequence of module definitions, interspersed with *marks*, followed by running processes; those module definitions, together with implicit module definitions for standard libraries, are the resources. Marks essentially name the sequence of module definitions preceding them. Marshal operations are each with respect to a mark; the modules below that mark are shipped and references to modules above that mark are rebound, to whatever local definitions may be present at the receiver. The mark `"StdLib"` used in §3 is declared at the end of the standard library; both this mark and library are in scope in all examples.

In the following example the sender declares a module `M` with a `y` field of type `int` and value `6`. It then marshals and sends the value `fun ()->M.y`. This `marshal` is with respect to mark `"StdLib"`, which lies above the definition of module `M`, so a copy of the `M` definition is marshalled up with the value `fun ()->M.y`. Hence, when this function is applied to `()` in the receiver the evaluation of `M.y` can use that copy, resulting in `6`.

```
  module M : sig val y:int end = struct let y=6 end
  IO.send( marshal "StdLib" (fun ()->M.y))
—
  (unmarshal (IO.receive ()) as unit -> int) ()
```

On the other hand, references to modules above the specified mark can be rebound. In the simplest case, one can rebind to an identical copy of a module that is already present on the receiver (for (3) or (1)). In the example below, the `M1.y` reference to `M1` is rebound, whereas the first definition of `M2` is copied and sent with the marshalled value. This results in `()` and `((6,3),4)` for the two programs.

```
  module M1:sig val y:int end = struct let y=6 end
  mark "MK"
  module M2:sig val z:int end = struct let z=3 end

  IO.send( marshal "MK" (fun ()-> (M1.y,M2.z))
                                : unit->int*int)
—
  module M1:sig val y:int end = struct let y=6 end
  module M2:sig val z:int end = struct let z=4 end
  ((unmarshal(IO.receive()) as unit->int*int)(),M2.z)
```

Note that we must permit running programs to contain multiple modules with the same source-code name and interface but with different definitions — here, after the unmarshal, the receiver has two versions of `M2` present, one used by the unmarshalled code and the other by the original receiver code.

In more interesting examples one may want to rebind to a local definition of `M1` even if it is not identical, to pick up some truly location-dependent library. The code below shows this, terminating with `()` and `(7,3)`.

```
module M1:sig val y:int end = struct let y=6 end
import M1:sig val y:int end version * = M1
mark "MK"
module M2:sig val z:int end = struct let z=3 end
IO.send( marshal "MK" (fun ()-> (M1.y,M2.z))
                                : unit->int*int )
—
module M1:sig val y:int end = struct let y=7 end
module M2:sig val z:int end = struct let z=4 end
(unmarshal (IO.receive ()) as unit->int*int) ()
```

The sender has two modules, `M1` and `M2`, with `M1` above the mark `MK`. It marshals a value `fun ()-> (M1.y,M2.z)`, that refers to both of them, with respect to that mark. This treats `M2.z` statically and `M1.y` dynamically at the marshal/unmarshal point: a copy of `M2` is sent along, and on unmarshalling at the receiver the value is rebound to the local definition of `M1`, in which `y=7`. To permit this rebinding we add an explicit *import*

```
import M1  :  sig val y:int end version * = M1
```

An import introduces a module identifier (the left `M1`) with a signature; it may or may not be linked to an earlier module or import (this one is, to the earlier `M1`). The `version *` overrides the default behaviour, which would constrain rebinding only to identical copies of `M1`. Marks are simply string constants, not binders subject to alpha equivalence, as they need to be dynamically rebound. For example, if one marshals a function that has an embedded `marshal` with respect to `"StdLib"`, and then unmarshals and executes it elsewere, one typically wants the embedded `marshal` to act with respect to the now-local `"StdLib"`.

4.3 Evaluation strategy: the relative timing of variable instantiation and marshalling

A language with rebinding cannot use a standard call-by-value operational semantics, which substitutes out identifier definitions as it comes to them, as some definitions may need to be rebound later. Two alternative CBV reduction strategies were developed in [BHS$^+$03] in a simple lambda-calculus setting: *redex-time*, in which one instantiates an identifier with its value only when the identifier occurs in redex-position, and *destruct-time* where instantiation may occur even later. Here, to make the semantics as intuitive as possible, we use the redex-time strategy for module references (local expression reduction remains standard CBV).

For example, the first occurrence of `M.y` in the first program below will be instantiated by `6` before the marshal happens, whereas the second occurrence would not appear in redex-position until a subsequent unmarshal and application of the function to `()`; the second occurrence is thus subject to rebinding. The results are `()` and `(6,2)`.

```
module M:sig val y:int end = struct let y=6 end
import M:sig val y:int end version * = M
mark "MK"
IO.send( marshal "MK" (M.y, fun ()-> M.y)
                           : int * (unit->int) )
—
module M:sig val y:int end = struct let y=2 end
let ((x:int),(f:unit->int)) =
  (unmarshal(IO.receive()) as int*(unit->int)) in
(x, f ())
```

4.4 The structure of marks and modules

A running Acute program has a linear sequence of evaluated definitions (marks, module definitions and imports) scoping in the running processes. Imports may be linked only to module definitions (or imports) that precede them in this sequence. When a value is unmarshalled that carried additional module definitions with it, those definitions are added to the end of the sequence.

This linear structure is not ideal. There are some obvious possible alternatives, whose exploration we leave for future work. An unordered set of module definitions would allow cyclic linking; or a tree structure would allow the usual structure of nested scopes to be expressed. In a sufficiently reflective language (i.e. one that would support negotiation, as mentioned above) one could think of coding up marks, dynamically maintaining particular sets of module names. One might well want explicit control over what must *not* be shipped, e.g. due to license restrictions or security concerns.

With any mark structure one has to decide where to put module definitions carried with values being unmarshalled. A useful criterion is to ensure that *repeated* marshalling/unmarshalling, moving code between many machines, behaves well. With the linear structure, putting definitions at the end of the sequence ensures they are inside all marks, and so will be picked up by subsequent marshals. In the hierarchical or unordered cases it is less clear what to do.

A further criterion is that the user of a module should not be required to know its dependency tree — in particular, if one specifies that the module be shipped, other modules that it may have dynamically loaded should be treated sensibly.

We also have to decide what to do with marks occurring between modules being marshalled: they can either be discarded or copied and sent. In Acute we take the latter semantics, but neither is fully satisfactory: in one, shipped module code may refer to marks that are not present locally; in the other there can be unwanted mark shadowing. This is a limitation of the linear structure.

4.5 Controlling when rebinding happens

We have to choose whether or not to allow execution of partial programs, which are those in which some imports are not linked to any earlier module definition (or import). Partial programs can arise in two ways. First, they can be written as such, as in conventional programs that use dynamic linking, where a library is omitted from the statically-linked code, to be discovered and loaded at runtime. For example:

```
import M : sig val y:int end version * = unlinked
fun () -> M.y
```

Secondly, they can be generated by marshalling, when one marshals a value that depends on a module above the mark (intending to rebind it on unmarshalling). For example, the final state of the receiver in

```
 module M:sig val y:int end = struct let y=6 end
 import M:sig val y:int end version * = M
 mark "MK"
 IO.send( marshal "MK" (fun ()->M.y) : unit->int )
—
 unmarshal (IO.receive ()) as unit->int
```

is roughly the program below.

```
import M : sig val y:int end version * = unlinked
fun ()-> M.y
```

If we disallow execution of partial programs then, when we unmarshal, all the unlinked imports that were sent with the marshalled value must be linked in to locally-available definitions; the unmarshal should fail if this is not possible.

Alternatively, if we allow execution of partial programs, we must be prepared to deal with an `M.x` in redex position where `M` is declared by an unlinked import. For any particular unmarshal, one might wish to force linking to occur at unmarshal time (to make any errors show up as early as possible) or defer it until the imported modules are actually used. The latter allows successful execution of a program where one happens not to use any functionality that requires libraries which are not present locally. Moreover, the 'usage point' could be expressed either explicitly (as with a call to the Unix `dlopen` dynamic loader) or implicitly, when a module field appears in redex-position.

A full language should support this per-marshal choice, but for simplicity Acute supports only one of the alternatives: it allows execution of partial programs, and no linking is forced at unmarshal time. Instead, when an unlinked `M.x` appears in redex position we look for an `M` to link the import to.

4.6 Controlling what to rebind to

How to look for such an `M` is specified by a *resolvespec* that can (optionally) be included in the import. By default it will be looked for just in the running program, in the sequence of modules defined above the import. Sometimes, though, one may wish to search in the local filesystem (e.g. for conventional shared-object names such as `libc.so.6`), or even at a web URI. In Acute we make an ad-hoc choice of a simple *resolvespec* language: a resolvespec is a finite list of *atomic resolvespecs*, each of which is either `Static_Link`, `Here_Already` or a URI. Lookup attempts proceed down the list, with `Static_Link` indicating the import should already be linked, `Here_Already` prompting a search for a suitable module (with the right name, signature and version) in the running program, and a URI prompting a file to be fetched and examined for the presence of a suitable module.

There is a tension between a restricted and a general *resolvespec* language. Sometimes one may need the generality of arbitrary computation (as in Java classloaders), e.g. for the negotiation scenario above, or as in browsers that dynamically discover where to obtain a newly-required plugin. On the other hand, a restricted language makes it possible to analyse a program to discover an upper bound on the set of modules it may require — necessary if one is marshalling it to a disconnected device, say. A full language should support both, though the majority of programs might only need the analysable sublanguage.

This *resolvespec* data is added to imports, for example:

```
import M : sig val y:int end version * by
  "http://www.cl.cam.ac.uk/users/pes20/acute/M.ac"
  = unlinked
M.y + 3
```

Here the `M.y` is in redex-position, so the runtime examines the *resolvespec* list associated with the import of `M`. That list has just a single element, the URI `http://www.cl.cam.ac.uk/users/pes20/acute/M.ac`. The file there will be fetched and (if it contains a definition of `M` with the right signature) the modules it contains will be added to the running program just before the import, which will be linked to the definition of `M`. The `M.y` can then be instantiated with its value.

URI *resolvespec*s are, of course, a limited form of distributed reference.

Note that this mechanism is not an exception — after `M` is loaded, the `M.y` is instantiated in its original evaluation context `_ + 3`. It could be encoded (with exceptions and affine continuations, or by encoding imports as option references) but here we focus on the user language.

One would like to be able to limit the resources that a particular unmarshal could rebind to, e.g. to sandboxed versions of libraries, to securely encapsulate untrusted code. This was possible in our earlier λ-calculus work [BHS$^+$03], but to support sufficiently-flexible limits here it seems necessary to have more structure than the Acute linear sequence of marks and modules.

4.7 The relationship between modules and the filesystem

Programs are decomposed not just into modules, but into separate source files. We have to choose whether (1) source files correspond to modules (as in OCaml, where a file named `foo.ml` implicitly defines a module `Foo`), or (2) source files contain sequences of module definitions, and are logically concatenated together in the build process, or (3) both are possible. As we shall see in the following sections, to deal with version change we sometimes need to refer to the results of previous builds. For Acute we take the simplest possible structure that supports this, following (2) with files containing compilation units:

```
compilationunit ::=
  empty
  e
  sourcedefinition ;; compilationunit
  includesource sourcefilename ;; compilationunit
  includecompiled compiledfilename ;; compilationunit
```

The result of compilation is a compiled unit which is just a sequence of compiled module definitions followed by an optional expression.

```
compiledunit ::=
  empty
  e
  definition ;; compiledunit
```

This means that the decomposition of a program into files does not affect its semantics, except that when code is loaded by a URI *resolvespec* an entire compiled unit is loaded.

In Acute any modules shipped with a marshalled value are loaded into the local runtime, but are not saved to local persistent store to be available to future runtime instances. One could envisage a closer integration of communication and package installation.

4.8 Module initialisation

In ML, module evaluation can involve arbitrary computation. For example, in

```
module fresh M : sig val x: int ref  val y:unit             end
            = struct let x=ref 3     let y=IO.print_int !x  end
```

the structure associates non-value expressions to both x and y; the evaluation to a structure value involves expression evaluation which has both store and IO effects. The store effect enables per-module state to be created.

This is also possible in Acute, though as we shall see in §5 it is necessary to distinguish between modules that have such initialisation effects and modules that do not. The evaluation order for a single sequential program is straightforward: a program is roughly a sequence of module definitions followed by an expression; the definitions are evaluated in that order, followed by the expression.

New module definitions can be introduced dynamically, both by unmarshalling and fetched via *resolvespec*s. The evaluation order ensures that any modules that must be marshalled have already been evaluated, and so unmarshalling only ever adds module value definitions to the program.

Consider now the definitions fetched via a *resolvespec*. As we do not have cyclic linking, these definitions must be added before the **import** that demanded them. One could allow such definitions to be compiled units of unevaluated definitions. In the sequential case this would be straightforward: simply by evaluating the extant definition list in order, any newly-added definitions would be evaluated before control returns to the program below. With concurrency, however, there may be multiple threads referring to an import that triggers the addition of new definitions, and some mechanism would be required to block linking of that import until they are fully evaluated (or, equivalently, block instantiation from each new definition until it is evaluated). This flow of control seems complex both from the programmer's point of view and to express in the semantics; we therefore do not allow non-evaluated definitions to be fetched via a *resolvespec*. We return to the interaction between module initialisation and concurrency in §9.8.

In a language with finer-grain control of linking (for the negotiation discussed in §4.2) one might want more control over initialisation, allowing clients to demand their own freshly-initialised occurrences of modules, but Acute does not support this at present.

4.9 Marshalling references

Acute contains ML-style references, so we have to deal with marshalling of values that include store locations. For example:

```
 let (x:int ref) = ref %[int] 5 in IO.send( marshal "StdLib" x : int ref)
—
 IO.print_int ( ! %[int] (unmarshal (IO.receive ()) as int ref ))
```

Here the best choice for the core language semantics seems to be for the marshalled value to include a copy of the reachable part of the store, to be disjointly added to the store of any unmarshaller. Just as in §4.1 we reject the alternative of building in automatic conversion of local references to distributed references, as no single distributed semantics (which here should include distributed garbage collection) will be satisfactory for all applications. A full language must be rich enough to express distributed store libraries above this, of course, and perhaps also other constructs such as those of [SY97, Bou03].

Some applications would demand distributed references together with distributed garbage collection (as JoCaml provides [Fes01]). We leave investigation of this, and of the type-theoretic support it requires, to future work.

One might well add more structure to the store to support more refined marshalling. In particular, one can envisage *regions* of local and of distributed store, perhaps related to the mark structure. We leave the development of this to future work also.

5 Naming: global module and type names

We now turn to marshalling and unmarshalling of values of abstract types. In ML, and in Acute, abstract types can be introduced by modules. For example, the module

```
  module EvenCounter
  : sig             = struct
      type t            type t=int
      val start:t       let start = 0
      val get:t->int    let get = fun (x:int)->x
      val up:t->t       let up = fun (x:int)->2+x
    end               end
```

provides an abstract type `EvenCounter.t` with representation type `int`; this representation type is not revealed in the signature above. The programmer might intend that all values of this type satisfy the 'even' invariant; they can ensure this, no matter how the module is used, simply by checking that the `start` and `up` operations preserve evenness.

Now, for values of type `EvenCounter.t`, what should the unmarshal-time dynamic type equality check of §3 be? It should ensure not just type safety w.r.t. the representation type, but also *abstraction safety* — respecting the invariants of the module. Within a single program, and for communication between programs with identical sources, one can compare such abstract types by their source-code paths, with `EvenCounter.t` having the same meaning in all copies (this is roughly what the manifest type and singleton kind static type systems of Leroy [Ler94] and Harper et al [HL94] do).

For distributed programming we need a notion of type equality that makes sense at runtime across the entire distributed system. This should respect abstraction: two abstract types with the same representation type but completely different operations will have different invariants, and should not be compatible. Moreover, we want common cases of interoperation to 'just work': if two programs share an (effect-free) module that defines an abstract type (and share its dependencies) but differ elsewhere, they should be able to exchange values of that type.

We see three cases, with corresponding ways of constructing globally-meaningful type names.

Case 1 For a module such as `EvenCounter` above that is effect-free (i.e. evaluation of the structure body involves no effects) we can use module *hashes* as global names for abstract types, generalising our earlier work [LPSW03]. The type `EvenCounter.t` is compiled to h`.t`, where the hash h is (roughly)

```
hash(
  module EvenCounter
  : sig             = struct
      type t            type t=int
      val start:t       let start = 0
      val get:t->int    let get = fun (x:int)->x
      val up:t->t       let up = fun(x:int)->2+x
    end               end
)
```

i.e. the hash of the module definition (in fact, of the abstract syntax of the module definition, up to alpha equivalence and type equality, together with some additional data). If one unmarshals a pair of type `EvenCounter.t *`

EvenCounter.t the unmarshal type equality check will compare with *h*.t**h*.t. This allows interoperation to just work between programs that share the EvenCounter source code, without further ado.

In constructing the hash for a module M we have to take into account any dependencies it has on other modules M', replacing any type and term references M'.t and M'.x. In our earlier work we did so by substituting out the definitions of all manifest types and terms (replacing abstract types by their hash type name). Now, to avoid doing that term substitution in the implementation, we replace M'.x by *h'*.x, where *h'* is the hash of the definition of M'. This gives a slightly finer, but we think more intuitive, notion of type equality. We still substitute out the definitions of manifest types from earlier modules. This is forced: in a context where M.t is manifestly equal to int, it should not make any difference to subsequent types which is used.

Case 2 Now consider effect-full modules such as the NCounter module below, where evaluating the up expression to a value involves an IO effect.

```
module fresh NCounter
: sig            = struct
    type t           type t=int
    val start:t      let start = 0
    val get:t->int   let get = fun (x:int)->x
    val up:t->t      let up =
                       let step=IO.read_int() in
                       fun (x:int)->step+x
  end              end
```

This reads an int from standard input at module initialisation time, and the invariant — that all values of type NCounter.t are a multiple of that int — depends on that effect. For such effect-full modules a fresh type name should be generated each time the module is initialised, at run-time, to ensure abstraction safety.

Case 3 Returning to effect-free modules, the programmer may wish to *force* a fresh type name to be generated, to avoid accidental type equalities between different 'runs' of the distributed system. A fresh name could be generated each time the module is initialised, as in the second case, or each time the module is compiled. This latter possibility, as in our earlier work [Sew01], enables interoperation between programs linked against the same compiled module, while forbidding interoperation between different builds.

For abstract types associated with modules it suffices to generate hashes or fresh names *h* per module, using the various *h*.t as the global type names for the abstract types of that module.

We let the programmer specify which of the three behaviours is required with a hash, fresh, or cfresh mode in the module definition, writing e.g. module hash EvenCounter. In general it would be abstraction-breaking to specify hash or cfresh for an effect-full module. To prevent this requires some kind of effect analysis, for which we use coarse but simple notions of *valuability*, following [HS00], and of *compile-time valuability*. We say a module is valuable if all of the expressions in its structure are and if its types are hash-generated. The set of valuable expressions is intermediate between the syntactic values and the expressions that a type-and-effect system could identify as effect-free, which in turn are a subset of the semantically effect-free expressions. They can include, e.g., applications of basic operators such as 2+2, providing useful flexibility.

The compile-time valuable, or *cvaluable*, modules can also include cfresh but otherwise are similar to the valuable modules. The *non-valuable* modules are those that are neither valuable nor cvaluable. If none of the fresh, hash or cfresh keywords are specified then a valuable module defaults to hash; a cvaluable module defaults to cfresh; and a non-valuable module must be fresh. On occasion it seems necessary to override the valuability checks, which we make possible with hash! and cfresh! modes. This is discussed in §8.3.

Acute also provides first-class System F existentials, as the experience with Pict [PT00] and Nomadic Pict [SWP99, US01] demonstrates these are important for expressing messaging infrastructures. For these a fresh type name will be constructed at each unpack, to correspond with the static type system.

6 Naming: expression names

Globally-meaningful *expression-level names* are also needed, primarily as interaction handles — dispatch keys for high-level interaction constructs such as asynchronous channels, location-independent communication, reliable messaging, multicast groups, or remote procedure (or function/method) calls. For any of these an interaction involves the communication of a pair of a handle and a value. Taking asynchronous channels as a simple example, these pairs comprise a channel name and a value sent on that channel. A receiver dispatches on the handle, using it to identify a local data structure for the channel (a queue of pending messages or of blocked readers). For type safety, the handle should be associated with a type: the type of values carried by the channel. (RPC is similar except that an additional affine handle must also be communicated for the return value.)

In Acute we build in support for the generation and typing of name expressions, leaving the various and complex dynamics of interaction constructs to be coded up above marshalling and byte-string interaction. As in FreshOCaml [SPG03], for any type *T* we have a type

```
  T name
```

of names associated with it. Values of these types (like type names) can be generated freshly at runtime, freshly at compile-time, or deterministically by hashing, with expression forms `fresh`, `cfresh`, `hash(M.x)`, `hash(`*T*`,`*e*`)`, and `hash(`*T*`,`*e*`,`*e*`)`. We detail these forms below, showing how they support several important scenarios. In each, the basic question is how one establishes a name shared between sender and receiver code such that testing equality of the name ensures the type correctness of communicated values.

The expression `fresh` evaluates to a fresh name at run-time The expression `cfresh` evaluates to a fresh name at compile-time. It is subject to the syntactic restriction that it can only appear in a compile-time valuable context. The expression `hash(M.x)` compiles to the hash of the pair of *n* and the label `x`, where *n* is the (hash- or fresh-)name associated with module `M`, which must have an `x` component. The expression `hash(`*T*`,`*e*`)` evaluates *e* to a string and then computes the hash of that string paired with the runtime representation of *T*. (Recall that a string can be injectively generated from an arbitrary value by marshalling). The expression `hash(`*T*`,`*e2*`,`*e1*`)` evaluates *e1* to a *T'* `name` and *e2* to a string, then hashes the triple of the two and *T*.

Each name form generates *T* `names` that are associated with a type *T*. For `fresh` and `cfresh` it is the type annotation; for `hash(M.x)` it is the type of the `x` component of module `M`; for `hash(`*T*`,`*e*`)` it is *T* itself; and for `hash(`*T*`,`*e2*`,`*e1*`)` it is *T*. Of these, `fresh` is non-valuable; `cfresh` is compile-time valuable; `hash(M.x)` has the same status as `M`; and `hash(`*T*`,`*e*`)` and `hash(`*T*`,`*e2*`,`*e1*`)` have the join of the status of their component parts.

(A purer collection of hash constructs, equally expressive, would be `hash(T)`, `hash(`*e1*`,`*e2*`)` (of a name and a string) and `hash(`*e1*`,`*T*`)` (of a name and a type). We chose the set above instead as they seem to be the combinations that one would commonly wish to use.)

6.1 Establishing shared names

For clarity we focus on distributed asynchronous messaging, supposing a module `DChan` which implements a distributed `DChan.send` by sending a marshalled pair of a channel name and a value across the network.

```
module hash DChan :
  sig
    val send : forall t. t name * t -> unit
    val recv : forall t. t name * (t -> unit) -> unit
  end
```

This uses names of type *T* `name` as channel names to communicate values of type *T*.[2]

Scenario 1 The sender and receiver both arise from a single execution of a single build of a single program. The execution was initiated on machine A, and the receiver is present there, but the sender was earlier transmitted to machine B (e.g. within a marshalled lambda abstraction).

[2]Acute does not yet support user-definable type constructors. If it did we would define an abstract type constructor `Chan.c:Type->Type` and have `send : forall t. t Chan.c name * t -> unit`.

Here the sender and receiver can originate from a single lexical scope and a channel name can be generated at runtime with a `fresh` expression. This might be at the expression level, e.g.

```
let (c : int name) = fresh in
```

with sender `DChan.send %[int] (c,v)` and receiver `DChan.recv %[int] (c,f)` for some `v:int` and `f:int->unit` (the `%[int]` is an explicit type application), or a module-level binder

```
module M : sig     val c : int name   end
         = struct  let c = fresh      end
```

generating the fresh name when the `let` is evaluated or the `module` is initialised respectively. This first scenario is basically that supported by JoCaml and Nomadic Pict.

Commonly one might have a single receiver function for a name, and tie together the generation of the name and the definition of the function, with an additional `DChan` field

```
val fresh_recv : forall t. (t -> unit) -> t name
```

implemented simply as

```
Function t -> fun f ->
  let c=fresh in DChan.recv %[t] (c,f); c
```

and used as below.

```
module M : sig  val c : int name  end
 = struct let c = DChan.fresh_recv %[int]
          (fun x -> IO.print_int x+1) end
```

Note that this `M` is an effect-full module, creating the name for `c` at module initialisation time.

Scenario 2 The sender and receiver are in different programs, but both are statically linked to a structure of names that was built previously, with expression `cfresh` for compile-time fresh generation.

Here one has a repository containing a compiled instance of a module such as

```
module cfresh M : sig val c : int name  end
            = struct let c = cfresh    end
```

in a file `m.aco`, which is included by the two programs containing the sender and receiver:

```
includecompiled "m.aco"
DChan.send %[int] (M.c,v)
```
—
```
includecompiled "m.aco"
DChan.recv %[int] (M.c,f)
```

Different builds of the sender and receiver programs will be able to interact, but rebuilding `M` creates a fresh channel name for `c`, so builds of the sender using one build of `M` will not interact with builds of the receiver using another build of `M`.

This can be regarded as a more disciplined alternative to the programmer making use of an explicit off-line name (or GUID) generator and pasting the results into their source code.

Scenario 3 The sender and receiver are in different programs, but both share the source code of a module that defines the function `f` used by the receiver; the hash of that module (and the identifier `f`) is used to generate the name used for communication.

This covers the case in which the sender and receiver are different execution instances of the same program (or minor variants thereof), and one wishes typed communication to work without any (awkward) prior exchange of names via the build process or at runtime. The shared code might be

```
module hash N : sig    val f : int -> unit    end
= struct let f = fun x->IO.print_int (x+100)  end
```

```
module hash M : sig    val c : int name         end
= struct let c = hash(int,"",hash(N.f) %[]) %[] end
```

in a file `nm.ac`, included by the two programs containing the sender and receiver:

```
 includesource "nm.ac"
 DChan.send %[int] (M.c,v)
—
 includesource "nm.ac"
 DChan.recv %[int] (M.c,N.f)
```

The `hash(N.f)` gives a T `name` where T = `int->unit` is the type of `N.f`; the surrounding hash coercion `hash(int,"",_)` constructs an `int name` from this.[3] This involves a certain amount of boiler-plate, with separate structures of functions and of the names used to access them, but it is unclear how that could be improved.

It might be preferable to regard the hash coercion as a family of polymorphic operators, indexed by pairs of type constructors $\Lambda\vec{t}.T_1$ and $\Lambda\vec{t}.T_2$ (of the same arity), of type $\forall\vec{t}.T_1$ `name` $\rightarrow T_2$ `name`.

Scenario 4 The sender and receiver are in different programs, sharing no source code except a type and a string; the hash of the pair of those is used to generate the name used for communication.

```
 let c = hash(int,"foo") %[] in
 DChan.send %[int] (c,v)
—
 let c = hash(int,"foo") %[] in
 DChan.recv %[int] (c,f)
```

This idiom requires the minimum shared information between the two programs. It can be seen as a disciplined, typed, form of the use of untyped “traders” to establish interaction media between separate distributed programs.

Scenario 5 The sender and receiver have established by some means a single typed shared name `c`, but need to construct many shared names for different communication channels. The hash coercion can be used for this also, constructing new typed names from old names, new types, and arbitrary strings. Whether this will be a common idiom is unclear, but it is easy to provide and seems interesting to explore. For example:

```
 let c1 = hash(int,"one",c)
 let c2 = hash(int,"two",c)
 let c3 = hash(bool,"",c)
 DChan.send %[int] (c1,v1);  DChan.send %[int] (c2,v2);  DChan.send %[bool] (c3,v3);
—
 let c1 = hash(int,"one",c)
 let c2 = hash(int,"two",c)
 let c3 = hash(bool,"",c)
 DChan.recv %[int] (c1,f1);  DChan.recv %[int] (c2,f2);  DChan.recv %[bool] (c3,f3);
```

Whether this will be a common idiom is unclear, but it is easy to provide and seems interesting to explore.

6.2 A refinement: ties

Scenario 3 of §6.1 above used a `hash(N.f)` as part of the construction of a name `M.c` used to access the `N.f` function remotely, linking the name and function together with a call `DChan.recv (M.c,N.f)`. It may be desirable to provide stronger language support for establishing this linkage, making it harder to accidentally use an unrelated name and function pair. For this, we propose a built-in abstract type

T `tie`

of those pairs, with an expression form `M@x` that constructs the pair of `hash(M.x)` and the value of `M.x` (of type T `tie` where `M.x :` T), and projections from the abstraction type `name_of_tie` and `val_of_tie`.

[3]Such coercions support `Chan.c` type constructors too, e.g. to construct an `int Chan.c name` from an `(int->unit) name`.

6.3 Polytypic name operations

We include the basic polytypic FreshOCaml expressions for manipulating names:

```
swap e1 and e2 in e3
e1 freshfor e2
support %[T] e
```

Here `swap` interchanges two names in an arbitrary value, `freshfor` determines whether a name does not occur free in an arbitrary value, and `support` calculates the set of names that do occur free in an arbitrary value (returning them as a duplicate-free list, at present).

We anticipate using these operations in the implementation of distributed communication abstractions. For example, when working with certain kinds of distributed channel one must send routing information along with every value, describing how any distributed channels mentioned in that value can be accessed.

We do not include the FreshOCaml name abstraction and pattern matching constructs just for simplicity — we foresee no difficulty in adding them.

In contrast to FreshOCaml, when one has values that mention store locations, the polytypic operations have effect over the reachable part of the Acute heap. This seems forced if we are to both (a) implement distributed abstractions, as above, and (b) exchange values of imperative data type implementations.

For constructing efficient datastructures over names, such as finite maps, we provide access to the underlying order relation, with a comparison between any two names of the same type.

```
compare_name %[T] : T name -> T name -> int
```

This cannot be preserved by name swapping, obviously, and so it would be an error to use it under any name abstraction, and in any other place subject to swapping. Nonetheless, the performance cost of not including it is so great we believe it is required. To ameliorate the problem slightly one might add a type

```
T nonswap
```

with a single constructor `Nonswap` that can be used to protect structures that depend on the ordering, with `swap` either stopping recursing or raising an exception if it encounters the `Nonswap` constructor. For the time being, however, `T nonswap` is not included in Acute.

6.4 Implementing names

In the implementation, all names are represented as fixed-length bit-strings (e.g. from 2^{160}) — both module-level and expression-level names, generated both by hashes and freshly. The representations of fresh names are generated randomly. More specifically: we do not want to require that the implementation generates each individual name randomly, as that would be too costly — we regard it as acceptable to generate a random start point at the initialisation of each compilation and the initialisation of each language runtime instance, and thereafter use a cheap pseudo-random number function for compile-time fresh and run-time fresh (the successor function would lead to poor behaviour in hash tables). This means that a low-level attacker would often be able to tell whether two names originated from the same point, and that (for making real nonces etc) a more aggressively random `fresh` would be required.

There is a possible optimisation which could be worthwhile if many names are used only locally: the bit-string representations could be generated lazily, when they are first marshalled, with a finite map associating local representations (just pointers) to the external names which have been exported or imported. This could be garbage-collected as normal. Whether the optimisation would gain very much is unclear, so we propose not to implement it now (but bear in mind that local channel communication should be made very cheap).

In order to implement the polytypic name operations the expression-level names must be implemented with explicit types.

7 Versions and version constraints

In a single-executable development process, one ensures the executable is built from a coherent set of versions of its component modules by choosing what to link together — in simple cases, by working with a single code directory tree.

In the distributed world, one could do the same: take sufficient care about which modules one links and/or rebinds to. Without any additional support, however, this is an error-prone approach, liable to end up with semantically-incoherent sets of versions of components interoperating. Typechecking can provide some basic sanity guarantees, but cannot capture these semantic differences.

One alternative is to permit rebinding only to identical copies of modules that the code was initially linked to. Usually, though, more flexibility will be required — to permit rebinding to modules with "small" or "backwards-compatible" changes to their semantics, and to pick up intentionally location-dependent modules. It is impractical to specify the semantics that one depends upon in interfaces (in general, theorem proving would be required at link time, though there are intermediate behavioural type systems). We therefore we introduce *versions* as crude approximations to semantic module specifications. We need a language of versions, which will be attached to modules, a language of version constraints, which will be attached to imports, a satisfaction relation, checked at static and dynamic link times, and an implication relation between constraints, for chains of imports.

Now, how expressive should these languages be? Analogously to the situation for *resolvespec*s, there is a tension between allowing arbitrary computation in defining the relations and supporting compile-time analysis. Ultimately, it seems desirable to make the basic module primitives parametric on abstract types of versions and constraints — in a particular distributed code environment, one may want a particular local choice for the languages. For Acute once again we choose not the most general alternative, but instead one which should be expressive enough for many examples, and which exposes some key design points.

Scenario 1 It is common to use version numbers which are supplied by the programmer, with no checked relationship to the code. As an arbitrary starting point, we take version numbers to be nonempty lists of natural numbers, and version constraints to be similar lists possibly ending in a wildcard `*` or an interval; satisfaction is what one would expect, with a `*` matching any (possibly empty) suffix. Many minor enhancements are possible and straightforward. Arbitrarily, we enhance version constraints with closed, left-open and right-open intervals, e.g. `1.5-7`, `1.8.-7`, and `2.4.7-`. These are certainly not exactly what one wants (they cannot express, for example, the set of all versions greater than `2.3.1`) but are indicative. The *meanings* of these numbers and constraints is dependent on some social process: within a single distributed development environment one needs a shared understanding that new versions of a module will be given new version numbers commensurate with their semantic changes.

Scenario 2 To support tighter version control than this, we can make use of the global module names (hash- or freshly-generated) introduced in §5: equality testing of these names is an implementable check for module semantic identity. We let version numbers include `myname` and version constraints include module identifiers `M` (those in scope, obviously). In each case the compiler or runtime writes in the appropriate module name. This supports a useful idiom in which code producers declare their exact identity as the least-significant component of their version number, and consumers can choose whether or not to be that particular. For example, a module `M` might specify it is version `2.3.myname`, compiled to `2.3.0xA564C8F3`; an import in that scope might require `2.3.M`, compiled to `2.3.0xA564C8F3`, or simply `2.3.*`; both would match it.

A key point is the balance of power between code producers and code consumers. The above leaves the code producer in control, who can "lie" about which version a module is — instead of writing `myname` they might write a name from a previous build. This is desirable if they know there are clients out there with an exact-name constraint but also know that their semantic change from that previous build will not break any of the clients.

Scenario 3 Finally, to give the code consumer more control, we allow constraints not only on the version field of a module but also on its actual name (which is unforgeable within the language). Typically one would have *a* definition of the desired version available in the filesystem (in Acute bringing it into scope as `M` with an `include`) and write `name=M`. (These exact-name constraints are also used to construct default `imports` when marshalling). One could also cut-and-paste a name in explicitly: `name=0xA564C8F3`. To guarantee that only mutually-tested collections of modules will be executed together, e.g. when writing safety-critical software, this would be the desired idiom everywhere, perhaps with development-environment support.

The current Acute version numbers and constraints, including all the above, are as follows.

$avne$ `::=`	Atomic version number expression
n	natural number literal
N	numeric hash literal
`myname`	name of this module

```
vne   ::=          Version number expression
  avne   |   avne.vne

avce   ::=         Atomic version constraint expression
  n                  natural number literal
  N                  numeric hash literal
  M                  name of module M

dvce ::=           Dotted version constraint
  avce | n-n' | -n | n- | * | avce.dvce

vce   ::=          Version constraint
  dvce               dotted version constraint
  name = M           exact-name version constraint
```

Version number and constraint expressions appear in modules and imports as below.

```
definition ::= ...
   module M:Sig version vne = Str  ...
 | import M:Sig version vce ...
     by resolvespec = Mo
```

In constructing hashes for modules we also take into account their version expressions, to prevent any accidental equalities. That version expression can mention `myname`, and, as we do not wish to introduce recursive hashes, the hash must be calculated before compilation replaces `myname` with the hash.

It turns out that one needs exact-name version constraints not just for user-specified tight version constraints, as in the idiom above, but also during marshalling, when one may have to generate imports for module bindings that cross a mark. Exact-name constraints seem to be the only reasonable default to use there.

One might wish to extend the version language further with conjunctive version number expressions and disjunctive constraints. One might also wish to support cryptographic signatures on version numbers. Both would affect the balance of power between code producer and consumer, and further experience is needed to discover what is most usable.

Finally, we have had to choose whether version numbers are hereditary or not. A hereditary version number for a module `M` would include the version numbers of all the modules it depends on (and the version constraints of all the imports it uses), whereas a non-hereditary version number is just a single entity, as in the grammar above. The hereditary option clearly provides more data to users of `M`, but, concomitantly, requires those users to understand the dependency structure — which usually one would like a module system to insulate them from. If one really needs hereditary numbers, perhaps the best solution would be to support version number expressions that can calculate a number for `M` in terms of the numbers of its immediate dependencies, e.g. adding tuples and `version(M)` expressions to the *avne* grammar.

Just as for *withspec*s one might need rich development-environment support. Local specifications of version constraints, spread over the imports in the source files of a large software system, could be very inconvenient. One might want to refer to the version numbers of a source-control system such as CVS, for example.

8 Interplay between abstract types, rebinding and versions

8.1 Definite and indefinite references

With conventional static linking, module references such as `M.t` are *definite*, in the terminology of [HP]: in any fully-linked executable there is just a single such `M`, though (with separate compilation) it may be unknown at compile-time which module definition for `M` it will be linked to. In contrast, the possibility of rebinding makes some references *indefinite* — during a single distributed execution, they may be bound to different modules.

For example, consider a module that declares an abstract type that depends on the term fields of some other module:

```
module M : sig    val f:int->int        end
         = struct let f=fun(x:int)->x+2 end
module EvenCounter
: sig             = struct
    type t            type t=int
    val start:t       let start = 0
    val get:t->int    let get = fun (x:int)->x
    val up:t->t       let up = fun (x:int)->M.f x
  end               end
```

In the absence of any rebinding, the runtime type name for the abstract type `EvenCounter.t` would be the hash of the `EvenCounter` abstract syntax with `M.f` replaced by h`.f`, where h is the hash of the abstract syntax of `M`. This dependence on the `M` operations guarantees type- and abstraction-preservation.

Now, however, if there is a mark between the two module definitions, a marshal can cut and rebind to any other module with the same signature, perhaps breaking the invariant of `EvenCounter.t` that its values are always even. The `M.f` module reference below is indefinite.

```
module M : sig    val f:int->int         end
         = struct let f=fun (x:int)->x+2 end
import M : sig val f:int->int end  version * = M
mark "MK"
module EvenCounter
: sig             = struct
    type t            type t=int
    val start:t       let start = 0
    val get:t->int    let get = fun (x:int)->x
    val up:t->t       let up = fun (x:int)->M.f x
  end               end
IO.send(marshal "MK" (fun ()->EvenCounter.get
(EvenCounter.up EvenCounter.start)):unit->int)
—
module M : sig    val f:int->int         end
         = struct let f=fun (x:int)->x+3 end
(unmarshal (IO.receive ()) as unit->int) ()
```

To prevent this kind of error one can use a more restrictive version constraint in the import of `M` that `EvenCounter` uses, either by using an exact-name constraint `name=M` to allow linking only to definitions of `M` that are identical to the definition in the sender, or by using some conventional numbering. If no import is given explicitly, an exact-name constraint is assumed.

The version constraint should be understood as an assertion by the code author that whatever `EvenCounter` is linked with, so long as it satisfies that constraint (and also has an appropriate signature, and is obtained following any *resolvespec* present), the intended invariants of `EvenCounter.t` will be preserved.

Now, what should the global type name for `EvenCounter.t` be here? Note that the original author might not have had any `M` module to hand, and even if they did (as above), that module is not privileged in any way: `EvenCounter` may be rebound during computation to other `M` matching the signature and version constraint. In generating the hash for `EvenCounter`, analogously to our replacement of definite references `M'.x` by the hash of the definition of `M'`, we replace indefinite import-bound references such as `M.f` by the hash of the *import*. This names the set of all `M` implementations that match that signature and version constraint.

In the case above this hash would be roughly

```
hash(import M:sig val f:int->int end version *  )
```

and where one imports a module with an abstract type field

```
import M : sig type t  val x:t  end
  version 2.4.7-  ...
```

the hash h =

```
hash(import M : sig type t  val x:t  end
       version 2.4.7-  ...)
```

provides a global name `h.t` for that type.

In the `EvenCounter` example, the imported module had no abstract type fields. Where they do, for type soundness we have to restrict the modules that the import can be linked to, to ensure that they all have the same representation types for these abstract type fields. We do so by requiring imports with abstract type fields to have a *likespec* (in place of the `...` above), giving that information. A compiled *likespec* is essentially a structure with a type field for each of the abstract type fields of the import.

At first sight this is quite unpleasant, requiring the importers of a module to 'know' representation types which one might expect should be hidden. With indefinite references to modules with abstract types, however, some such mechanism seems to be forced, otherwise no rebinding is possible. Moreover, in practice one would often have available a version of the imported library from which the information can be drawn. For example, one might be importing a graphics library that exists in many versions, but for which all versions that share a major version number also have common representations of the abstract types of `point`, `window`, etc. A typical import might have the form

```
import Graphics:sig type t end version 2.3.*
  like Graphics2_0
```

(with more types and operations) where `Graphics2_0` is the name of *a* graphics module implementation, which is present at the development site, and which can be used by the compiler to construct a structure with a representation for each of the abstract types of the signature.

While the abstraction boundaries are not as rigid as in ML, this should provide a workable idiom for dealing with large modular evolving systems, supporting rebinding but also allowing one to restrict type representation information to particular layers. The only alternative seems to be to make all types fully concrete at interfaces where rebinding may occur.

To correctly deal with abstract types defined by modules following an import, which use abstract type fields of the imported module in their representation types, compiled *likespecs* must be included in the hashes of imports.

On the other hand, we choose not to include *resolvespec*s in import hashes. This is debatable — the argument against including them is that it is useful to be able to change the location of code without affecting types, and so without breaking interoperation (e.g. to have a local code mirror, to change a web code repository to avoid a denial-of-service attack etc.).

Note that the indefinite character of our `import`s makes them quite different from module imports that are resolved by static linking, where typing can simply use module paths to name any abstract types and no *likespec* machinery is required. Both mechanisms are needed.

8.2 Breaking abstractions

In ongoing software evolution, implementations of an abstract type may need to be changed, to fix bugs or add functionality, while values of that type exist on other machines or in a persistent store. It is often impractical to simultaneously upgrade all machines to a new implementation version.

A simple case is that in which the representation of the abstract type is unchanged and where the programmer asserts that the two versions have compatible invariants, so it is legitimate to exchange values in both directions. This may be the case even if the two are not identical, e.g. for an efficiency improvement or bug fix. Here there should be some mechanism for forcing the old and new types to be identical, breaking the normal abstraction barrier.

In [Sew01, LPSW03] we proposed a *strong coercion* `with!` to do so, and Acute includes a variant of this. By analogy with ML sharing specifications, we allow a module definition to have a *withspec*, a list of equalities between abstract types and representations of modules constructed earlier (often this will be of previous builds of the same module).

```
definition   ::=  ...
```

```
  module M : Sig version vne = Str  withspec
withspec      ::=  empty | with! withspecbody
withspecbody ::=  empty | M.t=T,withspecbody
```

The compiler checks the representation type of these *M.t* are equal to the types specified (respecting any internal abstraction boundaries); if they are, the type equalities can be used in typechecking this definition.

For example, suppose the `EvenCounter` module definition of §5 was compiled to a file `p11_even.aco` and is widely deployed in a distributed system, and that later one needs a revised `EvenCounter` module, adding an operation or fixing a bug without making an incompatible type. A new module with an added `down` operation can be written as follows.

```
includecompiled "p11_even.aco"
module EvenCounter
: sig
    type t = EvenCounter.t
    val start:t
    val get:t->int
    val up:t->t
    val down:t->t
  end
= struct
    type t=int
    let start = 0
    let get = fun (x:int)->x
    let up = fun (x:int)->2+x
    let down = fun (x:int)-> x-2
  end
with! EvenCounter.t = int
```

In the interface here the type `t` of the new module is manifestly equal to the abstract type `t` of the previously-built module, and the `with!` enables the type equality between that abstract type and `int` to be used when typing the new module. The new type is compiled to be manifestly equal to (the internal hash-name of) the old type. (For this example, where the previous `EvenCounter` had a `hash`-generated type, one could include the source of the previous module rather than the compiled file, but if it were `cfresh`-generated the compiled file is obviously needed.)

The *withspec* is, in effect, a declaration by the programmer that the old and new implementations respect the same important invariants — here, that values of the representation type will always be even. In general they will not respect exactly the same invariants. For example, here the new module allows negative `int`s, but the programmer implicitly asserts that the clients of the old module will not be broken by this.

It would not suffice to check only that the new module respects at least the important invariants of the old, as if the types are made identical then values produced by either module can be acted upon by operations of the other.

In the more complex case where the old and new invariants are not compatible, or where the two representation types differ, the programmer will have to write an upgrade function. The same strong coercion can be used to make this possible, with a module that contains two types, one coerced to each. An example is given in [LPSW03].

There are several design options for *withspec*s. In our earlier proposals `with!` coerced an abstract type of the module being defined to be equal to an earlier abstract type. Here instead the `with!` simply introduces a type equality to the typechecking environment; manifest types in the signature of the new module can be used to make the type field of the compiled signature equal to the old. This simplifies the semantics slightly and may be conceptually clearer. We allow the *withspec* type equalities to be used both for typechecking the body of the new module and for checking that it does have the interface specified. One might instead only allow them to be used for the latter; it is unclear whether this would always be expressive enough. The programmer has to specify the representation type in a *withspec* explicitly. This is fine for small examples, e.g. the `int` above, but if the representation type is complex then it would be preferable to simply write `with! M.t`. That requires a somewhat more intricate semantics (as typechecking of modules with *withspec*s then depends on the representation types of earlier modules) and so we omit it for the time being. Finally, one might well want development-environment support, allowing collections of modules to be 'pinned' to the types in a particular earlier build without having to edit each module to add a *withspec* and make the types manifestly equal to the earlier ones.

8.3 Overriding valuability checks

The semantics for abstract type names outlined in §5 ensures that two instances of an effect-full module give rise to distinct abstract types. In general this is the only correct behaviour, as (as explained there) they may have very different invariants. In practice, however, one may often want to permit rebinding to modules which have some internal state. For example, in the communication library described in §11 the `Distributed_channel` module stores a `Tcp_string_messaging.handle option` which is set by calls to `Distributed_channel.init : Tcp.port -> unit`. One has to keep this as module state rather than threading a handle through the `Distributed_channel` interface calls so that those calls can be correctly rebound if (say) one marshals a function mentioning them. Despite the initialisation effect (evaluating `ref None`) we need the module name for `Distributed_channel` to be hash-generated, not fresh-generated, so that the abstract types in the interface are the same in different instance, so that rebinding can take place. The desired behaviour really is for the conceptually-distinct abstract types of different instances to be compatible. This could be expressed either

1. with module annotations **hash**! and **cfresh**!, which override the valuability check but otherwise are like **hash** and **cfresh**; or

2. with an expression form **ignore_effect**(e), transparent at runtime but concealing arbitrary effects as far as valuability goes.

We choose the former, to make the coercion clearer in the module source and to avoid polluting the expression grammar, but the latter has the advantage of localising the coercion to where it is really needed.

8.4 Exact matching or version flexibility?

In §6 we focussed on name-based dispatch. An alternative idiom for remote invocation simply makes use of the dynamic rebinding facilities provided in Acute, e.g. as in the code below where a thunk mentioning `N.f` is shipped from one machine to another.

```
 module N:sig val f:int->unit                  end
     = struct let f=fun x-> IO.print_int (x+1) end
 mark "MARK-N"
 IO.send (marshal "MARK-N" ((fun ()->N.f), 9))
—
 module N:sig val f:int->unit                  end
     = struct let f=fun x-> IO.print_int (x+1) end
 mark "MARK-N"
 let (g,(y:int))=unmarshal(IO.receive()) in g () y
```

As the `marshal` is with respect to a mark (`"MARK-N"`) below the definition of `N`, the pair of the thunk and `v` will be shipped together with an unlinked import for `N`; when the unmarshalled thunk is applied that import will become linked to the local definition of `N` on the receiver machine.

In the code as written the import will have an exact-name version constraint, but this could be liberalised by writing an explicit import in the sender, with an arbitrary version constraint.

This is quite different from the name-based dispatch of §6, where a simple name equality is checked for each communication. Here, a full link-ok check is involved, checking a subsignature relationship and a version constraint. It is therefore much more costly, but also allows much more flexible linking.

Another difference between the two schemes is that with name-based dispatch the receiver can express access-control checks by testing name equality, whereas here one would need to test equality of arbitrary incoming functions (against `fun ()->N.f` thunks), which we do not admit.

A common idiom may be to establish a shared structure of names by dynamic linking (including a version check) at the start of a lengthy interaction and thereafter to use name-based dispatch. Acute does not yet provide the low-level linking machinery needed for explicitly sending such a structure (see the discussion of negotiation elsewhere), so we do not explore this further here.

8.5 Marshalling inside abstraction boundaries

If one has a module defining an abstract type, and within that module marshals a value of that type, one has to choose whether it is marshalled abstractly or concretely. For example, in

```
module EvenCounter
: sig
    type t
    val start:t
    val get:t->int
    val up:t->t
    val send : t -> unit
    val recv : unit -> t
  end
= struct
    type t=int
    let start = 0
    let get = fun (x:int)->x
    let up = fun (x:int)->2+x
    let send = fun (x:t) -> IO.send( marshal "StdLib" x : t)
    let recv = fun () -> (unmarshal(IO.receive()) as t)
  end
EvenCounter.send (EvenCounter.start)
```

is the communicated value compatible with `int` or with `EvenCounter.t`? For Acute we take the former option: all types (in the absence of polymorphism) are fully normalised with respect to the ambient type equations before execution. Running the above in parallel with

```
IO.print_int(3+(unmarshal(IO.receive()) as int))
```

will therefore succeed.

One might well want more source-language control here, allowing the programmer to specify that such a `marshal` should be at the abstract type, but we leave this for future work (but cf. the comment on page **??**). In general, with nested modules and with `with!` specifications, there may be a complex type equation set structure to select from.

9 Concurrency, mobility, and `thunkify`

Distributed programming requires support for local concurrency: some form of threads and constructs for interaction between them.

9.1 Language-level concurrency vs OS threads

The first question here is whether to fix a direct relationship to the underlying OS threads or take language-level threads to be conceptually distinct, which might or might not be implemented with one OS thread each. The former has the advantages of a simple relationship with the OS scheduler (which may provide rich facilities, e.g. for QoS, that some programs need) and the potential to exploit multiple processors. It has the disadvantages of different concurrency models on different OSs, and of a nontrivial relationship between threading and the language garbage collector. The latter gives the language implementor much more freedom. In particular, to support lightweight concurrency (as in Erlang, Pict, JoCaml etc.), in which many parallel components simply send a message or two, it is desirable for parallel composition to not require the (costly) construction of a new OS thread. For Acute we adopt language-level concurrency.

9.2 Interaction primitives

There are two main styles of interaction between threads: shared memory and message passing. The latter is a better fit to large-scale distributed programming and, we believe, often leads to more transparent code. The former, however, is needed when dealing with large mutable datastructures, and suits the imperative nature of ML/OCaml programming. In large programs we expect both to be required. In Acute we initially provide shared-memory interaction, as OCaml does: references can be accessed from multiple threads, with atomic dereferencing and assignment, and mutexes and condition variables can be used for synchronization. These enable certain forms of message-passing interaction to be expressed as library modules, which suffices for the time being. In future we expect to build in support for message-passing. Indeed, some forms require direct language support (or a preprocessor-based implementation), e.g. Join patterns with their multi-way binding construct.

9.3 Thunkification

We want to make it possible to checkpoint and move running computations — for fault-tolerance, for working with intermittently-connected devices, and for system management. Several calculi and languages (JoCaml, Nomadic Pict, Ambients,etc.) provided a linear migration construct, which moved a computation between locations.

It now appears more useful to support marshalling of computations, which can then be communicated, checkpointed etc. using whatever communication and persistent store constructs are in use. Taking a step further, as we have marshalling of arbitrary values, marshalling of computations requires only the addition of a primitive for converting a running computation into a value. We call this *thunkification*. Checkpointing a computation can then be implemented by thunkifying it, marshalling the resulting value, and writing it to disk. Migration can be implemented by thunkification, marshalling, and communication. Note that these are not in general linear operations — if a computation has been checkpointed to disk it may be restarted multiple times.

There are many possible forms of thunkification. The simplest is to be both subjective and synchronous: executing `thunkify` in a single thread gives a thunk of that thread, essentially capturing the (single-thread) continuation of the `thunkify`. Typically, though, the computation which one wishes to thunkify will be composed of a group of threads. The programmer would then have to manually ensure that all the threads synchronize and then thunkify themselves, and collect together the results. This would be very heavy, requiring substantial rewriting of applications to make them amenable to checkpointing or migration. Accordingly, we think it preferable to have an objective and asynchronous `thunkify`, freezing a group of threads irrespective of their current behaviour.

A group of threads may be intertwined with interaction primitives (i.e. mutexes and condition variables) used for internal communication and synchronization. Accordingly, `thunkify` should also be applicable to those interaction primitives.

Thunkification is destructive, removing the threads, mutexes and condition variables that are thunkified.

Thunkification of a group must be atomic. To see the inadequacy of a `thunkify` that operates only on a single thread, consider thunkifying a pair of threads, the first of which is performing a thread operation (e.g. `kill`) on the second. If the second is thunkified before the first then the `kill` will fail, whereas with an atomic multi-thread `thunkify` it will always succeed, either before the `thunkify` happens or after the group is unthunkified later.

9.4 Naming and grouping

Threads must be structured in some fashion. The simplest option, taken by many process calculi, is to have a running system be a flat parallel composition of anonymous threads. In contrast, operating system threads are typically named, with names provided by the system at thread creation time; these names may be reused over time and between runtimes.

For Acute some naming structure is required, to allow threads to be manipulated (thunkified, killed, etc.). We see two main possibilities:

1. globally-unique names, created freshly by the system at thread creation time; or

2. locally-unique names, provided by the programmer at thread creation time, with an exception if they are already in use on this runtime.

The other two possibilities are not useful or not implementable: if names are being created freshly by the system they might as well be globally unique, with the same representation as we use for other names; if names are being provided by the programmer then it is not in general possible to check if they are in use on any runtime.

We expect (1) to be the most commonly desired semantics. Nonetheless, in Acute we choose (2). Firstly, given (2) the programmer can implement (1) simply by providing a fresh name at each thread creation point. The difference between the two shows up when one moves a group of threads, which internally record and manipulate the thread names of the group, from one machine to another. With (1) they necessarily receive new names at the destination, so to maintain correctness all records of their old names must be permuted with the new — which may be awkward if there are external records of these names. With (2), if this movement is known to be linear then the original names can be reused without further ado.

The same two possibilities exist for the naming of interaction primitives for synchronization and communication between threads, i.e. (at present) mutexes and condition variables, and we make the same choice of (2) for them.

Many distributed process calculi have exploited a hierarchical group structure over processes, with boundaries delimiting units of migration, units of failure, synchronization regions, secure encapsulation boundaries, and administrative domains. There is a basic tension between the need for communication across boundaries and the need for encapsulation and control over untrusted components, giving rise to a complex design space which is not well-understood. The tutorial [Sew00] gives a very preliminary overview. How this tension should be resolved and what group structure should be provided as primitive is a very interesting question for future work. We conjecture that groups for migration and synchronization units can be expressed rather easily in Acute with flat parallel compositions of named threads, and that is what the language currently provides.

Any group structure should — presumably — also structure the interaction primitives (mutexes, channels, etc.) but here there are additional complications, as these are necessarily going to be used for interaction across a boundary, so the interactands may be split apart by thunkification.

A further motivation for richer group structure comes from performance requirements. When programming in a message-passing style (as in the π-calculus and in the derived languages JoCaml, Pict, and Nomadic Pict) one may have many threads which contain only a single asynchronous output. For performance it may be necessary to optimise these, not always creating thread names and scheduler entries for them. If threads can discover their own names, e.g. by a

```
self : unit -> thread name
```

primitive, then this optimisation is nontrivial: a thread which outputs the value of an expression involving `self` must have been created with a name, whereas outputs of other values need not. This led us to explore grouping structures of named groups containing anonymous threads. Ultimately we rejected them, returning to the flat parallel compositions of named threads, as they seemed excessively complex and it seemed likely that a rather simple static analysis would be able to identify most non-self outputs.

9.5 Thread termination

Acute threads do not return values, and their termination cannot by synchronized upon. We have no strong opinion about these choices, making them for simplicity for the time being. Thread termination is observable indirectly, as `thunkify` and `kill` raise exceptions if called on non-existent threads.

9.6 Nonexistent threads, mutexes, and condition variables

In conventional single-machine programming it is straightforward to ensure that any mutexes and condition variables used must already exist — in OCaml, for example, the type system guarantees this. In Acute, however, this is no longer possible.

Firstly, mutex names may be marshalled (either alone or in a function such as `function () -> unlock m`) and then unmarshalled on another machine. In the absence of thunkification it is debatable whether this is useful: one might imagine forbidding such examples, either with a dynamic check at marshal-time or a rich type system that identifies non-marshallable types. With thunkification, however, one may certainly need to marshal a thunkified group of threads together with their internal mutexes. Secondly, thunkification can remove a mutex, leaving active threads that refer to it. This scenario seems inescapable: if one moves some threads, they typically are going to have been interacting, in some fashion, with other threads at the source.

Accordingly, the mutex and condition variable operations may fail dynamically, giving `Nonexistent_mutex` and `Nonexistent_cvar` exceptions. One would expect high-level communication libraries, e.g. of distributed communication channels and migration, to ensure such errors never occur.

9.7 References, names, marshalling, and thunkify

Semantically, it is tempting to treat store locations as another variety of name, similar to thread and mutex names. In Acute we do not make this identification as the cost seems under-motivated. A naive implementation, indirecting all access via a name lookup, would obviously be absurd. Even an optimisation, using local pointers but keeping a name with every store value, would be rather expensive — in a typical program there are many more store locations than mutexes or threads (it would be necessary to keep a name for each explicitly as garbage collection can relocate pointers but the name order must be preserved).

Further, the dynamic semantics is rather different: marshalling copies the reachable fragment of the store, whereas names are simply marshalled as the values that they are. Thunkifying threads and mutexes is destructive, removing them from the running system. Copying the reachable fragment of the store ensures that dereferencing and assignment can never fail dynamically (which we think would be unacceptable) whereas the implicit marshalling of entire threads seems unlikely to be desirable. Further practical experience is required to assess these choices.

9.8 Module initialisation, concurrency, and thunkify

Without module initialisation all threads are simply executing an expression. With initialisation, however, at least one thread might be executing a sequence of definitions (followed by an expression), evaluating expressions on the right-hand-side of structures in programs as below.

```
module fresh M : sig    val x: int ref      val y:unit            end
               = struct let x=ref 3         let y=IO.print_int !x  end
M.x := 7
```

These expressions may spawn other threads, which may interact (via the store, mutexes etc.) with the first.

In fact, as discussed in §4.8, no uninitialised definitions can be dynamically added to the system, so it is an invariant that at most one thread is executing in definitions (though the semantics actually allows definitions in all threads, for uniformity).

The initial thread has no other special status.

Now, what should **thunkify** do if invoked on such a thread? Acute has a second-class module system, so there is (unfortunately) no way to represent a suspended module-level computation in the expression language. The **thunkify** must therefore either abort or block until module initialisation is complete. For the time being we take the former choice, raising a `Thunkify_thread_in_definition` exception.

9.9 Thunkify and blocking calls

With any form of thread migration or (more generally) with our thunkification one has to deal with threads that are blocked in system calls. There are two possibilities:

1. have the `thunkify` block until the target thread returns, thunkifying its state just after the return; or

2. have the `thunkify` return immediately, thunkifying the state of the target thread with a raise of a `Thunkify_EINTR` exception replacing the blocked call, and discarding the eventual return value of the call. This is analogous to the Unix `EINTR` error, returned when a system call is interrupted by a signal, which applications must be prepared to deal with.

Both are desirable, in different circumstances, and so we allow a per-thread choice. Note that this applies only to blocking (or "slow") system calls such as `read()`, not to the many non-blocking system calls which return quickly. The language semantics must distinguish the two classes.

Taking this further, it is unpleasant for the system interface to be special in this way. For example, suppose one has a user library module that provides a wrapper around the system interface; one might want to identify some of the user module entry points as blocking and have similar `thunkify` behaviour. This would be conceptually straightforward if the functions provided by the module are all first-order and cannot be partially applied, in which case there is a straightforward notion of a thread executing 'in' the module. `thunkify` could behave as (2) as far as the calling thread is concerned and raise an asynchronous exception in the user library code. We believe this kind of mechanism is desirable, but have not explored it in detail.

9.10 Concurrency: the constructs

Putting these choices together, we have types

```
thread
mutex
cvar
thunkifymode
thunkkey
```

The first three types are empty; they are introduced to form types `thread name`, `mutex name`, and `cvar name`. A `thunkifymode` is either `Interrupting` or `Blocking`; type `thunkkey` has three constructors, `Thread`, `Mutex` and `CVar`, each taking a name of the associated type; the first takes also a `thunkifymode`.

We have operations for threads, mutexes, condition variables and thunkification as below.

```
create_thread : thread name -> (T->unit) -> T -> unit
self : unit -> thread name
kill : thread name -> unit

create_mutex : mutex name -> unit
lock : mutex name-> unit
try_lock : mutex name -> bool
unlock : mutex name -> unit

create_cvar : cvar name -> unit
wait : cvar name -> mutex name -> unit
signal : cvar name -> unit
broadcast : cvar name -> unit

thunkify :  thunkkey list -> thunkkey list -> unit

exit : int -> T
```

In addition, we have a control operator

```
e1 ||| e2
```

that spawns its first argument, as syntactic sugar for

```
create_thread fresh (function () -> e1); e2
```

Here `thunkify` takes a list of `thunkkeys` specifying which threads, mutexes and condition variables to thunkify; it returns a function which takes a list of the same shape specifying the names to give these entities and then atomically re-creates them.

9.11 Example

Below is a simple use of **thunkify**, capturing the state of a single running thread and an (unused) mutex.

```
let rec delay x = if x=0 then () else delay (x-1) in
let rec f x = IO.print_int x; IO.print_newline (); f (x+1) in
let t1 = fresh in
let m1 = fresh in
let _ = create_thread t1 f 0 in
let _ = create_mutex m1 in
let _ = delay 15 in
let v = thunkify ((Thread (t1,Blocking))::(Mutex m1)::[]) in
IO.send( marshal "StdLib" v : thunkkey list -> unit )
—
let rec delay x = if x=0 then () else delay (x-1) in
let exit_soon = create_thread fresh (fun () -> delay 15 ; exit 0) () in
let v = (unmarshal(IO.receive()) as thunkkey list -> unit) in
v ((Thread (fresh,Blocking))::(Mutex fresh)::[])
```

When run the first program prints `0 1 2 3 4` and the second `5 6 7 8`. The marshalled value, containing the thunk, is shown in §15.10.

10 Polymorphism

Ultimately, both subtype and parametric polymorphism should be included. Many version changes involve subtyping, e.g. the addition of fields to a manifest record type argument of a remote function; it should be possible to make these transparent to the callers. Parametric polymorphism is of course needed in some form for ML-style programming. In the distributed setting it seems to be particularly useful to have first-class universals, allowing polymorphic functions to be communicated, and first-class existentials. The latter support an idiom, common in Pict and Nomadic Pict, in which one packages a channel name and a value that can be sent on that channel, as a value of type $\exists\ t.t\ \ \mathsf{name} * t$. This lets one express communication infrastructure libraries that can uniformly forward messages of arbitrary types.

There are two substantial difficulties here. Firstly, type inference is challenging for such combinations of subtyping and parametric polymorphism. A partial type inference algorithm will be required, and it must be pragmatically satisfactory — inferring enough annotations, and unsurprising to the programmer. This is the subject of recent research on *local type inference* [PT98, HP99] and *coloured local type inference* [OZZ01]. Without subtyping, the MLF of Le Botlan and Rémy [LBR03] allows full System F but can infer types for all ML-typable programs.

Secondly, the interaction between subtyping and hash types requires further work — one can imagine, for instance, that a subhash order derived from subtype and subversion relationships needs to be dynamically propagated.

In Acute we sidestep both of these issues for the time being, making an interim choice that suffices for writing non-trivial examples, e.g. of polymorphic communication infrastructure modules. Acute has no subtyping. The basic scheme is monomorphic, but with type inference. The definition of the internal language has explicit type annotations, on pattern variables and on built-in constructors such as `[]` and `None`. In the external language these annotations can all be inferred by a unification-based algorithm. To this we add first class System F universals and existentials, with types `forall t.T` and `exists t.T` and explicit type abstractions, applications, packs and unpacks, with expression forms

```
Function t -> e
e %[T]
{T,e} as T'
let {t,x} = e1 in e2
```

There is no automatic generalisation, and the subsignature relation remains, as in the monomorphic case, without generalisation. We also have no user-definable type constructors. The expression forms could easily be more tightly integrated with the other pattern matching and function forms.

Traditional ML implementations can erase all types before execution. In contrast, an Acute runtime needs type representations at marshal and unmarshal points, to execute the expressions `marshal e : T` and `unmarshal e as T`. (These types can often be inferred). Type representations are also needed at `fresh`, `cfresh` and `hash(...)` points. Our prototype implementation keeps all type information, throughout execution, so that we can do runtime typechecking between reduction steps. A production implementation would probably do a flow analysis to determine where types are required, adding type representation parameters to functions as needed. The only operations that a production implementation needs to do on these type representations are (1) compare them for syntactic equality, (2) construct them when a polymorphic function is applied to its type parameter, and (3) take hashes of them. It is therefore not necessary to keep all the type structure. Indeed, one could (with a small probabilistic reduction in safety) work with hashes of types at runtime. Alternatively, if one keeps the structure it would be possible to add some form of runtime type analysis [Wei02] at little extra cost, at least for non-abstract types.

10.1 A refinement: marshal keys and name equality

In the implementation of distributed communication libraries one may often be communicating values of types such as `exists t. t name * T` (with the `t` potentially occurring in *T*) where the `t name` is used as a demultiplexing/dispatch key at the receiver.

To statically type the receiver code an enhanced conditional or matching form is needed: having compared that `t name` with the locally-stored name associated with (say) a channel data structure, typing the `true` branch must be in an environment where the two are known to be of the same type.

The enhanced form could be either an explicit type equality test or a name equality test. At present we do not see a strong argument either way. A type equality test is perhaps cleaner, but would lead to runtime type information being required at more program points. A general name equality test, `if e1=e2 then e3 else e4`, where *e1* and *e2* are of arbitrary `T1 name` and `T2 name` types, is the most obvious alternative, but this requires a slightly intricate treatment of multiple type equalities in the semantics. For the time being we combine name equality testing with existential unpacks, with

```
namecase e1 with  {t,(x1,x2)} when x1=e
   -> e2
   otherwise -> e3
```

where `e1:exists t. t name * T`, the `e:T' name` is evaluated first and used to build an equality pattern, and in the *e2* branch it is known that `t=T'`. Obviously such existentials are not uniformly parametric in Acute.

If one is communicating values of type `exists t. t name * t`, and is demultiplexing on the `t name`, the explicit type in the marshalled value (and the unmarshal-time type equality check) could be omitted; name equality gives an equally strong guarantee. If communicating many small values the performance gain of this could be worth direct language support for such ‘marshal keys’.

11 Pulling it all together: examples

To date, we have written many small examples in Acute (for automated testing), and three larger programs. The first two are `blockhead` and `minesweeper` games that mostly exercise local computation; the latter uses marshalling to save and restore the game state. The third is a communication infrastructure library which shows how most of the Acute features are needed and used. It has the following modules:

`Tcp_connection_management` maintains TCP connections to TCP addresses (IP address/port pairs), creating them on demand. `Tcp_string_messaging` uses that to provide asynchronous messaging of strings to TCP addresses. These are both `hash` modules, with abstract types of handles; they spawn daemons to deal with incoming communications.

Separately, a module `Local_channel` provides local (within a runtime) asynchronous messaging, again with an abstract type of channel management handles and with polymorphic `send:forall t. t name * t -> unit` and `recv:forall t. t name*(t->unit) -> unit` (to register a handler). Channel states are stored as existential packages of lists of pending messages or receptors; the `namecase` operation is used to manipulate them. Mutexes are needed for protection.

`Distributed_channel` pulls these together, with `send:forall t.string->(Tcp.addr*t name)->t->unit` (and a similar `recv`) for distributed asynchronous messaging to TCP addresses. The string names the mark to marshal with respect to. For a local address this simply uses `Local_channel`. For a remote address the `send` marshals its `t` argument and uses `Tcp_string_messaging`; the `recv` unmarshals and generates a local asynchronous output. This deals with the non-mobile case — active receivers cannot be moved from one runtime to another. However, code that uses this module, e.g. functions that invoke `send` and `recv`, can be marshalled and shipped between runtimes; the module initialisation state includes the TCP messaging handles and so rebinding to different instances of `send` and `recv` works correctly. A simple `RFI` module implements remote function invocation above distributed channels.

Clients of this libraries can use any of the various ways of creating shared typed names discussed in §6 and §8.4. Moreover, the use of first-class marks means that clients have the same flexible control over the marshalling that goes on as direct users of `marshal`.

Going further, a `Nomadic_pi` module supports mobility of running computations, with named *groups* of threads, each with a local channel manager, that can migrate between machines. Migration uses `thunkify` to capture the group's channel and thread state. Threads within a group can interact via local channels; groups can interact with a location-dependent `send_remote` that sends a message to a channel of a group assumed to be at a particular TCP address.

The location-independent messaging algorithms of JoCaml or high-level Nomadic Pict should be easy to express above this (the former requiring the polytypic `support` and `swap` operations to manipulate the free channel names of a communicated value).

12 Related work

Acute builds on our earlier work: compile-time fresh generation of abstract type names and channel names [Sew01]; hash-generation of effect-free abstract type names [LPSW03]; and dynamic rebinding [BHS+03]. There is extensive related work on module systems, dynamic binding, dynamic type tests, and distributed process calculi. For most of this we refer the reader to the discussion in those papers, confining our attention here to some of the most relevant distributed programming language developments.

Early work on adding local concurrency to ML resulted in Concurrent ML [Rep99] and the initial Facile, both based on the SML/NJ implementation. Facile was later extended with rich support for distributed execution, including a notion of *location* and computation mobility [TLK96]. dML [OK93] was another distributed extension of ML, implementable by translation into remote procedure calls without requiring communication at higher types.Erlang [AVWW96] supports concurrency, messaging and distribution, but without static typing.

The Pict experiment [PT00] investigated how one could base a usable programming language purely on local concurrency, with a π-calculus core instead of primitive functions or objects. The Distributed Join Calculus [FGL+96] and subsequent JoCaml implementation [JoC] modified the π primitives with a view to distribution, and added location hierarchies and location migration. The runtime involved a complex forwarding-pointer distributed infrastructure to ensure that, in the absence of failure, communication was location-independent. (Polyphonic C$^\sharp$ [BCF02] adds the Join Calculus local concurrency primitives to a class-based language.) Other work in the 1990s was also aimed at providing distribution transparency, notably Obliq [Car95], with network-transparent remote object references above Modula3's network objects.

Distribution transparency, while perhaps desirable in tightly-coupled reliable networks, cannot be provided in systems that are unreliable or span administrative boundaries. Work on Nomadic Pict [SWP99, US01] adopted a lower level of abstraction, showing how a wide variety of distributed infrastructure algorithms, including one similar to that of the JoCaml implementation, could be expressed in a high-level language; one was proved correct. The low level of abstraction means the core language can have a clean and easily-understood failure semantics; the work is a step towards the argument of §2.

A distinct line of work has focussed on typing the entire distributed system to prevent resource access failures, for Dπ [HRY04] and with modal types [MCHP04]. Even where this is possible, however, programmers must still deal with low-level network failure.

Work on Alice [Ali03, Ros03] is perhaps closest to ours, with ML modules, support for marshalling ('pickling') arbitrary values, and run-time fresh generation of abstract type names.

Many of the language designs cited above address distributed *execution*, with type-safe interaction within a single program that forks across the network, but there has been little work on distributed *development*, on typed interaction *between* programs[4], or on version change.

Both Java and .NET have some versioning support, though neither is integrated with the type system. Java serialisation, used in RMI, includes *serialVersionUID*s for classes of any serialised objects. These default to (roughly) hashes of the method names and types, not including the implementation. Class authors can override them with hashes of previous versions. Linking for Java, and in particular binary compatibility, has been studied by Drossopoulou et al. [DEW99]. The .NET framework supports versioning of *assemblies* [dot03]. Sharable assemblies must have *strong names*, which include a public key, file hashes, and a *major.minor.build.revision* version. Compile-time assembly references can be modified before use by XML policy files of the application, code publisher, and machine administrator; the semantics is complex.

Explicit versioning is common in package management, however. For example, both RedHat and Debian packages can contain version constraints on their dependencies, with numeric inequalities and capability-set membership. ELF shared objects express certain version constraints using pathname and symlink conventions. Vesta [ves] provides a rich configuration language.

As discussed in §3 Acute addresses the case in which complex values must be communicated and the interacting runtimes are not malicious. Much other work applies to the untrusted case, with various forms of proof-carrying code and wire-format XML typing which we cannot discuss here.

13 Conclusions and future work

We have addressed key issues in the design of high-level programming languages for distributed computation, discussing the language design space and presenting the Acute language. Acute is a synthesis of an OCaml core with several novel features: dynamic rebinding, global fresh and hash-based type and term naming, versions, type- and abstraction-safe marshalling, etc. It is an experimental language, not a proposal for a full production language, but (as demonstrated by our examples) it shows much of what is needed for higher-order typed distributed computation.

The new constructs should also admit an efficient implementation. The two main points are the tracking of run-time type information, and the implementation of redex-time reduction and rebinding. For the first, note that an implementation does not need to have types for all runtime values, but only (hashes of) the types that reach marshal and unmarshal points. The second would be a smooth extension of OCaml's existing CBV implementation: OCaml currently maintains each field reference `M.x` as a pointer until it is in redex position, when it is then dereferenced. Since field references inside a thunk remain as pointers, they could easily be rebound with only modest changes to the run-time. Of course compile-time inlining optimisations between parts of code separated by a mark would no longer be possible.

A great deal of future work remains. In the short term, more practical experience in programming in Acute is needed, and there are unresolved semantic issues in the interaction between explicit polymorphism, coloured brackets, and marshalling. Straightforward extensions would ease programming: user definable type operators and recursive datatypes, first-order functors, and richer version languages. A more efficient implementation runtime may be needed for larger examples. Improved tool support for the semantics would be of great value, for meta-typechecking, for conformance testing, and for proofs of soundness.

More fundamentally:

- We must study more refined low-level linking, for negotiation and for access control (escaping the linear mark/module structure). This may demand recursive modules.
- The Acute operational semantics is rather complex, as is the definition of compilation. In part this seems inevitable — the semantics deals with dynamic linking, marshalling, concurrency, thunkify, and coloured brackets, all of which are dynamically intricate (and few of which are covered by existing large-scale definitions). Additionally, our focus has been on a direct semantics of the user language, rather than a combination of a simpler core and a translation, and Acute has evolved through several phases. It should be possible to make the compilation semantics less algorithmic by appealing explicitly to type canonicalisation. The operational semantics for

[4]Several, including JoCaml and Nomadic Pict, have ad-hoc 'traders' for establishing initial connections between programs.

a language with lower-level linking might well be simpler than that presented here, factoring out the algorithmic issues of $resolvespec$s, for example.

- Subtyping is needed for many version-change scenarios, perhaps with corresponding subhash relations. As mentioned in §10, the proper integration of this with polymorphism is challenging, as is the question of what subtype information needs to be propagated at run-time.
- The Acute constructs for local concurrency are very low level, and it is unclear what should be added. Join patterns, CML-style events, π-style channels, and explicit automata; all are useful idioms.
- Some distributed abstractions, such as libraries of distributed references with distributed garbage collection, may challenge the type system.
- The constructs we have presented should be integrated with support for untrusted interaction.

A combination of what has been presented in Acute with solutions to these problems would support a wide range of distributed programming well.

Acknowledgements We acknowledge a Royal Society University Research Fellowship (Sewell), a St Catharine's College Heller Research Fellowship (Wansbrough), an INRIA grant "post-doc à l'étranger" (Zappa Nardelli), EP-SRC grant GRN24872, EC FET-GC project IST-2001-33234 PEPITO, and APPSEM 2. We thank Vilhelm Sjöberg and Christian Steinrücken for their implementation work on Acute's predecessors and Gilles Peskine for discussions on references and coloured brackets. Andrew Appel, Matthew Fairbairn, Jean-Jacques Lévy, Luc Maranget, Gilles Peskine, and Alisdair Wren provided comments.

Part II
Semantics

14 Semantics overview

The Acute definition, given in §16, describes syntax, typing, typed desugaring, errors from compilation and execution, compilation, and operational semantics. It also states type preservation and progress conjectures and gives semantic descriptions of two optimisations: for closures and for removing 'vacuous brackets'. This section outlines the main points of the semantics. It should be read in conjunction with the definition.

The definition involves several related languages:

1. The *concrete source* language is the language that programmers type, e.g. `function (x,y) -> x + y + M.z`. This is concrete — a set of character sequences.

2. The *sugared source internal* language is generated by parsing, scope resolution and type inference; for example $\mathbf{function}\ (x : \mathsf{int}, y : \mathsf{int}) \to (+)\ x\ ((+)\ y\ \mathrm{M}_M.\mathrm{z})$. This is an abstract grammar, up to alpha equivalence. The x, y and M are internal identifiers, subject to alpha equivalence; the z and M are external identifiers, which are not. (In fact operators are eta-expanded to ensure they are fully applied.)

3. The *source internal* language is generated by desugaring, for example $\mathbf{function}\ (u : \mathsf{int} * \mathsf{int}) \to \mathbf{match}\ u\ \mathbf{with}\ ((x : \mathsf{int}), (y : \mathsf{int})) \to (+)\ x\ ((+)\ y\ \mathrm{M}_M.\mathrm{z})$.

4. The *compiled* language is generated by compilation, which here computes global type names for hashed abstract types, carries out $\mathit{withspec}$ and $\mathit{likespec}$ checks, etc. The operational semantics is defined over elements of the compiled language.

   Note that the compiled language contains both compiled form and source internal form components. Specifically, a compiled program consists of compiled form definitions and/or source internal form $\mathbf{module\ fresh}$ definitions, and an optional compiled form expression.

The main definition is of the union of the grammars for the *source internal* and *compiled* languages.

14.1 Naming

The language makes heavy use of names: at the expression level (names for communication channels, RPC handles etc.), at the type level (for abstract type names), and at the module level (names associated with modules and with imports are used both to construct abstract type names and in version constraints and version expressions).

Names, of each variety, can be generated either from module or import hashes (deterministically), or by taking (psuedo-)random numbers, at either compile-time or run-time. In an implementation these names will all be represented uniformly, e.g. as 160-bit numbers.

Both hash-generation and random-generation allow names to be safely associated with type information across the global distributed system. If one wishes to establish a shared name (expression or type) across programs, it can either be hash-generated from shared source of a module or be compile-time fresh generated and the resulting `.aco` file included by both programs. Other names, on the other hand, must be run-time generated (for names of dynamically-created channels, and of generative types that depend on computational effects).

Using hashes and random name generation means that the correct operation of programs is only probabilistically guaranteed. The name representation must be chosen to be of enough bits to make the probability of accidental collision acceptably low (e.g. lower than the rate of hardware or cosmic-ray errors).

While a production implementation would represent names purely as 160-bit numbers, in order to define typability for states reachable by computation more structure is required. The semantics is therefore expressed in terms of *structured hashes* $\mathbf{hash}(...)$ and *abstract names* n; the metavariable h ranges over both. Structured hashes, e.g.

$\mathbf{hash}(\mathbf{hmodule}_{eqs}\ \mathrm{M} : \mathit{Sig}_0\ \mathbf{version}\ \ \mathit{vne} = \mathit{Str})$, are formal representations of hash values that preserve all their internal structure. For abstract names n, which have no internal structure, the semantics maintains a global type environment E_{n} mapping all those that have been created so far to their respective module, kind or type data. Our prototype implementation can work either with structured hashes (and maintain an E_{n}) or with literal numeric hashes (and discard the E_{n}). The former allows optional run-time typechecking, of the entire configuration on a machine after every reduction step, which is a valuable tool for debugging both the language definition and the implementation. It also allows the less costly option of unmarshal-time and resolve-time typechecking.

We do not work up to alpha-equivalence of the global abstract names in E_{n}, instead choosing fresh names non-deterministically from those that have not been used so far. Our E_{n} really is a global environment, affected (in the semantics but not the implementation) by *all* running machines. This contrasts with the usual π-calculus approach of extruding binders as necessary. We make this choice to avoid having to consider alpha-equivalence of marshalled values, which are simply byte-strings, but which can be unmarshalled to values containing names.

A further subtlety arises in the version expression and constraint languages. Here it is desirable to let the programmer paste in literal hashes, and there is no way for the language to ensure that these literals all arise as the hashes of well-formed modules.

14.2 Typing

(§16.3, page 91) Much of the type system is standard, using singleton kinds to express manifest and abstract types in modules [HL94, Ler94], and with a subsignature relation based on the subkind relation $\mathrm{EQ}(T) <: \mathrm{TYPE}$ allowing manifest type information to be forgotten.

In contrast to most previous work on abstract types and module systems the semantics constructs global names for abstract types, at compile-time or run-time, instead of erasing all types or substituting abstractions away. A source internal language type $\mathrm{M}_M.\mathrm{t}$ (the t type field of module M_M) is compiled or reduced to a global type name $h.\mathrm{t}$, where h is a hash or fresh abstract name. This is a dynamic analogue of the type-theoretic *selfification* rules in singleton-kind systems.

To establish greater confidence in the internal coherence of the semantics we preserve abstraction boundaries throughout execution, adapting and extending *coloured brackets* [GMZ00, LPSW03] to delimit subexpressions in which sets eqs of type equalities $h.\mathrm{t} \approx T$ between abstract types $h.\mathrm{t}$ and their representations can be used. Additionally, most type judgements, and the operational relations, are with respect to such sets of equalities eqs, reflecting which abstractions one is within. To type a coloured bracket $[e]^{T}_{\mathit{eqs}'}$, with type T and in an ambient colour eqs, one must have e of type T in colour eqs'.

The most interesting typing rules are for modules, imports, and hashes and abstract names. These latter two behave very like module identifiers, with rules for selfification and for constructing types $h.\mathrm{t}$ and terms $h.\mathrm{x}$ (the latter occuring only within other hashes, not in executable code). The type rules for the compiled language check that such h are used correctly, referring to their internal structure or the global type environment E_{n} respectively for hashes or abstract names. An implementation does not need this information, however — in particular, it is not required for the unmarshal-time type equality check.

Typing source internal language modules and imports is much as one would expect. Typing their compiled forms is more interesting, capturing a number of properties that are established by compilation.

Marshalling and unmarshalling are straightforward as far as their static typing goes, converting between arbitrary T and string.

In any given environment E and colour eqs, each semantic type may be represented by any member of an equivalence class of syntactic types defined by the relation $E \vdash_{\mathit{eqs}} T \approx T'$. Compilation ensures that the syntactic type chosen is always the *canonical type* from the relevant equivalence class. The canonical type is the one that is most concrete: it is the normal form under the rewrites $\{X.t \rightsquigarrow T \mid (X.t \approx T) \in \mathit{eqs}\}$, $M.t \rightsquigarrow T \mid M : \mathit{Sig} \in E \wedge t : \mathrm{EQ}(T) \in \mathit{Sig}$, and $t \rightsquigarrow T \mid t : \mathrm{EQ}(T) \in E$. This is important because in certain circumstances the syntactic representative chosen for a semantic type is significant.

14.3 Compilation

(§16.7, page 115) Compilation involves several activities (which are recursively intertwined in the definition):

- preprocessing (i.e., replacing $\mathbf{includesource}\ \ \mathit{sourcefilename}$ and $\mathbf{includecompiled}\ \ \mathit{compiledfilename}$ by the file bodies)
- desugaring
- type-checking
- traversing module definitions calculating (and using) the names to use for global type name of abstract type
- calculating fresh names for $\mathbf{cfresh}$ modules and expressions
- checking asserted $\mathit{withspec}$ equations are correct and that any module linking is legitimate. These are not checked by the type system as they depend on knowledge of the representation types of earlier abstract types, which is not recorded in type environments.

Formally, compilation is a relation from a name environment E_{n}, a $\mathit{sourcefilename}$, and a filesystem Φ to either a tuple of a source type environment E_0, a compiled type environment E_1, and a $\mathit{compiledunit}$, or an error.

Note that $\mathit{compiledunit}$ includes a name environment E_{n}: this environment contains $\mathbf{cfresh}$ names created during compilation. This name environment has no implementation significance: its sole purpose is to allow included compiled units to be appropriately typechecked and the configuration produced by compilation to be typechecked. These two checks are both necessary for runtime typechecking, but not otherwise.

Note that compilation is not a function because the choice of name environment in the $\mathit{compiledunit}$ is nondeterministic. This nondeterminism is common in many of the helper "functions" throughout, thus we take them all to be relations. For convenience, though, we write them as functions of their inputs, and use $\rightsquigarrow$ rather than $=$ to relate the "input arguments" to the "results".

Compilation has the form

$$\mathrm{compile}_{\Phi}(\mathit{sourcefilename})E_{\mathrm{n}} \rightsquigarrow (E_0', E_1', \mathit{compiledunit}')$$

defined to be

$$\mathrm{compile}_{\Phi\ \varnothing}^{\mathrm{empty}\ \ E_{\mathrm{n}}\ E_{\mathrm{const}}\ E_{\mathrm{const}}}(\mathbf{includesource}\ \ \mathit{sourcefilename}\ \mathbf{;;}\ \mathrm{empty}) \rightsquigarrow (E_0', E_1', \mathit{compiledunit}')$$

where the latter relation

$$\mathrm{compile}_{\Phi\ \mathit{sourcefilenames}}^{\mathit{definitions}\ E_{\mathrm{n}}\ E_0\ E_1}(\mathit{compilationunit}) \rightsquigarrow (E_0', E_1', \mathit{compiledunit}')$$

is defined inductively on the $\mathit{compilationunit}$. Here $\mathit{sourcefilenames}$ is the filenames we've been through (used to detect cyclic includes), $\mathit{definitions}$ is the accumulated compiled definitions, E_{n} is the accumulated name environment (all names created during compilation will be disjoint from $\mathrm{dom}(E_{\mathrm{n}})$), E_0 is the accumulated source type environment (including E_{const} at the start), E_1 is the accumulated compiled type environment (including E_{const} at the start), and $\mathit{compilationunit}$ is what we have left to do.

The behaviour of compilation on a module (or import, similarly) depends on whether it is annotated $\mathbf{hash}$, $\mathbf{cfresh}$ or $\mathbf{fresh}$ (which will generally depend on whether it is valuable, cvaluable or non-valuable). We first describe the $\mathbf{hash}$ case, with steps corresponding to those in §16.7. First and secondly, all types are normalised as far as possible, replacing any types $\mathrm{M}'_{M'}.\mathrm{t}$ defined in earlier modules by either the corresponding $h'.\mathrm{t}$ (if they are abstract) or the corresponding T (if they are manifest). References to earlier type fields in this module are also flattened where possible: in the structure all type definitions are substituted away; in the signature only manifest type fields can be substituted away. Thirdly, any $\mathit{withspec}$ is checked, and the resulting set of type equations, normalised, is recorded. Fourthly, the hash of this module can be constructed, first replacing any other-module expression dependencies $\mathrm{M}'_{M'}.\mathrm{x}$ by the corresponding $h'.\mathrm{x}$. Fifthly, that hash is used to selfify the remaining abstract type fields of the signature, replacing $\mathbf{type}\ \mathrm{t}_t : \textsc{Type}$ by $\mathbf{type}\ \mathrm{t}_t : \mathrm{EQ}(h.\mathrm{t})$. Sixthly, the version number expression of the module is evaluated, replacing **myhash** by the hash h. The result has the form

$$\mathbf{cmodule}_{h;\ eqs;\ Sig_0}\ \mathrm{M}_M : Sig_1\ \ \mathbf{version}\ \ vn = Str$$

where h is this module's hash, eqs are any extra equations added by the $withspec$, Sig_0 is the normalised but non-selfified signature, Sig_1 is the normalised and selfified signature, vn is the version number, and Str is the normalised structure.

The body of a hash thus does not exactly match either the source module or the compiled module. It cannot be the source module as it must be type-normalised, so that hash equality respects provable type equality. It cannot be the compiled module as that would require recursive hashes — the selfification during compilation uses the hash. (One could introduce a formal recursive hash, but it seems more intuitive not too.)

Compilation of a hash-import is broadly similar, with a $likespec$ rather than a $withspec$, resulting in a form

$$\mathbf{cimport}_{h;Sig_0}\ \mathrm{M}_M : Sig_1\ \ \mathbf{version}\ \ vc\ \ \mathbf{like}\ \ Str\ \ \mathbf{by}\ \ resolvespec = Mo$$

In the **cfresh** cases compilation constructs an h for the module nondeterministically instead of by hashing, taking any n not in the domain of the ambient E_n. Expression-level **cfresh** names are constructed similarly, and compilation is otherwise similar.

In the **fresh** case the h for the module is constructed nondeterministically at the start of its execution, whereupon it can be used to selfify and normalise types similarly.

14.4 Operational judgements

(§16.8.1, 16.8.2, and 16.8.3, page 125) The runtime configurations of a single machine have the form

$$\langle E_s,\ s,\ definitions,\ P\rangle$$

where E_s is the store typing (not required in a production implementation), s is the store, $definitions$ is the sequence of module definitions (all of which are definition values), and P is a multiset of named running threads, mutexes, and condition variables.

$$\begin{array}{lll} P & ::= & 0 \\ & & P_1 \mid P_2 \\ & & \mathbf{n} : definitions\ e \\ & & \mathbf{n} : \mathrm{MX}(\underline{b}) \\ & & \mathbf{n} : \mathrm{CV} \end{array}$$

The main operational judgements are as below. The first two of these are the main judgements; the other four are auxiliaries introduced so that most reduction axioms need only mention the relevant parts of a configuration. We sometimes call a tuple $\langle E_s,\ s,\ definitions,\ e\rangle$ a *pseudo-configuration*.

- $E_\mathrm{n}\ ;\ \langle E_s,\ s,\ definitions,\ P\rangle \xrightarrow{\mathbf{n}:\ell} {E_\mathrm{n}}'\ ;\ \langle {E'}_s,\ s',\ definitions',\ P'\rangle$ Process reduction.
- $E_\mathrm{n}\ ;\ \langle E_s,\ s,\ definitions,\ P\rangle \to \mathbf{TERM}$ Progam termination.
- $E_\mathrm{n}\ ;\ \langle E_s,\ s,\ definitions,\ e\rangle \xrightarrow{\ell}_{eqs} {E_\mathrm{n}}'\ ;\ \langle {E'}_s,\ s',\ definitions',\ e'\rangle$ Expression reduction.
- $e \xrightarrow{\ell}_{eqs} e'$
- $E_\mathrm{n}\ ;\ e \xrightarrow{\ell}_{eqs} {E_\mathrm{n}}'\ ;\ e'$

- $P \xrightarrow{\ell} P'$

where

ℓ	$::=$	empty	internal reduction step
		$x^n\ v_1^\varnothing\ ..\ v_n^\varnothing$ for $x^n \in \mathrm{dom}(E_{\mathrm{const}}) \wedge \mathrm{os}(x^n)$	invocation of OS call
		$\mathrm{Ok}(v^\varnothing)$	return from OS call
		$\mathrm{Ex}(v^\varnothing)$	return from OS call
		$\mathrm{GetURI}(\mathit{URI})$	request for code at URI
		$\mathrm{DeliverURI}(\mathit{definitions})$	resulting code
		$\mathrm{CannotFindURI}$	nothing found at URI

We write $\xrightarrow{\mathrm{empty}}$ simply as $\longrightarrow$.

The class of values is parameterised by colours, with v^{eqs} ranging over the values at colour eqs.
The dynamic semantics is expressed with evaluation contexts as follows.

- C_{eqs} is a single-level evaluation context at colour eqs. These are largely standard, for example $\_\ e$ and $v^{eqs}\ \_$.
- $C_{eqs_2}^{eqs_1}$ is a colour-changing single-level evaluation context, at colour eqs_1 but with a hole at colour eqs_2. The main case of these is the coloured brackets, $[\_]_{eqs_2}^{T}$, but there are several cases where we need to construct a value at colour $\varnothing$, e.g. to store or to pass to a primitive operator, so this grammar includes e.g. $l\ \mathbf{:=}'_T\ \_$ for $eqs_2 = \varnothing$.
- CC_{eqs} and $CC_{eqs_2}^{eqs_1}$ are multi-level evaluation contexts — simple compositions of the above.

$$CC_{eqs} \;::=\; \_ \;\mid\; CC_{eqs}.C_{eqs} \qquad\qquad CC_{eqs_2}^{eqs_1} \;::=\; \_ \;\mid\; CC_{eqs}^{eqs_1}.C_{eqs_2}^{eqs}$$

- SC_{eqs} is a structure evaluation context, allowing computation in the first non-value expression field of a structure.
- TC_{eqs} is a thread evaluation context. For a thread with body just a single expression e, computation can take place there; for a thread with a body $\mathit{definitions}\ e$ where the head of $\mathit{definitions}$ is a $\mathbf{cmodule}$, computation can take place in the first non-value expression field of the structure.
- TCC_{eqs} is a composition $TC_{eqs_2}.CC_{eqs}^{eqs_2}$, allowing computation within the expression in a TC_{eqs} hole.

14.5 Colours and bracket dynamics

The semantics preserves abstraction boundaries boundaries, generalising the *coloured brackets* of Grossman et al [GMZ00]. (At present this covers the entire Acute language except the System F polymorphism constructs.)

Coloured brackets make explicit the type equalities which are in scope for any subexpression. There is a bracket expression form

$$[e]_{eqs}^{T}$$

for type equations

$$eqs \;::=\; \varnothing \mid \mathrm{M}_M.\mathrm{t} \approx T \mid h.\mathrm{t} \approx T \mid eqs, eqs$$

giving the representation types of abstract types (source-language projections from a module identifier $\mathrm{M}_M.\mathrm{t}$ and compiled-language projections from a module name $h.\mathrm{t}$). From the outside $[e]_{eqs}^{T}$ is of type T; the type equations eqs can be used in typechecking e. We use "colour" and "type equations" interchangeably.

Brackets are not needed in a production implementation (our implementation can work with them or without them), and they are not strictly speaking necessary for the semantics — with the exception of the work of Grossman et al, and of our previous [LPSW03] and Rossberg's [Ros03], most operational semantics for existential types and for module systems forgets abstraction boundaries as it comes to them, e.g. with this rule for opening an existential package

$$\mathbf{let}\ \{t, x\} = (\{T, e\}\ \mathbf{as}\ \ T')\ \mathbf{in}\ \ e_2 \quad \rightarrow \quad \{T/t, e/x\}e_2$$

or by substituting out module definitions. Maintaining abstraction boundaries requires some complexity in the semantics, but we think it well worth while. Type preservation for an abstraction-preserving semantics is intuitively a much stronger property that for a standard semantics, and so a better check of internal consistency; and making type equations explicit in both the type system and runtime provides conceptual clarity.

Brackets are not a user source language construct. They are introduced primarily when instantiating a module field reference $\mathrm{M}_M.\mathrm{x}$ from a module M_M that introduced some abstract types (see *Module field instantiation – module case, via import sequence*, §16.8.6, page 136). For a simple example, consider the `EvenCounter` of §5, with fields `start : EvenCounter.t` and `up : EvenCounter.t->EvenCounter.t` Expressions `EvenCounter.start` and `EvenCounter.up` will be instantiated, when they appear in redex-position, to $[\mathtt{0}]^{h.\mathtt{t}}_{h.\mathtt{t=int}}$ and $[\mathtt{fun\ (x:int)\text{->}2+x}]^{h.\mathtt{t}\text{->}h.\mathtt{t}}_{h.\mathtt{t=int}}$ respectively. Here h is the hash-generated module name of `EvenCounter` as in §5.

Bracket semantics could be expressed either with a structural congruence or with reductions. We choose the latter, to support our prototype implementation. The basic points are the definition of values (§16.8.2, page 125) and the bracket-pushing reductions of §16.8.4, page 130. The latter push brackets through values in cases where the outermost value structure and the outermost type structure of the bracket type coincide, e.g.

$$[v_1^{eqs'} :: v_2^{eqs'}]^{T\ \mathsf{list}}_{eqs'} \quad \longrightarrow_{eqs} \quad [v_1^{eqs'}]^{T}_{eqs'} :: [v_2^{eqs'}]^{T\ \mathsf{list}}_{eqs'}$$

Bracket type revelation permits use of the ambient type equations to reveal an abstract bracket type, and bracket elimination removes redundant nested brackets.

The semantics must also suitably-bracket expressions used in substitutions to ensure they retain their original type equations. One sees this in the rule for pushing brackets through lambdas and in the reduction axioms for function application and recursive functions.

At several points it is necessary to take a value at some equations eqs and construct a value that makes sense at the empty set of equations $\varnothing$, e.g. when marshalling a value, passing a value to a primitive operator or an OS call, etc.

The treatment of store locations and names is discussed in §**??**.

14.6 Marshalling and unmarshalling

(§16.8.5, page 133) A marshalled value is a byte-string representation of an mv, containing data as below.

$$mv \quad ::= \quad \mathbf{marshalled}(E_\mathrm{n},\ E_s,\ s,\ \mathit{definitions},\ e,\ T) \quad \text{Marshalled value}$$

Here e is the core value being shipped, T its type, s a store, E_s a store typing, $\mathit{definitions}$ is a sequence of module definitions, and E_n is a name environment.

The E_n and E_s would not be shipped in an production implementation, but are needed to state type preservation and for runtime typechecking of reachable states. They are shipped in our implementation only if literal hashes are not being used.

As with the other syntactic objects, marshalled values are taken up to alpha equivalence. Here: the name environment E_n binds in everything to the right and internally contains no cycles; the store environment E_s binds in everything to the right and may contain internal cycles; the store s and the $\mathit{definitions}$ bind to the right and may mutually refer to each other; the s may contain internal cycles.

To characterise the wire format, we simply suppose a fixed partial function $\mathrm{raw\_unmarshal}$ from strings to marshalled values that includes all marshalled values in its range. The semantics for marshalling constructs an mv and then nondeterministically allows any string that is mapped to that mv. This permits small variations in the wire format (which a characterisation in terms of a function from marshalled values to strings would not). We use actual strings for wire-format marshalled values, instead of (say) adding a language type marshalled with elements of the form mv, so that the semantics can capture the behaviour of programs that do string operations — for example, extracting marshalled values from TCP byte streams.

The dynamic semantics for $\mathbf{marshal}\ e_1\ e_2\ :\ T$ first evaluates e_1 to a string MK — the mark at which module bindings will be cut. It then evaluates e_2 to a value in the ambient colour *eqs*, and then to a value in the empty colour $\varnothing$, giving a redex $\mathbf{marshalz}\ \mathrm{MK}\ v^{\varnothing}\ :\ T$ in a configuration of the form

$$E_{\mathrm{n}}\ \mathbf{;}\ \langle E_s,\ s,\ \mathit{definitions},\ P\,|\,\mathbf{n} : \mathit{TCC}_{\mathit{eqs}}.\mathbf{marshalz}\ \mathrm{MK}\ v^{\varnothing}\ :\ T\rangle$$

Suppose

$$\mathit{definitions} = \mathit{definitions}_1\ \mathbf{;;}\ \mathbf{mark}\ \ \mathrm{MK}\ \mathbf{;;}\ \mathit{definitions}_2$$
$$\mathbf{mark}\ \ \mathrm{MK}\ \notin\ \mathit{definitions}_2$$

In outline, what we do is prune $\mathit{definitions}_2$, omitting any modules that are not needed by the marshalled value, and on the way calculating which modules from $\mathit{definitions}_1$ are referred to. We then go through $\mathit{definitions}_1$ constructing an import for each of those. The constructed imports either have an exact-name constraint and HERE_ALREADY resolvespec, for a cut $\mathbf{cmodule}$ binding, or with the original version constraint and resolvespec, for a cut $\mathbf{cimport}$ binding. Note that this does not involve any definitions of executing threads, so the $\mathit{definitions}'$ that are shipped are guaranteed to be definition values. The shipped $\mathit{definitions}'$ includes (copies of) all the marks passed through in $\mathit{definitions}_2$, but not of the $\mathbf{mark}\ \ \mathrm{MK}$ being marshalled with respect to. The marshalled value also includes a copy of the reachable part of the store: the value $v^{\varnothing}$ may contain store locations. They may contain other store locations, but also module identifiers (under lambdas) from $\mathit{definitions}_1$ and $\mathit{definitions}_2$ which must be taken into account. Moreover, as *definitions* may be the result of module initialisation, it too may contain store locations.

Unmarshalling of a string $\underline{s}$, in a configuration of the form

$$E_{\mathrm{n}}\ \mathbf{;}\ \langle E_s,\ s,\ \mathit{definitions},\ P\,|\ \mathit{TCC}_{\mathit{eqs}}.\mathbf{unmarshal}\ \ \underline{s}\ \mathbf{as}\ T\rangle$$

takes the $\mathrm{raw\_unmarshal}$ image of $\underline{s}$, say $\mathbf{marshalled}(E_{\mathrm{n}}{}',\ E_{s'},\ s',\ \mathit{definitions}',\ v'^{\varnothing},\ T')$, adds the store fragment s' to the current store s (disjointly), adds the $\mathit{definitions}'$ to the end of *definitions* (avoiding clashes with alpha equivalence), and merges E_{n} with any new names from $E_{\mathrm{n}}{}'$. Note this depends on the fact that $\mathit{definitions}'$ are fully evaluated.

Existing marks will thus be shadowed by marks in $\mathit{definitions}'$, which is sometimes desirable but not always. This is a defect of the linear mark/module structure.

14.7 Module field instantiation

(§16.8.6, page 135) This specifies the runtime semantics for resolution of module field references, describing what happens when an $\mathrm{M}_M.\mathrm{x}$ appears in redex position. In general we have to chase through a (possibly-empty) sequence of linked imports until we arrive at either a module definition or an unlinked import. In the former case we instantiate the $\mathrm{M}_M.\mathrm{x}$ with its value from the module definition. In the latter, we work through the *resolvespec* attached to the import $\mathrm{M}'_{M'}$ that is unlinked. Each atomic resolve spec is dealt with in turn, as follows:

- STATIC_LINK – fail, raising an exception;
- HERE_ALREADY – look in the preceeding modules for one that matches the signature and version constraint. If there is one, link this import to it;
- *URI* – try to load a *compiledunit* from the *URI*. If we find one containing a module that matches the external name, signature and version constraint, and has $\mathit{eo} = \mathrm{empty}$, add it to the configuration's *definitions* just before the import, and link the import to it.

In the latter two cases, if there is a failure we try the subsequent atomic resolve specs, raising an exception if there are no more. Success leaves the $\mathrm{M}_M.\mathrm{x}$ again in redex position, where it can now be instantiated as in the former case.

Note that no additional linking is done, either to or of newly-loaded modules. Some user control of this would be desirable.

Resolution may involve IO, to pull a file containing compiled definitions from the web or filesystem. The semantics expresses this with labelled transitions $\mathbf{n}\ :\ \mathrm{GetURI}(\mathit{URI})$ for making a request for a

URI, $\mathbf{n}$: $\mathrm{DeliverURI}(E_{\mathrm{n}}{}', \mathit{definitions}')$ for receiving a name environment $E_{\mathrm{n}}{}'$ and $\mathit{definitions}'$, and $\mathbf{n}$: $\mathrm{CannotFindURI}$ if no file is found at the URI. The intermediate state is stored in the term, as a $\mathbf{resolve\_blocked}(\mathrm{M}_M.\mathrm{x}, \mathrm{M}'_{M'}, \mathit{resolvespec})$ for the blocked state and a $\mathbf{resolve}(\mathrm{M}_M.\mathrm{x}, \mathrm{M}'_{M'}, \mathit{resolvespec})$ for a state which is just about to make a request. The action must be split into send and receive events as the receive may be blocked arbitrarily, and the semantics must make sense with language threads are added (note that the reduction closure rules add thread ids to the transitions of axioms of this section).

There is a choice as to how we generate coloured brackets when passing through multiple imports: (1) we could make a single eqs set containing all the equalities we need; or, (2) we could have a nested sequence of brackets. Choice (1) might require the bracket pushing rules compare eqs sets by inclusion (or logical implication). The latter doesn't suffer from this problem, and so we choose (2).

On instantiating an $\mathrm{M}_M.\mathrm{x}$ via a chain of imports, where M_M is bound by a $\mathbf{cimport}$ which is linked to a $\mathbf{cimport}$ which ... is linked to a $\mathbf{cmodule}$, the equation set is the union of the weqs of that $\mathbf{cmodule}$, the equations of the signature/structure boundary of that $\mathbf{cmodule}$, and the equaltions of the signature/likespec boundary of the initial $\mathbf{cimport}$. The intermediate imports are not relevant.

There is a technical choice relating to the semantics of instantiation, of module initialisation, and of rebinding. For a module with internal expression dependencies, e.g.

```
module M : sig  val x:int  val f:int->int  end =
  struct
    let x = 3
    let f = fun (y:int) -> x + y
  end
M.f 10
```

we can either (1) substitute $\{$`3/x`$\}$ through the body of `f` in the structure at compilation or module-initialisation time (if `x` were bound to an effect-full computation it would have to be the latter), or (2) leave the body of `f` with a free occurrence of `x`. For (1) module field instantiation is straightforward, as when `M.f` is in redex position (as here) it can be replaced by the expression-identifier-closed `fun (y:int) -> 3+y`. For (2), instantiation of `M.f` would have to rewrite the `x` on the fly, either (a) to `M.x` or (b) to `3`. Option (2a), instantiating the `M.f` to `fun (y:int) -> M.x+y`, allows more rebinding that (1) or (2b), as `M` might be rebound before the `M.x` itself appears in redex position. If one is instantiating via an import, however, and if width subsignaturing were added to the language, it seems that one could not give a satisfactory semantics for (2a). The rewrite would have to use the module identifier, not the import identifier, and hence rebinding could often lead to link errors — it would not be enough to supply an implementation of the import one was working with as other fields of the module might be required.

14.8 Concurrency

(§16.8.8, page 140) The semantics for thread creation, termination, $\mathbf{self}$, and $\mathbf{kill}$ are technically straightforward, written as reduction axioms for the judgement $P \longrightarrow P'$ (but note that as some of the axioms need to check the set of *all* locally-used thread names, these transitions are not closed under parallel composition).

Our configurations keep the states of threads, mutexes and condition variables in a single multiset; each is named with a global name (which might be hash-, fresh- or cfresh-generated). This is notationally smoother than the alternative of having separate configuration components for each.

For mutexes, POSIX describes three semantics ("kinds" of mutex): *fast*, *recursive*, and *error checking*. The *fast* semantics blocks when a thread attempts to lock a locked mutex. Note that this leads to deadlock if a thread attempts to lock a mutex it has already locked. In the LinuxThreads implementation, any thread is allowed to unlock a locked mutex. However: "This is non-portable behaviour and must not be relied upon." — POSIX is quite clear that it is assumed the *owner* is unlocking the mutex. Unlocking an unlocked mutex has no effect (this is also non-portable: in POSIX it is undefined). The *recursive* semantics maintains the owner and a lock count in the mutex; if a thread locks a mutex it already holds, this succeeds immediately and the count is incremented; on unlock, the count is decremented (again, LinuxThreads non-portably does not check the owner here). (POSIX specifies that unlocking an unlocked mutex should fail with `EPERM`, but LinuxThreads' man page suggests that unlocking an unlocked mutex (non-portably) has no effect). The *error checking* semantics is like the fast semantics, except that locking a mutex already held by the calling thread results in an immediate `EDEADLK` error, and attempting to unlock a mutex not owned by the caller results in an immediate `EPERM` error. Unlocking an unlocked mutex results in an immediate `EPERM` error. POSIX does not specify which semantics is the default. On LinuxThreads the default is *fast*, and OCaml on our Linux install appears by default to have the *fast* POSIX semantics. It is this semantics (approximately) that is expressed in our semantics. We are not committed to this, but it is fine for now.

Application programmers using threads, mutexes and condition variables depend on some fairness properties. The semantics does not express these at present.

The semantics for **thunkify** uses an auxiliary function Thunkify to atomically construct a thunk encapsulating the state of the threads, mutexes and condition variables being thunkified. When that thunk is applied (to a thunkkey list giving the names at which to reify the various parts) it uses the auxiliary Unthunkify to (atomically) build a process and place it in parallel with that of the running configuration.

If a blocking thunkify is waiting, it does not at present have a 'lock' of any kind on the things it is trying to thunkify, though that might be desirable. Here, there just are no transitions for such a thunkify, or indeed a thunkify with a thread in a fast system call. Races between overlapping **thunkify**s are thus possible, and the liveness properties even of a single **thunkify** are very weak.

Note that **thunkify** fails when applied to a thread which contains some uninitialised definitions. One could instead have it block until the initialising thread is finished. (As thunks are simply part of the expression language, and Acute modules are second-class, allowing thunkification of module-initialising threads would entail substantial changes to the language — they simply cannot be expressed in the syntax as it stands, and the evaluation order for their usages is problematic.)

15 Semantics Examples

This Section illustrates some aspects of the semantics with examples generated by our implementation. At present it covers just the compilation and marshalled values of the examples earlier in the paper. They are rendered in a typewriter variant of the grammar used in the semantics, close to the concrete source language. In addition, the pretty-printer collects together occurrences of module hashes and abstract names, introducing metavariables *h* and *n*. Internal identifiers are rendered with numeric subscripts, e.g. $\mathtt{x}_0$. Internal identifiers of modules and imports that are not printed are rendered as `:M?:`.

15.1 Compilation: hash modules

The result of compiling module `EvenCounter` from §5, page 20, is below. Scope resolution has introduced internal identifiers $\mathtt{M}_0$, $\mathtt{t}_0$, $\mathtt{start}_0$, $\mathtt{x}_0$ etc. Compilation has calculated a module name *h0_EvenCounter* as a hash of an `hmodule` form, containing external module identifier, signature, version expression, and structure. This hash is taken up to alpha equivalence by choosing canonical strings for bound identifiers and up to type equality by substituting out earlier module names for identifiers and substituting out internal type dependencies. (The hash body shown is pretty-printed in a different mode to that used to build the actual hash to make it more readable, with identifiers based on the source language strings.) Both the symbolic and literal hash forms are shown. The compiled `cmodule EvenCounter` has two signatures, one in which source abstract types are still abstract and one in which they have been selfified using the module name and substituted through, e.g. the `type t[t₀] : Eq(`*h0_EvenCounter*`.t)` and `val start[start₀] :` *h0_EvenCounter*`.t`. The version of the compiled module has defaulted to its hash-generated name.

```
cmodule EvenCounter[M₀] h0_EvenCounter : {}
  sig
    type t[t₀] : Type
    val start[start₀] : t₀
    val get[get₀] : t₀ -> int
    val up[up₀] : t₀ -> t₀
  end (valuable, valuable)
  sig
    type t[t₀] : Eq(h0_EvenCounter.t)
    val start[start₀] : h0_EvenCounter.t
    val get[get₀] : h0_EvenCounter.t -> int
    val up[up₀] : h0_EvenCounter.t -> h0_EvenCounter.t
  end
  version h0_EvenCounter
= struct
    type t[t₀] = int
    let start[start₀] = 0
    let get[get₀] = function (x₀ : int) -> x₀
    let up[up₀] = function (x₀ : int) -> 2 + x₀
  end
```

where

```
    h0_EvenCounter = hash(hmodule EvenCounter : {}
                            sig
                              type t[t₀] : Type
                              val start[start₀] : t₀
                              val get[get₀] : t₀ -> int
                              val up[up₀] : t₀ -> t₀
                            end
                            version myname
                          = struct
                              type t[t₀] = int
                              let start[start₀] = 0
```

```
                             let get[get0] = function (x0 : int) -> x0
                             let up[up0] = function (x0 : int) -> 2 + x0
                           end)
                  = 0#E09083A42C03366FA0698C81E0063682
```

15.2 Compilation: fresh modules

The module `NCounter` from §5, page 21, compiles to:

```
module fresh NCounter[M0]
: sig
    type t[t0] : Type
    val start[start0] : t0
    val get[get0] : t0 -> int
    val up[up0] : t0 -> t0
  end
  version myname
= struct
    type t[t0] = int
    let start[start0] = 0
    let get[get0] = function (x0 : int) -> x0
    let up[up0] =
      match Pervasives[Lib_Pervasives].read_int () with (step0 : int) ->
        function (x0 : int) -> step0 + x0
  end
```

The first execution step of this involves generating a fresh name for the module and hashifying it, after which the `read_int` performs IO.

15.3 Compilation: hash module dependencies

The result of compiling modules `M` and `EvenCounter` from §8, page 28, is below. Two hashes are constructed to use as the names of the two modules, *h0_M* and *h1_EvenCounter*. Note that the up field of the `cmodule EvenCounter` structure refers to `M[M0].f x0`, whereas the up field of the `hmodule EvenCounter` in the body of its hash refers to *h0_M*`.f x0`, using the earlier hash.

```
cmodule M[M0] h0_M : {}
  sig
    val f[f0] : int -> int
  end (valuable, valuable)
  sig
    val f[f0] : int -> int
  end
  version h0_M
= struct
    let f[f0] = function (x0 : int) -> x0 + 2
  end
```

where

```
    h0_M = hash(hmodule M : {}
                  sig
                    val f[f0] : int -> int
                  end
                  version myname
                = struct
                    let f[f_0] = function (x_0 : int) -> x_0 + 2
                  end)
         = 0#FBCF6A65CCD4F06635C5188503EA9B72

cmodule EvenCounter[M_0] h1_EvenCounter : {}
  sig
    type t[t_0] : Type
    val start[start_0] : t_0
    val get[get_0] : t_0 -> int
    val up[up_0] : t_0 -> t_0
  end (valuable, valuable)
  sig
    type t[t_0] : Eq(h1_EvenCounter.t)
    val start[start_0] : h1_EvenCounter.t
    val get[get_0] : h1_EvenCounter.t -> int
    val up[up_0] : h1_EvenCounter.t -> h1_EvenCounter.t
  end
  version h1_EvenCounter
= struct
    type t[t_0] = int
    let start[start_0] = 0
    let get[get_0] = function (x_0 : int) -> x_0
    let up[up_0] = function (x_0 : int) -> M[M_0].f x_0
  end
```

where

```
    h1_EvenCounter = hash(hmodule EvenCounter : {}
                            sig
                              type t[t_0] : Type
                              val start[start_0] : t_0
                              val get[get_0] : t_0 -> int
                              val up[up_0] : t_0 -> t_0
                            end
                            version myname
                          = struct
                              type t[t_0] = int
                              let start[start_0] = 0
                              let get[get_0] = function (x_0 : int) -> x_0
                              let up[up_0] = function (x_0 : int) -> h0_M.f x_0
                            end)
                   = 0#F5EF4DE7D2DCB9E8D56EE8AAD19AE3E9
```

15.4 Compilation: cfresh modules

The `cfresh` code from Scenario 2, page 23, compiles to:

```
cmodule M[M_0] h0 : {}
  sig
    val c[c_0] : int name
  end (cvaluable, cvaluable)
  sig
```

```
    val c[c₀] : int name
  end
  version h0
= struct
    let c[c₀] = name_value(n1 %[int])
  end
```

where

```
    h0 = n0 = 0#2D3C130675CA1701BB285B45679B27BD
```

where

```
    n0 = 0$2D3C130675CA1701BB285B45679B27BD

    n1 = 0$5C334F890F66E794B27733D88A8228A7 %[int]
```

15.5 Compilation: constructing expression names from module hashes

The result of compiling the shared code from Scenario 3, page 23, is below. Note the hash *h0_N* involves the intension of `N.f`, and this appears within the `c` field of the `cmodule M` at the end. (Skip over the intervening hashes of standard library modules *h1_IO*, *h2_Pervasives*, and *h3_Persist*.)

```
cmodule N[M₀] h0_N : {}
  sig
    val f[f₀] : int -> unit
  end (valuable, valuable)
  sig
    val f[f₀] : int -> unit
  end
  version h0_N
= struct
    let f[f₀] = function (x₀ : int) -> IO[Lib_IO].print_int (x₀ + 100)
  end
```

where

```
    h0_N = hash(hmodule N : {}
                sig
                  val f[f₀] : int -> unit
                end
                version myname
              = struct
                  let f[f₀] = function (x₀ : int) -> h1_IO.print_int (x₀ + 100)
                end)
         = 0#75ABE6A8126FA4F96A02789EAC83E487
```

where

```
    h1_IO = hash(hmodule IO : {}
                 sig
                   val print_int[print_int₀] : int -> unit
                   val print_string[print_string₀] : string -> unit
                   val print_newline[print_newline₀] : unit -> unit
                   val send[send₀] : string -> unit
                   val receive[receive₀] : unit -> string
                 end
                 version myname
```

```
                = struct
                    let print_int[print_int$_0$] = function (x$_0$ : int) -> h2_Pervasives.print_int x$_0$
                    let print_string[print_string$_0$] =
                      function (s$_0$ : string) -> h2_Pervasives.print_string s$_0$
                    let print_newline[print_newline$_0$] =
                      function (ds$_0$ : unit) -> match ds$_0$ with () -> h2_Pervasives.print_newline ()
                    let send[send$_0$] = function (data$_0$ : string) -> h3_Persist.write data$_0$
                    let receive[receive$_0$] =
                      function (ds$_0$ : unit) -> match ds$_0$ with () -> h3_Persist.read ()
                  end)
         = 0#2808905E9138A8AA18FF6FF8E169EDED
```

where

```
    h2_Pervasives = hash(hmodule Pervasives : {}
                           sig
                             val string_of_int[string_of_int$_0$] : int -> string
                             val int_of_string[int_of_string$_0$] : string -> int
                             val print_string[print_string$_0$] : string -> unit
                             val print_int[print_int$_0$] : int -> unit
                             val print_endline[print_endline$_0$] : string -> unit
                             val print_newline[print_newline$_0$] : unit -> unit
                           end
                           version myname
                         = struct
                             let string_of_int[string_of_int$_0$] =
                               function (ds$_0$ : int) -> %"Apervasives_string_of_int" ds$_0$
                             let int_of_string[int_of_string$_0$] =
                               function (ds$_0$ : string) -> %"Apervasives_int_of_string" ds$_0$
                             let print_string[print_string$_0$] =
                               function (ds$_0$ : string) -> %"Apervasives_print_string" ds$_0$
                             let print_int[print_int$_0$] =
                               function (ds$_0$ : int) -> %"Apervasives_print_int" ds$_0$
                             let print_endline[print_endline$_0$] =
                               function (ds$_0$ : string) -> %"Apervasives_print_endline" ds$_0$
                             let print_newline[print_newline$_0$] =
                               function (ds$_0$ : unit) -> %"Apervasives_print_newline" ds$_0$
                           end)
                    = 0#4A5FE3EC8D80DFA70AD367461DD525AA
    h3_Persist = hash(hmodule Persist : {}
                        sig
                          val write[write$_0$] : string -> unit
                          val read[read$_0$] : unit -> string
                          val write2[write2$_0$] : string -> unit
                          val read2[read2$_0$] : unit -> string
                        end
                        version myname
                      = struct
                          let write[write$_0$] = function (ds$_0$ : string) -> %"Persist_write" ds$_0$
                          let read[read$_0$] = function (ds$_0$ : unit) -> %"Persist_read" ds$_0$
                          let write2[write2$_0$] = function (ds$_0$ : string) -> %"Persist_write2" ds$_0$
                          let read2[read2$_0$] = function (ds$_0$ : unit) -> %"Persist_read2" ds$_0$
                        end)
               = 0#D90A83203B41EF6E6E512B3E5FF54850



cmodule M[M$_0$] h4_M : {}
```

```
  sig
    val c[c_0] : int name
  end (valuable, valuable)
  sig
    val c[c_0] : int name
  end
  version h4_M
= struct
    let c[c_0] = hash(int, "", hash(h0_N.f) %[int -> unit]) %[int]
  end
```

where

```
    h4_M = hash(hmodule M : {}
                  sig
                    val c[c_0] : int name
                  end
                  version myname
                = struct
                    let c[c_0] = hash(int, "", hash(h0_N.f) %[int -> unit]) %[int]
                  end)
         = 0#2F7112C065BF44899C98205353679AD7
```

15.6 Compilation: type normalisation and marshalling within abstraction boundaries

The result of compilation for the example of marshalling within an abstraction boundary, §8.5, page 32, is below. Note here in the `cmodule struct` that the types at which marshalling and unmarshalling are done, in the `send` and `receive` fields, have both been normalised to `int` from the source-language `t`.

```
cmodule EvenCounter[M_0] h0_EvenCounter : {}
  sig
    type t[t_0] : Type
    val start[start_0] : t_0
    val get[get_0] : t_0 -> int
    val up[up_0] : t_0 -> t_0
    val send[send_0] : t_0 -> unit
    val recv[recv_0] : unit -> t_0
  end (valuable, valuable)
  sig
    type t[t_0] : Eq(h0_EvenCounter.t)
    val start[start_0] : h0_EvenCounter.t
    val get[get_0] : h0_EvenCounter.t -> int
    val up[up_0] : h0_EvenCounter.t -> h0_EvenCounter.t
    val send[send_0] : h0_EvenCounter.t -> unit
    val recv[recv_0] : unit -> h0_EvenCounter.t
  end
  version h0_EvenCounter
= struct
    type t[t_0] = int
    let start[start_0] = 0
    let get[get_0] = function (x_0 : int) -> x_0
    let up[up_0] = function (x_0 : int) -> 2 + x_0
    let send[send_0] = function (x_0 : int) -> IO[Lib_IO].send (marshal "StdLib" x_0 : int)
    let recv[recv_0] =
      function (ds_0 : unit) -> match ds_0 with () -> (unmarshal (IO[Lib_IO].receive ()) as int)
  end
```

where

```
    h0_EvenCounter = hash(hmodule EvenCounter : {}
                            sig
                              type t[t_0] : Type
                              val start[start_0] : t_0
                              val get[get_0] : t_0 -> int
                              val up[up_0] : t_0 -> t_0
                              val send[send_0] : t_0 -> unit
                              val recv[recv_0] : unit -> t_0
                            end
                            version myname
                          = struct
                              type t[t_0] = int
                              let start[start_0] = 0
                              let get[get_0] = function (x_0 : int) -> x_0
                              let up[up_0] = function (x_0 : int) -> 2 + x_0
                              let send[send_0] =
                                function (x_0 : int) -> h1_IO.send (marshal "StdLib" x_0 : int)
                              let recv[recv_0] =
                                function (ds_0 : unit) ->
                                  match ds_0 with () -> (unmarshal (h1_IO.receive ()) as int)
                            end)
                   = 0#A896BA1BA88F408A0AEE9744742E2717
```

where

```
    h1_IO = hash(hmodule IO : {}
                   sig
                     val print_int[print_int_0] : int -> unit
                     val print_string[print_string_0] : string -> unit
                     val print_newline[print_newline_0] : unit -> unit
                     val send[send_0] : string -> unit
                     val receive[receive_0] : unit -> string
                   end
                   version myname
                 = struct
                     let print_int[print_int_0] = function (x_0 : int) -> h2_Pervasives.print_int x_0
                     let print_string[print_string_0] =
                       function (s_0 : string) -> h2_Pervasives.print_string s_0
                     let print_newline[print_newline_0] =
                       function (ds_0 : unit) -> match ds_0 with () -> h2_Pervasives.print_newline ()
                     let send[send_0] = function (data_0 : string) -> h3_Persist.write data_0
                     let receive[receive_0] =
                       function (ds_0 : unit) -> match ds_0 with () -> h3_Persist.read ()
                   end)
          = 0#2808905E9138A8AA18FF6FF8E169EDED
```

where

```
    h2_Pervasives = hash(hmodule Pervasives : {}
                           sig
                             val string_of_int[string_of_int_0] : int -> string
                             val int_of_string[int_of_string_0] : string -> int
                             val print_string[print_string_0] : string -> unit
                             val print_int[print_int_0] : int -> unit
                             val print_endline[print_endline_0] : string -> unit
                             val print_newline[print_newline_0] : unit -> unit
```

```
             end
             version myname
            = struct
              let string_of_int[string_of_int0] =
                function (ds0 : int) -> %"Apervasives_string_of_int" ds0
              let int_of_string[int_of_string0] =
                function (ds0 : string) -> %"Apervasives_int_of_string" ds0
              let print_string[print_string0] =
                function (ds0 : string) -> %"Apervasives_print_string" ds0
              let print_int[print_int0] =
                function (ds0 : int) -> %"Apervasives_print_int" ds0
              let print_endline[print_endline0] =
                function (ds0 : string) -> %"Apervasives_print_endline" ds0
              let print_newline[print_newline0] =
                function (ds0 : unit) -> %"Apervasives_print_newline" ds0
             end)
         = 0#4A5FE3EC8D80DFA70AD367461DD525AA
  h3_Persist = hash(hmodule Persist : {}
            sig
              val write[write0] : string -> unit
              val read[read0] : unit -> string
              val write2[write20] : string -> unit
              val read2[read20] : unit -> string
            end
            version myname
           = struct
              let write[write0] = function (ds0 : string) -> %"Persist_write" ds0
              let read[read0] = function (ds0 : unit) -> %"Persist_read" ds0
              let write2[write20] = function (ds0 : string) -> %"Persist_write2" ds0
              let read2[read20] = function (ds0 : unit) -> %"Persist_read2" ds0
            end)
       = 0#D90A83203B41EF6E6E512B3E5FF54850
```

```
EvenCounter[M0].send EvenCounter[M0].start
```

15.7 Compilation: imports

The result of compiling the `M` and `EvenCounter` import example, §8, page 28, is below. Note here that the `cmodule M` and the `cimport M` have quite different names, the hashes *h0_M* and *h1_M* respectively. It is the latter that appears in the hash *h2_EvenCounter* of the `EvenCounter` module, and that thus would appear in the runtime type names of any source-language `EvenCounter.t` types (there are no such occurrences in this example).

```
cmodule M[M0] h0_M : {}
  sig
    val f[f0] : int -> int
  end (valuable, valuable)
  sig
    val f[f0] : int -> int
  end
  version h0_M
= struct
    let f[f0] = function (x0 : int) -> x0 + 2
  end
```

where

```
    h0_M = hash(hmodule M : {}
                  sig
                    val f[f_0] : int -> int
                  end
                  version myname
                = struct
                    let f[f_0] = function (x_0 : int) -> x_0 + 2
                  end)
         = 0#FBCF6A65CCD4F06635C5188503EA9B72
```

```
cimport M[M_1] h1_M
: sig
    val f[f_0] : int -> int
  end (valuable, valuable)
 sig
    val f[f_0] : int -> int
  end
  version *
  like  struct      end
  by Here_Already
  = M[M_0]
```

where

```
    h1_M = hash(himport M: sig    val f[f_0] : int -> int  end  version *  like  struct      end)
         = 0#BD28AD1B690255427DBA10F9471C765B
```

```
mark "MK"
cmodule EvenCounter[M_0] h2_EvenCounter : {}
  sig
    type t[t_0] : Type
    val start[start_0] : t_0
    val get[get_0] : t_0 -> int
    val up[up_0] : t_0 -> t_0
  end (valuable, valuable)
  sig
    type t[t_0] : Eq(h2_EvenCounter.t)
    val start[start_0] : h2_EvenCounter.t
    val get[get_0] : h2_EvenCounter.t -> int
    val up[up_0] : h2_EvenCounter.t -> h2_EvenCounter.t
  end
  version h2_EvenCounter
= struct
    type t[t_0] = int
    let start[start_0] = 0
    let get[get_0] = function (x_0 : int) -> x_0
    let up[up_0] = function (x_0 : int) -> M[M_1].f x_0
  end
```

where

```
    h2_EvenCounter = hash(hmodule EvenCounter : {}
                            sig
                              type t[t_0] : Type
                              val start[start_0] : t_0
                              val get[get_0] : t_0 -> int
                              val up[up_0] : t_0 -> t_0
                            end
```

```
                           version myname
                         = struct
                             type t[t₀] = int
                             let start[start₀] = 0
                             let get[get₀] = function (x₀ : int) -> x₀
                             let up[up₀] = function (x₀ : int) -> h1_M.f x₀
                           end)
                  = 0#A60A0BC55D9A1B0F753ED1FA69475D83

IO[Lib_IO].send
  (marshal "MK"
     (function (ds₀ : unit) ->
        match ds₀ with () -> EvenCounter[M₀].get (EvenCounter[M₀].up EvenCounter[M₀].start))
   : unit -> int)
```

15.8 Compilation: imports with abstract type fields

Here we show a fleshed-out version of the last two examples of §8.1, page 28. Consider the import below, which has a non-exact-name version and has a signature containing an abstract type field. It has a *likespec* `like M` specifying that the representation type for that type must be the same as that of the preceeding `module M` (one could equivalently give the *likespec* explicitly, writing `like struct type t=int end`). The import is initially linked to `M`.

```
  module M : sig    type t      val x:t  end
           version 2.4.9
           = struct type t=int  let x=17 end

  import M : sig    type t      val x:t  end
           version 2.4.7-
           like M
           = M
  mark "MK"
  (marshal "MK" M.x : M.t)
```

The result of compiling this code is below. Note that the *likespec* data appears in the import hash *h1_M*, which is used to form the type *h1_M*.`t`) at which the final `marshal` is done. Type errors caused by rebinding the import to modules with different representation types are thus excluded.

```
cmodule M[M₀] h0_M : {}
  sig
    type t[t₀] : Type
    val x[x₀] : t₀
  end (valuable, valuable)
  sig
    type t[t₀] : Eq(h0_M.t)
    val x[x₀] : h0_M.t
  end
  version 2.4.9
= struct
    type t[t₀] = int
    let x[x₀] = 17
  end
```

where

```
    h0_M = hash(hmodule M : {}
                  sig
```

```
                    type t[t_0] : Type
                    val x[x_0] : t_0
                  end
                  version 2.4.9
                = struct
                    type t[t_0] = int
                    let x[x_0] = 17
                  end)
         = 0#D17E216FA11DBB5BDDB0B020646900A3

cimport M[M_1] h1_M
: sig
    type t[t_0] : Type
    val x[x_0] : t_0
  end (valuable, valuable)
 sig
    type t[t_0] : Eq(h1_M.t)
    val x[x_0] : h1_M.t
  end
  version 2.4.7-
  like  struct    type t[t_0] = int  end
  by Here_Already
  = M[M_0]
```

where

```
    h1_M = hash(himport M
                : sig
                    type t[t_0] : Type
                    val x[x_0] : t_0
                  end
                  version 2.4.7-
                  like  struct    type t[t_0] = int  end)
         = 0#FE46E0350E1A6EAFB00547C6E836B6CB

mark "MK"
(marshal "MK" M[M_1].x : h1_M.t)
```

15.9 Compilation: breaking abstractions

The result of compiling the `with!` example of §8.2, page 29, is below. Here *h1_EvenCounter* is the hash of the original module and *h0_EvenCounter* is the hash of the new version with a `down` operation. The type equation {*h1_EvenCounter*.t=int} is recorded in the `cmodule`.

```
cmodule EvenCounter[M_0] h0_EvenCounter : {h1_EvenCounter.t=int}
  sig
    type t[t_0] : Eq(h1_EvenCounter.t)
    val start[start_0] : h1_EvenCounter.t
    val get[get_0] : h1_EvenCounter.t -> int
    val up[up_0] : h1_EvenCounter.t -> h1_EvenCounter.t
    val down[down_0] : h1_EvenCounter.t -> h1_EvenCounter.t
  end (valuable, valuable)
  sig
    type t[t_0] : Eq(h1_EvenCounter.t)
    val start[start_0] : h1_EvenCounter.t
    val get[get_0] : h1_EvenCounter.t -> int
    val up[up_0] : h1_EvenCounter.t -> h1_EvenCounter.t
    val down[down₀] : h1_EvenCounter.t -> h1_EvenCounter.t
  end
  version h0_EvenCounter
= struct
    type t[t₀] = int
    let start[start₀] = 0
    let get[get₀] = function (x₀ : int) -> x₀
    let up[up₀] = function (x₀ : int) -> 2 + x₀
    let down[down₀] = function (x₀ : int) -> x₀ - 2
  end
```

where

```
    h0_EvenCounter = hash(hmodule EvenCounter : {h1_EvenCounter.t=int}
                             sig
                               type t[t₀] : Eq(h1_EvenCounter.t)
                               val start[start₀] : h1_EvenCounter.t
                               val get[get₀] : h1_EvenCounter.t -> int
                               val up[up₀] : h1_EvenCounter.t -> h1_EvenCounter.t
                               val down[down₀] : h1_EvenCounter.t -> h1_EvenCounter.t
                             end
                             version myname
                           = struct
                               type t[t₀] = int
                               let start[start₀] = 0
                               let get[get₀] = function (x₀ : int) -> x₀
                               let up[up₀] = function (x₀ : int) -> 2 + x₀
                               let down[down₀] = function (x₀ : int) -> x₀ - 2
                             end)
                   = 0#E5E0448DECB46AC6F6E22081B274831D
```

where

```
    h1_EvenCounter = hash(hmodule EvenCounter : {}
                             sig
                               type t[t₀] : Type
                               val start[start₀] : t₀
                               val get[get₀] : t₀ -> int
                               val up[up₀] : t₀ -> t₀
                             end
                             version myname
                           = struct
                               type t[t₀] = int
                               let start[start₀] = 0
                               let get[get₀] = function (x₀ : int) -> x₀
                               let up[up₀] = function (x₀ : int) -> 2 + x₀
                             end)
                   = 0#E09083A42C03366FA0698C81E0063682
```

15.10 Marshalled values

In these examples the `-hack_optimise` option of the implementation is used to suppress most vacuous coloured brackets, as described in §16.11.

The marshalled value of the first example of §3, page 13, is below. It contains simply a value and a type.

```
marshalled ({  }, { }, {}, {}, 5, int)
```

The marshalled value of the first example of §4.2, page 15, is below. Here the module `M` is shipped together with a function that refers to it.

```
marshalled (
  {  },
  {cmodule M[M0] h0_M : {}
      sig
        val y[x] : int
      end (valuable, valuable)
      sig
        val y[x] : int
      end
      version h0_M
    = struct
        let y[x] = 6
      end

  }, {},
  {},

  (function (x : unit) -> match x with () -> M[M0].y),
  unit -> int)
```

The marshalled value of the second example of §4.2, page 15, is below. This includes an import for `M1` and the module for `M2`, and a function that refers to both. The former is automatically generated for the module binding of `M1` that is cut by the mark. It is constructed with an exact-name version constraint, here to the hash-generated name *h0_M1* of `M1`. The *likespec* of the import is also constructed based on the original module, though here that had no abstract types so the resulting *likespec* is empty.

```
marshalled (
  {  },
  {cimport M1[M0] h0_M1
    : sig
        val y[x] : int
      end (valuable, valuable)
     sig
        val y[x] : int
      end
      version name = h0_M1
      like  struct      end
      by Here_Already
      = unlinked
    cmodule M2[M0] h1_M2 : {}
      sig
        val z[x] : int
      end (valuable, valuable)
      sig
        val z[x] : int
      end
      version h1_M2
    = struct
        let z[x] = 3
      end
  }, {},
  {},

  (function (x : unit) -> match x with () -> (M1[$M_0$].y, M2[$M_0$].z)),
  unit -> int * int)
```

The marshalled value of the third example of §4.2, page 16, is below. Here the marshalled import is essentially that supplied by the user above the mark (hence, that of the binding that is cut by the mark), not an automatically-generated default import.

```
marshalled (
  {  },
  {cimport M1[$M_0$] h0_M1
    : sig
        val y[x] : int
      end (valuable, valuable)
     sig
        val y[x] : int
      end
      version name = h0_M1
      like  struct      end
      by Here_Already
      = unlinked
    cimport M1[$M_1$] h1_M1
    : sig
        val y[x] : int
      end (valuable, valuable)
     sig
        val y[x] : int
      end
      version *
      like  struct      end
      by Here_Already
      = unlinked
    cmodule M2[$M_0$] h2_M2 : {}
      sig
        val z[x] : int
      end (valuable, valuable)
      sig
        val z[x] : int
      end
      version h2_M2
    = struct
        let z[x] = 3
      end

  }, {},
  {},

  (function (x : unit) -> match x with () -> (M1[$M_1$].y, M2[$M_0$].z)),
  unit -> int * int)
```

The marshalled value of the first example of §4.3, page 16, is below. Here one can see that the `M.y` under the `fun` in the source language has not been instantiated (and an import is shipped, binding that `M`) whereas the unguarded `M.y` has been instantiated by its value `6` before marshalling took place.

```
marshalled (
  {  },
  {cimport M[M_0] h0_M
    : sig
        val y[x] : int
      end (valuable, valuable)
     sig
        val y[x] : int
      end
      version name = h0_M
      like  struct      end
      by Here_Already
      = unlinked
    cimport M[M_1] h1_M
    : sig
        val y[x] : int
      end (valuable, valuable)
     sig
        val y[x] : int
      end
      version *
      like  struct      end
      by Here_Already
      = unlinked

  }, {},
  {},

  (6, (function (x : unit) -> match x with () -> M[M_1].y)),
  int * (unit -> int))
```

The marshalled value of the example of §4.5, page 17, is below, with just an import being sent.

```
marshalled (
  {  },
  {cimport M[M_0] h0_M
    : sig
        val y[x] : int
      end (valuable, valuable)
     sig
        val y[x] : int
      end
      version name = h0_M
      like  struct      end
      by Here_Already
      = unlinked
    cimport M[M_1] h1_M
    : sig
        val y[x] : int
      end (valuable, valuable)
     sig
        val y[x] : int
      end
      version *
      like  struct      end
      by Here_Already
      = unlinked
  }, {},
  {},

  (function (x : unit) -> match x with () -> M[M₁].y),
  unit -> int)
```

The marshalled value of the example of §4.9, page 19, is below, with a store fragment mapping a single location to the value 5 and a store typing associating that location with type `int`. The expression part is just that location.

```
marshalled ({  }, { }, {(<1> : int ref)}, {(<1> := 5)}, <1>, int ref)
```

The marshalled value of the thunkify example of §9.11, page 37, is below. The body is a function which takes a thunkkey list containing a thread name and a mutex name and reconstructs the original thread and mutex state at those names.

```
marshalled (
  {  },
  {cimport Pervasives[Lib_Pervasives] h0_Pervasives
    : sig
        val string_of_int[x] : int -> string
        val int_of_string[x₀] : string -> int
        val print_string[x₁] : string -> unit
        val print_int[x₂] : int -> unit
        val print_endline[x₃] : string -> unit
        val print_newline[x₄] : unit -> unit
      end (valuable, valuable)
     sig
        val string_of_int[x] : int -> string
        val int_of_string[x₀] : string -> int
        val print_string[x₁] : string -> unit
        val print_int[x₂] : int -> unit
        val print_endline[x₃] : string -> unit
        val print_newline[x₄] : unit -> unit
      end
      version name = h0_Pervasives
      like  struct      end
      by Here_Already
      = unlinked
    cimport Persist[Lib_Persist] h1_Persist
    : sig
        val write[x] : string -> unit
        val read[x₀] : unit -> string
        val write2[x₁] : string -> unit
        val read2[x₂] : unit -> string
      end (valuable, valuable)
     sig
        val write[x] : string -> unit
        val read[x₀] : unit -> string
        val write2[x₁] : string -> unit
        val read2[x₂] : unit -> string
      end
      version name = h1_Persist
      like  struct      end
      by Here_Already
      = unlinked
    cimport IO[Lib_IO] h2_IO
    : sig
        val print_int[x] : int -> unit
        val print_string[x0] : string -> unit
        val print_newline[x1] : unit -> unit
        val send[x2] : string -> unit
        val receive[x3] : unit -> string
      end (valuable, valuable)
     sig
        val print_int[x] : int -> unit
        val print_string[x0] : string -> unit
        val print_newline[x1] : unit -> unit
        val send[x2] : string -> unit
        val receive[x3] : unit -> string
      end
      version name = h2_IO
      like  struct      end
      by Here_Already
      = unlinked

  }, {},
  {},

  (function (x : thunkkey list) ->
     match x with Thread ((x2 : thread name), (x1 : thunkifymode))::Mutex (x0 : mutex name)::([] %[
                  thunkkey]) ->
       unthunkify
       (Thunked_thread (x2,
                        (function (x3 : unit) ->
                           (let rec
                              x4 : int -> unit =
                                function
                                  (x5 : int) ->
                                    IO[Lib_IO].print_int x5;  IO[Lib_IO].print_newline ();  x4 (x5 + 1)
                            in
                            x4) ([4 ]^{int}_{{}} + 1))) ::
        Thunked_mutex (x0, false) :: ([] %[thunklet]))),
  thunkkey list -> unit)
where
    h0_Pervasives = hash(hmodule Pervasives : {}
                           sig
                             val string_of_int[x] : int -> string
                             val int_of_string[x0] : string -> int
                             val print_string[x1] : string -> unit
                             val print_int[x2] : int -> unit
                             val print_endline[x3] : string -> unit
                             val print_newline[x4] : unit -> unit
                           end
                           version myname
                         = struct
                             let string_of_int[x] = function (x : int) -> %"Apervasives_string_of_int" x
                             let int_of_string[x0] =
                               function (x0 : string) -> %"Apervasives_int_of_string" x0
                             let print_string[x1] =
                               function (x1 : string) -> %"Apervasives_print_string" x1
                             let print_int[x2] = function (x2 : int) -> %"Apervasives_print_int" x2
                             let print_endline[x3] =
                               function ($x_3$ : string) -> %"Apervasives_print_endline" $x_3$
                             let print_newline[$x_4$] =
                               function ($x_4$ : unit) -> %"Apervasives_print_newline" $x_4$
                           end)
                  = 0#4A5FE3EC8D80DFA70AD367461DD525AA
    h1_Persist = hash(hmodule Persist : {}
                        sig
                          val write[x] : string -> unit
                          val read[$x_0$] : unit -> string
                          val write2[$x_1$] : string -> unit
                          val read2[$x_2$] : unit -> string
                        end
                        version myname
                      = struct
                          let write[x] = function (x : string) -> %"Persist_write" x
                          let read[$x_0$] = function ($x_0$ : unit) -> %"Persist_read" $x_0$
                          let write2[$x_1$] = function ($x_1$ : string) -> %"Persist_write2" $x_1$
                          let read2[$x_2$] = function ($x_2$ : unit) -> %"Persist_read2" $x_2$
                        end)
               = 0#D90A83203B41EF6E6E512B3E5FF54850
    h2_IO = hash(hmodule IO : {}
                   sig
                     val print_int[x] : int -> unit
                     val print_string[$x_0$] : string -> unit
                     val print_newline[$x_1$] : unit -> unit
                     val send[$x_2$] : string -> unit
                     val receive[$x_3$] : unit -> string
                   end
                   version myname
                 = struct
                     let print_int[x] = function (x : int) -> h0_Pervasives.print_int x
                     let print_string[$x_0$] = function ($x_0$ : string) -> h0_Pervasives.print_string $x_0$
                     let print_newline[$x_1$] =
                       function ($x_1$ : unit) -> match $x_1$ with () -> h0_Pervasives.print_newline ()
                     let send[$x_2$] = function ($x_2$ : string) -> h1_Persist.write $x_2$
                     let receive[$x_3$] = function ($x_3$ : unit) -> match $x_3$ with () -> h1_Persist.read ()
                   end)
          = 0#2808905E9138A8AA18FF6FF8E169EDED
```

Part III
Definition

16 Language Definition

16.1 Metavariables

A	set of module identifiers M_M and locations l
BC	bracket context
C	single-level evaluation context, or $definitions$
CC	evaluation context
$CVAL$	compile-time valuable context
E	type environment
E_n	type environment of global abstract names
K	kind
L	set of store locations
M	module identifier (external)
MK	mark (string literal $\underline{s}$)
M	module identifer (internal)
Mo	module identifier option
Ms	sequence of module identifier
$\underline{N}$	numeric hash
P	process
Φ	filesystem
S	set of module identifier
SC	structure evaluation context
Sig	signature
Str	structure
T	type
TC	thread top-level evaluation context
TCC	thread evaluation context
$TpubfromC$	type
$TrepfromC$	type
URI	Uniform Resource Identifier
X	module name or hash
$ahvc$	atomic hash version constraint
$ahvce$	atomic hash version constraint expression
$atomicresolvespec$	atomic resolve spec
avc	atomic version constraint
$avce$	atomic version constraint expression
avn	atomic version number

$avne$	atomic version number expression
$\underline{b}$	boolean literal
$\underline{c}$	character literal
$compilationunit$	compilation unit
$compilationunit$	compilation unit
$compileddefinition$	compiled definition
$compiledfilename$	filename of compiled file
$compiledunit$	compiled unit
$config$	runtime configuration
$definition$	module definition
$definitions$	module definitions
dvc	dotted version constraint
$dvce$	dotted version constraint expression
e	expression
e_k	expression
ℓ	transition label
eo	expression option
eq	type equation
eqs	type equation set
h	hash
i	index (from $\mathbb{N}$)
$\underline{i}$	integer literal
j	index (from $\mathbb{N}$)
k	index (from $\mathbb{N}$)
l	store location
$likespec$	likespec
$likestr$	structure (in a likespec)
m	index (from $\mathbb{N}$)
$mode$	module or import mode
$mtch$	match
mv	marshalled value
n	index (from $\mathbb{N}$)
$\underline{n}$	natural number literal (from $\mathbb{N}_{2^{31}}$)
n	abstract name (from $\mathcal{N}$)
$\mathbf{n}$	abstract or hash name value
$\mathbf{nn}$	abstract or hash name
ns	name list
$nset$	name set
op	operator
p	pattern
$resolvespec$	resolvespec
ρ	substitution of T's for M_M.t's
s	store
σ	substitution of h.x's for M_M.x's
sig	signature body
$sourcedefinition$	source language definition
$sourcefilename$	filename of source file

$sourcefilenames$	set of $sourcefilename$s
$\underline{s}$	string literal
str	structure body
$strval$	structure body value
t	type identifier (internal)
θ	arbitrary syntactic entity
thk	thunk
$thks$	thunk list
tk	thunkkey
tks	thunkkey list
$tmode$	thunkify mode
t	type identifier (external)
u	expression identifier (internal)
v	value
$valuability$	valuability
$valuabilities$	valuabilities
vc	version constraint
vce	version constraint expression
vn	version number
vne	version number expression
$weqs$	withspec type equation set
$withspec$	withspec
x	expression identifier (internal)
xo	expression identifier option
x	expression identifier (external)
y	expression identifier (internal)
y	expression identifier (external)
z	expression identifier (internal)
z	expression identifier (external)

16.2 Syntax

The definition involves several related languages:

1. The *concrete source* language is the language that programmers type, e.g. `function (x,y) -> x + y + M.z`. This is concrete — a set of character sequences.
2. The *sugared source internal* language is generated by parsing, scope resolution and type inference; for example $\mathbf{function}\ (x : \mathsf{int}, y : \mathsf{int}) \rightarrow (+)\ x\ ((+)\ y\ \mathrm{M}_M.\mathrm{z})$. This is an abstract grammar, up to alpha equivalence. The x, y and M are internal identifiers, subject to alpha equivalence; the z and M are external identifiers, which are not. (In fact operators are eta-expanded to ensure they are fully applied.)
3. The *source internal* language is generated by desugaring, for example $\mathbf{function}\ (u : \mathsf{int} * \mathsf{int}) \rightarrow \mathbf{match}\ u\ \mathbf{with}\ ((x : \mathsf{int}), (y : \mathsf{int})) \rightarrow (+)\ x\ ((+)\ y\ \mathrm{M}_M.\mathrm{z})$.
4. The *compiled* language is generated by compilation, which here computes global type names for hashed abstract types, carries out $withspec$ and $likespec$ checks, etc. The operational semantics is defined over elements of the compiled language.

   Note that the compiled language contains both compiled form and source internal form components. Specifically, a compiled program consists of compiled form definitions and/or source internal form $\mathbf{module\ fresh}$ definitions, and an optional compiled form expression.

The main definition is of the union of the grammars for the *source internal* and *compiled* languages. The differences are signposted with "*Source internal form:*" (or S) and "*Compiled form:*" (or C) respectively. The main type system is defined over this union.

Sugared forms (the *sugared source internal* additions to the *source internal* language) are signposted "*Sugared source internal language:*" (or G). Additional rules specify typing for the sugared forms.

Differences between the *sugared source internal* and the concrete syntax of the *concrete source* language are signposted "*Concrete source language:*".

For any syntactic entity θ, we say $\mathrm{sugaredsourceinternalform}(\theta)$, $\mathrm{sourceinternalform}(\theta)$ or $\mathrm{compiledform}(\theta)$ to mean that entity is an element of the respective language.

Some syntactic requirements are not easily expressed in the BNF grammar itself. They are are instead placed in the body of the text in paragraphs signposted with "*Syntactic requirement:*".

Concrete source language: This definition does not fix character sets, comments, whitespace etc. The implementation generally follows OCaml.

> *Comment:* The syntax generally follows OCaml for standard features. We have tried to resist any temptation to change or improve it, for three reasons: (1) to avoid time-consuming and unproductive syntactic debate; (2) to enable automated testing of the implementation of those standard features against OCaml's behaviour; and (3) so that we and others can write Acute code without needing to learn new syntactic conventions. There are quite a number of things that should in principle be improved, however.

Identifiers

x	expression identifier (external)
x	expression identifier (internal)
t	type identifier (external)
t	type identifier (internal)
M	module identifier (external)
M	module identifier (internal)

Concrete source language: Expression and type identifiers are not split into internal and external forms; they are uncapitalized. External module identifiers are capitalized; they can have an optional internal identifier, also capitalized.

We use $x, \mathrm{x}, t, \mathrm{t}$ etc. both as metavariables (in most of this document) and as elements of their respective syntactic categories (in examples). Hence x_x (qua metavariables) ranges over x_x, x_y, y_x, etc (qua elements) – just because the two metavariables look similar does not mean that any concrete instance must be a pair related by the obvious isomorphism.

In ASCII when we need to write M_M, x_x, and t_t they are rendered, respectively, `MM[M]`, `xx[x]`, and `tt[t]`.

Kinds

K	$::=$	TYPE	kind of all types
		EQ(T)	kind of types equal to T

Types

T	$::=$	TC_0		TC_0	$::=$	int		TC_1	$::=$	list
		$T\ \mathrm{TC}_1$				bool				option
		$T_1 * .. * T_n$	$n \geq 2$			string				ref
		$T_1 + .. + T_n$	$n \geq 2$			unit				name
		n				char				tie
		$T \to T'$				void				
		$\mathrm{M}_M.\mathrm{t}$				exn				
		$h.\mathrm{t}$	C			thread				
		t				mutex				
		$\forall\ t.T$				cvar				
		$\exists\ t.T$				thunkifymode				
						thunkkey				
						thunklet	C			
						unixerrorcode				

Here $\mathrm{M}_M.\mathrm{t}$ is a type field t from module M_M, and t (used within a structure or signature) is a type defined in a previous field.

Source internal form: $h.\mathrm{t}$ is a global type name built from a module name; it is not permitted in source programs.

Compiled form: $\mathrm{M}_M.\mathrm{t}$ is not permitted in compiled form.

Type Environments

E	$::=$	empty	empty type environment
		$E, x : T$	
		$E, l : T$ ref	
		$E, \mathrm{M}_M : \mathit{Sig}$	
		$E, t : K$	

We write E, E' for the concatenation of two type environments, thereby asserting also that E and E' have disjoint domains. The domain $\mathrm{dom}(E)$ of an E is a set of internal value identifiers, locations, module external/internal identifier pairs, and internal type identifiers.

Names

Take an infinite set $\mathcal{N}$ of abstract names, ranged over by n. These are used to represent runtime and compile-time freshly-generated names.

We introduce a global type environment E_n associating abstract names with types, kind TYPE, or module/import data. Note that these can occur inside "closed" types, hashes etc.

E_n	$::=$	empty
		$E_\mathrm{n}, \mathrm{n} : \mathbf{nmodule}_{\mathit{eqs}}\ \mathrm{M} : \mathit{Sig}_0\ \ \mathbf{version}\ \ \mathit{vne} = \mathit{Str}$
		$E_\mathrm{n}, \mathrm{n} : \mathbf{nimport}\ \mathrm{M} : \mathit{Sig}_0\ \ \mathbf{version}\ \ \mathit{vc}\ \ \mathbf{like}\ \ \mathit{Str}$
		$E_\mathrm{n}, \mathrm{n} : \mathrm{TYPE}$
		$E_\mathrm{n}, \mathrm{n} : T$ name

Comment: In the absence of first-class existentials freshly generated type names would not be required, as ML abstract types are a module-level feature.

Comment: We often write just E to stand for the pair E_n, E of a name environment and a type environment. In this case, $\mathrm{namepart}(E)$ denotes the name environment component of this pair..

We let $\mathbf{nn}$ range over term names (hash- or fresh-generated) and $\mathbf{n}$ over their bracket-closure:

$\mathbf{nn}$	$::=$	$\mathbf{hash}(h.\mathrm{x})_T$
		$\mathbf{hash}(T', \underline{s})_T$
		$\mathbf{hash}(T', \underline{s}, \mathbf{nn})_T$
		n_T
$\mathbf{n}$	$::=$	$\mathbf{nn}$
		$[\mathbf{n}]^{T}_{eqs}$

In building these and other hashes we hash the abstract syntax up to alpha equivalence.

These are subgrammars of the e grammar; the e typing judgements apply.

Define the auxiliary $\mathrm{typeof}(\mathbf{n})$ to give the type subscript of the inner $\mathbf{nn}$.

We suppose there is a fixed total order $\leq$ over the $\mathbf{n}$, taken (in the implementation) to depend on the hash / n only, ignoring the T subscripts).

Comment: Later we will also add name-indexed hashtables, which should respect the order.

We let h range over module names (hash- or fresh-generated).

h	$::=$	$\mathbf{hash}(\mathbf{hmodule}_{eqs}\ \mathrm{M} : \mathit{Sig}_0\ \ \mathbf{version}\ \ \mathit{vne} = \mathit{Str})$
		$\mathbf{hash}(\mathbf{himport}\ \mathrm{M}\ : \mathit{Sig}_0\ \ \mathbf{version}\ \ \mathit{vc}\ \ \mathbf{like}\ \ \mathit{Str})$
		n

Comment: Note that only the external identifier M of a definition is included in the hash; the internal identifier is not. We will only ever deal with hashes in which eqs, Sig_0 and Str have been module-identifier-closed by substituting for $\mathrm{M}'_{M'}.\mathrm{t}$ and $\mathrm{M}'_{M'}.\mathrm{x}$. Furthermore, any internal module field references to type abbreviations will have been normalised away.

The version in a module hash is a version number expression, to permit it to include $\mathbf{myname}$. This avoids the need for a recursive hash construction. In any context, $\mathbf{myname}$ may simply be interpreted as the hash in which it occurs. In contrast, the version in an import hash is not a version constraint expression but a version constraint. This is to force the evaluation of any M_M references, replacing them by the hash of the module named. Otherwise, the meaning of M_M would be (undesirably) context-dependent.

For the term part, we substitute $h.\mathrm{x}$ for $\mathrm{M}_M.\mathrm{x}$ where h is the hash of whatever is bound to M_M. That gives a slightly more discriminating type equivalence than the ICFP calculus (which substituted code, not hash) for types that depend on the code containing that $h.\mathrm{x}$, but it seems more intuitive, and is cheaper to implement. For the type part:

- where $\mathrm{M}_M.\mathrm{t}$ is abstract (of kind TYPE in the source definition) we substitute in $h.\mathrm{t}$.
- where $\mathrm{M}_M.\mathrm{t}$ is concrete we must substitute in the type representation, otherwise we won't have enough type equalities later.

We assume a fixed function $\mathrm{HASH}(\cdot)$ which takes a structured hash h to a numeric hash $\underline{N}$, where $\underline{N} \in \mathbb{H}$ for some set $\mathbb{H}$. Typically $\mathrm{HASH}(\cdot)$ would be a well-known hash function such as MD5 or SHA1, and numeric hashes would just be long bit strings (128- or 160-bit, respectively).

With numeric hashes, runtime type safety for the language is only probabilistically guaranteed (though with rather high probability for reasonable usage); it depends on the assumption that $\mathrm{HASH}(\cdot)$ is injective for the set of structured hashes in use.

The language is not intended to protect against the malicious forging of ill-formed hashes or marshalled values.

Implementation: In the implementation both n and all $\mathbf{hash}(...)$ forms will be represented by a long bitstring taken from $\mathbb{H}$. (So $\mathbf{hash}(h.\mathrm{x})$ is represented by the hash of the pair of h and the external name x, not the pair of h and x.)

In the implementation, the representations of abstract names n will be generated randomly. More specifically: we do not want to require that the implementation generates each individual name randomly, as that might be too costly — it is acceptable to generate a random start point at the initialisation of each compilation and the initialisation of each runtime instance, and thereafter use some fast generation function for compile-time new and run-time new respectively. (Ideally the generation function would not be successor, to avoid triggering worst-case performance of naïve finite map implementations.) Nonetheless, a low-level attacker would often be able to tell whether two names originated from the same point, and that (for making real nonces etc) a more aggressively random $\mathbf{fresh}$ would be required.

Name representations could be generated lazily: as earlier discussed for FreshOCaml marshalling, we only really need an element of $\mathbb{H}$ when a name is first marshalled; the implementation could keep a finite map associating internal-to-this-runtime names (represented just with pointers) and elements of $\mathbb{H}$ that have been marshalled or have been unmarshalled from the outside. Whether we would gain very much by this is unclear, and we do not do it now. (However, it is important to make local channel use very cheap).

Implementation: In a production implementation, all occurrences of h would be implemented by occurences of $\underline{N}$, with $\mathrm{HASH}(\cdot)$ used where necessary to compute an $\underline{N}$ from a $\mathbf{hash}(..)$ form of h.

In our current implementation we support both numeric hashes and structured hashes, the latter preserving all the structure above. A compiler option selects which are generated. This enables us (when using structured hashes) to typecheck the reachable intermediate states. Four points in the semantics describe a typecheck that can be performed if structured hashes are being used: in compilation when a compiled unit is imported; at runtime when a marshalled value is unmarshalled; at runtime during module field instantiation, when compiled definitions are taken from a URI; and at runtime after reduction steps (for the small-step evaluator, after every reduction step; for the big-step evaluator, only some of the intermediate points are reached). All these checks should always succeed, assuming that marshalled values and compiled files are not forged.

Our implementation takes a $\mathrm{HASH}(\cdot)$ function that calculates the MD5 of a canonical pretty-print of structured hashes.

Hash equations

eq	$::=$	$h.\mathrm{t} \approx T$	
		$\mathrm{M}_M.\mathrm{t} \approx T$	S
eqs	$::=$	$\varnothing$	
		eq	
		eqs, eqs	

The eqs grammar is treated up to associativity, commutativity, idempotence, and identity.

The domain $\mathrm{dom}(eqs)$ of an equation set is the set of types ($h.\mathrm{t}$ or $\mathrm{M}_M.\mathrm{t}$ on the left-hand sides of equations in the set).

Comment: We believe that in fact, any equation set will consist either entirely of h-equations, or entirely of M_M-equations; the two will never be mixed. This is because M_M-equations appear in source form, and h-equations in compiled form. However, we do not (yet) model this in the abstract syntax, because we suspect carrying this through the type system would be painful. We should revisit this decision once the type system is stable, as there might be a clarity gain.

We occasionally use the metavariable X to stand for either h or M_M:

X	$::=$	M_M	
		h	C

Compiled form: The M_M case of X is not permitted in compiled form.

Constructors The constructors are:

$()$: unit	
$\underline{i}$: int	
$\underline{b}$: bool	
$\underline{c}$: char	
$\underline{s}$: string	
$[\,]_T$: T list	
NONE$_T$: T option	
SOME : $T \rightarrow T$ option	
TIECON : T name $*$ $T \rightarrow T$ tie	C
INJ$_i^{(T_1+..+T_n)}$: $T_i \rightarrow T_1 + .. + T_n$ $\quad n \geq 2 \wedge i \in 1..n$	
INTERRUPTING : thunkifymode	
BLOCKING : thunkifymode	
THREAD : thread name $*$ thunkifymode $\rightarrow$ thunkkey	
MUTEX : mutex name $\rightarrow$ thunkkey	
CVAR : cvar name $\rightarrow$ thunkkey	
THUNKED_THREAD : thread name $*$ (unit $\rightarrow$ unit) $\rightarrow$ thunklet	C
THUNKED_MUTEX : mutex name $*$ bool $\rightarrow$ thunklet	C
THUNKED_CVAR : cvar name $\rightarrow$ thunklet	C
RESOLVE_FAILURE : exn	
MATCH_FAILURE : string $*$ int $*$ int $\rightarrow$ exn	
LIBRARY_ERROR : string $\rightarrow$ exn	
MARSHAL_FAILURE : exn	
UNMARSHAL_FAILURE : string $\rightarrow$ exn	
FAILURE : string $\rightarrow$ exn	
INVALID_ARGUMENT : string $\rightarrow$ exn	
NOT_FOUND : exn	
SYS_ERROR : string $\rightarrow$ exn	
END_OF_FILE : exn	
DIVISION_BY_ZERO : exn	
SYS_BLOCKED_IO : exn	
NONEXISTENT_THREAD : exn	
NONEXISTENT_MUTEX : exn	
NONEXISTENT_CVAR : exn	
MUTEX_EPERM : exn	
EXISTENT_NAME : exn	
THUNKIFY_EINTR : exn	
THUNKIFY_SELF : exn	
THUNKIFY_KEYLISTS_MISMATCH : exn	
THUNKIFY_THREAD_IN_DEFINITION : exn	
UNIXERROR : unixerrorcode $*$ string $*$ string $\rightarrow$ exn	

The unix error codes, all constructors of type unixerrorcode, are:

E2BIG
EACCES
EADDRINUSE
EADDRNOTAVAIL
EAFNOSUPPORT
EAGAIN
EWOULDBLOCK
EALREADY
EBADF

EBADMSG
EBUSY
ECANCELED
ECHILD
ECONNABORTED
ECONNREFUSED
ECONNRESET
EDEADLK
EDESTADDRREQ
EDOM
EDQUOT
EEXIST
EFAULT
EFBIG
EHOSTUNREACH
EIDRM
EILSEQ
EINPROGRESS
EINTR
EINVAL
EIO
EISCONN
EISDIR
ELOOP
EMFILE
EMLINK
EMSGSIZE
EMULTIHOP
ENAMETOOLONG
ENETDOWN
ENETRESET
ENETUNREACH
NFILE
ENOBUFS
ENODATA
ENODEV
ENOENT
ENOEXEC
ENOLCK
ENOLINK
ENOMEM
ENOMSG
ENOPROTOOPT
ENOSPC
ENOSR
ENOSTR
ENOSYS

ENOTCONN
ENOTDIR
ENOTEMPTY
ENOTSOCK
ENOTSUP
ENOTTY
ENXIO
EOPNOTSUPP
EOVERFLOW
EPERM
EPIPE
EPROTO
EPROTONOSUPPORT
EPROTOTYPE
ERANGE
EROFS
ESPIPE
ESRCH
ESTALE
ETIME
ETIMEDOUT
ETXTBSY
EXDEV
ESHUTDOWN
EHOSTDOWN
EUNKNOWN_UNIX_ERROR

Here $\underline{i}$ ranges over integer literals (the same as the underlying FreshOCaml ints), $\underline{s}$ ranges over strings of characters, and $\underline{b}$ ranges over $\{\mathbf{true}, \mathbf{false}\}$.

In addition to $\underline{s}$, we let MK also range over string constants.

Note that constructors are all of arity 0 (C_0), arity 1 (C_1), or equal to :: or $(\_, .., \_)$. The typing and reduction rules treat the C_0 and C_1 cases uniformly and have special rules for the others.

Concrete source language: The type annotation subscripts are optional. If they are included (both here and in later forms), the linear ASCII rendering is e.g. `None %[T]`.

The string $*$ int $*$ int in the v' for the MATCH_FAILURE case gives the position in the source file of the match code.

Many of the exception constructors are raised by embedded OCaml library functions, as follows:

- INVALID_ARGUMENT is raised by library functions to signal that the given arguments do not make sense.
- FAILURE is raised by library functions to signal that they are undefined on the given arguments.
- NOT_FOUND is raised by search functions when the desired object could not be found.
- SYS_ERROR is raised by the input/output functions to report an operating system error.
- END_OF_FILE is raised by input functions to signal that the end of file has been reached.
- DIVISION_BY_ZERO is raised by division and remainder operations when their second argument is null.
- SYS_BLOCKED_IO is a special case of SYS_ERROR raised when no I/O is possible on a non-blocking I/O channel.
- UNIXERROR carries the errors raised by the TCP libraries.
- LIBRARY_ERROR carries any unrecognised error raised by an E_{const}.

  Comment: The Unix error codes above are the set of all those on our current Linux install, and the translation from integers to constructors is hard-wired into the Acute implementation (in `library.mlp`). This should be made more portable — at the least, that part of `library.mlp` should be automatically generated from the C header file.

Comment: Note that the polymorphic constructors exist as indexed families rather than using explicit polymorphism. This is a historical artifact.

Comment: We are not entirely consistent about the type annotations on constructors, operators, and expression forms. Acute was originally monomorphic, though with type inference for these annotations; the semantics was originally written to ensure that all values have unique types. That is no longer the case: the **raise** is *not* type-annotated (as to maintain that annotation during reduction would require notationally-heavy annotation of evaluation contexts and the other expression forms), so function values with a **raise** in the body may not be uniquely typable.

Standard Library

We suppose there is a fixed collection of special constants E_{const}, which is a finite partial map from internal value identifiers to types. Each is equipped with a natural-number arity, written x^n if x has arity n. The special constants are partitioned by a predicate $\text{os}(x^n)$ into the OS calls, which have labelled transitions in the semantics, and the internal built-in library calls, which have delta rules. The OS calls are further partitioned by a predicate $\text{fast}(x^n)$ specifying whether each is a fast or slow call.

Their internal identifiers are never shadowed, as specified below when we discuss binding.

The types of E_{const}s must be first order.

Comment: Before the addition of concurrency we permitted higher-order E_{const}s, e.g. to automatically embed the FreshOCaml `List.map` into Acute, but with concurrency that would be unduly complex.

Suppose further that there is a fixed list of library definitions $\mathit{definitions}_{\text{lib}}$, a finite list of module definitions. These have a special status in that their code can mention special constants from $\text{dom}(E_{\text{const}})$ whereas user-defined modules cannot (the running expression can also mention them, of course).

Note that the internal identifiers of $\mathit{definitions}_{\text{lib}}$ are fixed globally.

We generate names for these modules in the usual way when they are compiled (note there will be free internal identifiers inside, but that is not a problem).

Let E_{lib} be the partial map from module external/internal identifier pairs to signatures such that $E_{\text{const}} \vdash \mathit{definitions}_{\text{lib}} \rhd E_{\text{lib}}$.

The upshot of this is that all types defined in a $\mathit{definitions}_{\text{lib}}$ module must have representation types that are expressible within the language, but the code can make use of E_{const}. We do not require that $\mathit{definitions}_{\text{lib}}$ terms are pure E_{const}s. Programs can rebind to user-land replacements for $\mathit{definitions}_{\text{lib}}$ modules if needed, and can use them in *withspec* and *likespec*s.

Implementation: In the implementation $\mathit{definitions}_{\text{lib}}$ is composed of two parts. The first (`definitions_lib_auto.ac`) is automatically generated from a collection of OCaml interface files; each value component in these gives rise to an E_{const}. These interfaces are described in Section 21. Most are simple fragments of OCaml standard library interfaces, and are linked to those; some are linked to hand-written OCaml modules. Type embeddings and projections are dealt with automatically. The second part consists of various hand-written Acute modules. The two are combined into `definitions_lib.ac` as below.

```
includesource "definitions_lib_auto.ac"
(* includesource "io_template.ac"                (* simple IO for tcp              *) *)
includesource "io_persist.ac"           (* simple IO for persistent store *)
```

E_{const}s of arity 0 are now supported by the automated generation tool but they give non-value expressions in the `definitions_lib.ac` structures rather than the actual values, so we use the `hash!` mode for these modules.

Comment: Note that the semantics has immutable strings, whereas OCaml has mutable strings. Our string library contains only the non-mutating part of the OCaml string library.

Operators Take *operators* op^n

$\mathbf{ref}_T$	:	$T \to T$ ref	
$(=_T)$	:	$T \to T \to$ bool	
$(<), (\leq), (>), (\geq)$	:	int $\to$ int $\to$ bool	
$(+), (-), (*), (/), (\mathbf{mod})$	:	int $\to$ int $\to$ int	
$(\mathbf{land}), (\mathbf{lor}), (\mathbf{lxor})$	:	int $\to$ int $\to$ int	
$(\mathbf{lsl}), (\mathbf{lsr}), (\mathbf{asr})$	:	int $\to$ int $\to$ int	
$-$	:	int $\to$ int	
$(@_T)$	:	T list $\to T$ list $\to T$ list	
$(\hat{})$	:	string $\to$ string $\to$ string	
$\mathbf{compare\_name}_T$	:	T name $\to T$ name $\to$ int	
$\mathbf{create\_thread}_T$	:	thread name $\to (T \to$ unit$) \to T \to$ unit	
self	:	unit $\to$ thread name	
kill	:	thread name $\to$ unit	
create_mutex	:	mutex name $\to$ unit	
lock	:	mutex name $\to$ unit	
try_lock	:	mutex name $\to$ bool	
unlock	:	mutex name $\to$ unit	
create_cvar	:	cvar name $\to$ unit	
wait	:	cvar name $\to$ mutex name $\to$ unit	
signal	:	cvar name $\to$ unit	
broadcast	:	cvar name $\to$ unit	
$\mathbf{thunkify}^1$	:	thunkkey list $\to$ (thunkkey list $\to$ unit)	
unthunkify	:	thunklet list $\to$ thunkkey list $\to$ unit	C
$\mathbf{exit}\ _T$	:	int $\to T$	

The superscript is the arity of the operator. Note in particular that **thunkify** has arity 1, not 2.

Concrete source language: The binary operators in brackets may be written infix, e.g. $e =_T e'$ for $(=_T)\ e\ e'$; we use Ocaml's precedence rules. If $\mathbf{ref}_T$ is not saturated, then it must be enclosed in parentheses in source forms. Same for **mod**, **land**, **lor**, **lxor**, **lsl**, **lsr**, **asr**. The type subscripts can be omitted, as above.

> *Comment:* With locally-unique naming, there is no point in parameterising the **create_thread** function argument on its identity.

> *Comment:* The type of **compare_name** follows the type of **compare** in OCaml.

The operators come in two families: the *type indexed*, consisting of those bearing a type subscript, and the unindexed. We write op^n for both.

> *Comment:* The definition does not at present follow a consistent policy as to what should appear as an operator and what as an expression form (cf. the treatment of coloured arguments by the atomic evaluation contexts). Ultimately it should. The distinction between operators and E_{const}s comes from the implementation: the former are implemented within the Acute runtime; the latter by calling out to FreshOCaml.

Expressions

e	$::=$	C_0	C_0 a constructor of arity 0
		$\mathrm{C}_1\ e$	C_1 a constructor of arity 1
		$e_1 :: e_2$	Cons
		$(e_1, .., e_n)$	Tuple $(n \geq 2)$
		$\mathbf{function}\ (x : T) \to e$	Function
		$op^n\ e_1\ ...\ e_n$	*op* an operator
		$x^n\ e_1\ ...\ e_n$	x^n an external constant
		x	Identifier

$\mathrm{M}_M.\mathrm{x}$		Module projection
$h.\mathrm{x}$	*	Module hash projection
if e_1 **then** e_2 **else** e_3		Conditional
while e_1 **do** e_2 **done**		Loop
e_1 && e_2		Boolean short-circuit and
e_1 || e_2		Boolean short-circuit or
e_1 **;** e_2		Sequence
$e_1\ e_2$		Application
$!_T\, e$		Deref
e_1 **:=**$_T$ e_2		Assign
e_1 **:=**$'_T$ e_2	C	Assign uncoloured
l	C	Location
match e **with** $mtch$		Pattern match
let rec $x_1 : T =$ **function** $(x_2 : T') \to e_1$ **in** e_2		Recursive definition
raise e		Raise exception
try e **with** $mtch$		Handle exception(s)
marshal $e_1\ e_2 : T$		Marshal
marshalz $\underline{s}\ e : T$	C	Marshal (expression in uncoloured context)
unmarshal e **as** T		Unmarshal
fresh$_T$		run-time fresh name generation
cfresh$_T$	S	compile-time fresh name generation
hash$(X.\mathrm{x})_{T'}$		create name from module value field
hash$(T, e_1)_{T'}$		create name from type and string
hash$(T, e_1, e_2)_{T'}$		create name from type, string, and name
n_T	C	abstract name
swap e_1 **and** e_2 **in** e_3		polytypic swap
e_1 **freshfor** e_2		polytypic freshness test
support$_T\, e$		polytypic typed-name support
$\mathrm{M}_M@\mathrm{x}$	S	tie construction
name_of_tie e		tie inspection
val_of_tie e		tie inspection
$\Lambda\ t \to e$		type abstraction
$e\ T$		type application
$\{T, e\}$ **as** T'		existential package
let $\{t, x\} = e_1$ **in** e_2		unpackaging
namecase e_1 **with** $\{t,(x_1,x_2)\}$ **when** $x_1 = e \to e_2$ **otherwise** $\to e_3$		unpackaging and name equality
function $mtch$	G	$(mtch \neq (x' : T' \to e))$
fun $p_1..p_n \to e'$	G	$(n \geq 1)$
let $p = e'$ **in** e''	G	
let $x : T\ p_1..p_n = e'$ **in** e''	G	$(n \geq 1)$
let rec $x : T =$ **function** $mtch$ **in** e	G	$(mtch \neq (x' : T' \to e'))$
let rec $x : T\ p_1..p_n = e'$ **in** e''	G	$(n \geq 1)$
$e_1 \|\|\| e_2$	G	spawn e_1
op$(op^n)^n\ e_1\ ..\ e_n$	C	Primitive application of an operator
op$(x^n)^n\ e_1\ ..\ e_n$	C	Primitive application of an external constant

$[e]^{T}_{eqs}$	C	Coloured brackets
$\mathbf{resolve}(\mathrm{M}_M.\mathrm{x}, \mathrm{M}'_{M'}, \textit{resolvespec})$	C	*resolvespec* in progress
$\mathbf{resolve\_blocked}(\mathrm{M}_M.\mathrm{x}, \mathrm{M}'_{M'}, \textit{resolvespec})$	C	*resolvespec* blocked waiting for data
$\mathbf{RET}_T$	C	Await return from a 'fast' OS routine
$\mathbf{SLOWRET}_T$	C	Await return from a 'slow' OS routine

We write x^n to denote an $x \in E_{\mathrm{const}}$ that has arity n.

Sometimes we write $\mathbf{op}(e)^n$ ... and in this case the e ranges over op^n and x^n only.

Concrete source language: The type annotations in $!_T\ e$, $e_1 \mathrel{\mathbf{:=}_T} e_2$, $\mathbf{function}\ (x : T) \rightarrow e$ and $\mathbf{let\ \ rec}\ \ x_1 : T = \mathbf{function}\ (x_2 : T') \rightarrow e_1\ \ \mathbf{in}\ \ e_2$ are all optional. In ASCII a type abstraction $\Lambda\ t \rightarrow e$ is written as `Function t -> e` and a type application $e\ T$ is written `e %[T]`.

Sugared source internal language: The $: T$ type annotation in $\mathbf{let}\ \ x : T\ p_1..p_n$ and $\mathbf{let\ \ rec}\ \ x : T\ p_1..p_n$ is prohibited (to be compatible with Ocaml). These type annotations are inserted by type inference and used in the desugaring process.

Sugared source internal language: The $\mathbf{hash}(\mathrm{M}_M.\mathrm{x})_{T'}$ and $\mathrm{M_M@x}$ forms are only permitted within structures, not in the main expression. (This is not essential, but simplifies the hashify semantics.)

Compiled form: Module hash projections $h.\mathrm{x}$ may only occur within other hashes (they are not executable).

Sugared source internal language: In source programs, $!_T$, $\mathbf{:=}_T$, $||$, and $\&\&$ may all be written as prefix functions by wrapping them in parentheses, e.g. $(\mathbf{:=}_T)$. In source programs, operators, external constants, $!_T$, $\mathbf{:=}_T$, $||$, and $\&\&$ can be partially applied. The desugaring process is responsible for eta-expanding these. Type annotations in these sugared forms are likewise optional.

See Section 16.4 for details of the desugarings.

> *Comment:* It is an invariant that constructible values v^{eqs} satisfy $\mathrm{compiledform}(v^{eqs})$ and in addition contain none of $\mathbf{RET}_T$, $\mathbf{SLOWRET}_T$, $\mathbf{resolve}(...)$, or $\mathbf{resolve\_blocked}(...)$ Values can contain the forms l, $[e]^{T}_{eqs}$, n_T, and also the (transient) $e_1 \mathrel{\mathbf{:=}'_T} e_2$, $\mathbf{marshalz}\ \underline{s}\ e\ : T$, $\mathbf{op}(e)^n\ e_1\ ..\ e_n$. (See the semantics for $\mathbf{thunkify}$, which cannot create a thunk containing the first group.) Likewise, marshalled and stored values contain none of the first group.

Marshalled values

mv	::=	$\mathbf{marshalled}(E_{\mathrm{n}}, E_s, s, \textit{definitions}, e, T)$	Marshalled value

We suppose a fixed partial function $\mathrm{raw\_unmarshal}$ from strings to marshalled values that includes all marshalled values in its range.

Syntactic requirement: The components θ of a marshalled value all satisfy $\mathrm{compiledform}(\theta)$.

> *Comment:* Here e is the core value being shipped, T its type, s a store, E_s a store typing, *definitions* is a sequence of module definitions, and E_{n} is a name environment.
>
> The E_{n} and E_s would not be shipped in an production implementation, but are needed to state type preservation and for runtime typechecking of reachable states. They are shipped in our implementation only if literal hashes are not being used.
>
> As with the other syntactic objects, marshalled values are taken up to alpha equivalence. Here: the name environment E_{n} binds in everything to the right and internally contains no cycles; the store environment E_s binds in everything to the right and may contain internal cycles; the store s and the *definitions* bind to the right and may mutually refer to each other; the s may contain internal cycles.
>
> *Implementation:* The implementation of marshalled values should include a global type name for the Acute implementation representation type. As we are not bootstrapping, we should do this manually.

Matches

$mtch$	$::=$	$p \to e$
		$p \to e \mid mtch$

Concrete source language: An initial bar may be added.

Patterns

p	$::=$	$(\_ : T)$	Wildcard
		$(x : T)$	Identifier
		C_0	C_0 a constructor of arity 0
		$\mathrm{C}_1\ p$	C_1 a constructor of arity 1
		$p_1 :: p_2$	Cons
		$(p_1, .., p_n)$	Tuple $(n \geq 2)$
		$(p : T)$	Typed pattern

Syntactic requirement: These are subject to the condition that all identifiers occuring in a pattern are distinct.

Concrete source language: the type annotations on wildcard and identifier patterns can be omitted.

Signatures

sig	$::=$	empty	Empty signature body
		$\textbf{val}\ \mathrm{x}_x : T\ sig$	Signature body extended with val spec
		$\textbf{type}\ \mathrm{t}_t : K\ sig$	Signature body extended with type spec
Sig	$::=$	$\textbf{sig}\ sig\ \textbf{end}$	Signature

Concrete source language: We write $t : \mathrm{TYPE}$ as t, write $t : \mathrm{EQ}(T)$ as $t = T$, and allow optional **;;** between each non-empty spec in a sig. We write a single identifier in place of x_x and t_t.

Structures

str	$::=$	empty		Empty structure body
		$\textbf{type}\ \mathrm{t}_t = T\ str$		Structure body extended with type component
		$\textbf{let}\ \mathrm{x}_x = e\ str$		Structure body extended with expression component
		$\textbf{let}\ \mathrm{x}_x : T\ p_1..p_n = e'$	G	$(n \geq 1)$
Str	$::=$	$\textbf{struct}\ str\ \textbf{end}$		Structure

Concrete source language: We allow optional **;;** between each non-empty spec in a str. We write a single identifier in place of x_x and t_t. To match Ocaml the $: T$ is prohibited (but inserted by the type inference system).

Resolve specs

$atomicresolvespec$	$::=$		atomic resolve spec
		STATIC_LINK	code should be statically linked
		HERE_ALREADY	code should be here already, fail if not
		URI	load module from file or web
$resolvespec$	$::=$		resolve spec (nonempty list of atomic ones)
		$atomicresolvespec$	
		$atomicresolvespec, resolvespec$	
URI	$::=$		a string literal of a URI...
			(a subgrammar of RFC2396's `absoluteURI`)

Implementation: The current implementation supports `file`, `http`, and `ftp` URIs.

Version languages We define version number and constraint *expressions* as follows.

$avne$	$::=$			Atomic version number expression
		$\underline{n}$		natural number literal in $\mathbb{N}_{2^{31}}$
		$\underline{N}$		numeric name literal in $\mathbb{H}$
		h	C	structured name literal
		$\mathbf{myname}$		the compiler will write the name of this module in as a literal
vne	$::=$			Version number expression
		$avne$		atomic version
		$avne.vne$		dotted version
$ahvce$	$::=$			Atomic hash version constraint expression
		$\underline{N}$		numeric name literal in $\mathbb{H}$
		h	C	structured name literal
		M_M		the compiler will write the hash of M_M in as a literal
$avce$	$::=$			Atomic version constraint expression
		$ahvce$		atomic name version constraint expression
		$\underline{n}$		natural number literal in $\mathbb{N}_{2^{31}}$
$dvce$	$::=$			Dotted version constraint
		$avce$		atomic constraint
		$\underline{n}$–$\underline{n}'$		closed interval
		–$\underline{n}$		left-open interval
		$\underline{n}$–		right-open interval
		$*$		anything
		$avce.dvce$		dotted version constraint
vce	$::=$			Version constraint
		$dvce$		dotted version constraint
		$\mathbf{name} = ahvce$		exact-name version constraint

Syntactic requirement: We define the version number and constraint *values* $avn, vn, avc, ahvc, dvc, vc$ to be the relevant subgrammars with the $\mathbf{myname}$ and M_M clauses removed.

Source internal form: A user source program may not have an exact-name constraint of the form $\mathbf{name} = \underline{N}$, or $\mathbf{name} = h$, only $\mathbf{name} = \mathrm{M}_M$, as an in-scope module is required to provide the data to construct a $likestr$.

> *Comment:* There is an important distinction between h and $\underline{N}$. In the semantics a structured name h can be supplied only by the compiler, and thus we may ensure and assume it is generated from a well-formed and well-typed module or import. A numeric name $\underline{N}$ in a version expression may be supplied by the user as an arbitrary element of $\mathbb{H}$ (e.g., `0#60139C0047463B6261112944981EBF92`), and thus (for type-safety purposes) cannot be assumed to arise from a well-formed structured name (i.e. be either the $\mathrm{HASH}(\cdot)$ of a well-formed structured hash or be an appropriate abstract name).
>
> *Comment:* Note that the semantics of an exact-name version constraint $\mathbf{name} = ahvce$ is rather different from the other vces in that it is a constraint on the name, not the version, of the modules and imports that can be linked to an import with this constraint.
>
> *Comment:* The basic part of the version grammar should be improved: the intervals are not very useful as given here.

Define an equivalence relation over avc (note that avn and avc coincide) as the least equivalence such that $h \cong \underline{N}'$ if $\mathrm{HASH}(h) = \underline{N}'$. More explicitly:

$$\begin{array}{rcl}
\underline{n} \cong \underline{n}' & \iff & \underline{n} = \underline{n}' \\
\underline{N} \cong \underline{N}' & \iff & \underline{N} = \underline{N}' \\
h \cong h' & \iff & h = h' \\
h \cong \underline{N}' & \iff & \mathrm{HASH}(h) = \underline{N}' \\
\underline{N} \cong h' & \iff & \underline{N} = \mathrm{HASH}(h')
\end{array}$$

Define the set of vn denoted by each dvc as follows.

$$\begin{array}{rcl}
[\![\underline{N}]\!] & = & \{avn | avn \cong \underline{N}\} \\
[\![h]\!] & = & \{avn | avn \cong h\} \\
[\![\underline{n}]\!] & = & \{\underline{n}\} \\
[\![\underline{n}_1\text{–}\underline{n}_2]\!] & = & \{\underline{n} | \underline{n}_1 \leq \underline{n} \leq \underline{n}_2\} \\
[\![\underline{n}_1\text{–}]\!] & = & \{\underline{n} | \underline{n}_1 \leq \underline{n}\} \\
[\![\text{–}\underline{n}_2]\!] & = & \{\underline{n} | \underline{n} \leq \underline{n}_2\} \\
[\![*]\!] & = & \{vn | \mathbf{true}\} \\
[\![avc.dvc]\!] & = & \{avn.vn | avn \cong avc \wedge vn \in [\![dvc]\!]\} \cup \{avn | avn \cong avc \wedge dvc = *\}
\end{array}$$

We write $vn \in dvc$ for $vn \in [\![dvc]\!]$.

Say $vc \subseteq vc'$ if either (1) $vc = dvc$, $vc' = dvc'$ and $[\![dvc]\!] \subseteq [\![dvc']\!]$, or (2) $vc = (\mathbf{name} = ahvc)$ and $vc' = (\mathbf{name} = ahvc')$ and $ahvc \cong ahvc'$.

Modes and Valuabilities

$mode$ ::=	
hash	hash the structure of the module or import
cfresh	calculate a fresh name at compile time
fresh	calculate a fresh name at run time
hash!	hash the structure of the module or import, ignoring valuability
cfresh!	calculate a fresh name at compile time, ignoring valuability

vub	::=	valuable	is statically determined
		cvaluable	is statically determined after compile-time new
		nonvaluable	can only be calculated at run-time

We write $vubs$ for a pair of valuabilities. The first element of the pair refers to the status of the terms, the second to the status of the types.

Definitions Source definitions and compiled definitions (the latter ranged over by *definitions*) are as follows.

$sourcedefinition$ ::=	
module $mode$ M_M : Sig **version** $vne = Str$ $withspec$	Module declaration
import $mode$ M_M : Sig **version** vce $likespec$ **by** $resolvespec = Mo$	Module import
mark MK	Mark
module M_M : $Sig = \mathrm{M}'_{M'}$	Module alias declaration

$definition$ ::=	
$\mathbf{cmodule}_{h;eqs;Sig_0}$ $vubs$ M_M : Sig_1 **version** $vn = Str$	Module declaration
$\mathbf{cimport}_{h;Sig_0}$ $vubs$ M_M : Sig_1 **version** vc **like** Str **by** $resolvespec = Mo$	Module import
module fresh M_M : Sig **version** $vne = Str$ $withspec$	... (initialisation-time fresh)
import fresh M_M : Sig **version** vce $likespec$ **by** $resolvespec = Mo$	... (initialisation-time fresh)
mark MK	Mark

In the **cmodule** the eqs are any equations arising from the **with** ! clause; the Sig_0 is the semicompiled signature, not name-selfified but otherwise normalised as far as possible, and Sig_1 is the fully compiled signature.

$weqs$	$::=$	$\varnothing$	user coercion spec
		$\mathrm{M}_M.\mathrm{t} \approx T, weqs$	
$withspec$	$::=$	**with** $!weqs$	
$likespec$	$::=$	empty	like spec
		like M_M	
		like Str	
Mo	$::=$	M_M	linked to M_M
		UNLINKED	unlinked

Syntactic requirement: Here $weqs$ is up to associativity, commutativity and identity.

Concrete source language: The **version** vne in a **module** can be omitted, in which case it defaults to **version myname**. The $withspec$ in a **module** is either empty (in which case it defaults to **with** $!\varnothing$) or **with** $!weqs$ in which case $weqs$ is not empty. The **version** vce in an **import** can be omitted, in which case it defaults to **version** $*$. The **by** $resolvespec$ in an **import** can be omitted, in which case it defaults to **by** HERE_ALREADY. The $= Mo$ in an **import** can be omitted, in which case it defaults to $=$ UNLINKED. If an **import** has an exact-name constraint **name** $= \mathrm{M}_M$ then the $likespec$ must be empty.

Compilation Units

$compilationunit$	$::=$	eo
		$sourcedefinition$ **;;** $compilationunit$
		includesource $sourcefilename$ **;;** $compilationunit$
		includecompiled $compiledfilename$ **;;** $compilationunit$
$compiledunit$	$::=$	$(E_{\mathrm{n}}, definitions\ eo)$
$definitions$	$::=$	empty
		$definition$ **;;** $definitions$
eo	$::=$	empty
		e **;;**

Concrete source language: We allow optional and repeated **;;** (different rules apply for structures and signatures).

Compiled form: All **;;** are omitted.

There must be at most one final e, which may be e.g. at the end of an include at the end of the top-level compilation unit.

In the current implementation, programs with no final expression are not executed; in particular, no module initialisation is performed for such programs. This restriction should be relaxed.

Filesystems

Say a *filesystem* Φ is a finite partial map from $sourcefilename$ to $compilationunit$s and from $compiledfilename$ to $compiledunit$s.

Conventionally, $sourcefilename$s are of the form `foo.ac` and $compiledfilename$s are of the form `foo.aco`.

Processes

$$\begin{array}{lll} P & ::= & 0 \\ & & P_1 | P_2 \\ & & \mathbf{n} : \textit{definitions}\ e \\ & & \mathbf{n} : \mathrm{MX}(\underline{b}) \\ & & \mathbf{n} : \mathrm{CV} \end{array}$$

We work up to the structural congruence on processes which is the least congruence for | containing $P|0 \equiv P$, $P_1|P_2 \equiv P_2|P_1$, and $P_1|(P_2|P_3) \equiv (P_1|P_2)|P_3$.

Write dom(P) for the set of names of entities in P, i.e. dom$(0) = \varnothing$, dom$(P_1|P_2) =$ dom$(P_1) \cup$ dom(P_2), dom$(\mathbf{n} : ...) = \{\mathbf{n}\}$.

Stores

Say a *store* s is a finite partial map from locations l to $\varnothing$-coloured values (values are defined on page 125).

Configurations (or States)

Take tuples $\langle E_s,\ s,\ \textit{definitions},\ P\rangle$ of some module $\textit{definitions}$, a store typing E_s, a store s, and a process P. The store typing is not needed in an implementation. Note that (as we have module initialisation) the E_s, s scope in s, P and $\textit{definitions}$, and the $\textit{definitions}$ scope in s and P.

16.2.1 Binding

Syntactic requirement: We work up to alpha equivalence throughout.

Syntactic requirement: The external constants $x^n \in$ dom(E_{const}) may not appear in a binding position.

Syntactic requirement: For expression and type identifiers we have internal identifiers x and t as normal binders rather than the external/internal pair a binder (as in [BHS$^+$03]). For module identifiers we have the M_M pairs be binders.

We write fv$(...)$ for the set of free identifiers x, t, and M_M in ....

Syntactic requirement: In expression $\mathbf{function}\ (x : T) \to e$ the x binds in e. In expression $\mathbf{let\ \ rec}\ \ x_1 : T = \mathbf{function}\ (x_2 : T') \to e_1\ \ \mathbf{in}\ \ e_2$ the x_1 binds in e_1 and e_2, and the x_2 binds in e_1.

Syntactic requirement: In sugar expression $\mathbf{let}\ \ p = e_1\ \ \mathbf{in}\ \ e_2$ the internal value identifiers of p bind in e_2. In sugar expression $\mathbf{let\ \ rec}\ \ x : T = \mathbf{function}\ \ \textit{mtch}\ \ \mathbf{in}\ \ e$ the x binds in $\textit{mtch}$ and in e. In sugar expression $\mathbf{fun}\ \ p_1..p_n \to e$ the internal value identifiers of $p_1..p_n$ bind in e. In sugar expression $\mathbf{let}\ \ x\ p_1..p_n = e_1\ \ \mathbf{in}\ \ e_2$ the identifier x binds in e_2, and the internal value identifiers of $p_1..p_n$ bind in e1. In sugar expression $\mathbf{let\ \ rec}\ \ x\ p_1..p_n = e_1\ \ \mathbf{in}\ \ e_2$ the identifier x binds in e_1 and e_2, and the internal value identifiers of $p_1..p_n$ bind in e1.

Syntactic requirement: In $\Lambda\ t \to e$ the t binds in the e. In $\mathbf{let}\ \{t, x\} = e_1\ \ \mathbf{in}\ \ e_2$ the t and x bind in the e_2. In $\mathbf{namecase}\ \ e_1\ \ \mathbf{with}\ \{t, (x_1, x_2)\}\ \mathbf{when}\ \ x_1 = e \to e_2\ \ \mathbf{otherwise}\ \to e_3$, type identifier t, expression identifier x_2, and the first occurrence of expression identifier x_1 all bind in e_2; the second occurrence of x_1 is bound by the first occurrence. Note that e_1, e, and e_3 all live in the outer scope (no extra bindings).

Syntactic requirement: In match $p \to e$ the internal expression identifiers of p bind in e

Syntactic requirement: In signatures, in $\mathbf{val}\ \ \mathrm{x}_x : T\ \textit{sig}$ the x binds in $\textit{sig}$ and in $\mathbf{type}\ \ \mathrm{t}_t : K\ \textit{sig}$ the t binds in $\textit{sig}$.

Syntactic requirement: In structures, in $\mathbf{let}\ \ \mathrm{x}_x = e\ \textit{str}$ the x binds in $\textit{str}$, in $\mathbf{let}\ (\mathrm{x}_x : T)p_1..p_n = e\ \textit{str}$ the internal value identifiers of $p_1..p_n$ bind in e and the x binds in $\textit{str}$; and in $\mathbf{type}\ \ \mathrm{t}_t = T\ \textit{str}$ the t binds in $\textit{str}$.

Syntactic requirement: For any occurrence of a $\mathit{sourcedefinition}$ or $\mathit{definition}$, the M_M binds in subsequent definitions. It also binds in any subsequent store typing E_s, store s and expression or process e or P, e.g. when the $\mathit{definitions}$ appear in a configuration or marshalled body.

Comment: Note that $\mathbf{mark}\ \mathrm{MK}$ does not involve any binding – marks are just strings, as marks must be shared across programs.

We've (arbitrarily) chosen not to have the store bind the locations of its domain, as would have to chose whether the E_s or the s bind, or agglomerate the two.

16.3 Typing

The typing judgements are listed in the contents pages. The typing rules are in Figures below, with particularly interesting rules flagged ★.

Most judgements are parameterised on a set eqs of type equations. These are kept as a subscript instead of as part of E so they be be easily removed when one passes through brackets – unlike binders, they are not additive.

There is no $E \vdash compilationunit\ \ \mathbf{ok}$ as we need to substitute file contents in (recursively) before typechecking, not having introduced separate interfaces.

The source language type system must be considered together with the checks performed by compilation: several checks are not carried out in the type system because they involve the representation types of abstract types, and version data, from previous modules; these are not recorded in type environments and so are not accessible in the type system. Specifically: (i) formation of the equation $E \vdash \mathrm{M}_M.\mathrm{t} \approx T\ \ \mathbf{ok}$ (used especially for the $weqs$ in the $\mathbf{module}$ rule), and (ii) link-checking of a loaded import in the $\mathbf{import}$ rule, are only weakly constrained by the source type system.

The compiled language type system checks these explicitly, and enforces additional facts, e.g. that in compiled form occurrences of $\mathrm{M}_M.\mathrm{t}$ have been hashified to the $h.\mathrm{t}$ form. Also, the $h.\mathrm{x}$ form appears only within hashes.

The dynamic semantics is only intended to make sense for configurations that typecheck in the compiled language type system.

16.3.1 Typing for Source Internal and Compiled Forms

$\boxed{E_n \vdash \textbf{ok}}$

$$\frac{n \notin \mathrm{dom}(E_n) \quad E_n \vdash_\varnothing T : \textsc{Type}}{E_n, n : T \ \ \mathsf{name} \vdash \textbf{ok}} \qquad \frac{n \notin \mathrm{dom}(E_n)}{E_n, n : \textsc{Type} \vdash \textbf{ok}} \qquad \frac{n \notin \mathrm{dom}(E_n) \quad E_n \vdash \textbf{nmodule}_{eqs}\, \mathrm{M} : \mathit{Sig}_0 \ \textbf{version} \ \mathit{vne} = \mathit{Str} \ \textbf{ok}}{E_n, n : \textbf{nmodule}_{eqs}\, \mathrm{M} : \mathit{Sig}_0 \ \textbf{version} \ \mathit{vne} = \mathit{Str} \vdash \textbf{ok}}$$

$$\frac{}{\mathrm{empty} \vdash \textbf{ok}} \qquad \frac{n \notin \mathrm{dom}(E_n) \quad E_n \vdash \textbf{nimport}\, \mathrm{M} : \mathit{Sig}_0 \ \textbf{version} \ \mathit{vc} \ \textbf{like} \ \mathit{Str} \ \textbf{ok}}{E_n, n : \textbf{nimport}\, \mathrm{M} : \mathit{Sig}_0 \ \textbf{version} \ \mathit{vc} \ \textbf{like} \ \mathit{Str} \vdash \textbf{ok}}$$

$\boxed{E_n, E \vdash \textbf{ok}}$

$$\frac{E_n \vdash \textbf{ok}}{E_n, \mathrm{empty} \vdash \textbf{ok}} \qquad \frac{x \notin \mathrm{dom}(E) \quad E_n, E \vdash_\varnothing T : \textsc{Type}}{E_n, E, x : T \vdash \textbf{ok}} \qquad \frac{l \notin \mathrm{dom}(E) \quad E_n, E \vdash_\varnothing T : \textsc{Type}}{E_n, E, l : T \ \ \mathsf{ref} \vdash \textbf{ok}}$$

$$\frac{\mathrm{M}_M \notin \mathrm{dom}(E) \quad E_n, E \vdash \mathit{Sig} \ \textbf{ok}}{E_n, E, \mathrm{M}_M : \mathit{Sig} \vdash \textbf{ok}} \qquad \frac{t \notin \mathrm{dom}(E) \quad E_n, E \vdash K \ \textbf{ok}}{E_n, E, t : K \vdash \textbf{ok}}$$

$\boxed{E_n \vdash \textbf{nmodule}_{eqs}\, \mathrm{M} : \mathit{Sig}_0 \ \textbf{version} \ \mathit{vne} = \mathit{Str} \ \textbf{ok}}$ $\boxed{E_n \vdash \textbf{nimport}\ \mathit{eqs}\, \mathrm{M} : \mathit{Sig}_0 \ \textbf{version} \ \mathit{vc} \ \textbf{like} \ \mathit{Str} \ \textbf{ok}}$

$$\frac{E_n, E_{\mathrm{const}} \vdash_{eqs} \mathit{Str} : \mathit{Sig}_0 \quad \vdash \mathit{Str} \ \textbf{flat} \quad \vdash \mathit{Sig}_0 \ \textbf{flat}}{E_n \vdash \textbf{nmodule}_{eqs}\, \mathrm{M} : \mathit{Sig}_0 \ \textbf{version} \ \mathit{vne} = \mathit{Str} \ \textbf{ok}} \qquad \frac{E_n \vdash_\varnothing \mathit{Str} : \mathrm{limitdom}\,(\mathit{Sig}_0) \quad E_n \vdash \mathit{Sig}_0 \ \textbf{ok} \quad \vdash \mathit{Str} \ \textbf{flat} \quad \vdash \mathit{Sig} \ \textbf{flat}}{E_n \vdash \textbf{nimport}\ \mathit{eqs}\, \mathrm{M} : \mathit{Sig}_0 \ \textbf{version} \ \mathit{vc} \ \textbf{like} \ \mathit{Str} \ \textbf{ok}}$$

$\boxed{E_n \vdash h \ \textbf{ok}}$

$$\frac{h = \textbf{hash}(\textbf{hmodule}_{eqs}\, \mathrm{M} : \mathit{Sig}_0 \ \textbf{version} \ \mathit{vne} = \mathit{Str}) \quad E_n, E_{\mathrm{const}} \vdash_{eqs} \mathit{Str} : \mathit{Sig}_0 \quad \vdash \mathit{Str} \ \textbf{flat} \quad \vdash \mathit{Sig}_0 \ \textbf{flat}}{E_n \vdash h \ \textbf{ok}}$$

$$\frac{h = n \quad (n : \textbf{nmodule}_{eqs}\, \mathrm{M} : \mathit{Sig}_0 \ \textbf{version} \ \mathit{vne} = \mathit{Str} \ \textbf{ok}) \in E_n \quad E_n, E_{\mathrm{const}} \vdash_{eqs} \mathit{Str} : \mathit{Sig}_0 \quad \vdash \mathit{Str} \ \textbf{flat} \quad \vdash \mathit{Sig}_0 \ \textbf{flat}}{E_n \vdash h \ \textbf{ok}} \ \bigstar$$

$$\frac{h = \textbf{hash}(\textbf{himport}\, \mathrm{M} : \mathit{Sig}_0 \ \textbf{version} \ \mathit{vc} \ \textbf{like} \ \mathit{Str}) \quad E_n \vdash_\varnothing \mathit{Str} : \mathrm{limitdom}\,(\mathit{Sig}_0) \quad E_n \vdash \mathit{Sig}_0 \ \textbf{ok} \quad \vdash \mathit{Str} \ \textbf{flat} \quad \vdash \mathit{Sig} \ \textbf{flat}}{E_n \vdash h \ \textbf{ok}}$$

$$\frac{h = n \quad (n : \textbf{nimport}\, \mathrm{M} : \mathit{Sig}_0 \ \textbf{version} \ \mathit{vc} \ \textbf{like} \ \mathit{Str}) \in E_n \quad E_n \vdash_\varnothing \mathit{Str} : \mathrm{limitdom}\,(\mathit{Sig}_0) \quad E_n \vdash \mathit{Sig}_0 \ \textbf{ok} \quad \vdash \mathit{Str} \ \textbf{flat} \quad \vdash \mathit{Sig} \ \textbf{flat}}{E_n \vdash h \ \textbf{ok}} \ \bigstar$$

Figure 1: Typing Rules – Type Environments, Hashes

$\boxed{E \vdash K\ \ \mathbf{ok}}$

$$\frac{E \vdash\ \mathbf{ok}}{E \vdash \textsc{Type}\ \ \mathbf{ok}} \qquad \frac{E \vdash_{\varnothing}\ T : \textsc{Type}}{E \vdash \mathrm{EQ}(T)\ \mathbf{ok}}$$

$\boxed{E \vdash_{eqs} K \approx K'}$

$$\frac{E \vdash\ \mathit{eqs}\ \ \mathbf{ok}}{E \vdash_{eqs} \textsc{Type} \approx \textsc{Type}} \qquad \frac{E \vdash_{eqs} T \approx T'}{E \vdash_{eqs} \mathrm{EQ}(T) \approx \mathrm{EQ}(T')}$$

$\boxed{E \vdash_{eqs} K <: K'}$

$$\frac{\begin{array}{c} E \vdash_{\varnothing}\ T : \textsc{Type} \\ E \vdash\ \mathit{eqs}\ \ \mathbf{ok} \end{array}}{E \vdash_{eqs} \mathrm{EQ}(T) <: \textsc{Type}} \qquad \frac{E \vdash_{eqs} K \approx K'}{E \vdash_{eqs} K <: K'} \qquad \text{trans is derivable}$$

Figure 2: Typing Rules – Kinds

We define a metafunction $\mathrm{limitdom}\,(\ )$ that limits a signature to its abstract type fields as follows:

$$\begin{array}{lcl}
\mathrm{limitdom}\,(\mathbf{type}\ \ \mathrm{t}_t : \textsc{Type}\ \mathit{sig}) & = & \mathbf{type}\ \ \mathrm{t}_t : \textsc{Type}\ \mathrm{limitdom}\,(\mathit{sig}) \\
\mathrm{limitdom}\,(\mathbf{type}\ \ \mathrm{t}_t : \mathrm{EQ}(T)\ \mathit{sig}) & = & \mathrm{limitdom}\,(\mathit{sig}) \\
\mathrm{limitdom}\,(\mathbf{val}\ \ \mathrm{x}_x : T\ \mathit{sig}) & = & \mathrm{limitdom}\,(\mathit{sig}) \\
\mathrm{limitdom}\,(\mathrm{empty}) & = & \mathrm{empty}
\end{array}$$

We define $\mathrm{t}\ \ \mathrm{abstract\,in}_{E_\mathrm{n}}\ \ h$ to hold if for some t' we have $(\mathbf{type}\ \ \mathrm{t}_{t'} : \textsc{Type}) \in \mathit{Sig}_0$ where either $h = \mathbf{hash}(\mathbf{hmodule}_{eqs}\ \mathrm{M} : \mathit{Sig}_0\ \ \mathbf{version}\ \ \mathit{vne} = \mathit{Str})$, $h = \mathrm{n}$ and $(\mathrm{n} : \mathbf{nmodule}_{eqs}\ \mathrm{M} : \mathit{Sig}_0\ \ \mathbf{version}\ \ \mathit{vne} = \mathit{Str}) \in E_\mathrm{n}$, $h = \mathbf{hash}(\mathbf{himport}\ \mathrm{M}\ : \mathit{Sig}_0\ \ \mathbf{version}\ \ \mathit{vc}\ \ \mathbf{like}\ \ \mathit{Str})$, or $h = \mathrm{n}$ and $(\mathrm{n} : \mathbf{nimport}\ \mathrm{M} : \mathit{Sig}_0\ \ \mathbf{version}\ \ \mathit{vc}\ \ \mathbf{like}\ \ \mathit{Str}) \in E_\mathrm{n}$.

$$\begin{array}{l}
\mathrm{selfifysig}_X(\mathbf{type}\ \ \mathrm{t}_t : \textsc{Type}\ \mathit{sig}) = (\mathbf{type}\ \ \mathrm{t}_t : \mathrm{EQ}(X.\mathrm{t}))\ \ \mathrm{selfifysig}_X(\mathit{sig}) \\
\mathrm{selfifysig}_X(\mathbf{type}\ \ \mathrm{t}_t : \mathrm{EQ}(T)\ \mathit{sig}) = (\mathbf{type}\ \ \mathrm{t}_t : \mathrm{EQ}(T))\ \ \mathrm{selfifysig}_X(\mathit{sig}) \\
\mathrm{selfifysig}_X(\mathbf{val}\ \ \mathrm{x}_x : T\ \mathit{sig}) = (\mathbf{val}\ \ \mathrm{x}_x : T)\ \ (\mathrm{selfifysig}_X(\mathit{sig})) \\
\mathrm{selfifysig}_X(\mathrm{empty}) = \mathrm{empty} \\
\mathrm{selfifysig}_X(\mathbf{sig}\ \ \mathit{sig}\ \ \mathbf{end}) = \mathbf{sig}\ \ \mathrm{selfifysig}_X(\mathit{sig})\ \mathbf{end}
\end{array}$$

Figure 3: Typing Rules – Auxiliaries

$\boxed{E \vdash_{eqs} T : K}$

$$\frac{E \vdash \mathit{eqs}\ \ \mathbf{ok}}{E \vdash_{eqs} \mathrm{TC}_0 : \textsc{Type}} \qquad \frac{E \vdash_{eqs} T : \textsc{Type}}{E \vdash_{eqs} T\ \ \mathrm{TC}_1 : \textsc{Type}} \qquad \frac{E \vdash_{eqs} T_i : \textsc{Type}\ i = 1..n, n \geq 2}{\begin{array}{c} E \vdash_{eqs} T_1 * .. * T_n : \textsc{Type} \\ E \vdash_{eqs} T_1 + .. + T_n : \textsc{Type} \end{array}} \qquad \frac{\begin{array}{c} E \vdash_{eqs} T : \textsc{Type} \\ E \vdash_{eqs} T' : \textsc{Type} \end{array}}{E \vdash_{eqs} T \rightarrow T' : \textsc{Type}}$$

$$\frac{\begin{array}{l} E, t : \textsc{Type} \vdash_{\varnothing} T : \textsc{Type} \\ E \vdash \mathit{eqs}\ \ \mathbf{ok} \end{array}}{\begin{array}{c} E \vdash_{eqs} \forall\ t.T : \textsc{Type} \\ E \vdash_{eqs} \exists\ t.T : \textsc{Type} \end{array}} \qquad \frac{(\mathrm{n} : \textsc{Type}) \in\ \mathrm{namepart}(E)}{E \vdash_{eqs} \mathrm{n} : \textsc{Type}}$$

$$\frac{\begin{array}{l} E \vdash K\ \ \mathbf{ok} \\ E \vdash_{eqs} \mathrm{M}_M : \mathit{Sig} \\ (\mathrm{t}_t : K) \in\ \mathit{Sig} \end{array}}{E \vdash_{eqs} \mathrm{M}_M.\mathrm{t} : K} \qquad \frac{\begin{array}{l} E \vdash K\ \ \mathbf{ok} \\ E \vdash_{eqs} h : \mathit{Sig} \\ (\mathrm{t}_t : K) \in\ \mathit{Sig} \\ \mathrm{t}\ \ \mathrm{abstract\ in}_{\mathrm{namepart}(E)}\ h \end{array}}{E \vdash_{eqs} h.\mathrm{t} : K} \bigstar$$

Note that M_M and h are treated similarly, here and elsewhere, except that $h.\mathrm{t}$ can only be formed if t is abstract in h.

The later uses of $\mathrm{abstract\ in}$ could be replaced by uses of type formation, but it seems clearer to be more explicit.

$$\frac{E, t : K, E' \vdash \mathit{eqs}\ \ \mathbf{ok}}{E, t : K, E' \vdash_{eqs} t : K} \qquad \frac{\begin{array}{l} E, t : \textsc{Type} \vdash_{\varnothing} T : \textsc{Type} \\ E \vdash \mathit{eqs}\ \ \mathbf{ok} \end{array}}{\begin{array}{c} E \vdash_{eqs} \forall\ t.T : \textsc{Type} \\ E \vdash_{eqs} \exists\ t.T : \textsc{Type} \end{array}}$$

$$\frac{\begin{array}{l} E \vdash_{eqs} T : K \\ E \vdash_{eqs} K <: K' \end{array}}{E \vdash_{eqs} T : K'} \qquad \frac{E \vdash_{eqs} T \approx T'}{E \vdash_{eqs} T : \mathrm{EQ}(T')}$$

$\boxed{E \vdash_{eqs} T \approx T'}$

$$\frac{E \vdash_{eqs} T : \mathrm{EQ}(T')}{E \vdash_{eqs} T \approx T'} \qquad \frac{E, t : \textsc{Type} \vdash_{eqs} T \approx T'}{\begin{array}{c} E \vdash_{eqs} \forall\ t.T \approx \forall\ t.T' \\ E \vdash_{eqs} \exists\ t.T \approx \exists\ t.T' \end{array}}$$

plus sym, trans and congruence over arrow, tuple, list, option, ref. (refl is derivable)

$$\frac{E \vdash \mathit{eq}, \mathit{eqs}\ \ \mathbf{ok}}{E \vdash_{eq,eqs}\ \ \mathit{eq}} \bigstar$$

Figure 4: Typing Rules – Types, Type Equality

$\boxed{E \vdash \mathit{eqs}\ \ \mathbf{ok}}$

$$\frac{E \vdash\ \mathbf{ok}}{E \vdash \varnothing\ \mathbf{ok}} \qquad \frac{\begin{array}{l} E \vdash \mathit{eq}\ \ \mathbf{ok} \\ E \vdash \mathit{eqs}\ \ \mathbf{ok} \\ \neg\, \exists (\mathrm{M}_M.\mathrm{t}) \in \mathrm{dom}(\mathit{eq}) \cap \mathrm{dom}(\mathit{eqs}) \end{array}}{E \vdash \mathit{eq}, \mathit{eqs}\ \ \mathbf{ok}} \bigstar \qquad \frac{\begin{array}{l} E \vdash\ \mathbf{ok} \\ E = E_1, \mathrm{M}_M : \mathit{Sig}, E_2 \\ (\mathbf{type}\ \ \mathrm{t}_t : \mathrm{T{\small YPE}}) \in\ \mathit{Sig} \\ E_1 \vdash_{\varnothing}\ T : \mathrm{T{\small YPE}} \end{array}}{E \vdash \mathrm{M}_M.\mathrm{t} \approx T\ \ \mathbf{ok}} \bigstar$$

$$\frac{\begin{array}{l} E \vdash\ \mathbf{ok} \\ \mathrm{namepart}(E) \vdash h\ \ \mathbf{ok} \\ h = \mathbf{hash}(\mathbf{hmodule}_{\mathit{eqs}}\ \mathrm{M} : \mathit{Sig}_0\ \ \mathbf{version}\ \ \mathit{vne} = \mathit{Str}) \\ \mathrm{t\ \ abstract\ in}_{\mathrm{namepart}(E)}\ \ h \\ (\mathbf{type}\ \ \mathrm{t}_t = T) \in\ \mathit{Str} \end{array}}{E \vdash h.\mathrm{t} \approx T\ \ \mathbf{ok}} \bigstar$$

$$\frac{\begin{array}{l} E \vdash\ \mathbf{ok} \\ \mathrm{namepart}(E) \vdash h\ \ \mathbf{ok} \\ h = \mathrm{n} \\ (\mathrm{n} : \mathbf{nmodule}_{\mathit{eqs}}\ \mathrm{M} : \mathit{Sig}_0\ \ \mathbf{version}\ \ \mathit{vne} = \mathit{Str}) \in\ \mathrm{namepart}(E) \\ \mathrm{t\ \ abstract\ in}_{\mathrm{namepart}(E)}\ \ h \\ (\mathbf{type}\ \ \mathrm{t}_t = T) \in\ \mathit{Str} \end{array}}{E \vdash h.\mathrm{t} \approx T\ \ \mathbf{ok}} \bigstar$$

$$\frac{\begin{array}{l} E \vdash\ \mathbf{ok} \\ \mathrm{namepart}(E) \vdash h\ \ \mathbf{ok} \\ h = \mathrm{n} \\ (\mathrm{n} : \mathbf{nimport}\ \mathrm{M} : \mathit{Sig}_0\ \ \mathbf{version}\ \ \mathit{vc}\ \ \mathbf{like}\ \ \mathit{Str}) \in\ \mathrm{namepart}(E) \\ \mathrm{t\ \ abstract\ in}_{\mathrm{namepart}(E)}\ \ h \\ (\mathbf{type}\ \ \mathrm{t}_t = T) \in\ \mathit{Str} \end{array}}{E \vdash h.\mathrm{t} \approx T\ \ \mathbf{ok}} \bigstar$$

$$\frac{\begin{array}{l} E \vdash\ \mathbf{ok} \\ \mathrm{namepart}(E) \vdash h\ \ \mathbf{ok} \\ h = \mathbf{hash}(\mathbf{himport}\ \mathrm{M}\ : \mathit{Sig}_0\ \ \mathbf{version}\ \ \mathit{vc}\ \ \mathbf{like}\ \ \mathit{Str}) \\ \mathrm{t\ \ abstract\ in}_{\mathrm{namepart}(E)}\ \ h \\ (\mathbf{type}\ \ \mathrm{t}_t = T) \in\ \mathit{Str} \end{array}}{E \vdash h.\mathrm{t} \approx T\ \ \mathbf{ok}} \bigstar$$

Figure 5: Typing Rules – Equation sets

$\boxed{E \vdash \mathit{sig}\ \ \mathbf{ok}}$

$$\frac{E \vdash\ \mathbf{ok}}{E \vdash \text{empty}\ \ \mathbf{ok}} \qquad \frac{\begin{array}{l} E, x : T \vdash \mathit{sig}\ \ \mathbf{ok} \\ \text{x} \notin \text{dom}(\mathit{sig}) \end{array}}{E \vdash \mathbf{val}\ \ \text{x}_x : T\ \mathit{sig}\ \ \mathbf{ok}} \qquad \frac{\begin{array}{l} E, t : K \vdash \mathit{sig}\ \ \mathbf{ok} \\ \text{t} \notin \text{dom}(\mathit{sig}) \end{array}}{E \vdash \mathbf{type}\ \ \text{t}_t : K\ \mathit{sig}\ \ \mathbf{ok}}$$

$\boxed{E \vdash_{\mathit{eqs}} \mathit{sig} <: \mathit{sig}'}$

$$\frac{E \vdash \mathit{eqs}\ \ \mathbf{ok}}{E \vdash_{\mathit{eqs}} \text{empty} <: \text{empty}} \qquad \frac{\begin{array}{l} E \vdash_{\mathit{eqs}} T \approx T' \\ E, x : T \vdash_{\mathit{eqs}} \mathit{sig} <: \mathit{sig}' \\ \text{x} \notin \text{dom}(\mathit{sig}) \end{array}}{E \vdash_{\mathit{eqs}} \mathbf{val}\ \ \text{x}_x : T\ \mathit{sig} <: \mathbf{val}\ \ \text{x}_x : T'\ \mathit{sig}'} \qquad \frac{\begin{array}{l} E \vdash_{\mathit{eqs}} K <: K' \\ E, t : K \vdash_{\mathit{eqs}} \mathit{sig} <: \mathit{sig}' \\ \text{t} \notin \text{dom}(\mathit{sig}) \end{array}}{E \vdash_{\mathit{eqs}} \mathbf{type}\ \ \text{t}_t : K\ \mathit{sig} <: \mathbf{type}\ \ \text{t}_t : K'\ \mathit{sig}'}$$

refl and trans are derivable

$\boxed{E \vdash_{\mathit{eqs}} \mathit{sig} \approx \mathit{sig}'}$

$$\frac{E \vdash \mathit{eqs}\ \ \mathbf{ok}}{E \vdash_{\mathit{eqs}} \text{empty} \approx \text{empty}} \qquad \frac{\begin{array}{l} E \vdash_{\mathit{eqs}} T \approx T' \\ E, x : T \vdash_{\mathit{eqs}} \mathit{sig} \approx \mathit{sig}' \\ \text{x} \notin \text{dom}(\mathit{sig}) \end{array}}{E \vdash_{\mathit{eqs}} \mathbf{val}\ \ \text{x}_x : T\ \mathit{sig} \approx \mathbf{val}\ \ \text{x}_x : T'\ \mathit{sig}'} \qquad \frac{\begin{array}{l} E \vdash_{\mathit{eqs}} K \approx K' \\ E, t : K \vdash_{\mathit{eqs}} \mathit{sig} \approx \mathit{sig}' \\ \text{t} \notin \text{dom}(\mathit{sig}) \end{array}}{E \vdash_{\mathit{eqs}} \mathbf{type}\ \ \text{t}_t : K\ \mathit{sig} \approx \mathbf{type}\ \ \text{t}_t : K'\ \mathit{sig}'}$$

refl and trans are derivable

$\boxed{E \vdash_{\mathit{eqs}} \mathit{str} : \mathit{sig}}$

$$\frac{E \vdash \mathit{eqs}\ \ \mathbf{ok}}{E \vdash_{\mathit{eqs}} \text{empty} : \text{empty}} \qquad \frac{\begin{array}{l} E, x : T \vdash_{\mathit{eqs}} \mathit{str} : \mathit{sig} \\ E \vdash_{\mathit{eqs}} v : T \\ \text{x} \notin \text{dom}(\mathit{sig}) \end{array}}{E \vdash_{\mathit{eqs}} \mathbf{let}\ \ \text{x}_x = v\ \mathit{str} : \mathbf{val}\ \ \text{x}_x : T\ \mathit{sig}} \qquad \frac{\begin{array}{l} E, t : \text{EQ}(T) \vdash_{\mathit{eqs}} \mathit{str} : \mathit{sig} \\ E \vdash_{\mathit{eqs}} T : K \\ \text{t} \notin \text{dom}(\mathit{sig}) \end{array}}{E \vdash_{\mathit{eqs}} \mathbf{type}\ \ \text{t}_t = T\ \mathit{str} : \mathbf{type}\ \ \text{t}_t : K\ \mathit{sig}}$$

$$\frac{\begin{array}{l} E \vdash_{\mathit{eqs}} T \approx T_1 \to .. \to T_n \to T_0 \\ E \vdash p_i : T_i\ \rhd\ E_i \qquad i = 1..n \\ E, x : T \vdash \mathit{str} : \mathit{sig} \\ E, E_1, .., E_n \vdash_{\mathit{eqs}} e : T_0 \\ \text{x} \notin \text{dom}(\mathit{sig}) \\ n \geq 1 \end{array}}{E \vdash_{\mathit{eqs}} \mathbf{let}\ \ \text{x}_x : T\ p_1..p_n = e\ \mathit{str} : \mathbf{val}\ \ \text{x}_x : T\ \mathit{sig}}$$

Figure 6: Typing Rules – Signatures, Subsignaturing (part 1)

$\boxed{E \vdash \mathit{Sig}\ \ \mathbf{ok}}$

$$\frac{E \vdash \mathit{sig}\ \ \mathbf{ok}}{E \vdash \mathbf{sig}\ \ \mathit{sig}\ \ \mathbf{end}\ \ \mathbf{ok}}$$

$\boxed{E \vdash_{eqs} \mathit{Sig} <: \mathit{Sig}'}$

$$\frac{E \vdash_{eqs} \mathit{sig} <: \mathit{sig}'}{E \vdash_{eqs} \mathbf{sig}\ \ \mathit{sig}\ \ \mathbf{end} <: \mathbf{sig}\ \ \mathit{sig}'\ \ \mathbf{end}}$$

refl and trans are derivable

$\boxed{E \vdash_{eqs} \mathit{Sig} \approx \mathit{Sig}'}\boxed{E \vdash_{eqs} \mathit{Str} : \mathit{Sig}}$

$$\frac{E \vdash_{eqs} \mathit{sig} \approx \mathit{sig}'}{E \vdash_{eqs} \mathbf{sig}\ \ \mathit{sig}\ \ \mathbf{end} \approx \mathbf{sig}\ \ \mathit{sig}'\ \ \mathbf{end}} \qquad \frac{E \vdash_{eqs} \mathit{str} : \mathit{sig}}{E \vdash_{eqs} \mathbf{struct}\ \ \mathit{str}\ \ \mathbf{end} : \mathbf{sig}\ \ \mathit{sig}\ \ \mathbf{end}}$$

Perhaps we should collapse the *sig* /*Sig* and *str*/ *Str* distinction. It is needed with functors, which we do not have at present.

Figure 7: Typing Rules – Signatures, Subsignaturing (part 2)

$\boxed{\vdash \mathit{str}\ \ \mathbf{flat}}\boxed{\vdash \mathit{Str}\ \ \mathbf{flat}}$

$$\frac{}{\vdash \text{empty}\ \ \mathbf{flat}} \qquad \frac{\vdash \mathit{str}\ \ \mathbf{flat}}{\vdash \mathbf{let}\ \ \mathrm{x}_x = v\ \mathit{str}\ \ \mathbf{flat}} \qquad \frac{\begin{array}{c} t \notin \text{fv}(\mathit{str}) \\ \vdash \mathit{str}\ \ \mathbf{flat} \end{array}}{\vdash \mathbf{type}\ \ \mathrm{t}_t = T\ \mathit{str}\ \ \mathbf{flat}} \qquad \frac{\vdash \mathit{str}\ \ \mathbf{flat}}{\vdash \mathbf{struct}\ \ \mathit{str}\ \ \mathbf{end}\ \ \mathbf{flat}}$$

$\boxed{\vdash \mathit{sig}\ \ \mathbf{flat}}\boxed{\vdash \mathit{Sig}\ \ \mathbf{flat}}$

$$\frac{}{\vdash \text{empty}\ \ \mathbf{flat}} \qquad \frac{\vdash \mathit{sig}\ \ \mathbf{flat}}{\vdash \mathbf{val}\ \ \mathrm{x}_x : T\ \mathit{sig}\ \ \mathbf{flat}} \qquad \frac{\begin{array}{c} t \notin \text{fv}(\mathit{sig}) \\ \vdash \mathit{sig}\ \ \mathbf{flat} \end{array}}{\vdash \mathbf{type}\ \ \mathrm{t}_t : \text{EQ}(T)\ \mathit{sig}\ \ \mathbf{flat}} \qquad \frac{\vdash \mathit{sig}\ \ \mathbf{flat}}{\vdash \mathbf{type}\ \ \mathrm{t}_t : \text{TYPE}\ \mathit{sig}\ \ \mathbf{flat}}$$

$$\frac{\vdash \mathit{sig}\ \ \mathbf{flat}}{\vdash \mathbf{sig}\ \ \mathit{sig}\ \ \mathbf{end}\ \ \mathbf{flat}}$$

Figure 8: Typing Rules – **flat** predicates

$\boxed{E \vdash_{eqs} \mathrm{M}_M : \mathit{Sig}}$

$$\frac{E_1, \mathrm{M}_M : \mathit{Sig}, E_2 \vdash \mathit{eqs}\ \ \mathbf{ok}}{E_1, \mathrm{M}_M : \mathit{Sig}, E_2 \vdash_{eqs} \mathrm{M}_M : \mathit{Sig}}$$

$$\frac{E \vdash_{eqs} \mathrm{M}_M : \mathbf{sig}\ \ \mathit{sig}_1\ \mathbf{type}\ \ \mathrm{t}_t : K\ \mathit{sig}_2\ \ \mathbf{end}}{E \vdash_{eqs} \mathrm{M}_M : \mathbf{sig}\ \ \mathit{sig}_1\ \mathbf{type}\ \ \mathrm{t}_t : \mathrm{EQ}(\mathrm{M}_M.\mathrm{t})\ \mathit{sig}_2\ \ \mathbf{end}} \qquad \frac{\begin{array}{l} E \vdash_{eqs} \mathrm{M}_M : \mathit{Sig} \\ E \vdash_{eqs} \mathit{Sig} <: \mathit{Sig}' \end{array}}{E \vdash_{eqs} \mathrm{M}_M : \mathit{Sig}'}$$

$\boxed{E \vdash_{eqs} h : \mathit{Sig}}$

$$\frac{\begin{array}{l} h = \mathbf{hash}(\mathbf{hmodule}_{eqs'}\ \mathrm{M} : \mathit{Sig}\ \ \mathbf{version}\ \ \mathit{vne} = \mathit{Str}) \\ \mathrm{namepart}(E) \vdash h\ \ \mathbf{ok} \\ E \vdash \mathit{eqs}\ \ \mathbf{ok} \end{array}}{E \vdash_{eqs} h : \mathit{Sig}} \bigstar$$

$$\frac{\begin{array}{l} h = \mathrm{n} \\ (\mathrm{n} : \mathbf{nmodule}_{eqs'}\ \mathrm{M} : \mathit{Sig}\ \ \mathbf{version}\ \ \mathit{vne} = \mathit{Str}) \in\ \mathrm{namepart}(E) \\ \mathrm{namepart}(E) \vdash h\ \ \mathbf{ok} \\ E \vdash \mathit{eqs}\ \ \mathbf{ok} \end{array}}{E \vdash_{eqs} h : \mathit{Sig}} \bigstar$$

$$\frac{\begin{array}{l} h = \mathbf{hash}(\mathbf{himport}\ \mathrm{M}\ : \mathit{Sig}\ \ \mathbf{version}\ \ \mathit{vc}\ \ \mathbf{like}\ \ \mathit{Str}) \\ \mathrm{namepart}(E) \vdash h\ \ \mathbf{ok} \\ E \vdash \mathit{eqs}\ \ \mathbf{ok} \end{array}}{E \vdash_{eqs} h : \mathit{Sig}} \bigstar$$

$$\frac{\begin{array}{l} h = \mathrm{n} \\ (\mathrm{n} : \mathbf{nimport}\ \mathrm{M} : \mathit{Sig}\ \ \mathbf{version}\ \ \mathit{vc}\ \ \mathbf{like}\ \ \mathit{Str}) \in\ \mathrm{namepart}(E) \\ \mathrm{namepart}(E) \vdash h\ \ \mathbf{ok} \\ E \vdash \mathit{eqs}\ \ \mathbf{ok} \end{array}}{E \vdash_{eqs} h : \mathit{Sig}} \bigstar$$

$$\frac{\begin{array}{l} E \vdash_{eqs} h : \mathbf{sig}\ \ \mathit{sig}_1\ \mathbf{type}\ \ \mathrm{t}_t : K\ \mathit{sig}_2\ \ \mathbf{end} \\ \mathrm{t}\ \text{abstract in}_{\mathrm{namepart}(E)}\ h \end{array}}{E \vdash_{eqs} h : \mathbf{sig}\ \ \mathit{sig}_1\ \mathbf{type}\ \ \mathrm{t}_t : \mathrm{EQ}(h.\mathrm{t})\ \mathit{sig}_2\ \ \mathbf{end}} \bigstar \qquad \frac{\begin{array}{l} E \vdash_{eqs} h : \mathit{Sig} \\ E \vdash_{eqs} \mathit{Sig} <: \mathit{Sig}' \end{array}}{E \vdash_{eqs} h : \mathit{Sig}'}$$

Again h behaves much like M_M.
For both this and M_M there is a stylistic choice as to how much selfification we do in one go; the rules deal with just a single field at a time. This judgement is wrt an E for uniformity (see the $h.\mathrm{x}$ rule).

Figure 9: Typing Rules – Signatures of module identifiers and hashes

$$\boxed{E \vdash_{eqs} e : T}$$

$$\frac{\begin{array}{l} E \vdash_{\varnothing} T : \textsc{Type} \\ \mathrm{C}_0 : T \\ E \vdash \mathit{eqs}\ \mathbf{ok} \end{array}}{E \vdash_{eqs} \mathrm{C}_0 : T} \qquad \frac{\begin{array}{l} E \vdash_{\varnothing} T \to T' : \textsc{Type} \\ \mathrm{C}_1 : T \to T' \\ E \vdash_{eqs} e : T \end{array}}{E \vdash_{eqs} \mathrm{C}_1\ e : T'} \qquad \frac{\begin{array}{l} E \vdash_{eqs} e_1 : T \\ E \vdash_{eqs} e_2 : T\ \mathsf{list} \end{array}}{E \vdash_{eqs} e_1 :: e_2 : T\ \mathsf{list}} \qquad \frac{\begin{array}{l} E \vdash_{eqs} e_k : T_k\ k \in 1..n \\ n \geq 2 \end{array}}{E \vdash_{eqs} (e_1, .., e_n) : T_1 * .. * T_n}$$

$$\frac{\begin{array}{l} x \notin \mathrm{dom}(E_{\mathrm{const}}) \\ E_1, x : T, E_2 \vdash \mathit{eqs}\ \mathbf{ok} \end{array}}{E_1, x : T, E_2 \vdash_{eqs} x : T} \qquad \frac{\begin{array}{l} E \vdash_{eqs} \mathrm{M}_M : \mathbf{sig}\ \mathit{sig}\ \mathbf{val}\ \mathrm{x}_x : T\ \mathit{sig}'\ \mathbf{end} \\ E \vdash_{\varnothing} T : \textsc{Type} \end{array}}{E \vdash_{eqs} \mathrm{M}_M.\mathrm{x} : T} \qquad \frac{E_1, l : T, E_2 \vdash \mathit{eqs}\ \mathbf{ok}}{E_1, l : T, E_2 \vdash_{eqs} l : T}$$

$$\frac{\begin{array}{l} E \vdash_{eqs} e_1 : \mathsf{bool} \\ E \vdash_{eqs} e_2 : T \\ E \vdash_{eqs} e_3 : T \end{array}}{E \vdash_{eqs} \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 : T} \qquad \frac{\begin{array}{l} E \vdash_{eqs} e_1 : \mathsf{bool} \\ E \vdash_{eqs} e_2 : \mathsf{unit} \end{array}}{E \vdash_{eqs} \mathbf{while}\ e_1\ \mathbf{do}\ e_2\ \mathbf{done} : \mathsf{unit}} \qquad \frac{\begin{array}{l} E \vdash_{eqs} e_1 : \mathsf{unit} \\ E \vdash_{eqs} e_2 : T \end{array}}{E \vdash_{eqs} e_1\ \mathbf{;}\ e_2 : T}$$

$$\frac{\begin{array}{l} E \vdash_{eqs} e_1 : \mathsf{bool} \\ E \vdash_{eqs} e_2 : \mathsf{bool} \end{array}}{\begin{array}{l} E \vdash_{eqs} e_1\ \&\&\ e_2 : \mathsf{bool} \\ E \vdash_{eqs} \mathit{eq}\ ||\ e_2 : \mathsf{bool} \end{array}} \qquad \frac{E, x : T \vdash_{eqs} e : T'}{E \vdash_{eqs} \mathbf{function}\ (x : T) \to e : T \to T'} \qquad \frac{\begin{array}{l} E \vdash_{eqs} e_1 : T \to T' \\ E \vdash_{eqs} e_2 : T \end{array}}{E \vdash_{eqs} e_1\ e_2 : T'}$$

$$\frac{\begin{array}{l} \mathit{op}^n : T_1 \to .. \to T_n \to T \\ E \vdash \mathit{eqs}\ \mathbf{ok} \\ E \vdash_{eqs} e_j : T_j \quad j \in 1..n \end{array}}{E \vdash_{eqs} \mathit{op}^n e_1\ ..\ e_n : T} \bigstar \qquad \frac{\begin{array}{l} x^n \in \mathrm{dom}(E_{\mathrm{const}}) \\ E_1, x^n : T', E_2 \vdash_{eqs} T' \approx T_1 \to .. \to T_n \to T \\ E_1, x^n : T', E_2 \vdash_{eqs} e_j : T_j \quad j \in 1..n \end{array}}{E_1, x^n : T', E_2 \vdash_{eqs} x^n e_1\ ..\ e_n : T} \bigstar$$

$$\frac{\begin{array}{l} E \vdash \mathit{eqs}\ \mathbf{ok} \\ E \vdash_{\varnothing} e_0 : T_1 \to .. \to T_n \to T \\ E \vdash_{\varnothing} e_j : T_j \quad j \in 1..n \end{array}}{E \vdash_{eqs} \mathbf{op}(e_0)^n\ e_1\ ..\ e_n : T} \bigstar \qquad \frac{\begin{array}{l} E \vdash_{\varnothing} T_1 + .. + T_n : \textsc{Type} \\ E \vdash_{eqs} e : T_i \end{array}}{E \vdash_{eqs} \mathrm{INJ}_i^{(T_1 + .. + T_n)} e : T_1 + .. + T_n}$$

$$\frac{\begin{array}{l} E \vdash_{eqs} e : T \\ E \vdash_{eqs} \mathit{mtch} : T \to T' \end{array}}{E \vdash_{eqs} \mathbf{match}\ e\ \mathbf{with}\ \mathit{mtch} : T'} \qquad \frac{\begin{array}{l} E \vdash_{eqs} T_1 \approx T_2 \to T_3 \\ E, x_1 : T_1, x_2 : T_2 \vdash_{eqs} e_3 : T_3 \\ E, x_1 : T_1 \vdash_{eqs} e_4 : T_4 \end{array}}{E \vdash_{eqs} \mathbf{let}\ \mathbf{rec}\ x_1 : T_1 = \mathbf{function}\ (x_2 : T_2) \to e_3\ \mathbf{in}\ e_4 : T_4}$$

$$\frac{E \vdash_{eqs} e : \mathsf{exn}}{E \vdash_{eqs} \mathbf{raise}\ e : T} \qquad \frac{\begin{array}{l} E \vdash_{eqs} e : T \\ E \vdash_{eqs} \mathit{mtch} : \mathsf{exn} \to T \end{array}}{E \vdash_{eqs} \mathbf{try}\ e\ \mathbf{with}\ \mathit{mtch} : T} \qquad \frac{\begin{array}{l} E \vdash_{\varnothing} T : \textsc{Type} \\ E \vdash \mathit{eqs}\ \mathbf{ok} \end{array}}{\begin{array}{l} E \vdash_{eqs} \mathbf{RET}_T : T \\ E \vdash_{eqs} \mathbf{SLOWRET}_T : T \end{array}} \bigstar$$

$$\frac{\begin{array}{l} E \vdash_{eqs} e_1 : \mathsf{string} \\ E \vdash_{eqs} e_2 : T \end{array}}{E \vdash_{eqs} \mathbf{marshal}\ e_1\ e_2 : T : \mathsf{string}} \bigstar \qquad \frac{E \vdash_{eqs} e : \mathsf{string}}{E \vdash_{eqs} \mathbf{unmarshal}\ e\ \mathbf{as}\ T : T} \bigstar \qquad \frac{E \vdash_{\varnothing} e : T}{E \vdash_{eqs} \mathbf{marshalz}\ \underline{s}\ e : T : \mathsf{string}} \bigstar$$

$$\frac{\begin{array}{l} E \vdash \mathit{eqs}\ \mathbf{ok} \\ E \vdash_{\varnothing} T : \textsc{Type} \\ E \vdash_{eqs'} e : T \end{array}}{E \vdash_{eqs} [e]_{eqs'}^{T} : T} \bigstar \qquad \frac{\begin{array}{l} E \vdash_{eqs} e : T \\ E \vdash_{eqs} T \approx T' \end{array}}{E \vdash_{eqs} e : T'}$$

Figure 10: Typing Rules – Expressions (part 1)

$\boxed{E \vdash_{eqs} e : T \quad \text{continued...}}$

$$\frac{\begin{array}{l} E \vdash_{eqs} e_1 : T\ \mathsf{ref} \\ E \vdash_{eqs} e_2 : T \end{array}}{E \vdash_{eqs} e_1 \mathbin{\mathbf{:=}_T} e_2 : \mathsf{unit}} \qquad \frac{\begin{array}{l} E \vdash_{eqs} e_1 : T\ \mathsf{ref} \\ E \vdash_{\varnothing} e_2 : T \end{array}}{E \vdash_{eqs} e_1 \mathbin{\mathbf{:=}'_T} e_2 : \mathsf{unit}} \qquad \frac{E \vdash_{eqs} e_1 : T\ \mathsf{ref}}{E \vdash_{eqs} \mathbf{!}_T e_1 : T}$$

$$\frac{\begin{array}{l} E \vdash_{eqs} h : \mathbf{sig}\ \ \mathit{sig}\ \mathbf{val}\ \ \mathrm{x}_x : T\ \mathit{sig}'\ \ \mathbf{end} \\ E \vdash_{\varnothing} T : \textsc{Type} \end{array}}{E \vdash_{eqs} h.\mathrm{x} : T} \bigstar \qquad \frac{\begin{array}{l} E \vdash_{eqs} \mathrm{M}'_{M'} : \mathit{Sig} \\ E \vdash_{eqs} \mathrm{M}_M.\mathrm{x} : T \end{array}}{\begin{array}{l} E \vdash_{eqs} \mathbf{resolve}(\mathrm{M}_M.\mathrm{x}, \mathrm{M}'_{M'}, \mathit{resolvespec}) : T \\ E \vdash_{eqs} \mathbf{resolve\_blocked}(\mathrm{M}_M.\mathrm{x}, \mathrm{M}'_{M'}, \mathit{resolvespec}) : T \end{array}} \bigstar$$

$$\frac{\begin{array}{l} E \vdash_{eqs} e_1 : T\ \mathsf{name} \\ E \vdash_{eqs} e_2 : T\ \mathsf{name} \\ E \vdash_{eqs} e_3 : T' \end{array}}{E \vdash_{eqs} \mathbf{swap}\ \ e_1\ \ \mathbf{and}\ \ e_2\ \ \mathbf{in}\ \ e_3 : T'} \qquad \frac{\begin{array}{l} E \vdash_{eqs} e_1 : T\ \mathsf{name} \\ E \vdash_{eqs} e_2 : T' \end{array}}{E \vdash_{eqs} e_1\ \ \mathbf{freshfor}\ \ e_2 : \mathsf{bool}} \qquad \frac{\begin{array}{l} E \vdash_{\varnothing} T : \textsc{Type} \\ E \vdash_{eqs} e : T' \end{array}}{E \vdash_{eqs} \mathbf{support}_T e : T\ \mathsf{name}\ \ \mathsf{list}}$$

$$\frac{E \vdash_{eqs} \mathrm{M}_M.\mathrm{x} : T}{E \vdash_{eqs} \mathrm{M}_M@\mathrm{x} : T\ \mathsf{tie}} \qquad \frac{E \vdash_{eqs} e : T\ \mathsf{tie}}{\begin{array}{l} E \vdash_{eqs} \mathbf{name\_of\_tie}\ e : T\ \mathsf{name} \\ E \vdash_{eqs} \mathbf{val\_of\_tie}\ e : T \end{array}} \qquad \frac{E \vdash_{eqs} e_1 : \mathsf{unit} \qquad E \vdash_{eqs} e_2 : T}{E \vdash_{eqs} e_1 | e_2 : T}$$

$$\frac{\begin{array}{l} E \vdash\ \mathit{eqs}\ \ \mathbf{ok} \\ E, t : \textsc{Type} \vdash_{eqs} e : T \end{array}}{E \vdash_{eqs} \Lambda\ t \to e : \forall\ t.T} \qquad \frac{\begin{array}{l} E \vdash_{eqs} e : \forall\ t.T_1 \\ E \vdash_{\varnothing} T_2 : \textsc{Type} \end{array}}{E \vdash_{eqs} e\ \ T_2 : \{T_2/t\}T_1} \qquad \frac{\begin{array}{l} E \vdash_{\varnothing} T_2 : \textsc{Type} \\ E \vdash_{eqs} e : \{T_2/t\}T_1 \end{array}}{E \vdash_{eqs} \{T_2, e\}\ \mathbf{as}\ \ \exists\ t.T_1 : \exists\ t.T_1}$$

$$\frac{\begin{array}{l} E \vdash_{eqs} e_1 : \exists\ t.T \\ E, t : \textsc{Type}, x : T \vdash_{eqs} e_2 : T_2 \end{array}}{E \vdash \mathbf{let}\ \{t, x\} = e_1\ \ \mathbf{in}\ \ e_2 : T_2} \qquad \frac{\begin{array}{l} E \vdash_{eqs} e : T'\ \mathsf{name} \\ E \vdash_{eqs} e_1 : \exists\ t.t\ \ \mathsf{name} * T \\ E, t : \mathrm{EQ}(T'), x_1 : T'\ \mathsf{name}, x_2 : T \vdash_{eqs} e_2 : T_2 \\ E \vdash_{eqs} e_3 : T_2 \end{array}}{E \vdash \mathbf{namecase}\ \ e_1\ \ \mathbf{with}\ \{t, (x_1, x_2)\}\ \mathbf{when}\ \ x_1 = e \to e_2\ \ \mathbf{otherwise}\ \to e_3}$$

$$\frac{\begin{array}{l} E \vdash\ \mathit{eqs}\ \ \mathbf{ok} \\ E \vdash_{\varnothing} T : \textsc{Type} \end{array}}{E \vdash_{eqs} \mathbf{fresh}_T : T\ \mathsf{name}} \qquad \frac{\begin{array}{l} E \vdash\ \mathit{eqs}\ \ \mathbf{ok} \\ E \vdash_{\varnothing} T : \textsc{Type} \end{array}}{E \vdash_{eqs} \mathbf{cfresh}_T : T\ \mathsf{name}} \qquad \frac{E \vdash_{eqs} X.\mathrm{x} : T}{E \vdash_{eqs} \mathbf{hash}(X.\mathrm{x})_T : T\ \mathsf{name}}$$

$$\frac{\begin{array}{l} E \vdash_{\varnothing} T : \textsc{Type} \\ E \vdash_{eqs} e : \mathsf{string} \end{array}}{E \vdash_{eqs} \mathbf{hash}(T, e)_T : T\ \mathsf{name}} \qquad \frac{\begin{array}{l} E \vdash_{\varnothing} T' : \textsc{Type} \\ E \vdash_{eqs} e_1 : \mathsf{string} \\ E \vdash_{eqs} e_2 : T\ \mathsf{name} \end{array}}{E \vdash_{eqs} \mathbf{hash}(T', e_1, e_2)_{T'} : T'\ \mathsf{name}} \qquad \frac{E_1, \mathrm{n} : T\ \mathsf{name}, E_2 \vdash\ \mathit{eqs}\ \ \mathbf{ok}}{E_1, \mathrm{n} : T\ \mathsf{name}, E_2 \vdash_{eqs} \mathrm{n}_T : T\ \mathsf{name}}$$

Figure 11: Typing Rules – Expressions (part 2)

$$\boxed{E \vdash_{eqs} e : T \quad \text{Sugared source forms}}$$

$$\frac{E \vdash_{eqs} \mathit{mtch} : T \to T'}{E \vdash_{eqs} \textbf{function}\ \ \mathit{mtch} : T \to T'}$$

$$\frac{\begin{array}{l} E \vdash_{eqs} T_1 \approx T_2 \to T_3 \\ E, x : T_1 \vdash_{eqs} \mathit{mtch} : T_2 \to T_3 \\ E, x : T_1 \vdash_{eqs} e_4 : T_4 \end{array}}{E \vdash_{eqs} \textbf{let}\ \ \textbf{rec}\ \ x : T_1 = \textbf{function}\ \ \mathit{mtch}\ \ \textbf{in}\ \ e_4 : T_4}$$

$$\frac{\begin{array}{l} E \vdash_{eqs} T \approx T_1 \to .. \to T_n \to T' \\ E \vdash p_i : T_i \ \ \rhd\ \ E_i \\ E, x : T, E_1, .., E_n \vdash_{eqs} e' : T' \\ E, x : T \vdash_{eqs} e'' : T'' \end{array}}{E \vdash_{eqs} \textbf{let}\ \ \textbf{rec}\ \ x : T\ p_1..p_n = e'\ \ \textbf{in}\ \ e'' : T''}$$

$$\frac{\begin{array}{l} E \vdash p : T_1 \ \ \rhd\ \ E' \\ E \vdash_{eqs} e_1 : T_1 \\ E, E' \vdash_{eqs} e_2 : T_2 \end{array}}{E \vdash_{eqs} \textbf{let}\ \ p = e_1\ \ \textbf{in}\ \ e_2 : T_2}$$

$$\frac{\begin{array}{l} E \vdash_{eqs} T \approx T_1 \to .. \to T_n \to T' \\ E \vdash p_i : T_i \ \ \rhd\ \ E_i \qquad i = 1..n \\ E, E_1, .., E_n \vdash_{eqs} e' : T' \\ E, x : T \vdash_{eqs} e'' : T'' \end{array}}{E \vdash_{eqs} \textbf{let}\ \ x : T\ p_1..p_n = e'\ \ \textbf{in}\ \ e'' : T''}$$

$$\frac{\begin{array}{l} op^n : T_1 \to .. \to T_n \to T \\ E \vdash \mathit{eqs}\ \ \textbf{ok} \\ E \vdash_{eqs} e_j : T_j \qquad j \ \in\ 1..k, k \ \in\ 0..n-1 \end{array}}{E \vdash_{eqs} op^n e_1\ ..\ e_k : T_{k+1} \to .. \to T_n \to T} \ \bigstar$$

$$\frac{\begin{array}{l} x^n \in \mathrm{dom}(E_{\mathrm{const}}) \\ E_1, x^n : T', E_2 \vdash_{eqs} T' \approx T_1 \to .. \to T_n \to T \\ E_1, x^n : T', E_2 \vdash_{eqs} e_j : T_j \qquad j \ \in\ 1..k, k \ \in\ 0..n-1 \end{array}}{E_1, x^n : T', E_2 \vdash_{eqs} x^n e_1\ ..\ e_k : T_{k+1} \to .. \to T_n \to T} \ \bigstar$$

Figure 12: Typing Rules – Sugared Forms

$$\boxed{E \vdash p : T \ \ \rhd\ \ E'}$$

$$\frac{E \vdash_{\varnothing} T : \textsc{Type}}{E \vdash (\_ : T) : T \ \ \rhd\ \ \text{empty}} \qquad \frac{E \vdash_{\varnothing} T : \textsc{Type}}{E \vdash (x : T) : T \ \ \rhd\ \ x : T} \qquad \frac{\begin{array}{c} \mathrm{C}_0 : T \\ E \vdash\ \textbf{ok} \end{array}}{E \vdash \mathrm{C}_0 : T \ \ \rhd\ \ \text{empty}} \qquad \frac{\begin{array}{c} \mathrm{C}_1 : T \to T' \\ E \vdash p : T \ \ \rhd\ \ E' \end{array}}{E \vdash \mathrm{C}_1\ p : T' \ \ \rhd\ \ E'}$$

$$\frac{\begin{array}{l} E \vdash p_1 : T \ \ \rhd\ \ E_1 \\ E \vdash p_2 : T\ \ \mathsf{list}\ \ \rhd\ \ E_2 \end{array}}{E \vdash p_1 :: p_2 : T\ \ \mathsf{list}\ \ \rhd\ \ E_1, E_2} \qquad \frac{\begin{array}{l} E \vdash p_k : T_k \ \ \rhd\ \ E_k\ k \ \in\ 1..n \\ n \geq 2 \end{array}}{E \vdash (p_1, .., p_n) : T_1 * .. * T_n \ \ \rhd\ \ E_1, .., E_n} \qquad \frac{E \vdash p : T \ \ \rhd\ \ E'}{E \vdash (p : T) : T \ \ \rhd\ \ E'}$$

$$\boxed{E \vdash_{eqs} \mathit{mtch} : T \to T'}$$

$$\frac{\begin{array}{l} E \vdash p : T \ \ \rhd\ \ E' \\ E, E' \vdash_{eqs} e : T' \end{array}}{E \vdash_{eqs} p \to e : T \to T'} \qquad \frac{\begin{array}{l} E \vdash_{eqs} p \to e : T \to T' \\ E \vdash_{eqs} \mathit{mtch} : T \to T' \end{array}}{E \vdash_{eqs} p \to e | \mathit{mtch} : T \to T'}$$

Figure 13: Typing Rules – Patterns, Matches

$$\boxed{E_{\mathrm{n}} \vdash \mathit{avne}\ \ \mathbf{ok}}$$

$$\frac{}{E_{\mathrm{n}} \vdash \underline{n}\ \ \mathbf{ok}} \qquad \frac{}{E_{\mathrm{n}} \vdash \underline{N}\ \ \mathbf{ok}} \qquad \frac{E_{\mathrm{n}} \vdash h\ \ \mathbf{ok}}{E_{\mathrm{n}} \vdash h\ \ \mathbf{ok}} \qquad \frac{}{E_{\mathrm{n}} \vdash \mathbf{myname}\ \ \mathbf{ok}}$$

$$\boxed{E_{\mathrm{n}} \vdash \mathit{vne}\ \ \mathbf{ok}}$$

$$\frac{E_{\mathrm{n}} \vdash \mathit{avne}\ \ \mathbf{ok}}{E_{\mathrm{n}} \vdash \mathit{avne}\ \ \mathbf{ok}} \qquad \frac{\begin{array}{c} E_{\mathrm{n}} \vdash \mathit{avne}\ \ \mathbf{ok} \\ E_{\mathrm{n}} \vdash \mathit{vne}\ \ \mathbf{ok} \end{array}}{E_{\mathrm{n}} \vdash \mathit{avne}.\mathit{vne}\ \ \mathbf{ok}}$$

$$\boxed{E \vdash \mathit{ahvce}\ \ \mathbf{ok}}$$

$$\frac{E \vdash\ \mathbf{ok}}{E \vdash \underline{N}\ \ \mathbf{ok}} \qquad \frac{\begin{array}{l} \mathrm{namepart}(E) \vdash h\ \ \mathbf{ok} \\ E \vdash\ \mathbf{ok} \end{array}}{E \vdash h\ \ \mathbf{ok}} \bigstar \qquad \frac{E_1, \mathrm{M}_M : \mathit{Sig}, E_2 \vdash\ \mathbf{ok}}{E_1, \mathrm{M}_M : \mathit{Sig}, E_2 \vdash \mathrm{M}_M\ \ \mathbf{ok}} \bigstar$$

$$\boxed{E \vdash \mathit{avce}\ \ \mathbf{ok}}$$

$$\frac{E \vdash\ \mathbf{ok}}{E \vdash \underline{n}\ \ \mathbf{ok}} \qquad \frac{E \vdash \mathit{ahvce}\ \ \mathbf{ok}}{E \vdash \mathit{ahvce}\ \ \mathbf{ok}}$$

$$\boxed{E \vdash \mathit{dvce}\ \ \mathbf{ok}}$$

$$\frac{E \vdash \mathit{avce}\ \ \mathbf{ok}}{E \vdash \mathit{avce}\ \ \mathbf{ok}} \qquad \frac{\begin{array}{c} E \vdash \mathit{avce}\ \ \mathbf{ok} \\ E \vdash \mathit{dvce}\ \ \mathbf{ok} \end{array}}{E \vdash \mathit{avce}.\mathit{dvce}\ \ \mathbf{ok}} \qquad \frac{E \vdash\ \mathbf{ok}}{\begin{array}{l} E \vdash \underline{n}\text{–}\underline{n}'\ \ \mathbf{ok} \\ E \vdash \text{–}\underline{n}'\ \ \mathbf{ok} \\ E \vdash \underline{n}\text{–}\ \mathbf{ok} \\ E \vdash *\ \mathbf{ok} \end{array}}$$

$$\boxed{E \vdash \mathit{vce}\ \ \mathbf{ok}}$$

$$\frac{E \vdash \mathit{dvce}\ \ \mathbf{ok}}{E \vdash \mathit{dvce}\ \ \mathbf{ok}} \qquad \frac{E \vdash \mathit{ahvce}\ \ \mathbf{ok}}{E \vdash \mathbf{name} = \mathit{ahvce}\ \ \mathbf{ok}}$$

Figure 14: Typing Rules – Version number and constraint expressions

$\boxed{E \vdash \mathit{likespec}\ \ \mathbf{ok}}$

$$\frac{E \vdash\ \mathbf{ok}}{E \vdash \text{empty}\ \ \mathbf{ok}} \qquad \frac{E_1, \mathrm{M}_M : \mathit{Sig}, E_2 \vdash\ \mathbf{ok}}{E_1, \mathrm{M}_M : \mathit{Sig}, E_2 \vdash\ \mathbf{like}\ \ \mathrm{M}_M\ \ \mathbf{ok}}\ \bigstar \qquad \frac{E \vdash_\varnothing \mathit{str} : \mathit{sig}}{E \vdash\ \mathbf{like}\ \ \mathbf{struct}\ \ \mathit{str}\ \ \mathbf{end}\ \ \mathbf{ok}}\ \bigstar$$

$\boxed{E \vdash\!= \mathit{Mo} : \mathit{Sig}}$

$$\frac{E \vdash \mathit{Sig}\ \ \mathbf{ok}}{E \vdash\!= \text{UNLINKED} : \mathit{Sig}} \qquad \frac{E_1, \mathrm{M}_M : \mathit{Sig}, E_2 \vdash\ \mathbf{ok}}{E_1, \mathrm{M}_M : \mathit{Sig}, E_2 \vdash\!= \mathrm{M}_M : \mathit{Sig}} \qquad \frac{\begin{array}{l} E \vdash_\varnothing \mathit{Sig} <: \mathit{Sig}' \\ E \vdash\!= \mathrm{M}_M : \mathit{Sig} \end{array}}{E \vdash\!= \mathrm{M}_M : \mathit{Sig}'}$$

Figure 15: Typing Rules – Definition auxiliaries

$\boxed{E \vdash \mathit{sourcedefinition}\ \ \triangleright\ \ E'}$

$$\frac{\begin{array}{l} E \vdash \mathit{vne}\ \ \mathbf{ok} \\ E \vdash_{\mathit{weqs}} \mathit{Str} : \mathit{Sig} \end{array}}{E \vdash \mathbf{module}\ \mathrm{M}_M : \mathit{Sig}\ \ \mathbf{version}\ \ \mathit{vne} = \mathit{Str}\ \ \mathbf{with}\ !\mathit{weqs}\ \ \triangleright\ \ \mathrm{M}_M : \mathit{Sig}}\ \bigstar$$

$$\frac{\begin{array}{l} E \vdash \mathit{vce}\ \ \mathbf{ok} \\ E \vdash \mathit{likespec}\ \ \mathbf{ok} \\ E \vdash\!= \mathit{Mo} : \mathit{Sig} \end{array}}{E \vdash \mathbf{import}\ \mathrm{M}_M : \mathit{Sig}\ \ \mathbf{version}\ \ \mathit{vce}\ \mathit{likespec}\ \ \mathbf{by}\ \ \mathit{resolvespec} = \mathit{Mo}\ \ \triangleright\ \ \mathrm{M}_M : \mathit{Sig}}\ \bigstar$$

We could additionally check that for all abstract type fields in Sig there is a corresponding type in an in-line $\mathit{likestr}$, or a type (which could reasonably be required to be abstract) in an $\mathrm{M'}_{M'}\ \mathit{likestr}$. At present this is left to compilation.

$$\frac{E \vdash\ \mathbf{ok}}{E \vdash \mathbf{mark}\ \ \text{MK}\ \ \triangleright\ \ \text{empty}}\ \bigstar$$

$$\frac{\begin{array}{l} E_1, \mathrm{M'}_{M'} : \mathit{Sig}, E_2 \vdash\ \mathbf{ok} \\ \mathit{Sig}' = \text{selfifysig}_{\mathrm{M'}_{M'}}(\mathit{Sig}) \end{array}}{E_1, \mathrm{M'}_{M'} : \mathit{Sig}, E_2 \vdash \mathbf{module}\ \mathrm{M}_M : \mathit{Sig}' = \mathrm{M'}_{M'}\ \ \triangleright\ \ \mathrm{M}_M : \mathit{Sig}'}\ \bigstar$$

Note that we have to selfify in the alias rule to avoid introducing new abstract types. We do not allow subsignaturing as we do not want to think about sealing here. We could allow sig equality.

(selfifysig ( ) is defined on page 93.)

Figure 16: Typing Rules – Source Definitions

16.3.2 Typing for Compiled and Executing Forms

$\boxed{E \vdash \mathit{definition} \ \rhd \ E'}$

$$\frac{\begin{array}{l}\mathrm{namepart}(E) \vdash h \ \ \mathbf{ok} \\ \mathrm{namepart}(E) \vdash \mathit{eqs} \ \ \mathbf{ok} \\ \mathrm{namepart}(E) \vdash \mathit{Sig}_0 \ \ \mathbf{ok} \\ \mathit{eqs}_0 = \mathrm{eqs\_of\_sign\_str}(h, \mathit{Sig}_0, \mathit{Str}) \\ \mathit{Sig}_1 = \mathrm{typeflattensig}(\mathrm{selfifysig}_h(\mathit{Sig}_0)) \\ E \vdash_{\mathit{eqs}_0, \mathit{eqs}} \ \mathit{Str} : \mathit{Sig}_1 \\ \vdash \mathit{Str} \ \ \mathbf{flat} \\ \mathrm{compiledform}(\mathit{eqs}, \mathit{Sig}_0, \mathit{Sig}_1, \mathit{Str})\end{array}}{E \vdash \mathbf{cmodule}_{h;\mathit{eqs};\mathit{Sig}_0} \ \mathit{vubs} \ \mathrm{M}_M : \mathit{Sig}_1 \ \ \mathbf{version} \ \ \mathit{vn} = \mathit{Str} \ \ \rhd \ \ \mathrm{M}_M : \mathit{Sig}_1} \ \bigstar$$

The Sig_1 is now computable from the h and Sig_0. We keep it in the data for the time being, however, as it has a clear conceptual role.

An alternative rule here (corresponding to that for user modules above) would have $E \vdash_{\mathit{eqs}_0} \ \mathit{Str} : \mathit{Sig}'_1$ and $E \vdash_{\mathit{eqs}_0, \mathit{eqs}} \mathit{Sig}'_1 \approx \mathit{Sig}_1$.

In typing compiled and hashed modules there is a stylistic choice as to how many of the properties that compilation establishes are captured in the typing rules. Here we choose to be rather tight, at the cost of some baroqueness. Note that in the $E \vdash_{\mathit{eqs}} \ \mathit{Str} : \mathit{Sig}$ premise the E allows term components of earlier modules to be used (as is required), but also allows type components to be used. We prevent the latter with the compiledform(...) premise, as they have been hashified by compilation.

$$\frac{\begin{array}{l}\mathrm{namepart}(E) \vdash h \ \ \mathbf{ok} \\ \mathit{Sig}_1 = \mathrm{typeflattensig}(\mathrm{selfifysig}_h(\mathit{Sig}_0)) \\ \mathrm{namepart}(E) \vdash \mathit{Sig}_0 \ \ \mathbf{ok} \\ \mathrm{namepart}(E) \vdash_{\varnothing} \ \mathit{Str} : \mathrm{limitdom}\,(\mathit{Sig}_0) \\ \vdash \mathit{Str} \ \ \mathbf{flat} \\ \mathrm{compiledform}(\mathit{Sig}_0, \mathit{Sig}_1, \mathit{Str}) \\ E \vdash = \mathit{Mo} : \mathit{Sig}_0\end{array}}{E \vdash \mathbf{cimport}_{h;\mathit{Sig}_0} \ \mathit{vubs} \ \mathrm{M}_M : \mathit{Sig}_1 \ \ \mathbf{version} \ \ \mathit{vc} \ \ \mathbf{like} \ \ \mathit{Str} \ \ \mathbf{by} \ \ \mathit{resolvespec} = \mathit{Mo} \ \ \rhd \ \ \mathrm{M}_M : \mathit{Sig}_1} \ \bigstar$$

Note that compilation has cut down the *Str* in *likespec*. The fact that this ensures it doesn't include any code (or extraneous types) legitimizes the empty. This rule does not explicitly check type coherence between the *likespec* and the *Mo* implementation, as the latter is not available in E, but note that Sig_1 is hashified.

$$\frac{E \vdash \ \mathbf{ok}}{E \vdash \mathbf{mark} \ \ \mathrm{MK} \ \ \rhd \ \ \mathrm{empty}}$$

$$\frac{\begin{array}{l}\mathit{definition} = \mathbf{module\ fresh} \ \mathrm{M}_M : \mathit{Sig} \ \ \mathbf{version} \ \ \mathit{vne} = \mathit{Str} \ \ \mathbf{with} \ !\mathit{weqs} \\ \text{no } \mathbf{cfresh}_T \\ E \vdash \mathit{vne} \ \ \mathbf{ok} \\ E \vdash_{\mathit{weqs}} \mathit{Str} : \mathit{Sig}\end{array}}{E \vdash \mathit{definition} \ \ \rhd \ \ \mathrm{M}_M : \mathit{Sig}} \ \bigstar$$

$$\frac{\begin{array}{l}\mathit{definition} = \mathbf{import\ fresh} \ \mathrm{M}_M : \mathit{Sig} \ \ \mathbf{version} \ \ \mathit{vce} \ \mathit{likespec} \ \ \mathbf{by} \ \ \mathit{resolvespec} = \mathit{Mo} \\ E \vdash \mathit{vce} \ \ \mathbf{ok} \\ E \vdash \mathit{likespec} \ \ \mathbf{ok} \\ E \vdash = \mathit{Mo} : \mathit{Sig}\end{array}}{E \vdash \mathit{definition} \ \ \rhd \ \ \mathrm{M}_M : \mathit{Sig}} \ \bigstar$$

where $\mathrm{eqs\_of\_sign\_str}(h, \mathit{Sig}_0, \mathit{Str}) = \{h.\mathrm{t} \approx T \mid \exists\, t.(\mathbf{type} \ \ \mathrm{t}_t : \mathrm{TYPE} \ \in \ \mathit{Sig}_0 \wedge (\mathbf{type} \ \ \mathrm{t}_t = T) \in \ \mathit{Str}\}$

Figure 17: Typing Rules – Compiled Definitions

Define linkok $(E_{\mathrm{n}}, \mathit{definition}', \mathit{definition})$ if

1. $\mathit{definition}$ is of the form $\mathbf{cimport}_{h;\mathit{Sig}_0}\ \mathit{vubs}\ \mathrm{M}_M : \mathit{Sig}_1\ \ \mathbf{version}\ \ \mathit{vc}\ \ \mathbf{like}\ \ \mathit{Str}\ \ \mathbf{by}\ \ \mathit{resolvespec} = \mathit{Mo}$
2. $\mathit{definition}'$ is either a $\mathbf{cmodule}$ or a $\mathbf{cimport}$ as below.

   $\mathbf{cmodule}_{h';\mathit{eqs}';\mathit{Sig}'_0}\ \mathit{vubs}'\ \mathrm{M}'_{M'} : \mathit{Sig}'_1\ \ \mathbf{version}\ \ \mathit{vn}' = \mathit{Str}'$
   $\mathbf{cimport}_{h';\mathit{Sig}'_0}\ \mathit{vubs}'\ \mathrm{M}'_{M'} : \mathit{Sig}'_1\ \ \mathbf{version}\ \ \mathit{vc}'\ \ \mathbf{like}\ \ \mathit{Str}'\ \ \mathbf{by}\ \ \mathit{resolvespec}' = \mathit{Mo}'$

3. the external names match: $\mathrm{M}' = \mathrm{M}$.
   It is unclear whether we always want to require the above.
4. the interfaces match: $E_{\mathrm{n}} \vdash_{\varnothing} \mathit{Sig}'_0 <: \mathit{Sig}_0$. In an implementation, we check only syntacticsubsig $\mathit{Sig}'_0\ \mathit{Sig}_0$.
5. the versions match:
   - Case: the vc is not an exact-name constraint, i.e. $\mathit{vc} = \mathit{dvc}$ for some dvc. If $\mathit{definition}'$ is a $\mathbf{cmodule}$ check $\mathit{vn}' \in \mathit{dvc}$, otherwise if $\mathit{definition}'$ is a $\mathbf{cimport}$ check $\mathit{vc}' \subseteq \mathit{vc}$.
   - Case: the vc is an exact-name constraint, i.e. $\mathbf{name} = \mathit{ahvc}$ for some ahvc. Check $h' \cong \mathit{ahvc}$.
6. the representation types match: $\forall (\mathbf{type}\ \ \mathrm{t}_t = T) \in \mathit{Str}. \exists\, t', T'. (\mathbf{type}\ \ \mathrm{t}_{t'} = T') \in \mathit{Str}' \wedge T = T'$.

Define linkok $(E_{\mathrm{n}}, \mathit{definitions})$ if whenever

$$\begin{array}{lcl} \mathit{definitions} & = & \mathit{definitions}_1\ \mathbf{;;}\ \mathit{definition}\ \mathbf{;;}\ \mathit{definitions}_2 \\ \mathit{definition} & = & \mathbf{cimport}_{h;\mathit{Sig}_0}\ \mathit{vubs}\ \mathrm{M}_M : \mathit{Sig}_1\ \ \mathbf{version}\ \ \mathit{vc}\ \ \mathbf{like}\ \ \mathit{Str}\ \ \mathbf{by}\ \ \mathit{resolvespec} = \mathrm{M}'_{M'} \end{array}$$

there exists a $\mathit{definition}'$ for $\mathrm{M}'_{M'}$ in $\mathit{definitions}_1$ with linkok $(E_{\mathrm{n}}, \mathit{definition}', \mathit{definition})$.
Define (on flattened signatures only) syntacticsubsig $\mathit{Sig}'_0\ \mathit{Sig}_0$, a weak version of $E_{\mathrm{n}} \vdash_{\varnothing} \mathit{Sig}'_0 <: \mathit{Sig}_0$:

$$\frac{\begin{array}{c} T' = T \\ \text{syntacticsubsig}\ \mathit{sig}'\ \mathit{sig} \end{array}}{\text{syntacticsubsig}\ (\mathbf{val}\ \ \mathrm{x}_x : T'\ \mathit{sig}')\ (\mathbf{val}\ \ \mathrm{x}_x : T\ \mathit{sig})} \qquad \frac{\text{syntacticsubsig}\ \mathit{sig}'\ \mathit{sig}}{\text{syntacticsubsig}\ (\mathbf{sig}\ \ \mathit{sig}'\ \ \mathbf{end})\ (\mathbf{sig}\ \ \mathit{sig}\ \ \mathbf{end})}$$

$$\frac{\text{syntacticsubsig}\ \mathit{sig}'\ \mathit{sig}}{\text{syntacticsubsig}\ (\mathbf{type}\ \ \mathrm{t}_t : \textsc{Type}\ \mathit{sig}')\ (\mathbf{type}\ \ \mathrm{t}_t : \textsc{Type}\ \mathit{sig})} \qquad \frac{}{\text{syntacticsubsig empty empty}}$$

$$\frac{\begin{array}{c} T' = T \\ \text{syntacticsubsig}\ \mathit{sig}'\ \mathit{sig} \end{array}}{\text{syntacticsubsig}\ (\mathbf{type}\ \ \mathrm{t}_t : \mathrm{EQ}(T')\ \mathit{sig}')\ (\mathbf{type}\ \ \mathrm{t}_t : \mathrm{EQ}(T)\ \mathit{sig})}$$

$$\frac{\text{syntacticsubsig}\ \mathit{sig}'\ (\{T'/t\}\mathit{sig})}{\text{syntacticsubsig}\ (\mathbf{type}\ \ \mathrm{t}_t : \mathrm{EQ}(T')\ \mathit{sig}')\ (\mathbf{type}\ \ \mathrm{t}_t : \textsc{Type}\ \mathit{sig})}$$

Comment: The substitution is required in the case of a concrete type on the left and an abstract on the right, in order that inequalities such as the following are treated correctly: $\mathbf{sig}\ \ \mathbf{type}\ \ t = \mathsf{int}\ \mathbf{type}\ \ u = \mathsf{int}\ \ \mathbf{end} <: \mathbf{sig}\ \ \mathbf{type}\ \ t\ \mathbf{type}\ \ u = t\ \ \mathbf{end}$. There is no need to apply the substitution in the other concrete case, because compilation has already flattened Sig'_0 and Sig_0. An implementation may avoid the type substitution by carrying a type environment. We don't check that the external value and type names are distinct, since compilation has already ensured that both signatures are well-formed.
Comment: Note that an implementation need not refer to E_{n} while doing the subsignature check. The E_{n} is required only to provide type equalities $\mathrm{n.t} : \mathrm{EQ}(T)$ for concrete type fields of freshly-named modules. But hashify (step 3, page 119) has ensured that all concrete type fields in Sig_0 have already been substituted out. Thus only abstract type field references remain; and in this case it is sufficient to assume $\mathrm{n.t} : \textsc{Type}$ for all n, t. The same reasoning confirms that an implementation need not inspect the body of a $\mathbf{hash}(..)$-form hash h.
Comment: Note that this does not permit a $\mathbf{cimport}$ to be linked to a $\mathbf{module}\ \ \mathbf{fresh}$. This is slightly unpleasant from the user's point of view, though such imports can usually be written with a HERE_ALREADY resolvespec. The restriction avoids the need to check subsignature or version of a $\mathbf{module}\ \ \mathbf{fresh}$, which (as they have no name) is problematic.

Figure 18: Link Checking

$\boxed{E\ \mathbf{;}\ E_s \vdash s\ \ \text{store}}$

$$\frac{\begin{array}{l}\text{dom}(E_s)\text{ contains only locations}\\ l\ \in \text{dom}(E_s)\ \iff\ l\ \in \text{dom}(s)\\ \forall\ l : T\ \ \mathsf{ref}\ \in\ E_s.E, E_s \vdash_{\varnothing} s(l) : T \wedge \text{compiledform}(s(l))\end{array}}{E\ \mathbf{;}\ E_s \vdash s\ \ \text{store}}\ \bigstar$$

$\boxed{E \vdash \mathit{definitions}\ \ \triangleright\ \ E'}$

$$\frac{E \vdash\ \mathbf{ok}}{E \vdash \text{empty}\ \ \triangleright\ \ \text{empty}} \qquad \frac{\begin{array}{l}E \vdash \mathit{definition}\ \ \triangleright\ \ E'\\ E, E' \vdash \mathit{definitions}\ \ \triangleright\ \ E''\\ \text{dom}(E'), \text{dom}(E'')\text{ disjoint}\end{array}}{E \vdash \mathit{definition}\ \mathbf{;;}\ \mathit{definitions}\ \ \triangleright\ \ E', E''}$$

$\boxed{E \vdash \mathit{definitions}\ eo\ \ \mathbf{ok}}$

$$\frac{E \vdash \mathit{definitions}\ \ \triangleright\ \ E'}{E \vdash \mathit{definitions}\ \ \mathbf{ok}} \qquad \frac{\begin{array}{l}E \vdash \mathit{definitions}\ \ \triangleright\ \ E'\\ E, E' \vdash e : T\end{array}}{E \vdash \mathit{definitions}\ e\ \ \mathbf{ok}}$$

$\boxed{\vdash E_{\text{n}}\ \mathbf{;}\ \langle E_s,\ s,\ \mathit{definitions},\ e\rangle : T}$

$$\frac{\begin{array}{l}E_{\text{n}}, E_{\text{const}}, E_s \vdash \mathit{definitions}\ \ \triangleright\ \ E\\ \text{linkok}\,(E_{\text{n}}, \mathit{definitions})\\ E_{\text{n}}, E_{\text{const}}, E\ \mathbf{;}\ E_s \vdash s\ \ \text{store}\\ E_{\text{n}}, E_{\text{const}}, E, E_s \vdash_{\varnothing} e : T\\ \text{compiledform}(e)\end{array}}{\vdash E_{\text{n}}\ \mathbf{;}\ \langle E_s,\ s,\ \mathit{definitions},\ e\rangle : T}\ \bigstar$$

$\boxed{\vdash E_{\text{n}}\ \mathbf{;}\ \langle E_s,\ s,\ \mathit{definitions},\ P\rangle : T}$

$$\frac{\begin{array}{l}E_{\text{n}}, E_{\text{const}}, E_s \vdash \mathit{definitions}\ \ \triangleright\ \ E\\ \text{linkok}\,(E_{\text{n}}, \mathit{definitions})\\ E_{\text{n}}, E_{\text{const}}, E\ \mathbf{;}\ E_s \vdash s\ \ \text{store}\\ E_{\text{n}}, E_{\text{const}}, E, E_s \vdash P\ \ \mathbf{ok}\\ \text{compiledform}(e)\end{array}}{\vdash E_{\text{n}}\ \mathbf{;}\ \langle E_s,\ s,\ \mathit{definitions},\ P\rangle : \mathsf{unit}}\ \bigstar$$

Figure 19: Typing Rules – Store, Configurations

$\boxed{\vdash \mathit{mv}\ \ \mathbf{ok}}$

$$\frac{\begin{array}{l} E_{\mathrm{n}}, E_{\mathrm{const}}, E_s \vdash \mathit{definitions}\ \ \triangleright\ \ E' \\ \mathrm{linkok}\,(E_{\mathrm{n}}, \mathit{definitions}) \\ E_{\mathrm{n}}, E_{\mathrm{const}}, E', E_s \vdash_{\varnothing}\ e : T \\ \mathrm{compiledform}(e) \\ E_{\mathrm{n}} \vdash_{\varnothing}\ T : \textsc{Type} \\ E_{\mathrm{n}}, E_{\mathrm{const}}, E'\ \mathbf{;}\ E_s \vdash s\ \ \text{store} \end{array}}{\vdash \mathbf{marshalled}(E_{\mathrm{n}},\ E_s,\ s,\ \mathit{definitions},\ e,\ T)\ \mathbf{ok}}\ \bigstar$$

Figure 20: Typing Rules – Marshalled Values

$\boxed{E \vdash P\ \ \mathbf{ok}}$

$$\frac{E \vdash\ \ \mathbf{ok}}{E \vdash 0\ \ \mathbf{ok}} \qquad \frac{\begin{array}{l} E \vdash P_1\ \ \mathbf{ok} \\ E \vdash P_2\ \ \mathbf{ok} \\ \mathsf{dom}(P_1) \cap \mathsf{dom}(P_2) = \varnothing \end{array}}{E \vdash P_1 | P_2\ \ \mathbf{ok}}$$

$$\frac{\begin{array}{l} E \vdash \mathbf{n} : \mathsf{thread\ \ name} \\ E \vdash \mathit{definitions}\ \ \triangleright\ \ E' \\ E, E' \vdash_{\varnothing}\ e : \mathsf{unit} \end{array}}{E \vdash (\mathbf{n} : \mathit{definitions}\ e)\ \mathbf{ok}} \qquad \frac{E \vdash \mathbf{n} : \mathsf{mutex\ \ name}}{E \vdash (\mathbf{n} : \mathrm{MX}(\underline{b}))\ \mathbf{ok}} \qquad \frac{E \vdash \mathbf{n} : \mathsf{cvar\ \ name}}{E \vdash (\mathbf{n} : \mathrm{CV})\ \mathbf{ok}}$$

Comment: The implementation optionally allows a non-unit thread for testing convenience.

Figure 21: Typing Rules – Processes

16.4 Typed Desugaring

Desugaring is a function that takes a source expression and yields a source expression with all sugared forms eliminated. It is applied after type inference and typechecking and before hashification. Thus we assume that type annotations have been inserted that did not appear in user source programs.

Desugaring behaves as follows:

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathrm{desugar}(e_1 \mid\mid\mid e_2)} \\
= & \textbf{create\_thread fresh}_{\mathsf{thread}}\ (\textbf{function}\ () \to \mathrm{desugar}(e_1))\ ()\ \textbf{;}\ \mathrm{desugar}(e_2) \\
\multicolumn{2}{l}{\mathrm{desugar}(\textbf{function}\ \ mtch)} \\
= & \textbf{function}\ (x : T) \to \textbf{match}\ \ x\ \ \textbf{with}\ \mathrm{desugarmtch}(mtch) \\
& \text{where } mtch \neq (x' : T' \to e),\ x \text{ fresh},\ T = \mathrm{matchty}\,(mtch) \\
\multicolumn{2}{l}{\mathrm{desugar}(\textbf{fun}\ \ p_1..p_n \to e')} \\
= & (\textbf{function}\ (x_1 : T_1) \to \textbf{match}\ \ x_1\ \ \textbf{with}\ \ p_1 \to\ .. \\
& \textbf{function}\ (x_n : T_n) \to \textbf{match}\ \ x_n\ \ \textbf{with}\ \ p_n \to \mathrm{desugar}(e')) \\
& \text{where } n \geq 1,\ T_i = \mathrm{patty}\,(p_i) \text{ for } 1 \leq i \leq n, \text{ and } x_1..x_n \text{ fresh} \\
\multicolumn{2}{l}{\mathrm{desugar}(\textbf{let}\ \ p = e_1\ \ \textbf{in}\ \ e_2)} \\
= & \textbf{match}\ \mathrm{desugar}(e_1)\ \textbf{with}\ \ p \to \mathrm{desugar}(e_2) \\
\multicolumn{2}{l}{\mathrm{desugar}(\textbf{let}\ \ x : T\ p_1..p_n = e'\ \ \textbf{in}\ \ e'')} \\
= & \textbf{match}\ (\textbf{function}\ (x_1 : T_1) \to \textbf{match}\ \ x_1\ \ \textbf{with}\ \ p_1 \to\ .. \\
& \textbf{function}\ (x_n : T_n) \to \textbf{match}\ \ x_n\ \ \textbf{with}\ \ p_n \to \mathrm{desugar}(e')) \\
& \textbf{with}\ (x : T) \to \mathrm{desugar}(e'') \\
& \text{where } n \geq 1,\ T_i = \mathrm{patty}\,(p_i) \text{ for } 1 \leq i \leq n, \text{ and } x_1..x_n \text{ fresh} \\
\multicolumn{2}{l}{\mathrm{desugar}(\textbf{let}\ \ \textbf{rec}\ \ x : T = \textbf{function}\ \ mtch\ \ \textbf{in}\ \ e)} \\
= & \textbf{let}\ \ \textbf{rec}\ \ x : T = \textbf{function}\ (x' : T') \to (\textbf{match}\ \ x'\ \ \textbf{with}\ \mathrm{desugarmtch}(mtch))\ \textbf{in}\ \mathrm{desugar}(e) \\
& \text{where } mtch \neq (x'' : T'' \to e'),\ x' \text{ fresh},\ T' = \mathrm{matchty}\,(mtch) \\
\multicolumn{2}{l}{\mathrm{desugar}(\textbf{let}\ \ \textbf{rec}\ \ x : T\ p_1..p_n = e'\ \ \textbf{in}\ \ e'')} \\
= & \textbf{let}\ \ \textbf{rec}\ \ x : T = (\textbf{function}\ (x_1 : T_1) \to \textbf{match}\ \ x_1\ \ \textbf{with}\ \ p_1 \to\ .. \\
& \textbf{function}\ (x_n : T_n) \to \textbf{match}\ \ x_n\ \ \textbf{with}\ \ p_n \to \mathrm{desugar}(e')) \\
& \textbf{in}\ \mathrm{desugar}(e'') \\
& \text{where } n \geq 1,\ T_i = \mathrm{patty}\,(p_i) \text{ for } 1 \leq i \leq n, \text{ and } x_1..x_n \text{ fresh} \\
\multicolumn{2}{l}{\mathrm{desugarmtch}(p_1 \to e_1 | .. | p_n \to e_n)} \\
= & p_1 \to \mathrm{desugar}(e_1) | .. | p_n \to \mathrm{desugar}(e_n)
\end{array}$$

Desugaring of structure items:

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathrm{desugar}(\textbf{let}\ \ \mathrm{x}_x : T\ p_1..p_n = e')} \\
= & \textbf{let}\ \ \mathrm{x}_x : T = \textbf{function}\ (x_1 : T_1) \to \textbf{match}\ \ x_1\ \ \textbf{with}\ \ p_1 \to\ .. \\
& \textbf{function}\ (x_n : T_n) \to \textbf{match}\ \ x_n\ \ \textbf{with}\ \ p_n \to \mathrm{desugar}(e')) \\
& \text{where } n \geq 1,\ T_i = \mathrm{patty}\,(p_i) \text{ for } 1 \leq i \leq n, \text{ and } x_1..x_n \text{ fresh}
\end{array}$$

Desugar also saturates operators, by performing eta-expansions.

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathrm{desugar}(e\ e_1\ ..\ e_m)} \\
= & \textbf{function}\ (x_{m+1} : T_{m+1}) \to .. \to \textbf{function}\ (x_n : T_n) \to (e\ e_1\ ..\ e_m\ x_{m+1}\ ..\ x_n) \\
& \text{where } e : T_1 \to ... \to T_n \to T_0 \text{ and } m < n
\end{array}$$

for e an op or $e \in E_{\mathrm{const}}$.

Likewise, we saturate partial applications of $(\texttt{:=}_T) : T\ \ \mathsf{ref} \to T \to \mathsf{unit}$ and $(!_T) : T\ \ \mathsf{ref} \to T$ and $(\&\&) : \mathsf{bool} \to \mathsf{bool} \to \mathsf{bool}$ and $(||) : \mathsf{bool} \to \mathsf{bool} \to \mathsf{bool}$.

In all other cases, it merely descends recursively into the term without altering its structure. In particular,

$$\text{desugar}(\textbf{function } (x : T) \to e) = \textbf{function } (x : T) \to \text{desugar}(e)$$

> *Comment:* We diverge from Ocaml in alternating between "function" and "match"; the expression 'let f (Some x) (y:int) = x+1 in f None' raises MATCH_FAILURE in Acute while Ocaml returns a thunk.

$\text{desugar}(.)$ makes use of a function $\text{matchty}\,(.)$ that determines the argument type of a match:

$$\begin{array}{lcll}
\text{matchty}\,(p \to e|mtch) & = & \text{matchty}\,(p \to e) & \\
\text{matchty}\,(p \to e) & = & \text{patty}\,(p) & \\
\text{patty}\,(\_ : T) & = & T & \\
\text{patty}\,(x : T) & = & T & \\
\text{patty}\,(\mathrm{C}_0) & = & T & \text{where } \mathrm{C}_0 : T \\
\text{patty}\,(\mathrm{C}_1\ p) & = & T' & \text{where } \mathrm{C}_1 : T \to T' \\
\text{patty}\,(p_1 :: p_2) & = & \text{patty}\,(p_1)\ \mathsf{list} & \\
\text{patty}\,(p_1, .., p_n) & = & \text{patty}\,(p_1) * .. * \text{patty}\,(p_n) & \\
\text{patty}\,(p : T) & = & T &
\end{array}$$

16.5 Errors

The possible outcomes of compilation and execution are as follows:

SUCCESS

DEADLOCK

LIBRARY

FAILURE

- COMPILE
  - LEX
  - PARSE
  - TYPE
  - INCLUDE
    - CYCLE
    - SYSTEM_ERROR
  - NONFINAL_EXPRESSION
  - HASHIFY
    - WITHSPEC_EQUATION_FROM_IMPORT
    - WITHSPEC_WRT_BAD_TYPE_FIELD
    - WITHSPEC_TYPES_NOT_EQUAL
    - LIKESPEC_MISSING_TYPE_FIELDS
    - LINKOK_NOT
    - BAD_SOURCEDEF_VALUABILITY
    - NENV_MERGE_OF_COMPILEDUNIT
  - TYPECHECK_OF_COMPILEDUNIT
  - RUNTIME_MISMATCH
- RUN
  - TYPECHECK_OF_CONFIGURATION
  - TYPECHECK_ON_MARSHAL
  - TYPECHECK_ON_UNMARSHAL
  - TYPECHECK_ON_GET_URI
- INTERNAL
  - NEVER_HAPPEN
  - STUCK
  - UNIMPLEMENTED

with the appropriate additional data (respectively a configuration, a set of definitions , or further information about the failure).

Of these, FAILURE.COMPILE.TYPECHECK_OF_COMPILEDUNIT, FAILURE.RUN.TYPECHECK_OF_CONFIGURATION, FAILURE.RUN.TYPECHECK_ON_UNMARSHAL, and FAILURE.RUN.TYPECHECK_ON_GET_URI, are signalled if a compile-time compiled module typecheck or a run-time typecheck fails. They should never happen (assuming that marshalled values and compiled files are not forged), and in a production implementation using numeric hashes these typechecks cannot be performed. They are therefore not currently mapped to internal Acute exceptions.

The FAILURE.INTERNAL.$*$ errors should never happen.

The FAILURE.COMPILE.RUNTIME_MISMATCH error occurs when we try and parse something (a marshalled value, an included file, or an imported file) which was created with an runtime incompatible with the current one — for example one created using literal hashing when we are using structured hashing. If a resolvespec meets this error Acute fails immediately rather than proceeding to the next resolvespec.

Only some kinds of deadlock are detected by Acute.

Individual threads may exit cleanly, be killed, or raise an exception; none of these (in themselves) result in program termination, although they may cause a debugging message to be written to the console.

16.6 Valuability helper functions

Comment: These definitions should be disentangled.

We define $\mathrm{find\_valuabilities}(\mathit{definitions}, \mathrm{M}_M)$ which looks up the valuabilities vubs of M_M; $\mathrm{check\_valuability\_expr}(\mathit{definitions}, e, \mathit{vub})$ which checks whether e can have valuability vub; and $\mathrm{derive\_valuabilities}(\mathit{definitions}, \mathit{sourcedefinition})$ which calculates the valuabilities vubs of the $\mathit{sourcedefinition}$ for $\mathit{sourcedefinition}$ not of the form **mark** MK.

As usual, we conflate the $\mathit{definitions}$ with a finite map C from module names to definitions.

First, we define $\mathrm{find\_valuabilities}(\mathit{definitions}, \mathrm{M}_M)$ as follows.

To do this, we perform case analysis on $C(\mathrm{M}_M)$ to calculate vubs':

- Case $\mathbf{cmodule}_{h;\mathit{eqs};\mathit{Sig}_0}\ \mathit{vubs}\ \mathrm{M}_M : \mathit{Sig}_1\ \ \mathbf{version}\ \ \mathit{vn} = \mathit{Str}$: let $\mathit{vubs}' = \mathit{vubs}$.
- Case $\mathbf{cimport}_{h;\mathit{Sig}_0}\ \mathit{vubs}\ \mathrm{M}_M : \mathit{Sig}_1\ \ \mathbf{version}\ \ \mathit{vc}\ \ \mathbf{like}\ \ \mathit{Str}\ \mathbf{by}\ \ \mathit{resolvespec} = \mathit{Mo}$: let $\mathit{vubs}' = \mathit{vubs}$.
- Case $\mathbf{module\ fresh}\ \mathrm{M}_M : \mathit{Sig}\ \ \mathbf{version}\ \ \mathit{vne} = \mathit{Str}\ \mathit{withspec}$: let $\mathit{vubs}' = (\mathrm{nonvaluable}, \mathrm{nonvaluable})$.
- Case $\mathbf{import\ fresh}\ \mathrm{M}_M : \mathit{Sig}\ \ \mathbf{version}\ \ \mathit{vce}\ \mathit{likespec}\ \ \mathbf{by}\ \mathit{resolvespec} = \mathit{Mo}$: let $\mathit{vubs}' = (\mathrm{nonvaluable}, \mathrm{nonvaluable})$.

Now we define $\mathrm{check\_valuability\_expr}(\mathit{definitions}, e, \mathit{vub})$, which checks where e can have valuability vub.

Comment: The valuabilities are linearly ordered, with valuable implying cvaluable and cvaluable implying nonvaluable.

Say a *cval context* is a linear expression context of the grammar defined from the $v^{\varnothing}$ grammar by taking all clauses with a sub-$v^{\varnothing}$ metaveriable, replacing one by a _ and all others by an e, i.e. :

$$
\begin{array}{lll}
\mathit{CVAL} & ::= & \_ \\
 & & \mathrm{c}_1\ \mathit{CVAL} \\
 & & \mathit{CVAL} :: e \\
 & & e :: \mathit{CVAL} \\
 & & (\mathit{CVAL}, .., e_n)..(e_1, .., \mathit{CVAL}) \\
 & & \mathbf{hash}(T, \mathit{CVAL})_{T'} \\
 & & \mathbf{hash}(\mathit{CVAL}, T, e)_{T'} \\
 & & \mathbf{hash}(e, T, \mathit{CVAL})_{T'} \\
 & & \{T, \mathit{CVAL}\}\ \mathbf{as}\ \ T'
\end{array}
$$

The cval contexts are used in the following to limit occurrences of **cfresh** to unguarded positions.

We perform case analysis on vub:

- Case $\mathit{vub} = \mathrm{valuable}$: check that
  - e is a term of the grammar consisting of
    - the clauses of the $v^{\varnothing}$ grammar
    - together with $\mathrm{M}_M.\mathrm{x}$
    - together with $\mathrm{M}_M@\mathrm{x}$
    - together with x (such as might occur if the expression is in a module structure and refers to an earlier field)

  (This list allows $\mathrm{M}_M.x$, etc., to occur in unguarded positions, which is not allowed if one confines e to just the value grammar.)

  - for all $\mathrm{M}_M.\mathrm{x}$ and $\mathrm{M}_M@\mathrm{x}$ occurring in e we have $\mathrm{fst}(\mathrm{find\_valuabilities}(\mathit{definitions}, \mathrm{M}_M)) = \mathrm{valuable}$;
  - for all $\mathrm{M}_M.\mathrm{t}$ occurring in e we have $\mathrm{snd}(\mathrm{find\_valuabilities}(\mathit{definitions}, \mathrm{M}_M)) = \mathrm{valuable}$.
  - there are no occurrences of $\mathbf{cfresh}_T$.
- Case $vub = \mathrm{cvaluable}$: check that
  - e is a term of the grammar consisting of the clauses of
    * the $v^{\varnothing}$ grammar
    * together with $\mathbf{cfresh}_T$ where all occurrences of $\mathbf{cfresh}_T$ give a decomposition of e of the form $\mathit{CVAL}.\mathbf{cfresh}_T$ (i.e. all are in unguarded positions)
    * together with $\mathrm{M}_M.\mathrm{x}$
    * together with $\mathrm{M}_M@\mathrm{x}$
    * together with x (i.e. an earlier field)
  - for all $\mathrm{M}_M.\mathrm{x}$ and $\mathrm{M}_M@\mathrm{x}$ occurring in e we have $\mathrm{fst}(\mathrm{find\_valuabilities}(\mathit{definitions}, \mathrm{M}_M)) \in \{\mathrm{valuable}, \mathrm{cvaluable}\}$;
  - for all $\mathrm{M}_M.\mathrm{t}$ occurring in e we have $\mathrm{snd}(\mathrm{find\_valuabilities}(\mathit{definitions}, \mathrm{M}_M)) \in \{\mathrm{valuable}, \mathrm{cvaluable}\}$.
- Case $vub = \mathrm{nonvaluable}$: check that all occurrences of $\mathbf{cfresh}_T$ give a decomposition of e of the form $\mathit{CVAL}.\mathbf{cfresh}_T$.

  Comment: Note that we impose conditions on the valuability of *all* $\mathrm{M}_M.\mathrm{x}$ and $\mathrm{M}_M@\mathrm{x}$ occurring in e, not just the ones in unguarded positions, since we need to be sure that we can replace these by an appropriate $h.\mathrm{x}$ in hashification during compilation.

Now we define $\mathrm{derive\_valuabilities}(\mathit{definitions}, \mathit{sourcedefinition})$ by case analysis on $\mathit{sourcedefinition}$:

- Case $\mathit{sourcedefinition} = (\mathbf{module}\ \mathit{mode}\ \mathrm{M}_M : \mathit{Sig}\ \ \mathbf{version}\ \ \mathit{vne} = \mathit{Str}\ \mathit{withspec})$. We consider several subcases:
  - Case $\mathit{mode} = \mathbf{hash}$: Check for all e on the rhs of Str we have $\mathrm{check\_valuability\_expr}(\mathit{definitions}, e, \mathrm{valuable})$. Check that for all $\mathrm{M'}_{M'}.\mathrm{t}$ occurring anywhere we have $\mathrm{snd}(\mathrm{find\_valuabilities}(\mathit{definitions}, \mathrm{M'}_{M'}))) = \mathrm{valuable}$. The result is $(\mathrm{valuable}, \mathrm{valuable})$.
  - Case $\mathit{mode} = \mathbf{hash!}$: Check for all $\mathrm{M}_M.\mathrm{x}$ and $\mathrm{M}_M@\mathrm{x}$ occurring in an e on the rhs of Str we have $\mathrm{fst}(\mathrm{find\_valuabilities}(\mathit{definitions}, \mathrm{M}_M)) = \mathrm{valuable}$. Check that for all $\mathrm{M'}_{M'}.\mathrm{t}$ occurring anywhere we have $\mathrm{snd}(\mathrm{find\_valuabilities}(\mathit{definitions}, \mathrm{M'}_{M'}))) = \mathrm{valuable}$. The result is $(\mathrm{valuable}, \mathrm{valuable})$.
  - Case $\mathit{mode} = \mathbf{cfresh}$: Check for all e on the rhs of Str we have $\mathrm{check\_valuability\_expr}(\mathit{definitions}, e, \mathrm{cvaluable})$. Check that for all $\mathrm{M'}_{M'}.\mathrm{t}$ occurring anywhere we have $\mathrm{snd}(\mathrm{find\_valuabilities}(\mathit{definitions}, \mathrm{M'}_{M'}))) \in \{\mathrm{valuable}, \mathrm{cvaluable}\}$.

    The result is $(\mathrm{cvaluable}, \mathrm{cvaluable})$
  - Case $\mathit{mode} = \mathbf{cfresh!}$: Check for all e on the rhs of Str: (a) for all $\mathrm{M}_M.\mathrm{x}$ and $\mathrm{M}_M@\mathrm{x}$ occurring in e we have $\mathrm{fst}(\mathrm{find\_valuabilities}(\mathit{definitions}, \mathrm{M}_M)) \in \{\mathrm{valuable}, \mathrm{cvaluable}\}$; and (b) all occurrences of $\mathbf{cfresh}_T$ give a decomposition of e of the form $\mathit{CVAL}.\mathbf{cfresh}_T$ (i.e. all are in unguarded positions). Check that for all $\mathrm{M'}_{M'}.\mathrm{t}$ occurring anywhere we have $\mathrm{snd}(\mathrm{find\_valuabilities}(\mathit{definitions}, \mathrm{M'}_{M'}))) \in \{\mathrm{valuable}, \mathrm{cvaluable}\}$. The result is $(\mathrm{cvaluable}, \mathrm{cvaluable})$
  - Case $\mathit{mode} = \mathbf{fresh}$: Check for all e on the rhs of Str we have $\mathrm{check\_valuability\_expr}(\mathit{definitions}, e, \mathrm{nonvaluable})$. The result is $(\mathrm{nonvaluable}, \mathrm{nonvaluable})$

    Comment: For the **hash**! case we do not ignore valuability checking altogether, as to build (at compile-time) a hash for this module requires names for any referenced modules and imports. There is an alternative possibility, deferring the hash construction until run-time if necessary, but it seems that that would be confusing (too different from the **hash** semantics). We do similarly for the **cfresh**! case.

- Case $\mathit{sourcedefinition} = (\mathbf{import}\ \mathit{mode}\ \mathrm{M}_M : \mathit{Sig}\ \mathbf{version}\ \mathit{vce}\ \mathit{likespec}\ \mathbf{by}\ \mathit{resolvespec} = \mathit{Mo})$:

  Comment: The valuabilities of $\mathrm{M}''_{M''}$, where $\mathit{Mo} = \mathrm{M}''_{M''}$, are unimportant.

  - Case $\mathit{mode} = \mathbf{hash}$ and $\mathit{mode} = \mathbf{hash}!$: We perform case analysis on the $\mathit{likespec}$:
    - Case $\mathit{likespec} = \mathrm{empty}$: true.
    - Case $\mathit{likespec} = \mathbf{like}\ \mathit{Str}$: Check that for all $\mathrm{M}'_{M'}.\mathrm{t}$ occurring anywhere in those fields of Str that are in the domain of $\mathrm{limitdom}\,(\mathit{Sig})$ we have $\mathrm{snd}(\mathrm{find\_valuabilities}(\mathit{definitions}, \mathrm{M}'_{M'})) = \mathrm{valuable}$.
    - Case $\mathit{likespec} = \mathbf{like}\ \mathrm{M}'_{M'}$: check $\mathrm{snd}(\mathrm{find\_valuabilities}(\mathit{definitions}, \mathrm{M}'_{M'})) = \mathrm{valuable}$

    The result is $(\mathrm{valuable}, \mathrm{valuable})$.
  - Case $\mathit{mode} = \mathbf{cfresh}$ and $\mathit{mode} = \mathbf{cfresh}!$: We perform case analysis on the $\mathit{likespec}$:
    - Case $\mathit{likespec} = \mathrm{empty}$: true.
    - Case $\mathit{likespec} = \mathbf{like}\ \mathit{Str}$: Check that for all $\mathrm{M}'_{M'}.\mathrm{t}$ occurring anywhere in those fields of Str that are in the domain of $\mathrm{limitdom}\,(\mathit{Sig})$ we have $\mathrm{snd}(\mathrm{find\_valuabilities}(\mathit{definitions}, \mathrm{M}'_{M'})) \in \{\mathrm{valuable}, \mathrm{cvaluable}\}$.
    - Case $\mathit{likespec} = \mathbf{like}\ \mathrm{M}'_{M'}$: check $\mathrm{snd}(\mathrm{find\_valuabilities}(\mathit{definitions}, \mathrm{M}'_{M'})) \in \{\mathrm{valuable}, \mathrm{cvaluable}\}$.

    The result is $(\mathrm{cvaluable}, \mathrm{cvaluable})$
  - Case $\mathit{mode} = \mathbf{fresh}$: The result is $(\mathrm{nonvaluable}, \mathrm{nonvaluable})$.

    Comment: We used to regard expression projections from an import as always nonvaluable (as there is no unique value they are guaranteed to reduce to, in the presence of rebinding, except in the exact-name version constraint case). Now, we regard the expression and type valuabilities as the same, and so could return to a single valuability rather than a pair throughout.
- Case $\mathit{sourcedefinition} = (\mathbf{module}\ \mathrm{M}_M : \mathit{Sig} = \mathrm{M}'_{M'})$: Check $C(\mathrm{M}'_{M'})$ is not of the form $\mathbf{cimport}\ {}_{;}$. The result is $\mathrm{find\_valuabilities}(\mathit{definitions}, \mathrm{M}'_{M'})$ if neither element of this pair is equal to nonvaluable.

If any of these checks fail, we have the exception COMPILE.HASHIFY.BAD_SOURCEDEF_VALUABILITY.

16.7 Compilation

Formally, compilation is a relation from a name environment E_{n}, a *sourcefilename*, and a filesystem Φ to either a tuple of a source type environment E_0, a compiled type environment E_1, and a *compiledunit*, or an error.

Note that *compiledunit* includes a name environment E_{n}: this environment contains **cfresh** names created during compilation. This name environment has no implementation significance: its sole purpose is to allow included compiled units to be appropriately typechecked and the configuration produced by compilation to be typechecked. These two checks are both necessary for runtime typechecking, but not otherwise.

Note that compilation is not a function because the choice of name environment in the *compiledunit* is nondeterministic. This nondeterminism is common in many of the helper "functions" throughout, thus we take them all to be relations. For convenience, though, we write them as functions of their inputs, and use $\rightsquigarrow$ rather than $=$ to relate the "input arguments" to the "results".

Compilation has the form

$$\mathrm{compile}_{\Phi}(\mathit{sourcefilename})E_{\mathrm{n}} \rightsquigarrow (E_0', E_1', \mathit{compiledunit}')$$

defined to be

$$\mathrm{compile}_{\Phi\ \varnothing}^{\mathrm{empty}\ E_{\mathrm{n}}\ E_{\mathrm{const}}\ E_{\mathrm{const}}}(\textbf{includesource}\ \ \mathit{sourcefilename}\ \textbf{;;}\ \mathrm{empty}) \rightsquigarrow (E_0', E_1', \mathit{compiledunit}')$$

where the latter relation

$$\mathrm{compile}_{\Phi\ \mathit{sourcefilenames}}^{\mathit{definitions}\ E_{\mathrm{n}}\ E_0\ E_1}(\mathit{compilationunit}) \rightsquigarrow (E_0', E_1', \mathit{compiledunit}')$$

is defined inductively on the $\mathit{compilationunit}$. Here $\mathit{sourcefilenames}$ is the filenames we've been through (used to detect cyclic includes), $\mathit{definitions}$ is the accumulated compiled definitions, E_{n} is the accumulated name environment (all names created during compilation will be disjoint from $\mathrm{dom}(E_{\mathrm{n}})$), E_0 is the accumulated source type environment (including E_{const} at the start), E_1 is the accumulated compiled type environment (including E_{const} at the start), and $\mathit{compilationunit}$ is what we have left to do.

It uses auxiliaries $\mathrm{E}_0(\mathit{definitions})$, $\mathrm{E}_1(\mathit{definitions})$, $\mathrm{derive\_valuabilities}(\mathit{definitions}, \mathit{sourcedefinition})$, $\rho^{\mathit{definitions}}$, $\mathrm{hashify\_ties\_and\_hashes}(\mathit{definitions},\ e)$, and $\mathrm{hashify}_{\mathit{definitions}}(E_{\mathrm{n}}, \mathit{sourcedefinition}, \mathit{vubs})$ defined in the rest of this section.

Consider the cases of $\mathit{compilationunit}$:

- Case empty.
  1. $(E_0, E_1, (E_{\mathrm{n}}, (\mathit{definitions}\ \ \mathrm{empty})))$.
- Case e.
  1. Check that for some T we have $E_0 \vdash_{\varnothing} e : T$ (otherwise COMPILE.TYPE).
  2. Calculate ${E_{\mathrm{n}}}' = E_{\mathrm{n}}$ and $e' = \mathrm{desugar}(e)$.
  3. An expression is just like a field in a fresh module. Thus at initialisation time, when we have shunted all modules across, we should then apply rho and rewrite $\mathrm{M}'_{M'}@x$, etc, in e'. For now we don't do that, so we have to be very conservative:
     - Check that e' has no **cfresh** subexpressions.
     - Check that for all all $\mathrm{M}'_{M'}@\mathrm{x}$ and all $\textbf{hash}(\mathrm{M}'_{M'}.\mathrm{x})_T$ subexpressions, $\mathrm{fst}(\mathrm{find\_valuabilities}(\mathit{definitions}, \mathrm{M}'_{M'})) \in \{\mathrm{valuable}, \mathrm{cvaluable}\}$.
     - Check that for all $\mathrm{M}'_{M'}.\mathrm{t}$, we have $\mathrm{snd}(\mathrm{find\_valuabilities}(\mathit{definitions}, \mathrm{M}'_{M'})) \in \{\mathrm{valuable}, \mathrm{cvaluable}\}$.
  4. $(E_0, E_1, ({E_{\mathrm{n}}}', (\mathit{definitions}\ \ (\mathrm{hashify\_ties\_and\_hashes}(\mathit{definitions},\ \rho^{\mathit{definitions}}(e'))))))$
- Case $(\mathit{sourcedefinition}\ \textbf{;;}\ \mathit{compilationunit})$.
  1. Check that for some E_0' we have $E_0 \vdash \mathit{sourcedefinition}\ \ \triangleright\ \ E_0'$ (otherwise COMPILE.TYPE).
  2. Now we perform case analysis on the structure of of $\mathit{sourcedefinition}$. Each case constructs ${E_{\mathrm{n}}}'$ and $\mathit{definition}'$.
     - Case $\mathit{sourcedefinition} = \textbf{mark}\ \ \mathrm{MK}$: Let $\mathit{definition}' = \textbf{mark}\ \mathrm{MK}$ and ${E_{\mathrm{n}}}' = E_{\mathrm{n}}$.
     - Otherwise:
       (a) Calculate vubs where $\mathit{vubs} = \mathrm{derive\_valuabilities}(\mathit{definitions}, \mathit{sourcedefinition})$ (otherwise COMPILE.BAD_SOURCEDEF_VALUABILITY).

(b) Now we do case analysis on mode:

  $*$ Case $\mathit{mode} \in \{\mathbf{hash}, \mathbf{cfresh}, \mathbf{hash!}, \mathbf{cfresh!}\}$: Calculate $(E_{\mathrm{n}}', \mathit{definition}')$ where $\mathrm{hashify}_{\mathit{definitions}}(E_{\mathrm{n}}, \mathrm{desugar}(\mathit{sourcedefinition}), \mathit{vubs}) \rightsquigarrow (E_{\mathrm{n}}', \mathit{definition}')$ (otherwise any of the COMPILE.HASHIFY.$*$).

  $*$ Case $\mathit{mode} = \mathbf{fresh}$: Calculate $E_{\mathrm{n}}' = E_{\mathrm{n}}$ and $\mathit{definition}' = \mathit{sourcedefinition}$.

3. Calculate $E_1' = \mathrm{E}_1(\mathit{definition}')$.

4. Calculate a result of $\mathrm{compile}_{\Phi\ \mathit{sourcefilenames}}^{(\mathit{definitions}\ ;;\ \mathit{definition}')\ E_{\mathrm{n}}'\ (E_0, E_0')\ (E_1, E_1')}(\mathit{compilationunit})$ (otherwise any of the compilation errors).

- Case $(\mathbf{includesource}\ \ \mathit{sourcefilename}\ ;;\ \mathit{compilationunit})$.

  1. Check $\mathit{sourcefilename} \notin \mathit{sourcefilenames}$ (otherwise COMPILE.INCLUDE.CYCLE).
  2. Look up $\mathit{compilationunit}' = \Phi(\mathit{sourcefilename})$ (otherwise COMPILE.INCLUDE.SYSTEM_ERROR)
  3. Calculate $(E_0', E_1', (E_{\mathrm{n}}', \mathit{definitions}'\ \mathit{eo}'))$ where $\mathrm{compile}_{\Phi\ (\mathit{sourcefilenames}, \mathit{sourcefilename})}^{\mathit{definitions}\ E_{\mathrm{n}}\ E_0\ E_1}(\mathit{compilationunit}') \rightsquigarrow (E_0', E_1', (E_{\mathrm{n}}', \mathit{definitions}'\ \mathit{eo}'))$ (otherwise any of the compilation errors).
  4. Check $\mathit{eo}' = \mathrm{empty} \vee \mathit{compilationunit} = \mathrm{empty}$ (otherwise COMPILE.NONFINAL_EXPRESSION).
  5. Calculate a result of $\mathrm{compile}_{\Phi\ \mathit{sourcefilenames}}^{\mathit{definitions}'\ E_{\mathrm{n}}'\ E_0'\ E_1'}(\mathit{compilationunit}\ ;;\ \mathit{eo}')$ (otherwise any of the compilation errors).

- Case $(\mathbf{includecompiled}\ \ \mathit{compiledfilename}\ ;;\ \mathit{compilationunit})$

  1. Look up $(E_{\mathrm{n}}', (\mathit{definitions}'\ \ \mathit{eo}')) = \Phi(\mathit{compiledfilename})$ (otherwise COMPILE.INCLUDE.SYSTEM_ERROR)
  2. Check $\mathit{eo}' = \mathrm{empty} \vee \mathit{compilationunit} = \mathrm{empty}$ (otherwise COMPILE.NONFINAL_EXPRESSION).
  3. If using structured hashes,
     - Check $E_{\mathrm{n}}' \vdash \mathit{definitions}'\ \mathit{eo}'\ \ \mathbf{ok}$.
     - Check that eo' has no $\mathbf{cfresh}$ subexpressions.
     - Check that eo' has no ties $\mathrm{M}'_{M'}@\mathrm{x}$ and no unhashified hashes $\mathbf{hash}(\mathrm{M}'_{M'}.\mathrm{x})_T$.

     (otherwise COMPILE.TYPECHECK_OF_COMPILEDUNIT).
  4. Let $E_{\mathrm{n}}'' = \mathrm{merge\_nenvs}(E_{\mathrm{n}}, E_{\mathrm{n}}')$, raising COMPILE.NENV_MERGE_OF_COMPILEDUNIT in case of an error. The funtion merge_nenvs merges two name environments, checking that their ranges are identical where their domains intersect.
  5. Let $E_0' = \mathrm{E}_0(\mathit{definitions}')$.
  6. Let $E_1' = \mathrm{E}_1(\mathit{definitions}')$.
  7. Calculate a result of $\mathrm{compile}_{\Phi\ \mathit{sourcefilenames}}^{\mathit{definitions}\ ;;\ \mathit{definitions}'\ E_{\mathrm{n}}''\ (E_0, E_0')\ (E_1, E_1')}(\mathit{compilationunit}\ ;;\ \mathit{eo}')$ (otherwise any of the compilation errors).

Note that compilation as defined here operates on abstract syntax. The implementation operates on bytestrings, and so can result also in FAILURE.COMPILE.LEX and FAILURE.COMPILE.PARSE. We do not specify where these can arise in any more detail.

The definition is given rather algorithmically – it might be nice to rephrase it in a way that makes explicit use of type normalization.

Now we describe the helper functions and relations.

$\mathrm{E}_0(\mathit{definitions})$ and $\mathrm{E}_1(\mathit{definitions})$ extract the bindings for, respectively, the source and compiled signature of a definition

$$\begin{array}{lcl}
\mathrm{E}_i(\mathbf{cmodule}_{h;eqs;Sig_0}\ \mathit{vubs}\ \mathrm{M}_M : \mathit{Sig}_1\ \mathbf{version}\ \ \mathit{vn} = \mathit{Str}\ \mathbf{;;}\ \mathit{definitions}) & = & \mathrm{M}_M : \mathit{Sig}_i, \mathrm{E}_i(\mathit{definitions})\\
\mathrm{E}_i(\mathbf{cimport}_{h;Sig_0}\ \mathit{vubs}\ \mathrm{M}_M : \mathit{Sig}_1\ \mathbf{version}\ \ \mathit{vc}\ \mathbf{like}\ \mathit{Str}\ \mathbf{by}\ \mathit{resolvespec} = \mathit{Mo}\ \mathbf{;;}\ \mathit{definitions}) & = & \mathrm{M}_M : \mathit{Sig}_i, \mathrm{E}_i(\mathit{definitions})\\
\mathrm{E}_i(\mathbf{module\ fresh}\ \mathrm{M}_M : \mathit{Sig}\ \ \mathbf{version}\ \ \mathit{vne} = \mathit{Str}\ \mathit{withspec}\ \mathbf{;;}\ \mathit{definitions}) & = & \mathrm{M}_M : \mathit{Sig}, \mathrm{E}_i(\mathit{definitions})\\
\mathrm{E}_i(\mathbf{import\ fresh}\ \mathrm{M}_M : \mathit{Sig}\ \ \mathbf{version}\ \ \mathit{vce}\ \mathit{likespec}\ \ \mathbf{by}\ \mathit{resolvespec} = \mathit{Mo}\ \mathbf{;;}\ \mathit{definitions}) & = & \mathrm{M}_M : \mathit{Sig}, \mathrm{E}_i(\mathit{definitions})\\
\mathrm{E}_i(\mathbf{mark}\ \ \mathrm{MK}\ \mathbf{;;}\ \mathit{definitions}) & = & \mathrm{E}_i(\mathit{definitions})\\
\mathrm{E}_i(\mathrm{empty}) & = & \mathrm{empty}
\end{array}$$

In what follows we let C range over $\mathit{definitions}$.

The substitution $\rho^{\mathit{definitions}}$, or ρ^C, is defined by

$$\begin{array}{ll}
& \left\{ \mathrm{M'}_{M'}.\mathrm{t} \mapsto T \;\middle|\; \begin{array}{l} C(\mathrm{M'}_{M'}) = \mathbf{cmodule}_{h';eqs';Sig'_0}\ \mathit{vubs}'\ \mathrm{M'}_{M'} : \mathit{Sig}'_1\ \ \mathbf{version}\ \ \mathit{vn}' = \mathit{Str}' \\ \wedge(\mathbf{type}\ \ \mathrm{t}_t : \mathrm{EQ}(T)) \in\ \mathit{Sig}'_1\ \ \mathbf{end} \end{array} \right\}\\
\cup & \left\{ \mathrm{M'}_{M'}.\mathrm{t} \mapsto T \;\middle|\; \begin{array}{l} C(\mathrm{M'}_{M'}) = \mathbf{cimport}_{h';Sig'_0}\ \mathit{vubs}'\ \mathrm{M'}_{M'} : \mathit{Sig}'_1\ \ \mathbf{version}\ \ \mathit{vc}'\ \ \mathbf{like}\ \ \mathit{Str}'\ \ \mathbf{by}\ \ \mathit{resolvespec}' = \mathit{Mo}' \\ \wedge(\mathbf{type}\ \ \mathrm{t}_t : \mathrm{EQ}(T)) \in\ \mathit{Sig}'_1\ \ \mathbf{end} \end{array} \right\}
\end{array}$$

> *Comment:* ρ^C does have any affect on $\mathrm{M'}_{M'}.\mathrm{t}$ if $\mathrm{M'}_{M'}$ is a **module fresh** or **import fresh** definition in C. These cases will never arrive (thanks to valuability checking) when ρ^C is used in hashification.

The relation evalcfresh nondeterministically transforms a pair of a name environment and an expression. This expression is a value modulo the presence of **cfresh**s in cval contexts.

$$\mathrm{evalcfresh}(E_\mathrm{n}, e) \rightsquigarrow (E_\mathrm{n}{}', e')$$

It replaces all **cfresh** used in a cval context (see §**??**) by a fresh name and is extended in a standard way to structures:

$$\begin{array}{lcll}
\mathrm{evalcfresh}(E_\mathrm{n}, \mathbf{cfresh}_T) & \rightsquigarrow & ((E_\mathrm{n}, \mathrm{n} : T\ \ \mathsf{name}), \mathrm{n}) & \text{for } n\ \notin \mathrm{dom}(E_\mathrm{n})\\
\mathrm{evalcfresh}(E_\mathrm{n}, \mathrm{C}_1\ e) & \rightsquigarrow & (E_\mathrm{n}{}', \mathrm{C}_1\ e') & \text{for } \mathrm{evalcfresh}(E_\mathrm{n}, e) \rightsquigarrow (E_\mathrm{n}{}', e')\\
\mathrm{evalcfresh}(E_\mathrm{n}, \mathbf{hash}(T, e)) & \rightsquigarrow & (E_\mathrm{n}{}', \mathbf{hash}(T, e')) & \text{ditto}\\
\mathrm{evalcfresh}(E_\mathrm{n}, \{T, e\}\ \mathbf{as}\ \ T') & \rightsquigarrow & (E_\mathrm{n}{}', \{T, e'\}\ \mathbf{as}\ \ T') & \text{ditto}\\
\mathrm{evalcfresh}(E_\mathrm{n}, e_1 :: e_2) & \rightsquigarrow & (E'_{\mathrm{n}2}, e'_1 :: e'_2) & \text{for } \mathrm{evalcfresh}(E_\mathrm{n}, e_1) \rightsquigarrow (E'_{\mathrm{n}1}, e'_1)\\
 & & & \quad \mathrm{evalcfresh}(E'_{\mathrm{n}1}, e_2) \rightsquigarrow (E'_{\mathrm{n}2}, e'_2)\\
\mathrm{evalcfresh}(E_\mathrm{n}, \mathbf{hash}(T, e_1, e_2)_{T'}) & \rightsquigarrow & (E'_{\mathrm{n}2}, \mathbf{hash}(T, e'_1, e'_2)_{T'}) & \text{ditto}\\
\mathrm{evalcfresh}(E_\mathrm{n}, (e_1, ..., e_k)) & \rightsquigarrow & (E'_{\mathrm{n}k}, (e'_1, ..., e'_k)) & \text{for } E_{\mathrm{n}0} = E_\mathrm{n}\\
 & & & \quad \mathrm{evalcfresh}(E_{\mathrm{n}0}, e_1) \rightsquigarrow (E'_{\mathrm{n}1}, e'_1)\\
 & & & \quad ...\\
 & & & \quad \mathrm{evalcfresh}(E'_{\mathrm{n}(k-1)}, e_k) \rightsquigarrow (E'_{\mathrm{n}k}, e'_2)\\
\mathrm{evalcfresh}(E_\mathrm{n}, \mathbf{struct}\ \ \mathit{str}\ \ \mathbf{end}) & \rightsquigarrow & (E_\mathrm{n}{}', \mathbf{struct}\ \ \mathit{str}'\ \ \mathbf{end}) & \text{for } \mathrm{evalcfresh}(E_\mathrm{n}, \mathit{str}) \rightsquigarrow (E_\mathrm{n}{}', \mathit{str}')\\
\mathrm{evalcfresh}(E_\mathrm{n}, (\mathbf{let}\ \ \mathrm{x}_x = e)\ \mathit{str}) & \rightsquigarrow & (E_\mathrm{n}{}'', (\mathbf{let}\ \ \mathrm{x}_x = e')\ \mathit{str}') & \text{for } \mathrm{evalcfresh}(E_\mathrm{n}, e) \rightsquigarrow (E_\mathrm{n}{}', e')\\
 & & & \quad \mathrm{evalcfresh}(E_\mathrm{n}{}', \mathit{str}) \rightsquigarrow (E_\mathrm{n}{}'', \mathit{str}')\\
\mathrm{evalcfresh}(E_\mathrm{n}, (\mathbf{type}\ \ \mathrm{t}_t = T)\ \mathit{str}) & \rightsquigarrow & (E_\mathrm{n}{}', (\mathbf{type}\ \ \mathrm{t}_t = T)\ \mathit{str}') & \text{for } \mathrm{evalcfresh}(E_\mathrm{n}, \mathit{str}) \rightsquigarrow (E_\mathrm{n}{}', \mathit{str}')
\end{array}$$

Now we define a helper function $\mathrm{hashify\_ties\_and\_hashes}(C,\ e)$ which compiles all ties and hashes as follows: in e each $\mathrm{M'}_{M'}@\mathrm{x}$ is replaced by $\textsc{TieCon}(\mathbf{hash}(\sigma^C(\mathrm{M'}_{M'}.\mathrm{x}))_{T'}, \mathrm{M'}_{M'}.\mathrm{x})$ and each $\mathbf{hash}(\mathrm{M'}_{M'}.\mathrm{x})_T$ is replaced by $\mathbf{hash}(\sigma^C(\mathrm{M'}_{M'}.\mathrm{x}))_T$, where σ is defined below and T is the type of $\mathrm{M'}_{M'}.\mathrm{x}$ in the relevant signature in C. We extend the domain to structures, $\mathrm{hashify\_ties\_and\_hashes}(C,\ \mathit{str})$, by applying the expression-level version pointwise to the fields of str.

The relation $\mathrm{hashify}_C(E_\mathrm{n}, \mathit{sourcedefinition}, \mathit{vubs})$ is defined by case analysis as follows. Here C is a list of $\mathit{definitions}$, which we also regard as a partial function mapping each M_M to a $\mathit{definition}$. It returns a new $E_\mathrm{n}{}'$ (containing E_n and any new names) and a $\mathit{definition}$, or fails with one of FAILURE.COMPILE.HASHIFY.$*$.

Note that it is used

- during compilation for **hash** and **cfresh** modules and imports;
- during run time for initialisation of **fresh** modules and imports.

We consider the following cases:

- Case $\mathit{sourcedefinition} = \textbf{module}\ \mathit{mode}\ \mathrm{M}_M : \mathit{Sig}\ \ \textbf{version}\ \ \mathit{vne} = \mathit{Str}\ \mathit{withspec}$, where $\mathit{Sig} = \textbf{sig}\ \ \mathit{sig}\ \ \textbf{end}$ and $\mathit{Str} = \textbf{struct}\ \ \mathit{str}\ \ \textbf{end}$.

  (Convention: things subscripted by n are roughly the result of the n-th step.)

  1. Remove other-module type dependencies with ρ^C. Let $(\mathit{str}_1 : \mathit{sig}_1) = \rho^C(\mathit{str} : \mathit{sig})$.

     This replaces references to $\mathrm{M'}_{M'}.\mathrm{t}$ by the manifest type from the compiled module or import signature – and in compiled modules and imports, all types in the Sig_1 have been made manifest.

  2. Let $\mathit{str}'_1 = \mathrm{hashify\_ties\_and\_hashes}(C,\ \mathit{str}_1)$.

  3. Then, remove same-module type dependencies, i.e. references t to previous type fields, as far as possible, with $\mathrm{typeflattenstruct}(\ )$ and $\mathrm{typeflattensig}(\ )$.

     $$\begin{aligned}&\mathrm{typeflattenstruct}(\textbf{type}\ \ \mathrm{t}_t = T\ \mathit{str}) = (\textbf{type}\ \ \mathrm{t}_t = T)\ \ \mathrm{typeflattenstruct}(\{T/t\}\mathit{str}))\\ &\mathrm{typeflattenstruct}(\textbf{let}\ \ \mathrm{x}_x = v^{\mathit{eqs}}\ \mathit{str}) = (\textbf{let}\ \ \mathrm{x}_x = v^{\mathit{eqs}})\ \ \mathrm{typeflattenstruct}(\mathit{str})\\ &\mathrm{typeflattenstruct}(\mathrm{empty}) = \mathrm{empty}\end{aligned}$$

     $$\begin{aligned}&\mathrm{typeflattensig}(\textbf{type}\ \ \mathrm{t}_t : \mathrm{TYPE}\ \mathit{sig}) = (\textbf{type}\ \ \mathrm{t}_t : \mathrm{TYPE})\ \ \mathrm{typeflattensig}(\mathit{sig})\\ &\mathrm{typeflattensig}(\textbf{type}\ \ \mathrm{t}_t : \mathrm{EQ}(T)\ \mathit{sig}) = (\textbf{type}\ \ \mathrm{t}_t : \mathrm{EQ}(T))\ \ \mathrm{typeflattensig}(\{T/t\}\mathit{sig})\\ &\mathrm{typeflattensig}(\textbf{val}\ \ \mathrm{x}_x : T\ \mathit{sig}) = (\textbf{val}\ \ \mathrm{x}_x : T)\ (\mathrm{typeflattensig}(\mathit{sig}))\\ &\mathrm{typeflattensig}(\mathrm{empty}) = \mathrm{empty}\end{aligned}$$

     Let $\mathit{str}_2 : \mathit{sig}_2 = \mathrm{typeflattenstruct}(\mathit{str}'_1) : \mathrm{typeflattensig}(\mathit{sig}_1)$.

     This $\mathrm{typeflattensig}(\ )$ leaves internal references to abstract type fields – that is forced, as we cannot yet calculate the h required to build their replacements.

     This type normalisation (both the ρ and type flattening here, and selfification below) amounts to treating modules up to type equality when making **hmodule** s and **nmodule** s, instead of using exactly the abstract syntax. Working up to type equality seems intuitively preferable, though it makes compilation seem even more algorithmic.

  4. Generate **cfresh** names if needed. We proceed by case analysis on mode and establish str_3 and $E_{\mathrm{n}3}$:

     – Case $\mathit{mode} \in \{\textbf{hash}, \textbf{fresh}, \textbf{hash!}\}$:

     Let str_3 = str_2 and $E_{\mathrm{n}3} = E_\mathrm{n}$.

     Comment: Note that hashify (, ,) is called at run time, not compile time, for **module fresh**. For such modules, the **cfresh** subexpressions are thus eliminated by compilation and not here.

     – Case $\mathit{mode} = \textbf{cfresh}, \textbf{cfresh!}$:

     Calculate $E_{\mathrm{n}3}, \mathit{str}_3$ by applying evalcfresh to eliminate all **cfresh** expressions: $\mathrm{evalcfresh}(E_\mathrm{n}, \mathit{str}_2) \rightsquigarrow (E_{\mathrm{n}3}, \mathit{str}_3)$.

5. Check the $\mathit{withspec}$ (see-through semantics).

   Suppose $\mathit{withspec} = \ \mathbf{with}\ !\mathit{weqs}$.

   Let $\mathit{eqs} = \{\mathrm{M'}_{M'}.\mathrm{t} \approx \rho^C\, T | \mathrm{M'}_{M'}.\mathrm{t} \approx T \ \in\ \mathit{weqs}\}$.

   For all $\mathrm{M'}_{M'}.\mathrm{t} \approx T \ \in\ \mathit{weqs}$, if $C(\mathrm{M'}_{M'})$ is an import then fail (WITHSPEC_EQUATION_FROM_IMPORT). Otherwise, suppose

   $$C(\mathrm{M'}_{M'}) = \mathbf{cmodule}_{h';\mathit{eqs}';\mathit{Sig}'_0}\ \mathit{vubs}'\ \mathrm{M'}_{M'} : \mathit{Sig}'_1\ \ \mathbf{version}\ \ \mathit{vn}' = \mathit{Str}' \wedge (\mathbf{type}\ \ \mathrm{t}_t : \mathrm{TYPE}) \in\ \mathit{Sig}'_0 \\ \wedge(\mathbf{type}\ \ \mathrm{t}_t = \mathit{TrepfromC}) \in\ \mathit{Str}'$$

   Comment: $\mathrm{M'}_{M'}.\mathrm{t}$ exists and is abstract, since the source definition is well-typed. It's unclear whether abstractness is forced but there does not seem to be any reason to permit seeing through a non-abstract type.

   Check $\mathit{TrepfromC} = \rho^C\, T$ (or fail with WITHSPEC_TYPES_NOT_EQUAL).

   Comment: $\mathit{TrepfromC}$ was already closed by the ρ from an earlier stage, so applying (the current) ρ to the T in weqs means that syntactic type equality is the appropriate check.

   Comment: There are two possible semantics for see-through **with** !. Currently we permit eqs to be used anywhere in typing the structure part; alternatively one could allow it to be used only in a final subsignature step. Unclear which is preferable in practice.

   We need to construct a closed set of equations for use inside the **nmodule** or **hmodule** .

   Let $\mathit{eqs}' = \{\rho^C(\mathrm{M'}_{M'}.\mathrm{t}) \approx \rho^C\, T | (\mathrm{M'}_{M'}.\mathrm{t} \approx T) \in\ \mathit{weqs}\}$.

6. Remove other-module term dependencies with $\sigma^C =$

$$\left\{ \mathrm{M'}_{M'}.\mathrm{x} \mapsto h'.\mathrm{x} \ \middle|\ \begin{array}{l} C(\mathrm{M'}_{M'}) = \mathbf{cmodule}_{h';\mathit{eqs}';\mathit{Sig}'_0}\ \mathit{vubs}'\ \mathrm{M'}_{M'} : \mathit{Sig}'_1\ \ \mathbf{version}\ \ \mathit{vn}' = \mathit{Str}' \\ \wedge(\mathbf{val}\ \ \mathrm{x}_x : T) \in\ \mathit{Sig}'_0 \end{array} \right\}$$
$$\cup\ \left\{ \mathrm{M'}_{M'}.\mathrm{x} \mapsto h'.\mathrm{x} \ \middle|\ \begin{array}{l} C(\mathrm{M'}_{M'}) = \mathbf{cimport}_{h';\mathit{Sig}'_0}\ \mathit{vubs}'\ \mathrm{M'}_{M'} : \mathit{Sig}'_1\ \ \mathbf{version}\ \ \mathit{vc}'\ \ \mathbf{like}\ \ \mathit{Str}'\ \ \mathbf{by}\ \ \mathit{resolvespec}' = \mathit{Mo}' \\ \wedge(\mathbf{val}\ \ \mathrm{x}_x : T) \in\ \mathit{Sig}'_0 \end{array} \right\}$$

   Let $\mathit{str}_5 = \sigma^C \mathit{str}_3$.

7. We do case analysis on the mode, calcuting h and $E_{\mathrm{n}}{}'$.

   – Case $\mathit{mode} = \mathbf{hash}$:

     Let $E_{\mathrm{n}}{}' = E_{\mathrm{n}3}$ and $h = \mathbf{hash}(\mathbf{hmodule}\ \ \mathit{eqs}'\ \mathrm{M} : \mathbf{sig}\ \ \mathit{sig}_2\ \ \mathbf{end}\ \ \mathbf{version}\ \ \mathit{vne} = \mathbf{struct}\ \mathit{str}_5\ \ \mathbf{end})$.

     Comment: A possible alternative semantics would be to substitute the eqs out in the body of the hash, making hash equality slightly coarser. It is unclear whether that would be preferable or not.

   – Case $\mathit{mode} = \mathbf{cfresh}, \mathbf{fresh}$:

     Let n be a fresh name not in the domain of $E_{\mathrm{n}3}$ (to serve as the new name of this module). We extend $E_{\mathrm{n}3}$ to $E_{\mathrm{n}}{}'$ accordingly:

     $$E_{\mathrm{n}}{}' = E_{\mathrm{n}3}, \mathrm{n} : \mathbf{nmodule}_{\mathit{eqs}'}\ \mathrm{M} : \mathbf{sig}\ \ \mathit{sig}_2\ \ \mathbf{end}\ \ \mathbf{version}\ \ \mathit{vne} = \mathbf{struct}\ \ \mathit{str}_5\ \ \mathbf{end}$$

     Let $h = \mathrm{n}$.

8. Selfify with respect to that h, to remove same-module type references. (selfifysig ( ) is defined on page **??**.)

Let $\mathit{sig}_7 = \mathrm{typeflattensig}(\mathrm{selfifysig}_h(\mathit{sig}_2))$.

Comment: Note that **hmodule** and **nmodule** generation happens before abstract-type selfification (and evaluation of vne). That is forced, as we need the h to selfify abstract type components (and to do that evaluation). However, manifest type components do get substituted out before.

Comment: Stylistic choice: you have to flatten the sig again, either in the definition of $\mathrm{selfifysig}_h(\ )$ (as we used to), or with $\mathrm{typeflattensig}(...)$ again. We do the latter, so that $\mathrm{selfifysig}_h(\ )$ can be used in the module alias typing rule.

Comment: In previous versions, and in [Sew01], selfify not only replaced $\mathrm{T{\small YPE}}$ by $\mathrm{EQ}(h)$ in the signature, but also replaced t by $h.\mathrm{t}$ in the structure (letting h range over hashes and new names). In [Sew01] that was because functors took type fields and term fields from their argument *struct*, not their argument sig, and so to not replace t by $h.\mathrm{t}$ would have been broken. An unfortunate consequence of doing that $\{h.\mathrm{t}/t\}$ in the struct, however, is that you need to keep the t elsewhere (in [Sew01], formally in the global type environment) for representation type checking of **with** !. Now we realise that that was not really forced. As signatures have always been fully EQified before you apply a functor (one case) or construct a ρ^C to use later (the other case), we can have both functors and ρ pull out type fields from sigs instead of structs.

Comment: With respect to marshalling (or fresh name generation etc.) inside an abstraction boundary (cf. §8.5), however, doing $\{h.\mathrm{t}/t\}$ in the structure might well be preferable. That would require changes to the construction of eqs, for which one might want to do the substitution in expression fields but not in the definitions of abstract types.

9. Evaluate the version number expression.

   Let $\mathit{vn} = \{h/\mathbf{myname}\}(\mathit{vne})$.

10. Finally, put that all together, writing in the h and the equations.

    Let $\mathit{definition}' = \mathbf{cmodule}_{h;\mathit{eqs}';\mathbf{sig}\ \mathit{sig}_2\ \mathbf{end}}\ \mathit{vubs}\ \mathrm{M}_M\ :\ \mathbf{sig}\ \ \mathit{sig}_7\ \ \mathbf{end}\ \mathbf{version}\ \ \mathit{vn}\ =\ \mathbf{struct}\ \ \mathit{str}_2\ \ \mathbf{end}$ and let $E_\mathrm{n}{}'$ be as built above.

- Case $\mathit{sourcedefinition} = \textbf{import}\ \mathit{mode}\ \mathrm{M}_M : \mathit{Sig}\ \ \textbf{version}\ \ \mathit{vce}\ \mathit{likespec}\ \ \textbf{by}\ \ \mathit{resolvespec} = \mathit{Mo}$ and $\mathit{Sig} = \textbf{sig}\ \ \mathit{sig}\ \ \textbf{end}$.

  Comment: Note we have an h subscript on $\textbf{cimport}_{\,;}$ s too, for convenience during compilation.

  1. Normal case: the vce is not an exact-name constraint, ie $\mathit{vce} = \mathit{dvce}$ for some dvce.

     (a) Calculate a $\mathit{likestr}'$ without internal type dependencies or other-module type dependencies. There are three cases: either $\mathit{likespec}$ was empty, or an in-line structure, or a module identifier (in the last case, we allow import-bound identifiers, as there seems no reason to exclude them).

     – Case $\mathit{likespec} = \mathrm{empty}$. If this import is linked to $\mathrm{M}'_{M'}$, then the empty $\mathit{likespec}$ defaults to $\textbf{like}\ \mathrm{M}'_{M'}$ (see below). Otherwise, take $\mathit{likestr} = \mathrm{empty}$.

     – Case $\mathit{likespec} = \ \textbf{like}\ \ \textbf{struct}\ \ \mathit{str}\ \ \textbf{end}$.

       Use the auxiliary $\mathrm{typeflattenstruct}(\ )$ to substitute out occurences of internal type field names t within str.

       Let $\mathit{likestr} = \mathrm{typeflattenstruct}(\rho^C\, \mathit{str})$.

     – Case $\mathit{likespec} = \ \textbf{like}\ \ \mathrm{M}'_{M'}$. Either

$$\begin{array}{lcl} C(\mathrm{M}'_{M'}) & = & \textbf{cmodule}_{h';\mathit{eqs}';\mathit{Sig}'_0}\ \mathit{vubs}'\ \mathrm{M}'_{M'} : \mathit{Sig}'_1\ \ \textbf{version}\ \ \mathit{vn}' = \mathit{Str}' \\ \mathit{Str}' & = & \textbf{struct}\ \ \mathit{likestr}\ \ \textbf{end} \end{array}$$

       or

$$\begin{array}{lcl} C(\mathrm{M}'_{M'}) & = & \textbf{cimport}_{h';\mathit{Sig}'_0}\ \mathit{vubs}'\ \mathrm{M}'_{M'} : \mathit{Sig}'_1\ \ \textbf{version}\ \ \mathit{vc}'\ \ \textbf{like}\ \ \mathit{Str}'\ \ \textbf{by}\ \ \mathit{resolvespec}' = \mathit{Mo}' \\ \mathit{Str}' & = & \textbf{struct}\ \ \mathit{likestr}\ \ \textbf{end} \end{array}$$

       Comment: These are the only two cases we need consider: if we are being called at compile time then $\mathit{mode}\ \in \{\mathrm{valuable}, \mathrm{cvaluable}\}$; valuability then ensures that $\mathrm{M}'_{M'}$ cannot be a fresh module or import. If we are being called at run time, then all the previous definitions have been hashified already.

       Comment: In the second case it might be more intuitive to insist that the $\mathit{likestr}$ has exactly the needed fields, rather than (as here) permit it to have more.

     Now calculate a $\mathit{likestr}'$ by cutting down the $\mathit{likestr}$ to the abstract type part of Sig. To do that we define the auxiliary function $\mathrm{filter}\ \ \mathit{str}\ \mathit{sig}$ which calculates the subsequence of str with the external type fields of sig. It is assumed that sig contains no value fields. It is a partial function, failing if there are not enough type fields in str, and only constructs a sensible struct if the struct argument is type-flattened.

$$\begin{array}{l} \mathrm{filter}(\textbf{type}\ \ \mathrm{t}_t = T\ \mathit{str})(\textbf{type}\ \ \mathrm{t}_{t'} : K\ \mathit{sig}) = (\textbf{type}\ \ \mathrm{t}_t = T)\ (\mathrm{filter}\ \ \mathit{str}\ \mathit{sig}) \\ \mathrm{filter}(\textbf{type}\ \ \mathrm{t}_t = T\ \mathit{str})(\textbf{type}\ \ \mathrm{t}'_{t'} : K\ \mathit{sig}) = (\mathrm{filter}\ \ \mathit{str}(\textbf{type}\ \ \mathrm{t}'_{t'} : K\ \mathit{sig}))\ \text{if } \mathrm{t} \neq \mathrm{t}' \\ \mathrm{filter}(\textbf{let}\ \ \mathrm{x}_x = v^{\mathit{eqs}}\ \mathit{str})\mathit{sig} = \mathrm{filter}\ \ \mathit{str}\ \mathit{sig} \\ \mathrm{filter}(\textbf{type}\ \ \mathrm{t}_t = T\ \mathit{str})\ \mathrm{empty} = \mathrm{empty} \\ \mathrm{filter}\ \ \mathrm{empty}\ \ \mathrm{empty} = \mathrm{empty} \\ \mathrm{filter}\ \ \mathrm{empty}\ \ \mathit{sig}\ \text{undefined if } \mathit{sig} \text{ is non empty} \end{array}$$

     Let $\mathit{likestr}' = \mathrm{filter}\ \ \mathit{likestr}(\mathrm{limitdom}\,(\mathit{sig}))$ (or fail with LIKESPEC_MISSING_TYPE_FIELDS if this is undefined).

       Comment: This semantics permits the $\mathit{likestr}$ to contain more fields than are required (or will appear in the constructed $\mathit{likestr}'$ of this import when compiled). Inelegant?

       Comment: Because we cut $\mathit{likestr}$ down to a $\mathit{likestr}'$, a structure containing only type fields, we have no need to worry about $\mathit{likestr}'$ containing uses of **cfresh** or **fresh**.

(b) Let $\mathit{sig}_0 = \text{typeflattensig}(\rho^C \mathit{sig})$. Let $\mathit{Sig}_0 = \textbf{sig}\ \ \mathit{sig}_0\ \ \textbf{end}$.

(c) Let vc be the result of evaluating vce with respect to C, replacing any $\mathrm{M'}_{M'}$ by the hash associated with the module or import in C, i.e. $\mathit{vc} = \rho^C \mathit{vce}$.

(d) ** Now we do case analysis on mode to construct h and $E_\mathrm{n}{}'$:

– Case $\mathit{mode} \in \{\textbf{hash}, \textbf{hash!}\}$: Let $h = \textbf{hash}(\textbf{himport}\ \mathrm{M} : \mathit{Sig}_0\ \ \textbf{version}\ \ \mathit{vc}\ \ \textbf{like}\ \ \textbf{struct}\ \ \mathit{likestr}'\ \ \textbf{end})$ and let $E_\mathrm{n}{}' = E_\mathrm{n}$.

– Case $\mathit{mode} \in \{\textbf{cfresh}, \textbf{fresh}, \textbf{cfresh!}\}$: Let n be a fresh name not in the domain of E_n. Let $h = \mathrm{n}$ and let

$$E_\mathrm{n}{}' = E_\mathrm{n}, \mathrm{n} : \textbf{nimport}\ \mathrm{M} : \mathit{Sig}_0\ \ \textbf{version}\ \ \mathit{vc}\ \ \textbf{like}\ \ \textbf{struct}\ \ \mathit{likestr}'\ \ \textbf{end}$$

Comment: We choose not to include $\mathit{resolvespec}$s in **hmodule** s or **nmodule** s of imports. This is debatable – the argument against including them is that it is useful to be able to change location without breaking code (local code mirror, changing web site to avoid MSBlast.exe, etc.).

(e) Selfify the sig. Let $\mathit{Sig}_1 = \textbf{sig}\ \ \text{typeflattensig}(\text{selfifysig}_h(\mathit{sig}_0))\ \textbf{end}$.

(f) Calculate $\mathit{definition}' =$
$\textbf{cimport}_{h;\mathit{Sig}_0}\ \mathit{vubs}\ \mathrm{M}_M : \mathit{Sig}_1\ \ \textbf{version}\ \mathit{vc}\ \ \textbf{like}\ \ \textbf{struct}\ \ \mathit{likestr}'\ \ \textbf{end}\ \ \textbf{by}\ \ \mathit{resolvespec} = \mathit{Mo}$.

(g) If $\mathit{Mo} = \mathrm{M''}_{M''}$ then check linkok $(E_\mathrm{n}{}', \mathit{definition}'', \mathit{definition}')$ where $C(\mathrm{M''}_{M''}) = \mathit{definition}''$ (or fail with LINKOK_NOT). Otherwise if $\mathit{Mo} =$ UNLINKED check true.

(h) The result is $(E_\mathrm{n}{}', \mathit{definition}')$.

2. exact-name case: if the vce is an exact-name constraint $\textbf{name} = \mathrm{M'}_{M'}$, then we must have $\mathit{likespec} = \mathrm{empty}$ (this is enforced by a syntactic requirement).

The name of this import will be exactly the name of $\mathrm{M'}_{M'}$.

Construct

$\mathit{sourcedefinition}_1 = \textbf{import}\ \mathit{mode}\ \ \mathrm{M}_M : \mathit{Sig}\ \ \textbf{version}\ \ \textbf{name} = \mathrm{M'}_{M'}\ \ \textbf{like}\ \ \mathrm{M'}_{M'}\ \ \textbf{by}\ \ \mathit{resolvespec} = \mathit{Mo}$

and use the normal-case algorithm as above except that we take the h to be the one associated with the module or import $\mathrm{M'}_{M'}$ in C in the step marked **.

Comment: You could hash this import instead of using h in $\mathit{definition}'$. This gives a slightly coarser type equality, which might sometimes be handy, but when you come make up exact-name imports the choice is forced: for type preservation those have to have exactly the hash of the module they are made up from.

• Case $\mathit{sourcedefinition} = \textbf{module}\ \mathrm{M}_M : \mathit{Sig} = \mathrm{M'}_{M'}$.

Note that by typing $\mathrm{M'}_{M'}$ cannot be a **module fresh** or **import fresh**.

– Case $C(\mathrm{M'}_{M'}) = \textbf{cmodule}_{h';\mathit{eqs}';\mathit{Sig}_0'}\ \mathit{vubs}'\ \mathrm{M'}_{M'} : \mathit{Sig}_1'\ \ \textbf{version}\ \ \mathit{vn}' = \mathit{Str}'$

Take $\mathit{definition}' = \textbf{cmodule}_{h';\mathit{eqs}';\mathit{Sig}_0'}\ \mathit{vubs}'\ \mathrm{M}_M : \mathit{Sig}_1'\ \ \textbf{version}\ \ \mathit{vn}' = \mathit{Str}'$ (identical except for the M_M).

– Case $C(\mathrm{M'}_{M'}) = \textbf{cimport}_{h';\mathit{Sig}_0'}\ \mathit{vubs}'\ \mathrm{M''}_{M'} : \mathit{Sig}_1'\ \ \textbf{version}\ \ \mathit{vc}'\ \ \textbf{like}\ \ \mathit{Str}'\ \ \textbf{by}\ \ \mathit{resolvespec}' = \mathit{Mo}'$

Take $\mathit{definition}' = \textbf{cimport}_{h';\mathit{Sig}_0'}\ \mathit{vubs}'\ \mathrm{M}_M : \mathit{Sig}_1'\ \ \textbf{version}\ \ \mathit{vc}'\ \ \textbf{like}\ \ \mathit{Str}'\ \ \textbf{by}\ \ \mathit{resolvespec}' = \mathit{Mo}'$ (identical except for the M_M).

Comment: There are two options here: either copy the $\mathit{definition}$ from $\mathrm{M'}_{M'}$ – but that is semantically odd when one does any rebinding – or just keep the alias in the resulting code – but then both C and runtime lookup need to go through aliases transparently.

However, as aliases are present just to get module names into scope for $\mathbf{with}$! and version annotations, to avoid formalising a filesystem containing modules, for now it is not worth doing anything more elaborate than the above, even though it is strange to copy modules and imports across marks.

- Case $\mathit{sourcedefinition} = \mathbf{mark}\ \mathrm{MK}$.

  Take $\mathit{definition}' = \mathit{sourcedefinition}$.

16.8 Operational semantics

16.8.1 The judgements

We define a labelled transition system over configurations with judgements as follows.

- E_{n} **;** $\langle E_s,\ s,\ \mathit{definitions},\ P\rangle \xrightarrow{\mathbf{n}:\ell} {E_{\mathrm{n}}}'$ **;** $\langle {E'}_s,\ s',\ \mathit{definitions}',\ P'\rangle$ Process reduction.
- E_{n} **;** $\langle E_s,\ s,\ \mathit{definitions},\ P\rangle \rightarrow \mathbf{TERM}$ Progam termination.
- E_{n} **;** $\langle E_s,\ s,\ \mathit{definitions},\ e\rangle \xrightarrow{\ell}_{\mathit{eqs}} {E_{\mathrm{n}}}'$ **;** $\langle {E'}_s,\ s',\ \mathit{definitions}',\ e'\rangle$ Expression reduction.
- $e \xrightarrow{\ell}_{\mathit{eqs}} e'$
- E_{n} **;** $e \xrightarrow{\ell}_{\mathit{eqs}} {E_{\mathrm{n}}}'$ **;** e'
- $P \xrightarrow{\ell} P'$

where

ℓ	$::=$	empty	internal reduction step
		$x^n\ v_1^{\varnothing}\ ..\ v_n^{\varnothing}$ for $x^n \in \mathrm{dom}(E_{\mathrm{const}}) \wedge \mathrm{os}(x^n)$	invocation of OS call
		$\mathrm{Ok}(v^{\varnothing})$	return from OS call
		$\mathrm{Ex}(v^{\varnothing})$	return from OS call
		$\mathrm{GetURI}(\mathit{URI})$	request for code at URI
		$\mathrm{DeliverURI}(\mathit{definitions})$	resulting code
		CannotFindURI	nothing found at URI

We write $\xrightarrow{\text{empty}}$ simply as $\rightarrow$.

In addition the runtime implementation might fail with the RUN or INTERNAL errors, though it should not.

16.8.2 Values

The set of values is indexed by a colour, to control the adminstrative bracket-pushing reductions, as follows. Note that colour is a spectral phenomenon — two entities have the same colour iff they are indexed by the same set of equations

$$
\begin{array}{lll}
v^{\mathit{eqs}} & ::= & \mathrm{C}_0 \\
& & \mathrm{C}_1\ v^{\mathit{eqs}} \\
& & v^{\mathit{eqs}} :: v^{\mathit{eqs}} \\
& & (v_1^{\mathit{eqs}}, .., v_n^{\mathit{eqs}}) \\
& & \mathbf{function}\ (x : T) \rightarrow e \\
& & l \\
& & \mathbf{nn} \\
& & \Lambda\ t \rightarrow e \\
& & \{T, v^{\mathit{eqs}}\}\ \mathbf{as}\ T' \\
& & [v^{\mathit{eqs}'}]_{\mathit{eqs}'}^{T\ \mathsf{ref}} \\
& & [v^{\mathit{eqs}'}]_{\mathit{eqs}'}^{T\ \mathsf{name}} \\
& & [v^{\mathit{eqs}'}]_{\mathit{eqs}'}^{h.\mathrm{t}} \quad \text{where } h.\mathrm{t} \in \mathrm{dom}(\mathit{eqs}') \text{ and } h.\mathrm{t} \notin \mathrm{dom}(\mathit{eqs})
\end{array}
$$

Comment: The different families of values collapse into a single one when there are no coloured brackets, such is the case with user source programs.

This just says that within values, brackets may only appear at types with no visible structure or at a T ref or T name type. This is achieved by the bracket-pushing reductions below. For discussion of these and of the possible design choices for the T ref and T name cases, see §16.8.4 (page 130).

16.8.3 Reduction contexts and closure rules

We use redex-time instantiation for module identifiers, but as we have marks only between the (second-class) modules, there is no need to be anything other than conventional call-by-value λ_c *within* the running expression. Evaluation contexts are therefore conventional, except that we must track colours.

Single-level evaluation contexts and colour-changing evaluation contexts

C_{eqs}	$::=$	$\mathrm{C}_1\ \_$	C_1 a constructor of arity 1
		$\_ :: e$	
		$v^{eqs} :: \_$	
		$(e_1, .., e_{m-1}, \_, v^{eqs}_{m+1}, .., v^{eqs}_n)$	$n \geq 2, 1 \leq m \leq n$
		$\mathbf{if}\ \_\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2$	
		$\_ \,\&\&\, e$	
		$\_ \,\vert\vert\, e$	
		$\_ \,\mathbf{;}\, e$	
		$\_\ e$	
		$v^{eqs}\ \_$	
		$e\ e_1\ ...\ e_{m-1}\ \_\ v^{eqs}_{m+1}\ ...\ v^{eqs}_n$	$1 \leq m \leq n$, $e = op^n$ or $e = x^n$
		$!_T\ \_$	
		$\_\ \mathbf{:=}_T\ e$	
		$v^{eqs}\ \mathbf{:=}_T\ \_$	
		$\mathbf{match}\ \_\ \mathbf{with}\ mtch$	
		$\mathbf{raise}\ \_$	
		$\mathbf{try}\ \_\ \mathbf{with}\ mtch$	
		$\mathbf{marshal}\ \mathrm{MK}\ \_\ :\ T$	
		$\mathbf{marshal}\ \_\ e_2\ :\ T$	
		$\mathbf{unmarshal}\ \ \_\ \mathbf{as}\ T$	
		$\mathbf{swap}\ \_\ \mathbf{and}\ e_2\ \mathbf{in}\ e_3$	
		$\mathbf{swap}\ v^{eqs}_1\ \mathbf{and}\ \_\ \mathbf{in}\ e_3$	
		$\mathbf{swap}\ v^{eqs}_1\ \mathbf{and}\ v^{eqs}_2\ \mathbf{in}\ \_$	
		$\_\ \mathbf{freshfor}\ e_2$	
		$v^{eqs}_1\ \mathbf{freshfor}\ \_$	
		$\mathbf{support}_T\ \_$	
		$\mathbf{name\_of\_tie}\ \_$	
		$\mathbf{val\_of\_tie}\ \_$	
		$\_\ T$	
		$\mathbf{let}\ \{t, x\} = \_\ \mathbf{in}\ e_2$	
		$\mathbf{namecase}\ \_\ \mathbf{with}\ \{t, (x_1, x_2)\}\ \mathbf{when}\ x_1 = e \rightarrow e_2\ \mathbf{otherwise}\ \rightarrow e_3$	
		$\mathbf{namecase}\ v^{eqs}\ \mathbf{with}\ \{t, (x_1, x_2)\}\ \mathbf{when}\ x_1 = \_ \rightarrow e_2\ \mathbf{otherwise}\ \rightarrow e_3$	
$C^{eqs_1}_{eqs_2}$	$::=$	C_{eqs_1}	$eqs_1 = eqs_2$
		$[\_]^T_{eqs_2}$	
		$\mathbf{marshalz}\ \mathrm{MK}\ \_\ :\ T$	$eqs_2 = \varnothing$
		$l\ \mathbf{:=}'_T\ \_$	$eqs_2 = \varnothing$

$\mathbf{op}(e_0)^n\ e_1\ ..\ e_{i-1}\ \_\ v^{\varnothing}_{i+1}\ ..\ v^{\varnothing}_n$	$1 \leq i \leq n, eqs_2 = \varnothing$
$\mathbf{hash}(T, \_)_{T'}$	$eqs_2 = \varnothing$
$\mathbf{hash}(T, v^{\varnothing}_1, T)_{T'}$	$eqs_2 = \varnothing$
$\mathbf{hash}(T, \_, e_2)_{T'}$	$eqs_2 = \varnothing$

It is sometimes convenient to refer to bracket contexts (sequences of nested brackets).

Bracket contexts

$$\begin{array}{lll} BC & ::= & \_ \\ & & BC.[\_]^{T}_{eqs} \end{array}$$

We follow the evaluation order of `ocamlopt` (not `ocamlc`), except that `ocamlopt` treats saturated applications of operators (such as `(+)`) specially, whereas we treat all functions and operators uniformly. Specifically, we evaluate applications from left to right in all cases, whereas `ocamlopt` evaluates a saturated operator (either `e1 + e2` or `(+) e1 e2`, but not `((+) e1) e2`) from right-to-left, and `ocamlc` evaluates all applications from right-to-left. Note that in both, tuples are evaluated right-to-left.

Evaluation contexts and Colour changing evaluation contexts

$$\begin{array}{lll} CC_{eqs} & ::= & \_ \\ & & CC_{eqs}.C_{eqs} \end{array} \qquad \begin{array}{lll} CC^{eqs_1}_{eqs_2} & ::= & \_ \\ & & CC^{eqs_1}_{eqs}.C^{eqs}_{eqs_2} \end{array}$$

Structure evaluation contexts and thread evaluation contexts

$$\begin{array}{lll} TC_{eqs} & ::= & \_ \\ & & (\mathbf{cmodule}_{h;eqs;Sig_0}\ \mathit{vubs}\ \mathrm{M}_M : Sig_1\ \ \mathbf{version}\ \ vn = \mathbf{struct}\ \ SC_{eqs}\ \mathbf{end})\ \mathit{definitions}\ e \\ TCC_{eqs} & ::= & \_ \\ & & TC_{eqs_2}.CC^{eqs_2}_{eqs} \\ SC_{eqs} & ::= & \mathbf{let}\ \ \mathrm{x}_x = \_\ \mathit{str} \\ & & \mathbf{let}\ \ \mathrm{x}_x = v^{eqs}\ SC_{eqs} \quad \text{where } x \notin \text{fv}\ SC_{eqs} \\ & & \mathbf{type}\ \ \mathrm{t}_t = T\ SC_{eqs} \\ \\ \mathit{strval}_{eqs} & ::= & \mathbf{let}\ \ \mathrm{x}_x = v^{eqs}\ \mathit{strval}_{eqs} \\ & & \mathbf{type}\ \ \mathrm{t}_t = T\ \mathit{strval}_{eqs} \end{array}$$

Module and definition values Say a $\mathbf{cmodule}$ *value* is a *definition* of the form $\mathbf{cmodule}_{h;eqs;Sig_0}\ \mathit{vubs}\ \mathrm{M}_M : Sig_1\ \ \mathbf{version}\ \ vn = \mathbf{struct}\ \ \mathit{strval}_{eqs}\ \mathbf{end}$ where there are no internal expression field dependencies in strval_{eqs}.

Say a *definition value* is a $\mathbf{cmodule}$ value, a $\mathbf{cimport}$, or a $\mathbf{mark}\ \ \mathrm{MK}$.

These induce the following reductions.

$$\frac{\text{if } \mathit{definition} \text{ is a definition value}}{\begin{array}{l}E_{\mathrm{n}} \,;\, \langle E_s,\, s,\, \mathit{definitions}_0,\, P|\mathbf{n} : (\mathit{definition}\ \mathit{definitions}\ e)\rangle \xrightarrow{\mathrm{n:empty}} \\ E_{\mathrm{n}} \,;\, \langle E_s,\, s,\, \mathit{definitions}_0\ \mathit{definition},\, P|\mathbf{n} : (\mathit{definitions}\ e)\rangle\end{array}}$$

$$\frac{\begin{array}{l}\text{if } \mathit{definition} \text{ is of the form } \mathbf{module\ fresh} \text{ or } \mathbf{import\ fresh} \\ \text{and } \mathrm{hashify}_{\mathit{definitions}_0}(E_{\mathrm{n}}, \mathit{definition}, (\mathrm{nonvaluable}, \mathrm{nonvaluable})) \rightsquigarrow (E_{\mathrm{n}}{}', \mathit{definition}')\end{array}}{\begin{array}{l}E_{\mathrm{n}} \,;\, \langle E_s,\, s,\, \mathit{definitions}_0,\, P|\mathbf{n} : (\mathit{definition}\ \mathit{definitions}\ e)\rangle \xrightarrow{\mathbf{n}\mathrm{:empty}} \\ E_{\mathrm{n}}{}' \,;\, \langle E_s,\, s,\, \mathit{definitions}_0,\, P|\mathbf{n} : (\mathit{definition}'\ \mathit{definitions}\ e)\rangle\end{array}}$$

$$\frac{e \xrightarrow{\ell}_{\mathit{eqs}} e'}{E_{\mathrm{n}} \,;\, \langle E_s,\, s,\, \mathit{definitions},\, e\rangle \xrightarrow{\ell}_{\mathit{eqs}} E_{\mathrm{n}} \,;\, \langle E_s,\, s,\, \mathit{definitions},\, e'\rangle}$$

$$\frac{E_{\mathrm{n}} \,;\, e \xrightarrow{\ell}_{\mathit{eqs}} E_{\mathrm{n}}{}', e'}{E_{\mathrm{n}} \,;\, \langle E_s,\, s,\, \mathit{definitions},\, e\rangle \xrightarrow{\ell}_{\mathit{eqs}} E_{\mathrm{n}}{}' \,;\, \langle E_s,\, s,\, \mathit{definitions},\, e'\rangle}$$

$$\frac{E_{\mathrm{n}} \,;\, \langle E_s,\, s,\, \mathit{definitions},\, e\rangle \xrightarrow{\ell}_{\mathit{eqs}_2} E_{\mathrm{n}}{}' \,;\, \langle E'_s,\, s',\, \mathit{definitions}',\, e'\rangle}{E_{\mathrm{n}} \,;\, \langle E_s,\, s,\, \mathit{definitions},\, C^{\mathit{eqs}_1}_{\mathit{eqs}_2}.e\rangle \xrightarrow{\ell}_{\mathit{eqs}_1} E_{\mathrm{n}}{}' \,;\, \langle E'_s,\, s',\, \mathit{definitions}',\, C^{\mathit{eqs}_1}_{\mathit{eqs}_2}.e'\rangle}$$

$$\frac{E_{\mathrm{n}} \,;\, \langle E_s,\, s,\, \mathit{definitions},\, e\rangle \xrightarrow{\ell}_{\mathit{eqs}} E_{\mathrm{n}}{}' \,;\, \langle E'_s,\, s',\, \mathit{definitions}',\, e'\rangle}{E_{\mathrm{n}} \,;\, \langle E_s,\, s,\, \mathit{definitions},\, P|\mathbf{n} : \mathit{TC}_{\mathit{eqs}}.e\rangle \xrightarrow{\mathbf{n}:\ell} E_{\mathrm{n}}{}' \,;\, \langle E'_s,\, s',\, \mathit{definitions}',\, P|\mathbf{n} : \mathit{TC}_{\mathit{eqs}}.e'\rangle}$$

$$\frac{P \xrightarrow{\mathbf{n}:\ell} P'}{E_{\mathrm{n}} \,;\, \langle E_s,\, s,\, \mathit{definitions},\, P\rangle \xrightarrow{\mathbf{n}:\ell} E_{\mathrm{n}} \,;\, \langle E_s,\, s,\, \mathit{definitions},\, P'\rangle}$$

If the hashification of fresh definition *definition* in rule 2 above fails, the error is reported at toplevel as FAILURE.COMPILE.HASHIFY and the program terminates. This is unpleasant, but unavoidable in the absence of exception handlers around threads.

16.8.4 Simple expression forms

Eliminating internal field dependencies When performing module initialisation we evaluate each field in order, and for each replace all later uses of it by its value. This ensures we do not need to consider marshalling or placing in the store a thunk containing a free x. For simplicity, we do this systematically to all **cmodule**s even when it is not forced (which could be detected by looking at their valuabilities).

This stategy has some impact on rebinding: if one has a module $\mathrm{M}_M ... =$ **struct let** $\mathrm{x}_x = 3$ **let** $\mathrm{y}_y =$ **function** $() \to x$ **end** before initialisation then the x will be eliminated in the y_y field. Thus later externally instantiating M_M.y gives **function** $() \to 3$ rather than **function** $() \to \mathrm{M}_M$.x.

The strategy also has implication for **swap** — as there's no need to follow x uses in a value.

$$\begin{array}{ll} & E_{\mathrm{n}}\ \mathbf{;}\ \langle E_s,\ s,\ \mathit{definitions},\ P|\mathbf{n} : (\mathit{definition}\ \mathit{definitions}'\ e)\rangle \\ \xrightarrow{\mathbf{n}:\mathrm{empty}} & E_{\mathrm{n}}\ \mathbf{;}\ \langle E_s,\ s,\ \mathit{definitions},\ P|\mathbf{n} : (\mathit{definition}'\ \mathit{definitions}'\ e)\rangle \end{array}$$

where

$$\begin{array}{l} \mathit{definition} = (\mathbf{cmodule}_{h;eqs;Sig_0}\ \mathit{vubs}\ \mathrm{M}_M : \mathit{Sig}_1\ \ \mathbf{version}\ \ \mathit{vn} = \mathbf{struct}\ \ \mathit{str}\ \ \mathbf{end}) \\ \mathit{definition}' = (\mathbf{cmodule}_{h;eqs;Sig_0}\ \mathit{vubs}\ \mathrm{M}_M : \mathit{Sig}_1\ \ \mathbf{version}\ \ \mathit{vn} = \mathbf{struct}\ \ \mathit{str}'\ \ \mathbf{end}) \\ \mathit{str} = \mathit{strval}_{eqs}\ \mathbf{let}\ \ \mathrm{x}_x = v^{eqs}\ \mathit{str}_0 \\ \mathit{str}' = \mathit{strval}_{eqs}\ \mathbf{let}\ \ \mathrm{x}_x = v^{eqs}\ \{v^{eqs}/x\}\mathit{str}_0 \\ x\ \in\ \mathrm{fv}(\mathit{str}_0) \\ \text{dom}(\mathit{strval}_{eqs}) \text{ does not intersect the free expression identifiers of } \mathit{str} \end{array}$$

Note the $x\ \in\ \mathrm{fv}(\mathit{str}_0)$ condition to prevent divergence.

> *Comment:* This rule is terminating because of the $x\ \in\ \mathrm{fv}(\mathit{str})$ condition. This condition, in combination with the free identifier side condition on SC_{eqs} contexts used in module field initialisation forces the two reduction rules to be disjoint. It might be more tasteful to work with a **cmodule** that is split into the post- and pre-evaluation parts, but it would be notationally heavy.

Matching Define a partial function $\mathrm{matchsub}_{eqs}(\_,\_)$ taking a value of colour *eqs* and a pattern (in which all identifiers are distinct) and giving a set of substitutions, adding suitable brackets:

$$\begin{array}{lcl} \mathrm{matchsub}_{eqs}(v^{eqs}, (\_ : T)) & = & \varnothing \\ \mathrm{matchsub}_{eqs}(v^{eqs}, (x : T)) & = & \{[v^{eqs}]^T_{eqs}/x\} \\ \mathrm{matchsub}_{eqs}(v^{eqs}, (p : T)) & = & \mathrm{matchsub}_{eqs}(v^{eqs}, p) \\ \mathrm{matchsub}_{eqs}(\mathtt{C}_0, \mathtt{C}_0) & = & \varnothing \\ \mathrm{matchsub}_{eqs}(\mathtt{C}_1\ v^{eqs}, \mathtt{C}_1\ p) & = & \mathrm{matchsub}_{eqs}(v^{eqs}, p) \\ \mathrm{matchsub}_{eqs}(v_1^{eqs} :: v_2^{eqs}, p_1 :: p_2) & = & \mathrm{matchsub}_{eqs}(v_1^{eqs}, p_1) \cup\ \mathrm{matchsub}_{eqs}(v_2^{eqs}, p_2) \\ \mathrm{matchsub}_{eqs}((v_1^{eqs}, .., v_n^{eqs}), (p_1, .., p_n)) & = & \mathrm{matchsub}_{eqs}(v_1^{eqs}, p_1) \cup .. \cup\ \mathrm{matchsub}_{eqs}(v_n^{eqs}, p_n) \quad n \geq 2 \\ \mathrm{matchsub}_{eqs}(v^{eqs}, p) & & \text{undefined otherwise} \end{array}$$

Reduction Axioms

$$\begin{array}{llll} \mathbf{if}\ \ \mathbf{true}\ \ \mathbf{then}\ \ e_1\ \ \mathbf{else}\ \ e_2 & \longrightarrow_{eqs} & e_1 & \\ \mathbf{if}\ \ \mathbf{false}\ \ \mathbf{then}\ \ e_1\ \ \mathbf{else}\ \ e_2 & \longrightarrow_{eqs} & e_2 & \\ \mathbf{false}\ \&\&\ e & \longrightarrow_{eqs} & \mathbf{false} & \\ \mathbf{true}\ \&\&\ e & \longrightarrow_{eqs} & e & \\ \mathbf{false}\ ||\ e & \longrightarrow_{eqs} & e & \\ \mathbf{true}\ ||\ e & \longrightarrow_{eqs} & \mathbf{true} & \\ ()\ \mathbf{;}\ e & \longrightarrow_{eqs} & e & \\ \mathbf{while}\ \ e_1\ \ \mathbf{do}\ \ e_2\ \ \mathbf{done} & \longrightarrow_{eqs} & \mathbf{if}\ \ e_1\ \ \mathbf{then}\ (e_2\ \mathbf{;}\ \mathbf{while}\ \ e_1\ \ \mathbf{do}\ \ e_2\ \ \mathbf{done})\ \mathbf{else}\ () & \\ (\mathbf{function}\ (x : T) \rightarrow e)\ v^{eqs} & \longrightarrow_{eqs} & \{[v^{eqs}]^T_{eqs}/x\}e & \\ \mathbf{match}\ \ v^{eqs}\ \mathbf{with}\ \ p_1 \rightarrow e_1|..|p_n \rightarrow e_n & \longrightarrow_{eqs} & \mathrm{matchsub}_{eqs}(v^{eqs}, p_k)e_k & (a) \\ \mathbf{match}\ \ v^{eqs}\ \mathbf{with}\ \ p_1 \rightarrow e_1|..|p_n \rightarrow e_n & \longrightarrow_{eqs} & \mathbf{raise}\ \textsc{Match\_failure}\ v' & (b) \\ \mathbf{let}\ \ \mathbf{rec}\ \ x_1 : T = \mathbf{function}\ (x_2 : T') \rightarrow e_1\ \ \mathbf{in}\ \ e_2 & \longrightarrow_{eqs} & & \\ \multicolumn{4}{l}{\qquad \{[\{[\mathbf{let}\ \ \mathbf{rec}\ \ x_1 : T = \mathbf{function}\ (x_2 : T') \rightarrow e_1\ \ \mathbf{in}\ \ x_1]^T_{eqs}/x_1\}\mathbf{function}\ (x_2 : T') \rightarrow e_1]^T_{eqs}/x_1\}e_2} \\ C_{eqs}.\mathbf{raise}\ \ v^{eqs} & \longrightarrow_{eqs} & \mathbf{raise}\ \ v^{eqs} & (c) \\ [\mathbf{raise}\ \ v^{eqs'}]^T_{eqs'} & \longrightarrow_{eqs} & \mathbf{raise}\ [v^{eqs'}]^{\mathsf{exn}}_{eqs'} & \\ \mathbf{try}\ \ \mathbf{raise}\ \ v^{eqs}\ \mathbf{with}\ \ p_1 \rightarrow e_1|..|p_n \rightarrow e_n & \longrightarrow_{eqs} & \mathrm{matchsub}_{eqs}(v^{eqs}, p_k)e_k & (a) \\ \mathbf{try}\ \ v^{eqs}\ \mathbf{with}\ \ p_1 \rightarrow e_1|..|p_n \rightarrow e_n & \longrightarrow_{eqs} & v^{eqs} & \\ \mathbf{marshal}\ \mathrm{MK}\ v^{eqs}\ :\ T & \longrightarrow_{eqs} & \mathbf{marshalz}\ \mathrm{MK}\ [v^{eqs}]^T_{eqs}\ :\ T & \end{array}$$

(a) $\mathrm{matchsub}_{eqs}(v^{eqs}, p_k)$ is defined and there is no $k' < k$ with $\mathrm{matchsub}_{eqs}(v^{eqs}, p_{k'})$ defined

(b) (i) not exists $k \in 1..n$ such that $\mathrm{matchsub}_{eqs}(v^{eqs}, p_k)$ is defined, and

(ii) v'^{eqs} is an arbitrary value such that $\vdash v'^{eqs} : \mathsf{string} * \mathsf{int} * \mathsf{int}$

(c) there does not exist $p_1 \to e_1|..|p_n \to e_n$ and k st $C_{eqs} = \textbf{try}\ \_\ \textbf{with}\ p_1 \to e_1|..|p_n \to e_n$ and $\mathrm{matchsub}_{eqs}(v^{eqs}, p_k)$ defined

Comment: Note that in several places the semantics involves not-quite-value substitutions: substitutions of a value surrounded by an extra pair of brackets. Bracket reduction is effect-free and terminating, so this is not a problem – it would be notationally awkward to reduce before substituting.

Bracket-pushing (administrative) reductions

Brackets are used to represent abstraction boundaries, and their location is carefully controlled during evaluation. Brackets are purely annotations, however, and the semantics obtained by erasing them corresponds precisely to the given (coloured) semantics. An implementation will typically use the erased semantics; the coloured semantics and the correspondence property serve to give confidence that the implementation respects abstraction boundaries.

Frequently desired reductions will involve subterms on both sides of an abstraction boundary - for example, applying a module function $\mathrm{M}_M.z$ to a value outside that module will yield a term $[\textbf{function}\ x : T \to e]_{eqs}^{T' \to T''}\ v$. Rather than give reduction rules for each permutation of brackets, we give reduction rules only for the bracket-free case, and add administrative reductions to move brackets out of the way. Specifically, erased-values (that is, terms that correspond to values in the erased semantics) may not yet be values in the coloured semantics; to make them so, we push the brackets inwards by application of the bracket-pushing rules. In the example above, we may push the brackets inwards through the lambda, to obtain $(\textbf{function}\ x : T' \to [\{[x]_T^{eqs}/x\}e]_{\varnothing}^{T''})\ v$, and now the ordinary function application rule yields $[\{[v]_T^{eqs}/x\}e]_{\varnothing}^{T''}$. These administrative reductions apply only to erased-values.

The bracket-pushing rules are as follows:

pushing through constructors:		
$[[\,]_{T'}]_{eqs'}^{T\ \mathsf{list}}$	$\longrightarrow_{eqs}$	$[\,]_T$
$[\mathrm{NONE}_{T'}]_{eqs'}^{T\ \mathsf{option}}$	$\longrightarrow_{eqs}$	NONE_T
$[\mathrm{INJ}_i^{(T'_1+..+T'_n)}\ v^{eqs'}]_{eqs'}^{T_1+..+T_n}$	$\longrightarrow_{eqs}$	$\mathrm{INJ}_i^{(T_1+..+T_n)}\ [v^{eqs'}]_{eqs'}^{T_i}$
$[v_1^{eqs'} :: v_2^{eqs'}]_{eqs'}^{T\ \mathsf{list}}$	$\longrightarrow_{eqs}$	$[v_1^{eqs'}]_{eqs'}^{T} :: [v_2^{eqs'}]_{eqs'}^{T\ \mathsf{list}}$
$[(v_1^{eqs'}, .., v_n^{eqs'})]_{eqs'}^{T_1*..*T_n}$	$\longrightarrow_{eqs}$	$([v_1^{eqs'}]_{eqs'}^{T_1}, .., [v_n^{eqs'}]_{eqs'}^{T_n})\quad n \geq 2$
$[\mathrm{C}^n\ v_1^{eqs'}\ \dots\ v_n^{eqs'}]_{eqs'}^{T_0}$	$\longrightarrow_{eqs}$	$\mathrm{C}^n\ [v_1^{eqs'}]_{eqs'}^{T_1}\ \dots\ [v_n^{eqs'}]_{eqs'}^{T_n}$
		$\mathrm{C}^n : T_1 \to \dots \to T_n \to T_0$ any other constructor
pushing through lambda:		
$[\textbf{function}\ (x : T) \to e]_{eqs'}^{T' \to T''}$	$\longrightarrow_{eqs}$	$\textbf{function}\ (x : T') \to [\{[x]_{eqs'}^{T}/x\}e]_{eqs'}^{T''}$
pushing through type-lambda and pack:		
$[\Lambda\ t \to e]_{eqs'}^{\forall\ t.T}$	$\longrightarrow_{eqs}$	$\Lambda\ t \to [e]_{eqs'}^{T}$
$[\{T, v^{eqs'}\}\ \textbf{as}\ T']_{eqs'}^{\exists\ t.T''}$	$\longrightarrow_{eqs}$	$\{T, [v^{eqs'}]_{eqs'}^{\{T/t\}T''}\}\ \textbf{as}\ \exists\ t.T''$
bracket type revelation:		
$[v^{eqs'}]_{eqs'}^{h.\mathrm{t}}$	$\longrightarrow_{eqs}$	$[v^{eqs'}]_{eqs'}^{T}\quad (h.\mathrm{t} \approx T) \in eqs \wedge h.\mathrm{t} \in \mathrm{dom}(eqs')$
bracket elimination:		
$[[v^{eqs''}]_{eqs''}^{h.\mathrm{t}}]_{eqs'}^{h.\mathrm{t}}$	$\longrightarrow_{eqs}$	$[v^{eqs''}]_{eqs''}^{h.\mathrm{t}}\quad h.\mathrm{t} \notin \mathrm{dom}(eqs')$

It is straightforward to show that all these rules are type-preserving.

Comment: Note that brackets are handled specially in the case of names and store locations; there are no bracket-pushing rules for these forms.

Comment: Note that type revelation does not introduce non-termination, because the equation formation rules ensure that for any equation $X.t \approx T$, T is well-formed in the environment prior to the definition of X.

These rules ensure that any erased-value can be reduced to a (coloured) value, by pushing brackets inward as far as they can go, and eliminating double brackets.

Note that the rule for pushing brackets through a **function** depends on the fact that functions bind a single argument identifier, functions with more complex patterns being treated as syntactic sugar for a single-argument function with a **match** as the body. Without this, bracket pushing would have to be much more elaborate.

Store- and name-related bracket-pushing

Bracket handling for locations, dereferencing, assignment, and names is subtle. Notice, for example, that a module may return a location to its caller at an abstract type, and allow the caller to store abstract values in it, and then internally pull them out at the concrete one. Worse, a module may create a ref cell, and return its location twice, once at an abstract type and once at a concrete type. There seems no good reason to prohibit this arbitrary aliasing of pointers, where each alias may have different type transparency depending on the locally available eqs. In this respect we differ from Zdancewic et al. [GMZ00, §4.2]. For names the issue is simply that a name records its type as that at which it was created, but use at multiple types, according to context, must be permitted.

Since ref is treated as a vanilla operator (brackets are not involved), we do not discuss its semantics here.

In the value grammar we allow names and locations to be wrapped in brackets in order to express the variety of type transparency that aliases of the name or location may have. Thus, if we have a bracketted (!) or (:=), we *pull* the brackets outside, changing the type annotations accordingly. The goal is to peel away the brackets surrounding a location so as to expose the location itself to dereference or assignment:

$$\begin{array}{lll} !_T\ [v^{eqs'}]_{eqs'}^{T'\ \mathsf{ref}} & \longrightarrow_{eqs} & [!_{T'}\ v^{eqs'}]_{eqs'}^{T'} \\ [v'^{eqs'}]_{eqs'}^{T'\ \mathsf{ref}}\ \mathbf{:=}_T\ v^{eqs} & \longrightarrow_{eqs} & [v'^{eqs'}\ \mathbf{:=}_{T'}\ [v^{eqs}]_{eqs}^{T'}]_{eqs'}^{\mathsf{unit}} \end{array}$$

Comment: When bracket pulling through $!_T$ it is not immediately obvious why the bracket on the RHS is at T' and not T. It is correct (even though the type of the whole expression must be T) because we may deduce from the LHS that $E \vdash_{eqs} T \approx T'$, and it is necessary because we cannot deduce $E \vdash_{eqs'} T \approx T'$, which would be needed in order to type the alternative.

Values in the store are always black ($v^{\varnothing}$). When we get a raw location, $!_T$ can dereference it:

$$E_{\mathrm{n}}\ \mathbf{;}\ \langle E_s,\ (s, l \mapsto v^{\varnothing}),\ \mathit{definitions},\ !_T\ l\rangle \quad \longrightarrow_{eqs} \quad E_{\mathrm{n}}\ \mathbf{;}\ \langle E_s,\ (s, l \mapsto v^{\varnothing}),\ \mathit{definitions},\ v^{\varnothing}\rangle$$

Comment: Note that the correctness of this rule relies on the fact that typing is monotonic with respect to the eqs set. By hypothesis, $E_{\mathrm{n}}, E_s \vdash_{\varnothing} v^{\varnothing} : T_0$ where $E_s(l) = T_0$ and $E_{\mathrm{n}}, E_s \vdash eqs$ **ok** and $E_{\mathrm{n}}, E_s \vdash_{eqs} T_0 \approx T$. This implies $E_{\mathrm{n}}, E_s \vdash_{eqs} v^{\varnothing} : T_0$ since having more equalities can't hurt, hence $E_{\mathrm{n}}, E_s \vdash_{eqs} v^{\varnothing} : T$ as desired.

When we get a raw location, $\mathbf{:=}_T$ prepares the value to be put in the store; when that value becomes a value in $\varnothing$ (see the discussion of operator reduction below), we can install it in the store:

$$\begin{array}{lll} l\ \mathbf{:=}_T\ v^{eqs} & \longrightarrow_{eqs} & l\ \mathbf{:=}'_T\ [v^{eqs}]_{eqs}^{T} \\ E_{\mathrm{n}}\ \mathbf{;}\ \langle E_s,\ (s, l \mapsto v'^{\varnothing}),\ \mathit{definitions},\ l\ \mathbf{:=}'_T\ v^{\varnothing}\rangle & \longrightarrow_{eqs} & E_{\mathrm{n}}\ \mathbf{;}\ \langle E_s,\ (s, l \mapsto v^{\varnothing}),\ \mathit{definitions},\ ()\rangle \end{array}$$

For names there is no other argument to which the brackets must be transferred; instead, we define all operators which operate on names to ignore brackets surrounding those names.

Comment: In the current semantics, we treat $[v^{eqs'}]_{eqs'}^{T'\ \mathsf{ref}}$ as a value in eqs for arbitrary instantiations of the metavariables (and similarly for name). This is not desirable because it fails to distinguish between those T' ref brackets that are really necessary and those that are not. We have some ideas of how to proceed, but for the present leave the removal of this technical infelicity to future work.

Operators and Special Constants Before evaluating the application of a primitive operator or of a primitive constant, we make the arguments be $\varnothing$-coloured values. Once this is done, we perform the actual evaluation by a delta-rule.

$$\begin{array}{lll} e^n\ v_1^{eqs}\ ..\ v_n^{eqs} & \longrightarrow_{eqs} & \mathbf{op}(e^n)^n\ [v_1^{eqs}]_{eqs}^{T_1}\ ..\ [v_n^{eqs}]_{eqs}^{T_n} \end{array}$$

where $e^n : T_1 \rightarrow .. \rightarrow T_n \rightarrow T$ is x^n **in** E_{const} or op^n

Note the rule includes the case $x^0 : T$.

For simple operators the delta rules are as follows:

$$\begin{array}{llll} \mathbf{op}(=_T)^2\ v^\varnothing v'^\varnothing & \rightarrow_{eqs} & \mathbf{true} & \text{erase\_brackets}(v^\varnothing) = \text{erase\_brackets}(v'^\varnothing) \\ \mathbf{op}(=_T)^2\ v^\varnothing v'^\varnothing & \rightarrow_{eqs} & \mathbf{false} & \text{otherwise} \end{array}$$

with similar rules for the other arithmetic and logical operators (noting that equality raises INVALID_ARGUMENT if used on a function or existential package and division can raise DIVISION_BY_ZERO).

The delta rule for the reference operator is:

$$E_{\mathrm{n}}\ ;\ \langle E_s,\ s,\ \mathit{definitions},\ (\mathbf{op}(\mathbf{ref}_T)^1\ v^\varnothing)\rangle \quad \rightarrow_{eqs} \quad E_{\mathrm{n}}\ ;\ \langle (E_s, l : T\ \mathsf{ref}),\ (s, l \mapsto v^\varnothing),\ \mathit{definitions},\ l\rangle \quad l \notin \text{dom}(s)$$

Note that the rule for ref introduces nondeterminism. That could be avoided by working up to cyclic bindings – seems slightly simpler without, but there is little in it.

Special constants from E_{const} are of two classes. Some have are internal to the language; for these we should have further delta rules, but do not write them here. The others — the x^n such that $\mathrm{os}(x^n)$, which are all of function type — are calls to OS routines. For these we have labelled transitions for invocations and returns:

$$\begin{array}{llll} E_{\mathrm{n}}\ ;\ \mathbf{op}(x^n)^n\ \ v_1^\varnothing\ ..\ v_n^\varnothing & \xrightarrow{x^n\ v_1^\varnothing\ ..\ v_n^\varnothing}_{eqs} & E_{\mathrm{n}}\ ;\ \mathbf{RET}_T & (x^n : T_1 \rightarrow ..T_n \rightarrow T) \in\ E_{\mathrm{const}} \wedge \mathrm{os}(x^n) \wedge \mathrm{fast}(x^n) \\ E_{\mathrm{n}}\ ;\ \mathbf{RET}_T & \xrightarrow{\mathrm{Ok}(v^\varnothing)}_{eqs} & E_{\mathrm{n}}\ ;\ v^\varnothing & \text{if } E_{\mathrm{n}}, E_{\mathrm{const}} \vdash_\varnothing v^\varnothing : T \\ E_{\mathrm{n}}\ ;\ \mathbf{RET}_T & \xrightarrow{\mathrm{Ex}(v^\varnothing)}_{eqs} & E_{\mathrm{n}}\ ;\ \mathbf{raise}\ (v^\varnothing) & \text{if } E_{\mathrm{n}}, E_{\mathrm{const}} \vdash_\varnothing v^\varnothing : \mathsf{exn} \end{array}$$

and

$$\begin{array}{llll} E_{\mathrm{n}}\ ;\ \mathbf{op}(x^n)^n\ \ v_1^\varnothing\ ..\ v_n^\varnothing & \xrightarrow{x^n\ v_1^\varnothing\ ..\ v_n^\varnothing}_{eqs} & E_{\mathrm{n}}\ ;\ \mathbf{SLOWRET}_T & (x^n : T_1 \rightarrow ..T_n \rightarrow T) \in\ E_{\mathrm{const}} \wedge \mathrm{os}(x^n) \wedge \neg\ \mathrm{fast}(x^n) \\ E_{\mathrm{n}}\ ;\ \mathbf{SLOWRET}_T & \xrightarrow{\mathrm{Ok}(v^\varnothing)}_{eqs} & E_{\mathrm{n}}\ ;\ v^\varnothing & \text{if } E_{\mathrm{n}}, E_{\mathrm{const}} \vdash_\varnothing v^\varnothing : T \\ E_{\mathrm{n}}\ ;\ \mathbf{SLOWRET}_T & \xrightarrow{\mathrm{Ex}(v^\varnothing)}_{eqs} & E_{\mathrm{n}}\ ;\ \mathbf{raise}\ (v^\varnothing) & \text{if } E_{\mathrm{n}}, E_{\mathrm{const}} \vdash_\varnothing v^\varnothing : \mathsf{exn} \end{array}$$

Comment: The semantics allows the OS return values to be typed with respect to $E_{\mathrm{n}}, E_{\mathrm{const}}$, though for the extant OS call types this makes no difference.

Termination For program termination we have the axiom below.

$$E_{\mathrm{n}}\ ;\ \langle E_s,\ s,\ \mathit{definitions},\ P\rangle \quad \rightarrow \quad \mathbf{TERM}$$

where either (a) there are no threads $\mathbf{n} : \mathit{definitions}\ e$ in P, or (b) there is at least one thread $\mathbf{n} : \mathit{definitions}\ e$ in P but for all such we have $\mathbf{n}$ internally blocked in P.

Case (b) is a useful and sound but very coarse approximation to deadlock detection. In this case the implementation prints a warning.

In addition the programmer can force termination with **exit** as below.

$$E_{\mathrm{n}}\ ;\ \langle E_s,\ s,\ \mathit{definitions},\ P|\mathbf{n} : \mathit{TCC}_{eqs}.\mathbf{op}(\mathbf{exit}\ )\ \ v\rangle \rightarrow \mathbf{TERM}$$

Comment: At present we do not distinguish between successful and unsuccessful termination — cf. the thread exception semantics, which specifies that threads that reduce to a value or to a raised exception silently exit.

16.8.5 Marshalling and unmarshalling

Marshalling

Here we define the reduction step for E_{n} **;** $\langle E_s,\ s,\ \mathit{definitions},\ \mathbf{marshalz}\ \mathrm{MK}\ v^{\varnothing}\ :\ T\rangle$, where

$$\begin{array}{l} \mathit{definitions} = \mathit{definitions}_1\ \mathbf{;;}\ \mathbf{mark}\ \ \mathrm{MK}\ \mathbf{;;}\ \mathit{definitions}_2 \\ \mathbf{mark}\ \ \mathrm{MK}\ \notin\ \mathit{definitions}_2 \end{array}$$

that constructs a marshalled value.

If there is no $\mathbf{mark}$ MK in *definitions*, we fail with E_{n} **;** $\langle E_s,\ s,\ \mathit{definitions},\ \mathbf{raise}\ \ \mathrm{MARSHAL\_FAILURE}\rangle$.

In outline, what we do is prune $\mathit{definitions}_2$, omitting any modules that are not needed and on the way calculating which modules from $\mathit{definitions}_1$ we refer to. We then go through $\mathit{definitions}_1$ making up an import for each of those (this does not unload imports in $\mathit{definitions}_2$ that point within $\mathit{definitions}_1$, instead generating an additional import at the boundary).

Note that this does not involve any *definitions* of the executing thread.

Write fmv(...) for the set of free module external/internal identifier pairs in a gadget (note hashes are all fmv -closed). We make explicit some interesting cases of fmv (first on terms, then on *Mo*s):

$$\begin{array}{lll} \mathrm{fmv}(\mathrm{M}_M.\mathrm{x}) & = & \{\mathrm{M}_M\} \\ \mathrm{fmv}(h) & = & \varnothing \\ & & \\ \mathrm{fmv}(\mathrm{M}_M) & = & \{\mathrm{M}_M\} \\ \mathrm{fmv}(\mathrm{UNLINKED}) & = & \varnothing \end{array}$$

Write locs(...) for the set of locations occurring in a gadget.

Now, given the configuration above, with its *definitions* and s, define a reachability relation $\leadsto$ over the union of the set of M_M defined by the *definitions* and the l in the domain of the store s.

- For M_M defined in $\mathit{definitions}_2$ by $\mathit{definition} = (\mathbf{cmodule}_{h;\mathit{Sig}_0;\mathit{vubs}}\ \mathrm{M}_M : \mathit{Sig}_1\ \ \mathbf{version}\ \ \mathit{vn} = \mathit{Str})$:

$$\begin{array}{llll} \mathrm{M}_M & \leadsto & \mathrm{M}'_{M'} & \mathbf{if}\ \ \mathrm{M}'_{M'} \in \mathrm{fmv}(\mathit{Str}) \\ \mathrm{M}_M & \leadsto & l & \mathbf{if}\ \ \ell \in \mathrm{locs}(\mathit{definition}) \end{array}$$

- For M_M defined in $\mathit{definitions}_2$ by $\mathit{definition} = (\mathbf{cimport}_{h;\mathit{Sig}_0}\ \mathit{vubs}\ \mathrm{M}_M : \mathit{Sig}_1\ \ \mathbf{version}\ \ \mathit{vc}\ \ \mathbf{like}\ \ \mathit{Str}\ \ \mathbf{by}\ \ \mathit{resolvespec} = \mathit{Mo})$:

$$\begin{array}{ll} \mathrm{M}_M \leadsto \mathrm{M}'_{M'} & \mathbf{if}\ \ \mathit{Mo} = \mathrm{M}'_{M'} \\ \mathrm{M}_M \leadsto l & \mathbf{if}\ \ \ell \in \mathrm{locs}(\mathit{definition}) \end{array}$$

- For l,

$$\begin{array}{ll} l \leadsto \mathrm{M}'_{M'} & \mathbf{if}\ \ \mathrm{M}'_{M'} \in \mathrm{fmv}(s(l)) \\ l \leadsto l' & \mathbf{if}\ \ l' \in \mathrm{locs}(s(l)) \end{array}$$

(Note there are no clauses for M_M defined in $\mathit{definitions}_1$. Note that in the import case the free module identifiers of the *Str* will always be empty, as it is a struct that consists exclusively of hashified types, and, similarly, in both cases the free module identifiers of the signatures will be empty.) Let A be the smallest set containing $\mathrm{fmv}(v^{\varnothing}) \cup \mathrm{locs}(v^{\varnothing})$ and closed under $\rightsquigarrow$.

Let S_1, S_2, and L be the partition of A into its module identifiers defined by $\mathit{definitions}_1$, those defined by $\mathit{definitions}_2$, and the locations.

Let $\mathit{definitions}'_2$ be the subsequence of $\mathit{definitions}_2$ containing the definitions of modules in S_2 together with all **mark** s.

Now makeimports *definitions* S constructs imports for the needed modules based on their compiled $\mathit{definitions}_1$.

```
makeimports($\mathit{definitions}$ ;; $\mathbf{cmodule}_{h;eqs;Sig_0}$ $\mathit{vubs}$ $\mathrm{M}_M$ : $\mathit{Sig}_1$ version $\mathit{vn} = \mathit{Str}$)$S$ =
  if $\mathrm{M}_M \in S$ then
    (makeimports($\mathit{definitions}$)($S - \{\mathrm{M}_M\}$) ;;
    $\mathbf{cimport}_{h;Sig_0}$ $\mathrm{M}_M$ : $\mathit{Sig}_1$ version name = $h$ like filter $\mathit{Str}$(limitdom ($\mathit{Sig}_0$))
     by HERE_ALREADY = UNLINKED
    )
   else
    makeimports $\mathit{definitions}$ $S$

makeimports($\mathit{definitions}$ ;; $\mathbf{cimport}_{h;Sig_0}$ $\mathrm{M}_M$ : $\mathit{Sig}_1$ version $\mathit{vc}$ like $\mathit{Str}$ by $\mathit{resolvespec} = \mathit{Mo}$)$S$ =
  if $\mathrm{M}_M \in S$ then
    (makeimports($\mathit{definitions}$)($S - \{\mathrm{M}_M\}$) ;;
    $\mathbf{cimport}_{h;Sig_0}$ $\mathrm{M}_M$ : $\mathit{Sig}_1$ version $\mathit{vc}$ like $\mathit{Str}$ by $\mathit{resolvespec}$ = UNLINKED
    )
   else
    makeimports $\mathit{definitions}$ $S$

makeimports($\mathit{definitions}$ ;; mark MK)$S$ =
  makeimports($\mathit{definitions}$)$S$

makeimports(empty)$S$ = empty
```

> *Comment:* You might think that in the module-initialisation world, reachability would need to go via earlier fields of this module (occurrences of x under a lambda, say) and via top-level *definitions* (M_M.x, say). However, see module field instantiation, internal field case: we have chosen an x-substitution semantics, and so the former case does not arise (x has been substituted away by the time we reach it).

Let $\mathit{definitions}' = \mathrm{makeimports}(\mathit{definitions}_1)S_1$ **;;** $\mathit{definitions}'_2$

Below we write $X \downharpoonleft L$ for X restricted to L. Let $E_{s'} = E_s \downharpoonleft L$. Let $s' = s \downharpoonleft L$.

Let $E_\mathrm{n}{}'$ be the smallest subsequence of E_n including all the abstract names of $E'{}_s$, s', $\mathit{definitions}'$, $v^{\varnothing}$, T and all **nmodule**s in $E_\mathrm{n}{}'$.

The $E_\mathrm{n}{}'$ can be omitted in a production implementation.

Note that marshalling preserves all the original marks we pass through in $\mathit{definitions}_2$, putting them in $\mathit{definitions}'$ and thus in the marshalled value. It does not include the MK we are marshalling with respect to.

Finally, then, we have:

$$\begin{array}{ll} & E_\mathrm{n}\ \mathbf{;}\ \langle E_s,\ s,\ \mathit{definitions},\ \mathbf{marshalz}\ \mathrm{MK}\ v^{\varnothing}\ :\ T\rangle \\ \longrightarrow_{eqs} & E_\mathrm{n}\ \mathbf{;}\ \langle E_s,\ s,\ \mathit{definitions},\ \underline{s}\rangle \end{array}$$

where

$$\text{raw\_unmarshal}(\underline{s}) = \mathbf{marshalled}({E_\text{n}}',\, E_{s'},\, s',\, \mathit{definitions}',\, v^{\varnothing},\, T)$$

If marshal-time typechecking is specified, additionally check $\vdash \mathbf{marshalled}({E_\text{n}}',\, E_{s'},\, s',\, \mathit{definitions}',\, v'^{\varnothing},\, T)\ \mathbf{ok}$. Fail with RUN.TYPECHECK_ON_MARSHAL otherwise.

Unmarshalling

Choosing here to do linking as late as possible, so not doing any linking at unmarshal-time.

$$\begin{array}{ll} & E_\text{n}\ \mathbf{;}\ \langle E_s,\, s,\, \mathit{definitions},\, \mathbf{unmarshal}\ \ \underline{s}\ \mathbf{as}\ T\rangle \\ \longrightarrow_{eqs} & {E_\text{n}}''\ \mathbf{;}\ \langle (E_s, \sigma\ E_{s'}),\, (s, (\sigma\ s').\sigma^{-1}),\, (\mathit{definitions}\ \mathbf{;;}\ \mathit{definitions}'),\, (\sigma\ (v'^{\varnothing}))\rangle \end{array}$$

where

$\text{raw\_unmarshal}(\underline{s}) = \mathbf{marshalled}({E_\text{n}}',\, E_{s'},\, s',\, \mathit{definitions}',\, v'^{\varnothing},\, T')$
the module binders of $\mathit{definitions}'$ are distinct from those of $\mathit{definitions}$
σ is a location injection with domain dom(s') and with ran(σ) disjoint from dom(s)
$T = T'$
${E_\text{n}}'' = \text{merge\_nenvs}(E_\text{n}, {E_\text{n}}')$

(writing $\sigma\ X$ for the result of applying σ as a substitution to X, and so $\sigma\ s'$ for the result of doing that pointwise to the range of s').

As usual, the calculation of ${E_\text{n}}''$ is superfluous if we are not doing run-time type checking.

Note that marshalled modules are always fully evaluated so at unmarshal-time they can be added to the per-runtime definitions not to the thread definitions.

If marshal-time typechecking is specified, additionally check $\vdash \mathbf{marshalled}({E_\text{n}}',\, E_{s'},\, s',\, \mathit{definitions}',\, v^{\varnothing},\, T')\ \mathbf{ok}$, and check that the $\text{merge\_nenvs}(E_\text{n}, {E_\text{n}}')$ above succeeds. Fail with RUN.TYPECHECK_ON_UNMARSHAL otherwise.

$$\begin{array}{ll} & E_\text{n}\ \mathbf{;}\ \langle E_s,\, s,\, \mathit{definitions},\, \mathbf{unmarshal}\ \ \underline{s}\ \mathbf{as}\ T\rangle \\ \longrightarrow_{eqs} & E_\text{n}\ \mathbf{;}\ \langle E_s,\, s,\, \mathit{definitions},\, \mathbf{raise}\ \ \text{UNMARSHAL\_FAILURE}\ \underline{s'}\rangle \end{array}$$

where $\text{raw\_unmarshal}(\underline{s})$ undefined, or $\text{raw\_unmarshal}(\underline{s}) = \mathbf{marshalled}({E_\text{n}}',\, E_{s'},\, s',\, \mathit{definitions}',\, v'^{\varnothing},\, T')$ and $\neg\ T = T'$. $\underline{s'}$ is a string describing the cause of the unmarshal failure.

> *Comment:* Note that unmarshalling will cause existing marks to be shadowed by the marks contained in $\mathit{definitions}'$. This is sometimes desirable, but not always – really, this is a defect of the linear mark/module structure.
>
> *Comment:* Note that marshalling permits one to see through abstraction boundaries in limited fashion, by equality testing (or even more detailed examination) of the marshalled strings for abstract types.

16.8.6 Module field instantiation

Module field instantiation – module case, via import sequence

$$E_{\mathrm{n}} \,;\, \langle E_s,\ s,\ \mathit{definitions},\ \mathrm{M}_M.\mathrm{x}\rangle \quad \longrightarrow_{\mathit{eqs}} \quad E_{\mathrm{n}} \,;\, \langle E_s,\ s,\ \mathit{definitions},\ [v']^{T}_{\mathit{eqs}'}\rangle$$

where

$\mathit{definitions} = \mathit{definitions}_0 \;;;\; \mathit{definition} \;;;\; \mathit{definitions}_1 \;;;\; \mathit{definition}_1 \;;;\; \mathit{definitions}_2 \;;;\; ... \;;;\; \mathit{definition}_n \;;;\; \mathit{definitions}_n$
$\mathit{definition} = \mathbf{cmodule}_{h;\mathit{eqs}_0;\mathit{Sig}_0}\ \mathit{vubs}\ \mathrm{M}_{0\,M_0} : \mathit{Sig}_1 = \mathit{Str}$
$\forall\ i\ \in\ 1..n.\quad \mathit{definition}_i = \mathbf{cimport}_{h_i;\mathit{Sig}_{0\,i}}\ \mathrm{M}_{i\,M_i} : \mathit{Sig}_{1\,i}\ \ \mathbf{version}\ \ \mathit{vc}_i\ \ \mathbf{like}\ \ \mathit{Str}_i\ \ \mathbf{by}\ \ \mathit{resolvespec}_i = \mathrm{M}_{i-1\,M_{i-1}}$
$\mathrm{M}_M = \mathrm{M}_{n\,M_n}$, $\mathit{definitions}_n$ doesn't_define M_M
$(\mathbf{val}\ \mathrm{x}_x : T) \in \mathit{Sig}_{1\,n}$
$(\mathbf{let}\ \mathrm{x}_x = v^{\mathit{eqs}_0}) \in \mathit{Str}$
$v' = v$
$\mathit{eqs}' = \mathit{eqs}_0, \mathrm{eqs\_of\_sign\_str}(h, \mathit{Sig}_0, \mathit{Str}), \mathrm{eqs\_of\_sign\_str}(h_n, \mathit{Sig}_{0\,n}, \mathit{Str}_n)$

Comment: Note that we include eqs_of_sign_str from the **cmodule** and from the ultimate **cimport** (if any), not from any intermediate imports.

Comment: There are two choices here, dependent on the module initialisation semantics. Before, module values could have an expression value field containing free expression identifiers of earlier fields. Then the v' here had to be mutated, taking v with each y free in v replaced by $\mathrm{M}_{n\,M_n}$.y (if $n > 0$) or by M_M.y (if $n = 0$). Now module initialisation substitutes out fields as it goes, so this is no longer needed.

Note that in the earlier semantics that term selfification of the value v depends on that fact that the signature check in linkok does not allow width subsignaturing. What the behaviour should be if one allowed width subsignaturing is unclear.

Module field instantiation – unlinked import case; start looking

$$E_{\mathrm{n}} \,;\, \langle E_s,\ s,\ \mathit{definitions},\ \mathrm{M}_M.\mathrm{x}\rangle \quad \longrightarrow_{\mathit{eqs}} \quad E_{\mathrm{n}} \,;\, \langle E_s,\ s,\ \mathit{definitions},\ \mathbf{resolve}(\mathrm{M}_M.\mathrm{x}, \mathrm{M}_{0\,M_0}, \mathit{resolvespec})\rangle$$

where

$\mathit{definitions} = \mathit{definitions}_0 \;;;\; \mathit{definition}_0 \;;;\; \mathit{definitions}_1 \;;;\; \mathit{definition}_1 \;;;\; \mathit{definitions}_2 \;;;\; ... \;;;\; \mathit{definition}_n \;;;\; \mathit{definitions}_n$
$\mathit{definition} = \mathbf{cimport}_{h_0;\mathit{Sig}_0}\ \mathrm{M}_{0\,M_0} : \mathit{Sig}_1\ \ \mathbf{version}\ \ \mathit{vc}_0\ \ \mathbf{like}\ \ \mathit{Str}_0\ \ \mathbf{by}\ \ \mathit{resolvespec}_0 = \mathrm{UNLINKED}$
$\forall\ i\ \in\ 1..n.\quad \mathit{definition}_i = \mathbf{cimport}_{h_i;\mathit{Sig}_{0\,i}}\ \mathrm{M}_{i\,M_i} : \mathit{Sig}_{1\,i}\ \ \mathbf{version}\ \ \mathit{vc}_i\ \ \mathbf{like}\ \ \mathit{Str}_i\ \ \mathbf{by}\ \ \mathit{resolvespec}_i = \mathrm{M}_{i-1\,M_{i-1}}$
$\mathrm{M}_M = \mathrm{M}_{n\,M_n}$
$\mathit{definitions}_n$ doesn't_define M_M
$(\mathbf{val}\ \mathrm{x}_x : T) \in \mathit{Sig}_{1\,n}$

Module field instantiation – resolve URI

$$\begin{array}{rl} & E_{\mathrm{n}} \,;\, \langle E_s,\ s,\ \mathit{definitions},\ \mathbf{resolve}(\mathrm{M}_M.\mathrm{x}, \mathrm{M}'_{M'}, (\mathit{URI}, \mathit{resolvespec}))\rangle \\ \xrightarrow{\mathrm{GetURI}(\mathit{URI})}_{\mathit{eqs}} & E_{\mathrm{n}} \,;\, \langle E_s,\ s,\ \mathit{definitions},\ \mathbf{resolve\_blocked}(\mathrm{M}_M.\mathrm{x}, \mathrm{M}'_{M'}, \mathit{resolvespec})\rangle \end{array}$$

Module field instantiation – resolve case, HERE_ALREADY

$$E_{\mathrm{n}} \,;\, \langle E_s,\ s,\ \mathit{definitions},\ \mathbf{resolve}(\mathrm{M}_M.\mathrm{x}, \mathrm{M}'_{M'}, (\mathrm{HERE\_ALREADY}, \mathit{resolvespec}_0))\rangle \quad \longrightarrow_{\mathit{eqs}} \quad E_{\mathrm{n}} \,;\, \langle E_s,\ s,\ \mathit{definitions}_9,\ e\rangle$$

where

$\mathit{definitions} \quad = \quad \mathit{definitions}_1 \;;;\; \mathit{definition} \;;;\; \mathit{definitions}_2$
$\mathit{definitions}_2$ doesn't_define $\mathrm{M}'_{M'}$
$\mathit{definition} \quad = \quad \mathbf{cimport}_{h;\mathit{Sig}_0}\ \mathrm{M}'_{M'} : \mathit{Sig}_1\ \ \mathbf{version}\ \ \mathit{vc}\ \ \mathbf{like}\ \ \mathit{Str}\ \ \mathbf{by}\ \ \mathit{resolvespec} = \mathrm{UNLINKED}$

Let Ms be the sequence of the $\mathrm{M}''_{M''}$ defined by a $\mathit{definition}'$ in $\mathit{definitions}_1$ satisfying $\mathrm{linkok}\,(E_\mathrm{n}, \mathit{definition}', \mathit{definition}\ \oplus (= \mathrm{M}'_{M'}))$, where $\mathit{definition}\ \ \oplus\ (=\ \mathrm{M}'_{M'})$ is as $\mathit{definition}$ but with $= \mathrm{M}'_{M'}$ replacing $= \text{UNLINKED}$.

If Ms is nonempty then, taking $\mathrm{M}''_{M''}$ to be its last element,

$$\begin{array}{lcl} \mathit{definitions}_9 & = & \mathit{definitions}_1\ \mathbf{;;}\ \mathit{definition}_9\ \mathbf{;;}\ \mathit{definitions}_2 \\ \mathit{definition}_9 & = & \mathbf{cimport}_{h;\mathit{Sig}_0}\ \mathrm{M}'_{M'} : \mathit{Sig}_1\ \ \mathbf{version}\ \ \mathit{vc}\ \ \mathbf{like}\ \ \mathit{Str}\ \ \mathbf{by}\ \ \mathit{resolvespec} = \mathrm{M}''_{M''} \\ e & = & \mathrm{M}_M.\mathrm{x} \end{array}$$

otherwise if Ms is empty take

$\mathit{definitions}_9 = \mathit{definitions}$ and $e = \mathbf{resolve}(\mathrm{M}_M.\mathrm{x}, \mathrm{M}'_{M'}, \mathit{resolvespec}_0)$.

Module field instantiation – resolve case, STATIC_LINK

$$\begin{array}{ll} & E_\mathrm{n}\ \mathbf{;}\ \langle E_s,\ s,\ \mathit{definitions},\ \mathbf{resolve}(\mathrm{M}_M.\mathrm{x}, \mathrm{M}'_{M'}, (\text{STATIC\_LINK}, \mathit{resolvespec}_0))\rangle \\ \longrightarrow_{\mathit{eqs}} & E_\mathrm{n}\ \mathbf{;}\ \langle E_s,\ s,\ \mathit{definitions},\ \mathbf{raise}\ \ \text{RESOLVE\_FAILURE}\rangle \end{array}$$

The intention for imports with STATIC_LINK *resolvespec*s is that they should have been statically linked, so if we reach one at runtime it is an error. We do not yet define a separate static linking phase, however, so they are not yet very useful.

Module field instantiation – resolveblocked case, got some $\mathit{definitions}'$

$$\begin{array}{ll} & E_\mathrm{n}\ \mathbf{;}\ \langle E_s,\ s,\ \mathit{definitions},\ \mathbf{resolve\_blocked}(\mathrm{M}_M.\mathrm{x}, \mathrm{M}'_{M'}, \mathit{resolvespec}_0)\rangle \\ \xrightarrow{\mathrm{DeliverURI}(E_\mathrm{n}', \mathit{definitions}')}_{\mathit{eqs}} & E_{\mathrm{n}9}\ \mathbf{;}\ \langle E_s,\ s,\ \mathit{definitions}_9,\ e\rangle \end{array}$$

where

$$\begin{array}{lcl} \mathit{definitions} & = & \mathit{definitions}_1\ \mathbf{;;}\ \mathit{definition}\ \mathbf{;;}\ \mathit{definitions}_2 \\ \mathit{definitions}_2\ \ \text{doesn't\_define}\ \ \mathrm{M}'_{M'} & & \\ \mathit{definition} & = & \mathbf{cimport}_{h;\mathit{Sig}_0}\ \mathrm{M}'_{M'} : \mathit{Sig}_1\ \ \mathbf{version}\ \ \mathit{vc}\ \ \mathbf{like}\ \ \mathit{Str}\ \ \mathbf{by}\ \ \mathit{resolvespec} = \text{UNLINKED} \end{array}$$

Let $E_{\mathrm{n}9} = \mathrm{merge\_nenvs}(E_\mathrm{n}, E_\mathrm{n}{}')$. This is superfluous if we are not doing run-time type checking.

Note that we disallow module field instantiation with non-value definitions. This ensures the new $\mathit{definitions}'$ can be inserted into the existing per-runtime *definitions* before the **cimport** which must be linked to them, without breaking the invariant that the per-runtime *definitions* are always fully evaluated. To relax this would need additional mechanism to block instantiation from a linked but not-yet-evaluated module.

If

- $\mathit{definitions}'$ consists only of definition values.
- $E = E_1(\mathit{definitions}')$
- $\mathrm{dom}(\mathit{definitions}') \cap \mathrm{dom}(\mathit{definitions}) = \varnothing$ (achievable by alpha equivalence)
- Letting Ms be the sequence of the $\mathrm{M}''_{M''}$ defined by a $\mathit{definition}'$ in $\mathit{definitions}'$ satisfying $\mathrm{linkok}\,(E_{\mathrm{n}9}, \mathit{definition}', \mathit{definition}\ \oplus (= \mathrm{M}'_{M'}))$, we have Ms nonempty with a last element $\mathrm{M}''_{M''}$.

then

$$\begin{array}{lcl} \mathit{definitions}_9 & = & \mathit{definitions}_1\ \mathbf{;;}\ \mathit{definitions}'\ \mathbf{;;}\ \mathit{definition}_9\ \mathbf{;;}\ \mathit{definitions}_2 \\ \mathit{definition}_9 & = & \mathbf{cimport}_{h;\mathit{Sig}_0}\ \mathrm{M}'_{M'} : \mathit{Sig}_1\ \ \mathbf{version}\ \ \mathit{vc}\ \ \mathbf{like}\ \ \mathit{Str}\ \ \mathbf{by}\ \ \mathit{resolvespec} = \mathrm{M}''_{M''} \\ e & = & \mathrm{M}_M.\mathrm{x} \end{array}$$

else

$\mathit{definitions}_9 = \mathit{definitions}$ and $e = \mathbf{resolve}(\mathrm{M}_M.\mathrm{x}, \mathit{resolvespec}_0)$.

In the implementation, if a byte string that does not lex or parse as a $\mathit{definitions}'$ is returned for this URI it is treated as a $\mathrm{CannotFindURI}$ transition.

If doing run-time type checking, check additionally $E_{\mathrm{const}} \vdash \mathit{definitions}' \;\triangleright\; E$ and $\mathrm{linkok}\,({E_{\mathrm{n}}}', \mathit{definitions}')$ and that the $\mathrm{merge\_nenvs}$ succeeds (or fail with TYPECHECK_ON_GET_URI).

Module field instantiation – resolveblocked case, didn't get any $\mathit{definitions}'$

$$\begin{array}{ll} & E_{\mathrm{n}}\,\mathbf{;}\,\langle E_s,\ s,\ \mathit{definitions},\ \mathbf{resolve\_blocked}(\mathrm{M}_M.\mathrm{x}, \mathrm{M'}_{M'}, \mathit{resolvespec})\rangle \\ \xrightarrow{\mathrm{CannotFindURI}}_{\mathit{eqs}} & E_{\mathrm{n}}\,\mathbf{;}\,\langle E_s,\ s,\ \mathit{definitions},\ \mathbf{resolve}(\mathrm{M}_M.\mathrm{x}, \mathrm{M'}_{M'}, \mathit{resolvespec})\rangle \end{array}$$

Module field instantiation – run out of $\mathit{resolvespec}$

$$E_{\mathrm{n}}\,\mathbf{;}\,\langle E_s,\ s,\ \mathit{definitions},\ \mathbf{resolve}(\mathrm{M}_M.\mathrm{x}, \mathrm{M'}_{M'}, \mathrm{empty})\rangle \quad \longrightarrow_{\mathit{eqs}} \quad E_{\mathrm{n}}\,\mathbf{;}\,\langle E_s,\ s,\ \mathit{definitions},\ \mathbf{raise}\ \ \mathrm{RESOLVE\_FAILURE}\rangle$$

16.8.7 Name operations

We write $\mathrm{fn}\,(v)$ for the set of names $\mathbf{n}$ occuring in v and $\mathrm{fns}\,(v)$ for the set of simple names $\mathbf{nn}$ occurring in v. The primitive swapping function $\mathrm{SSwap}_{\mathit{eqs}}(BC_1.\mathbf{nn}_1, BC_2.\mathbf{nn}_2)$ in v yields the result of replacing, in v, all occurrences of $\mathbf{nn}_i$ with $\mathrm{revbc}_{\mathit{eqs}}(BC_i).BC_{2-i}.\mathbf{nn}_{2-i}$. These are defined homomorphically through the abstract syntax, save that they do *not* propagate though $\mathbf{hash}(\ldots)$. (Note that $\mathbf{marshalled}(\ldots)$ does not occur (hereditarily) in the abstract syntax for expressions). The auxiliary function revbc reverses the sequence of brackets in a bracket context, and is defined as follows:

$$\begin{array}{lcl} \mathrm{revbc}_{\mathit{eqs}}(\_) & = & \_ \\ \mathrm{revbc}_{\mathit{eqs}}([\_]^{T'}_{\mathit{eqs}'}.BC) & = & \mathrm{revbc}_{\mathit{eqs}'}(BC).[\_]^{T'}_{\mathit{eqs}} \end{array}$$

Given a configuration with $\mathit{definitions}$ and s, define a reachability relation $\leadsto$ over the domain of the store s.

- $l \leadsto l'$ **if** $l' \in \mathrm{locs}(s(l))$

Let the reachable locations from a value v^{eqs} be the smallest set A containing $\mathrm{locs}(v^{eqs})$ and closed under $\leadsto$.

$$\frac{\mathrm{n} \notin \mathrm{dom}(E_{\mathrm{n}})}{E_{\mathrm{n}}\ \mathbf{;}\ \langle E_s,\ s,\ \mathit{definitions},\ \mathbf{fresh}_T\rangle \longrightarrow_{eqs} E_{\mathrm{n}}, \mathrm{n} : T\ \ \mathsf{name}\ \mathbf{;}\ \langle E_s,\ s,\ \mathit{definitions},\ \mathrm{n}_T\rangle}$$

$$\frac{\begin{array}{l} L \text{ is the set of locations reachable from } v^{eqs} \\ \sigma \text{ is a location injection with domain } L \text{ and with } \mathrm{ran}(\sigma) \text{ disjoint from } \mathrm{dom}(s) \\ E_{s'} = \sigma(E_s \downharpoonright L) \\ s' = \lambda l\ \in \mathrm{ran}(\sigma).\, \mathrm{SSwap}_{eqs}(BC.\mathbf{nn}, BC'.\mathbf{nn}')\ \text{ in }\ \sigma\ s(\sigma^{-1}(l)) \\ v_2^{eqs} = \mathrm{SSwap}_{eqs}(BC.\mathbf{nn}, BC.\mathbf{nn}')\ \text{ in }\ \sigma^{-1}\ v^{eqs} \end{array}}{E_{\mathrm{n}}\ \mathbf{;}\ \langle E_s,\ s,\ \mathit{definitions},\ \mathbf{swap}\ (BC.\mathbf{nn})\ \mathbf{and}\ (BC'.\mathbf{nn}')\ \mathbf{in}\ \ v^{eqs}\rangle \longrightarrow_{eqs} E_{\mathrm{n}}\ \mathbf{;}\ \langle (E_s, E_{s'}),\ (s, s'),\ \mathit{definitions},\ v_2^{eqs}\rangle}$$

$$\frac{\begin{array}{l} L \text{ is the set of locations reachable from } v_2^{eqs} \\ \underline{b} = (\mathrm{erase\_brackets}(v_1^{eqs}) \in \mathrm{fns}\,(v_2^{eqs}) \cup \mathrm{fns}\,(\mathrm{ran}(s \downharpoonright L))) \end{array}}{E_{\mathrm{n}}\ \mathbf{;}\ \langle E_s,\ s,\ \mathit{definitions},\ v_1^{eqs}\ \mathbf{freshfor}\ \ v_2^{eqs}\rangle \longrightarrow_{eqs} E_{\mathrm{n}}\ \mathbf{;}\ \langle E_s,\ s,\ \mathit{definitions},\ \underline{b}\rangle}$$

$$\frac{\begin{array}{l} L \text{ is the set of locations reachable from } v^{eqs} \\ \mathit{nset} = \mathrm{fns}\,(v^{eqs}) \cup \mathrm{fn}\,(\mathrm{ran}(s \downharpoonright L)) \\ \{\mathbf{n}_1, ..., \mathbf{n}_k\} = \mathrm{filter}\ (\lambda\mathbf{n}.\, \mathrm{typeof}(\mathbf{n}) =\ T)\ \mathit{nset} \\ \forall\ i \neq j.\mathbf{n}_i \neq \mathbf{n}_j \\ v'^{eqs} = \mathbf{n}_1 :: ... :: \mathbf{n}_k :: [\,]_{T\ \ \mathsf{name}} \end{array}}{E_{\mathrm{n}}\ \mathbf{;}\ \langle E_s,\ s,\ \mathit{definitions},\ \mathbf{support}_T v^{eqs}\rangle \longrightarrow_{eqs} E_{\mathrm{n}}\ \mathbf{;}\ \langle E_s,\ s,\ \mathit{definitions},\ v'^{eqs}\rangle}$$

$\mathbf{hash}(T', \underline{s}, v^{\varnothing})$	$\longrightarrow_{eqs}$	$\mathbf{hash}(T', \underline{s}, \mathbf{n})$	$\mathrm{erase\_brackets}(v^{\varnothing}) = \mathbf{n} \wedge v^{\varnothing} \neq \mathbf{n}$
$\mathbf{compare\_name}_T\ v_1^{\varnothing}\ v_2^{\varnothing}$	$\longrightarrow_{eqs}$	0	$\mathrm{erase\_brackets}(v_1^{\varnothing}) = \mathrm{erase\_brackets}(v_2^{\varnothing})$
$\mathbf{compare\_name}_T\ v_1^{\varnothing}\ v_2^{\varnothing}$	$\longrightarrow_{eqs}$	-1	$\mathrm{erase\_brackets}(v_1^{\varnothing}) < \mathrm{erase\_brackets}(v_2^{\varnothing})$
$\mathbf{compare\_name}_T\ v_1^{\varnothing}\ v_2^{\varnothing}$	$\longrightarrow_{eqs}$	1	$\mathrm{erase\_brackets}(v_1^{\varnothing}) > \mathrm{erase\_brackets}(v_2^{\varnothing})$
$\mathbf{name\_of\_tie}\ \mathrm{TIECON}(v_1^{eqs}, v_2^{eqs})$	$\longrightarrow_{eqs}$	v_1^{eqs}	
$\mathbf{val\_of\_tie}\ \mathrm{TIECON}(v_1^{eqs}, v_2^{eqs})$	$\longrightarrow_{eqs}$	v_2^{eqs}	

Comment: Note that (in contrast to FreshOCaml) reachability here does go through the store.

Comment: Note that this makes the semantics of **support** potentially surprising: $\mathbf{support}_T e$ is the set of names in e that were *constructed at* type T, not all those that have type T in the present context. A positive consequence is that the reduction rule for **support** is simple to implement because it is independent of the presence of brackets, thus the same reductions are obtained after bracket erasure.

A negative consequence is that the rule fails to account for the type equalities introduced by the brackets surrounding a name, thus possibly not collecting all the relevant names; the rule can also see through abstraction boundaries, thus possibly collecting too many names.

Comment: With module initialisation, one has to decide whether reachability goes through *definitions*, e.g. if you have a store location containing **function** $() \rightarrow \mathrm{M}_M.\mathrm{x}$ and $\mathrm{M}_M.\mathrm{x}$ has either a store location or a **function** $() \rightarrow \mathrm{M'}_{M'}.\mathrm{x'}$. Here we choose not — note that this is a different notion of reachability from that used in marshalling.

Comment: Note that we treat fresh and hash-generated names uniformly here, allowing swapping etc. over (and between) both.

Comment: We can (indirectly) send and receive modules, but we have no way of swapping over them. This is clearly suspicious and is one more point in favour of more first-class modules.

Comment: Note that the polytypic **swap** , **support** and **freshfor** can see through abstraction boundaries.

Comment: One might want $\mathbf{support}$ to return a duplicate-free list w.r.t. the embedded simple names, not just w.r.t. $\mathbf{n}$ equality.

16.8.8 Concurrency

Basic thread operations: termination, $\mathbf{create\_thread}$, $\mathbf{self}$, $\mathbf{kill}$

Below we write $\mathbf{n} \ =_{\text{erased}} \ \mathbf{n}'$ for $\text{erase\_brackets}(\mathbf{n}) = \text{erase\_brackets}(\mathbf{n}')$, $\mathbf{n} \ \in_{\text{erased}} \ \mathit{nset}$ for $\text{erase\_brackets}(\mathbf{n}) \in \text{erase\_brackets}(\mathit{nset})$ and similarly for $\notin_{\text{erased}}$.

$P|\mathbf{n} : v^{\varnothing} \longrightarrow P|0$

$P|\mathbf{n} : \mathit{TC}_{\mathit{eqs}}.\mathbf{raise} \ \ v^{\varnothing} \longrightarrow P|0$

$P|\mathbf{n} : \mathit{TCC}_{\mathit{eqs}}.\mathbf{op}(\mathbf{create\_thread})^3 \ \ \mathbf{n}' \ v_1^{\varnothing} \ \ v_2^{\varnothing} \longrightarrow P|\mathbf{n} : \mathit{TCC}_{\mathit{eqs}}.()|\mathbf{n}' : (v_1^{\varnothing} \ \ v_2^{\varnothing}) \qquad \mathbf{n}' \ \notin_{\text{erased}} \{\mathbf{n}\}, \text{dom}(P)$

$P|\mathbf{n} : \mathit{TCC}_{\mathit{eqs}}.\mathbf{op}(\mathbf{self})^1 \ \ () \longrightarrow P|\mathbf{n} : \mathit{TCC}_{\mathit{eqs}}.\mathbf{n}$

$P|\mathbf{n}' : e_1|\mathbf{n} : \mathit{TCC}_{\mathit{eqs}}.\mathbf{op}(\mathbf{kill})^1 \ \ \mathbf{n}' \longrightarrow P|0|\mathbf{n} : \mathit{TCC}_{\mathit{eqs}}.()$

$P|\mathbf{n} : \mathit{TCC}_{\mathit{eqs}}.\mathbf{op}(\mathbf{kill})^1 \ \ \mathbf{n} \longrightarrow P|0$

$P|\mathbf{n} : \mathit{TCC}_{\mathit{eqs}}.\mathbf{op}(\mathbf{create\_thread})^3 \ \ \mathbf{n}' \ v_1^{\varnothing} \ \ v_2^{\varnothing} \longrightarrow P|\mathbf{n} : \mathit{TCC}_{\mathit{eqs}}.\mathbf{raise} \ \ \text{EXISTENT\_NAME} \qquad \mathbf{n}' \ \in_{\text{erased}} \{\mathbf{n}\}, \text{dom}(P)$

$P|\mathbf{n} : \mathit{TCC}_{\mathit{eqs}}.\mathbf{op}(\mathbf{kill})^1 \ \ \mathbf{n}' \longrightarrow P|\mathbf{n} : \mathit{TCC}_{\mathit{eqs}}.\mathbf{raise} \ \ \text{NONEXISTENT\_THREAD} \qquad \mathbf{n}' \ \notin_{\text{erased}} \{\mathbf{n}\}, \text{dom}(P)$

Comment: At present threads terminate silently, in both value and raised-exception cases. An alternative to the latter would be $P|\mathbf{raise} \ \ v^{\varnothing} \longrightarrow 0$, which is more fail-stop (a good thing, in principle) but possibly more annoying. At very least, a thread dying by with a raised exception should currently generate a warning to the console. Ultimately we should perhaps arrange some way to have exception handlers around threads.

Comment: We allow a thread can kill itself — may not make much difference, but this seems slightly more intuitive than the alternative of raising an exception. This is a difference from the $\mathbf{thunkify}$ semantics.

Comment: $\mathbf{kill}$ and $\mathbf{thunkify}$ are both dangerous operations in that they can remove threads which hold mutexes, which will then never be released. We expect them to be used only within the implementations of libraries that provide both $\mathbf{kill}$- or $\mathbf{thunkify}$-like operations together with safe thread interaction constructs. Otherwise, the preferred idiom for killing a thread should be to ask it to kill itself; it can then exit cleanly.

Thunkify

Say an e is an *atomic internal blocked form* in P if e is either $\mathbf{op}(\mathbf{lock})^1 \ \ \mathbf{n}'$ with $\mathbf{n}'' : \text{MX}(\mathbf{true})$ also in P for $\mathbf{n}'' \ =_{\text{erased}} \ \mathbf{n}'$, or $\mathbf{op}(\mathbf{waiting})^2 \ \ \mathbf{n}' \ \mathbf{n}''$.

Say an e is an *atomic blocked form* in P if e is either $\mathbf{SLOWRET}_T$, $\mathbf{op}(\mathbf{lock})^1 \ \ \mathbf{n}'$ with $\mathbf{n}'' : \text{MX}(\mathbf{true})$ also in P for some $\mathbf{n}'' \ =_{\text{erased}} \ \mathbf{n}'$, or $\mathbf{op}(\mathbf{waiting})^2 \ \ \mathbf{n}' \ \mathbf{n}''$, or $\mathbf{resolve\_blocked}(\text{M}_M.\text{x}, \text{M}'_{M'}, \mathit{resolvespec})$.

Say $\mathbf{n}$ is *internally blocked* in P if there is an $\mathbf{n} : \mathit{TCC}'^{\varnothing}_{\mathit{eqs}_2}.e$ in P where e is an atomic internal blocked form in P.

Say $\mathbf{n}$ is *blocked* in P if there is a $\mathbf{n} : \mathit{TCC}'^{\varnothing}_{\mathit{eqs}_2}.e$ in P where e is an atomic blocked form in P.

Say $\mathbf{n}$ is in a *fast call* in P if there is a $\mathbf{n} : {TCC'}^{\varnothing}_{eqs_2}.e$ in P where e is $\mathbf{RET}_T$ or $\mathbf{resolve}(\mathrm{M}_M.\mathrm{x}, \mathrm{M'}_{M'}, \mathit{resolvespec})$.

$P|\mathbf{n} : TCC_{eqs}.\mathbf{op}(\mathbf{thunkify})^1\ \ tks \rightarrow P'|\mathbf{n} : TCC_{eqs}.e$ if $\mathrm{Thunkify}\ \ tks\ P = (e, P')$

$P|\mathbf{n} : TCC_{eqs}.\mathbf{op}(\mathbf{thunkify})^1\ \ tks \rightarrow P|\mathbf{n} : TCC_{eqs}.\mathbf{raise}\ \ e$ if $\mathrm{Thunkify}\ \ tks\ P = \mathrm{FAIL}(e)$

$P|\mathbf{n} : TCC_{eqs}.\mathbf{op}(\mathbf{unthunkify})^1\ \ thks \rightarrow P|\mathbf{n} : TCC_{eqs}.()|P'$ if $\mathrm{Unthunkify}\ \ thks\ (\{\mathbf{n}\} \cup \mathsf{dom}(P)) = P'$

$P|\mathbf{n} : TCC_{eqs}.\mathbf{op}(\mathbf{unthunkify})^1\ \ thks \rightarrow P|\mathbf{n} : TCC_{eqs}.\mathbf{raise}\ \ e$ if $\mathrm{Unthunkify}\ \ thks\ (\{\mathbf{n}\} \cup \mathsf{dom}(P)) = \mathrm{FAIL}(e)$

The auxiliaries are defined in ML-like pseudocode below. Thunkify takes a list tks of thunkkeys and the process state. It gives either BLOCK, if this **thunkify** cannot execute now (there is no transition rule in this case, thus blocking progress until thunkification is possible), or $\mathrm{FAIL}(e)$, if it should raise an exception, or the abstract syntax of an Acute function that takes a list of names (of the right shape) and has a body that unthunkifies the thunked mutexes, cvars and threads with those names and the remaining (non-thunkified) objects. It uses DoThunkify, which is defined recursively on tks, P_1 and i, building up the pattern and body of Acute function as it goes.

Unthunkify calculates an Acute process to be added to the running program, or returns $\mathrm{FAIL}(e)$ if an exception should be raised.

There is no syntactic distinction between the pseudocode and object-language constructors; hopefully the context makes it clear.

```
$\mathrm{Thunkify}\ \ tks\ P =$
  $\mathbf{match}\ \ \mathrm{DoThunkify}\ \ tks\ P\ P\ 0\ \ \mathbf{with}$
    $\mathrm{BLOCK} \rightarrow \mathrm{BLOCK}$
  $|\,\mathrm{FAIL}(e) \rightarrow \mathrm{FAIL}(e)$
  $|(p, e, P_2) \rightarrow$
      $(\mathbf{function}\ \ x \rightarrow$
        $\mathbf{match}\ \ x\ \ \mathbf{with}$
            $p \rightarrow \mathbf{unthunkify}\ e$
          $|\ \_ \rightarrow \mathbf{raise}\ \ \mathrm{THUNKIFY\_KEYLISTS\_MISMATCH}), P_2$

$\mathrm{DoThunkify}\ \ tks\ P\ P_1\ i =$
  $\mathbf{match}\ \ tks\ \ \mathbf{with}$
    $[\,] \rightarrow [\,], (), P_1$
  $|\ tk :: tks_0 \rightarrow$
      $\mathbf{match}\ \ tk\ \ \mathbf{with}$
        $\mathrm{MUTEX}\ \mathbf{n}_0 \rightarrow$
          $\mathbf{if}\ \ \exists\ \mathbf{n}, P_0, \underline{b}.P_1 \equiv \mathbf{n} : \mathrm{MX}(\underline{b})|P_0 \wedge \mathbf{n}_0\ =_{\text{erased}}\ \mathbf{n}\ \ \mathbf{then}$
            $\mathbf{let}\ \ p, e, P_2 = \mathrm{DoThunkify}\ \ tk_0\ P\ P_0\ (i+1)\ \mathbf{in}$
            $(\mathrm{MUTEX}(x_i : \mathsf{mutex\ \ name})) :: p, \mathrm{THUNKED\_MUTEX}(x_i, \underline{b}) :: e, P_2$
           $\mathbf{else}$
            $\mathrm{FAIL}(\mathrm{NONEXISTENT\_MUTEX})$
      $|\mathrm{CVAR}\ \mathbf{n}_0 \rightarrow$
          $\mathbf{if}\ \ \exists\ \mathbf{n}, P_0, v^{\varnothing}.P_1 \equiv \mathbf{n} : \mathrm{CV}|P_0 \wedge \mathbf{n}_0\ =_{\text{erased}}\ \mathbf{n}\ \ \mathbf{then}$
            $\mathbf{let}\ \ p, e, P_2 = \mathrm{DoThunkify}\ \ tk_0\ P\ P_0\ (i+1)\ \mathbf{in}$
            $(\mathrm{CVAR}(x_i : \mathsf{cvar\ \ name})) :: p, \mathrm{THUNKED\_CVAR}(x_i) :: e, P_2$
           $\mathbf{else}$
            $\mathrm{FAIL}(\mathrm{NONEXISTENT\_CVAR})$
      $|\mathrm{THREAD}(\mathbf{n}_0, tmode) \rightarrow$
```

```
        if $\exists\ \mathbf{n}, P_0, e_0.P_1 \equiv \mathbf{n} : e_0|P_0 \wedge \mathbf{n}_0 \ =_{\text{erased}}\ \mathbf{n}$ then
          let $p, e, P_2 = \text{DoThunkify}\ \ tk_0\ P\ P_0\ (i+1)$ in
          if $\mathbf{n}$ not blocked in $P$ and not in a fast call in $P$ then
            $(\text{THREAD}(x_i : \mathsf{thread\ \ name},_{:}\ \mathsf{thunkifymode})) :: p,$
            $\text{THUNKED\_THREAD}(x_i, \mathbf{function}\ () \to e_0) :: e, P_2$
          else if $e_0 = CC'^{\varnothing}_{eqs_2}.e_1$ and $e_1$ is an atomic blocked form in $P$
              and $tmode = \text{INTERRUPTING}$ then
            $(\text{THREAD}(x_i : \mathsf{thread\ \ name},_{:}\ \mathsf{thunkifymode})) :: p,$
            $\text{THUNKED\_THREAD}(x_i, \mathbf{function}\ () \to CC'^{\varnothing}_{eqs_2}.\mathbf{raise}\ \ \text{EINTR}) :: e, P_2$
          else
            BLOCK
        else if $\exists\ P_0, \mathit{definition}, \mathit{definitions}, e_0.P_1 \equiv \mathbf{n} : \mathit{definition}\ \mathit{definitions}\ e_0|P_0$ then
          FAIL(THUNKIFY_THREAD_IN_DEFINITION)
        else
          FAIL(NONEXISTENT_THREAD)
```

where the x_i are all fresh.

```
Unthunkify $\mathit{thks}\ \mathit{ns}$ = match $\mathit{thks}$ with
   $[\,] \to 0$
|$\mathit{thk} :: \mathit{thks}_0 \to$
    (Unthunkify $\mathit{thks}_0\ \mathit{ns}$)
      |
    (match $\mathit{thk}$ with
      $\text{THUNKED\_MUTEX}(\mathbf{n}, \underline{b}) \to$ if $\mathbf{n} \notin_{\text{erased}} ns$ then $\mathbf{n} : \text{MX}(\underline{b})$ else FAIL(EXISTENT_NAME)
    |$\text{THUNKED\_CVAR}(\mathbf{n}) \to$ if $\mathbf{n} \notin_{\text{erased}} ns$ then $\mathbf{n} : \text{CV}$ else FAIL(EXISTENT_NAME)
    |$\text{THUNKED\_THREAD}(\mathbf{n}, v^{\varnothing}) \to$ if $\mathbf{n} \notin_{\text{erased}} ns$ then $\mathbf{n} : (v^{\varnothing}\ ())$ else FAIL(EXISTENT_NAME)
    )
```

Comment: There is a stylistic choice as to how a thunkified value is expressed. In principle it might just be a normal function in the language so far, but this requires non-trivial coding to ensure atomicity, e.g. to ensure one thread does not start before all the other threads are spawned and mutexes recreated. We therefore have a single semantic step, using some non-source-internal-language constructors to code the thunked state.

Comment: Maybe one would want a thunkifymode to apply also to mutexes and condition variables, eg to block until they reach a certain state.

Comment: If a thread tries to thunkify itself the NONEXISTENT_THREAD exception is raised.

Comment: If you feed the wrong things to a thunk you just get a match failure, not an unthunkify failure, which is slightly unpleasant.

Mutexes: $\mathbf{create\_mutex}$, $\mathbf{lock}$, $\mathbf{try\_lock}$, $\mathbf{unlock}$.

$$P|\mathbf{n} : TCC_{eqs}.\mathbf{op}(\mathbf{create\_mutex})^1\ \mathbf{n}' \quad\longrightarrow\quad P|\mathbf{n} : TCC_{eqs}.()|\mathbf{n}' : \mathrm{MX}(\mathbf{false}) \qquad \mathbf{n}' \notin_{\text{erased}} \mathrm{dom}(P), \{\mathbf{n}\}$$

$$P|\mathbf{n}' : \mathrm{MX}(\mathbf{false})|\mathbf{n} : TCC_{eqs}.\mathbf{op}(\mathbf{lock})^1\ \mathbf{n}_1' \quad\longrightarrow\quad P|\mathbf{n}' : \mathrm{MX}(\mathbf{true})|\mathbf{n} : TCC_{eqs}.() \qquad \mathbf{n}_1' =_{\text{erased}} \mathbf{n}'$$

$$P|\mathbf{n}' : \mathrm{MX}(\underline{b})|\mathbf{n} : TCC_{eqs}.\mathbf{op}(\mathbf{try\_lock})^1\ \mathbf{n}_1' \quad\longrightarrow\quad P|\mathbf{n}' : \mathrm{MX}(\mathbf{true})|\mathbf{n} : TCC_{eqs}.(\neg\ \underline{b}) \qquad \mathbf{n}_1' =_{\text{erased}} \mathbf{n}'$$

$$P|\mathbf{n}' : \mathrm{MX}(\mathbf{true})|\mathbf{n} : TCC_{eqs}.\mathbf{op}(\mathbf{unlock})^1\ \mathbf{n}_1'|\mathbf{n}'' : TC'_{eqs'}.\mathbf{op}(\mathbf{lock})^1\ \mathbf{n}_2'$$
$$\longrightarrow\quad P|\mathbf{n}' : \mathrm{MX}(\mathbf{true})|\mathbf{n} : TCC_{eqs}.()|\mathbf{n}'' : TC'_{eqs'}.() \qquad \mathbf{n}_1' =_{\text{erased}} \mathbf{n}' \wedge \mathbf{n}_2' =_{\text{erased}}$$

$$P|\mathbf{n}' : \mathrm{MX}(\mathbf{true})|\mathbf{n} : TCC_{eqs}.\mathbf{op}(\mathbf{unlock})^1\ \mathbf{n}_1' \quad\longrightarrow\quad P|\mathbf{n}' : \mathrm{MX}(\mathbf{false})|\mathbf{n} : TCC_{eqs}.() \qquad (*) \qquad \mathbf{n}_1' =_{\text{erased}} \mathbf{n}'$$

$$P|\mathbf{n}' : \mathrm{MX}(\mathbf{false})|\mathbf{n} : TCC_{eqs}.\mathbf{op}(\mathbf{unlock})^1\ \mathbf{n}_1' \quad\longrightarrow\quad P|\mathbf{n}' : \mathrm{MX}(\mathbf{false})|\mathbf{n} : TCC_{eqs}.() \qquad \mathbf{n}_1' =_{\text{erased}} \mathbf{n}'$$

$$P|\mathbf{n} : TCC_{eqs}.\mathbf{op}(\mathbf{create\_mutex})^1\ \mathbf{n}' \quad\longrightarrow\quad P|\mathbf{n} : TCC_{eqs}.\mathbf{raise}\ \ \mathrm{EXISTENT\_NAME} \qquad \mathbf{n}' \in_{\text{erased}} \mathrm{dom}(P), \{\mathbf{n}\}$$

$$P|\mathbf{n} : TCC_{eqs}.\mathbf{op}(\mathbf{lock})^1\ \mathbf{n}' \quad\longrightarrow\quad P|\mathbf{n} : TCC_{eqs}.\mathbf{raise}\ \ \mathrm{NONEXISTENT\_MUTEX} \qquad \mathbf{n}' \notin_{\text{erased}} \mathrm{dom}(P), \{\mathbf{n}\}$$

$$P|\mathbf{n} : TCC_{eqs}.\mathbf{op}(\mathbf{unlock})^1\ \mathbf{n}' \quad\longrightarrow\quad P|\mathbf{n} : TCC_{eqs}.\mathbf{raise}\ \ \mathrm{NONEXISTENT\_MUTEX} \qquad \mathbf{n}' \notin_{\text{erased}} \mathrm{dom}(P), \{\mathbf{n}\}$$

$$P|\mathbf{n} : TCC_{eqs}.\mathbf{op}(\mathbf{try\_lock})^1\ \mathbf{n}' \quad\longrightarrow\quad P|\mathbf{n} : TCC_{eqs}.\mathbf{raise}\ \ \mathrm{NONEXISTENT\_MUTEX} \qquad \mathbf{n}' \notin_{\text{erased}} \mathrm{dom}(P), \{\mathbf{n}\}$$

$(\ *\ )\ \neg\ \exists(\mathbf{n}'' : TCC'_{eqs'}.\mathbf{op}(\mathbf{lock})^1\ \mathbf{n}_2') \in\ P\ .\ \mathbf{n}_2' =_{\text{erased}} \mathbf{n}'$

Comment: These rules give an error if one $\mathbf{lock}$, $\mathbf{unlock}$, or $\mathbf{try\_lock}$ for a nonexistent mutex — which situation couldn't arise in the single-machine case, but can in ours. The simplest thing to do seems to be to have mutex and condition variable names global (which seems perfectly sensible, really), and to raise exceptions if one tries to use a nonexistent one.

Comment: Do we want to distinguish in the semantics between a $\mathbf{lock}\ m$ that has actually blocked and one that has not yet attempted to execute (introducing explicit "slow" states for them)? Can not see any need.

Condition variables: $\mathbf{create\_cvar}$, $\mathbf{wait}$, $\mathbf{signal}$, $\mathbf{broadcast}$.

$$P|\mathbf{n} : TCC_{eqs}.\mathbf{op}(\mathbf{create\_cvar})^1\ \mathbf{n}' \quad\longrightarrow\quad P|\mathbf{n} : TCC_{eqs}.()|\mathbf{n}' : \mathrm{CV}$$

$$P|\mathbf{n}' : \mathrm{CV}|\mathbf{n}'' : \mathrm{MX}(\mathbf{true})|\mathbf{n} : TCC_{eqs}.\mathbf{op}(\mathbf{wait})^2\ \mathbf{n}_1'\ \mathbf{n}_1''|\mathbf{n}''' : TCC'_{eqs'}.\mathbf{op}(\mathbf{lock})^1\ \mathbf{n}_2''$$
$$\longrightarrow\quad P|\mathbf{n}' : \mathrm{CV}|\mathbf{n}'' : \mathrm{MX}(\mathbf{true})|\mathbf{n} : TCC_{eqs}.\mathbf{op}(\mathbf{waiting})^2\ \mathbf{n}'\ \mathbf{n}''|\mathbf{n}''' : TCC'_{eqs'}.()$$
$$\mathbf{n}_1' =_{\text{erased}} \mathbf{n}' \wedge \mathbf{n}_1'' =_{\text{erased}} \mathbf{n}'' \wedge \mathbf{n}_2'' =_{\text{erased}} \mathbf{n}''$$

$$P|\mathbf{n}' : \mathrm{CV}|\mathbf{n}'' : \mathrm{MX}(\mathbf{true})|\mathbf{n} : TCC_{eqs}.\mathbf{op}(\mathbf{wait})^2\ \mathbf{n}_1'\ \mathbf{n}_1''$$
$$\longrightarrow\quad P|\mathbf{n}' : \mathrm{CV}|\mathbf{n}'' : \mathrm{MX}(\mathbf{false})|\mathbf{n} : TCC_{eqs}.\mathbf{op}(\mathbf{waiting})^2\ \mathbf{n}_1'\ \mathbf{n}_1''$$
$$(\mathbf{n}'' : TCC'_{eqs'}.\mathbf{op}(\mathbf{lock})^1\ \mathbf{n}_2'') \notin\ P$$
$$\mathbf{n}_1' =_{\text{erased}} \mathbf{n}' \wedge \mathbf{n}_1'' =_{\text{erased}} \mathbf{n}'' \wedge \mathbf{n}_2'' =_{\text{erased}} \mathbf{n}''$$

$$P|\mathbf{n}' : \mathrm{CV}|\mathbf{n} : TCC_{eqs}.\mathbf{op}(\mathbf{signal})^1\ \mathbf{n}_1' \quad\longrightarrow\quad \mathrm{restart\_one}(P, \mathbf{n}')|\mathbf{n}' : \mathrm{CV}|\mathbf{n} : TCC_{eqs}.() \qquad \mathbf{n}_1' =_{\text{erased}} \mathbf{n}'$$

$$P|\mathbf{n}' : \mathrm{CV}|\mathbf{n} : TCC_{eqs}.\mathbf{op}(\mathbf{broadcast})^1\ \mathbf{n}_1' \quad\longrightarrow\quad \mathrm{restart\_all}(P, \mathbf{n}')|\mathbf{n} : TCC_{eqs}.() \qquad \mathbf{n}_1' =_{\text{erased}} \mathbf{n}'$$

$$P|\mathbf{n} : TCC_{eqs}.\mathbf{op}(\mathbf{create\_cvar})^1\ \mathbf{n}' \quad \to \quad P|\mathbf{n} : TCC_{eqs}.\mathbf{raise}\ \text{EXISTENT\_NAME} \qquad \mathbf{n}' \in_{\text{erased}} \text{dom}(P), \{\mathbf{n}\}$$

$$P|\mathbf{n} : TCC_{eqs}.\mathbf{op}(\mathbf{wait})^2\ \mathbf{n}'\ \mathbf{n}'' \quad \to \quad P|\mathbf{n} : TCC_{eqs}.\mathbf{raise}\ \text{NONEXISTENT\_MUTEX} \qquad \forall\, \mathbf{n}''_1 =_{\text{erased}} \mathbf{n}''.\mathbf{n}''_1 : \text{MX}(\underline{b}) \notin P$$

$$P|\mathbf{n} : TCC_{eqs}.\mathbf{op}(\mathbf{wait})^2\ \mathbf{n}'\ \mathbf{n}'' \quad \to \quad P|\mathbf{n} : TCC_{eqs}.\mathbf{raise}\ \text{NONEXISTENT\_CVAR} \qquad \forall\, \mathbf{n}'_1 =_{\text{erased}} \mathbf{n}'.\mathbf{n}'_1 : \text{CV} \notin P$$

$$P|\mathbf{n} : TCC_{eqs}.\mathbf{op}(\mathbf{signal})^1\ \mathbf{n}' \quad \to \quad P|\mathbf{n} : TCC_{eqs}.\mathbf{raise}\ \text{NONEXISTENT\_CVAR} \qquad \forall\, \mathbf{n}'_1 =_{\text{erased}} \mathbf{n}'.\mathbf{n}' : \text{CV} \notin P$$

$$P|\mathbf{n} : TCC_{eqs}.\mathbf{op}(\mathbf{broadcast})^1\ \mathbf{n}' \quad \to \quad P|\mathbf{n} : TCC_{eqs}.\mathbf{raise}\ \text{NONEXISTENT\_CVAR} \qquad \forall\, \mathbf{n}'_1 =_{\text{erased}} \mathbf{n}'.\mathbf{n}' : \text{CV} \notin P$$

$$\begin{aligned}&P|\mathbf{n}' : \text{CV}|\mathbf{n}'' : \text{MX}(\mathbf{false})|\mathbf{n} : TCC_{eqs}.\mathbf{op}(\mathbf{wait})^2\ \mathbf{n}'_1\ \mathbf{n}'' \\ &\qquad \to \quad P|\mathbf{n}' : \text{CV}|\mathbf{n}'' : \text{MX}(\mathbf{false})|\mathbf{n} : TCC_{eqs}.\mathbf{raise}\ \text{MUTEX\_EPERM} \qquad \mathbf{n}'_1 =_{\text{erased}} \mathbf{n}'\end{aligned}$$

The auxiliaries are as follows:

$\text{restart\_one}(P, \mathbf{n}')$ gives P but with a single $\mathbf{n} : TCC^{()}_{eqs_2}.\mathbf{op}(\mathbf{waiting})^2\ \mathbf{n}'_1\ \mathbf{n}''_1$, if one exists with $\mathbf{n}' =_{\text{erased}} \mathbf{n}'_1$, replaced by $\mathbf{n} : TCC^{()}_{eqs_2}.\mathbf{op}(\mathbf{lock})^1\ \mathbf{n}''$. If none exist, then $\text{restart\_one}(P, \mathbf{n}') = P$.

$\text{restart\_all}(P, \mathbf{n}')$ gives P but with all $\mathbf{n} : TCC^{()}_{eqs_2}.\mathbf{op}(\mathbf{waiting})^2\ \mathbf{n}'_1\ \mathbf{n}''_1$ for $\mathbf{n}' =_{\text{erased}} \mathbf{n}'_1$ replaced by $\mathbf{n} : TCC^{()}_{eqs_2}.\mathbf{op}(\mathbf{lock})^1\ \mathbf{n}''$.

> *Comment:* Here $\mathbf{op}(\mathbf{wait})^2\ \mathbf{n}'\ \mathbf{n}''$ for nonexistent $\mathbf{n}'$ *and* $\mathbf{n}''$ nondeterministically gives one or the other error.
>
> *Comment:* POSIX specifies that waiting without holding the mutex passed is an error. LinuxThreads appears (from the man page) not to implement this check; replace MX(**true**) with MX($\underline{b}$) above and remove the MX(false) rule above to mimic this. (If you do this, it is almost certain that your code has a race, so it is nice for the OS to let you know).
>
> *Comment:* The "mutex handover" rule that atomically performs an unlock together with a lock is necessary for a sane and fair implementation.
>
> *Comment:* Should a restarted **wait** atomically lock its mutex or not?
>
> *Comment:* Applications may rely on some fairness property that this semantics does not express. Specifically, the threads waiting on a mutex or condition variable should be woken in FIFO order (i.e., for mutexes and **signal**, the first waiting thread should be woken; for **broadcast**, threads should be woken in such a way that the first waiting thread is first to be scheduled. Scheduling does not have to be this strict, but something like the fairness this implies is assumed by application programmers.

16.8.9 Polymorphism

We have not yet addressed the abstraction-preserving semantics for polymorphism, which will entail adding coloured brackets to the reduction axioms below and adding bracket-pushing rules for these constructs.

Without runtime type names or coloured brackets, the reduction axioms would be as below.

$$(\Lambda\ t \to e)\ T \quad \to_{eqs} \quad \{T/t\}e$$

$$\mathbf{let}\ \{t, x\} = (\{T, e\}\ \mathbf{as}\ \ T')\ \mathbf{in}\ \ e_2 \quad \to_{eqs} \quad \{T/t, e/x\}e_2$$

$$\begin{aligned}&\mathbf{namecase}\ (\{T, (\mathbf{n}', e)\}\ \mathbf{as}\ \ T')\ \mathbf{with}\ \{t, (x_1, x_2)\}\ \mathbf{when}\ \ x_1 = e \to e_2\ \mathbf{otherwise}\ \to e_3 \\ &\qquad \to_{eqs} \quad \{T/t, e/x\}e_2 \quad \text{if erase\_brackets}(\mathbf{n}) = \text{erase\_brackets}(\mathbf{n}') \\ &\qquad \to_{eqs} \quad e_3 \qquad\qquad\ \ \text{if erase\_brackets}(\mathbf{n}) \neq \text{erase\_brackets}(\mathbf{n}')\end{aligned}$$

Adding runtime type names, but still without coloured brackets, the unpack rule should be more like the rule below, which generates a fresh type name at each unpack to mirror the static semantics.

$$E_{\mathrm{n}}\ \mathbf{;}\ \langle E_s,\ s,\ \mathit{definitions},\ \mathbf{let}\ \{t, x\} = (\{T, e\}\ \mathbf{as}\ \ T')\ \mathbf{in}\ \ e_2\rangle \quad \rightarrow_{eqs} \quad E_{\mathrm{n}}, \mathrm{n} : \mathrm{EQ}(T)\ \mathbf{;}\ \langle E_s,\ s,\ \mathit{definitions},\ \{\mathrm{n}/t, e/x\}e_2\rangle$$

16.9 Type Preservation and Progress

We have not attempted to prove type preservation and progress results, as the definition is of a size where either a hand or machine proof would be a very major undertaking (and a hand proof would probably contain many errors). Indeed, there may well be problems in the definition. However, it is still worth while stating precisely the properties that we believe should hold.

Some confidence in the soundness of the definition comes from running the implementation with runtime typechecking, typechecking the configuration after every reduction step.

The statements of the conjectures are the basis for this runtime typechecking, and are also a useful guide to the intuition while developing the definitions.

Conjecture 16.1 (Typed Compilation)

1. If $\mathrm{compile}_\Phi(\mathit{sourcefilename})E_\mathrm{n} \rightsquigarrow (E'_0, E'_1, \mathit{compiledunit}')$ and $\mathit{compiledunit}' = {E_\mathrm{n}}'$ **;** $\mathit{definitions}'\ \mathit{eo}'$ then for some T and for $\mathrm{n} \notin \mathrm{dom}({E_\mathrm{n}}')$ we have $\vdash {E_\mathrm{n}}', \mathrm{n}_{\mathsf{thread}}$ **;** $\langle \mathrm{empty}, \mathrm{empty}, \mathit{definitions}, \mathrm{n}_{\mathsf{thread}} : \mathit{compiledunit}' \rangle : T$.

Conjecture 16.2 (Type Preservation) If

1. $\vdash E_\mathrm{n}$ **;** $\langle E_s,\ s,\ \mathit{definitions},\ P\rangle : T$ and
2. E_n **;** $\langle E_s,\ s,\ \mathit{definitions},\ P\rangle \xrightarrow{\ell}_\varnothing {E_\mathrm{n}}'$ **;** $\langle E'_s,\ s',\ \mathit{definitions}',\ P'\rangle$

then $\vdash {E_\mathrm{n}}'$ **;** $\langle E'_s,\ s',\ \mathit{definitions}',\ P'\rangle : T$.

Conjecture 16.3 (Progress) If $\vdash E_\mathrm{n}$ **;** $\langle E_s,\ s,\ \mathit{definitions},\ P\rangle : T$ and there exists a thread $\mathbf{n} : \mathit{definitions}\ e$ in P with $\mathbf{n}$ neither blocked in P nor in a fast call in P then there exists ${E_\mathrm{n}}'$ **;** $\langle \mathit{definitions}',\ E'_s,\ s',\ P'\rangle$ such that E_n **;** $\langle E_s,\ s,\ \mathit{definitions},\ P\rangle \xrightarrow{\ell}_\varnothing {E_\mathrm{n}}'$ **;** $\langle E'_s,\ s',\ \mathit{definitions}',\ P'\rangle$.

Comment: One could also formulate a result saying the compilation always succeeds for well-typed source programs that do not include files or involve linking or **with** ! etc. It would not be very informative, though.

Comment: The progress property should in principle be strengthened to ensure that in a well-typed configuration *every* non-blocked thread can make progress. To do so would require more data in the semantics, however, e.g. to track the threads involved in multi-thread reductions, so it is not worth doing now.

16.10 Runtime type checking

The main check is, for each configuration reached by the evaluator, that

$\vdash E_\mathrm{n}$ **;** $\langle E_s,\ s,\ \mathit{definitions},\ P\rangle : \mathsf{unit}$

(or indicate FAILURE.RUN.TYPECHECK_OF_CONFIGURATION).

We can also run typechecks during compilation on compiled units (or indicate FAILURE.COMPILE.TYPECHECK_OF_COMPILEDUNIT), at unmarshal time (or indicate FAILURE.RUN.TYPECHECK_ON_UNMARSHAL), and when compiled definitions are taken from a URI during module field instantiation (or indicate FAILURE.RUN.TYPECHECK_ON_GET_URI). These are described in §16.7, §16.8.5, and §16.8.5 respectively.

Failure of any of these checks indicates an error in the typesystem or the implementation.

The implementation has switches to control whether these checks are done. They all require structured names to be enabled.

16.11 Vacuous bracket optimization

The semantics above constructs coloured brackets in many circumstances where they are not required — where they do not change the colour. A production implementation would erase all brackets. For our implementation, while we need to keep the colour-changing brackets in order to do runtime typechecking, for execution speed it is useful to optimize away as many vacuous brackets as possible.

Accordingly, we define here an optimized variant semantics, which can, optionally, be used in our implementation.

1. in the rule **Module field instantiation – module case, via import sequence** (page 136), omit the brackets on the rhs if $eqs' = eqs$.
2. in the rule for $[\mathbf{raise}\ \ v^{eqs'}]^{T}_{eqs'}$ (page 129), omit the brackets on the rhs if $eqs' = eqs$.
3. in the rule for $\mathbf{marshal}\ \mathrm{MK}\ v^{eqs}\ :\ T$ (page 129), omit the brackets on the rhs if $eqs = \varnothing$.
4. in each of the 5 rules for pushing brackets through non-nullary constructors (page 130), omit the brackets on the rhs if $eqs' = eqs$.
5. in the rule for pushing brackets through lambda (page 130), omit the outer brackets on the rhs if $eqs' = eqs$.
6. in the rule for bracket type revelation (page 130), omit the brackets on the rhs if $eqs' = eqs$.
7. in the rule for bracket elimination (page 130), omit the brackets on the rhs if $eqs'' = eqs$.
8. in the two rules for op^n and x^n (page 131), omit the brackets on the rhs if $eqs = \varnothing$.

Note that brackets constructed to be used in a substitution (in the definition of $\mathrm{matchsub}$, the $\mathbf{function}$ rule, and the $\mathbf{let\ \ rec}$ rule) cannot be optimized away without analysis of the structure of the expression in which they are being substituted. We do not do this.

The Type Preservation property should still hold.

16.12 Closures

16.12.1 Value closures

For efficiency, the implementation uses an environment instead of substitution. This requires function values to be represented as closures. In this section we extend the small-step semantics given above to model this. We remain in the style of a calculus, rather than an abstract machine.

The reduction arrow now gains another index, the environment ρ, in addition to the colour eqs it already carries:

$$e \longrightarrow^{\rho}_{eqs} e'$$

We must also introduce a term form to change the environment – recall that a closure replaces the environment, rather than extending the existing one. This same form can be used also to implement other forms of binding that only extend the environment. We write $\mathbf{inenv}\ \ \rho'\ \ \mathbf{do}\ \ e$ to denote that e is evaluated in the environment ρ'. (Recall that colour changes are already handled by brackets.) $\mathbf{inenv}$ itself is an environment-changing evaluation context.

Evaluation contexts now carry an inner and an outer environment as well as an inner and outer colour; $\mathbf{inenv}\ \ \rho'\ \ \mathbf{do}\ \ e$ is an environment-changing context. This has the same effect as the following rule:

$$\frac{e \longrightarrow^{\rho'}_{eqs} e'}{\mathbf{inenv}\ \ \rho'\ \ \mathbf{do}\ \ e \longrightarrow^{\rho}_{eqs} \mathbf{inenv}\ \ \rho'\ \ \mathbf{do}\ \ e'}$$

Let ρ be an environment, i.e., a list of pairs $\{[v^{eqs}]^{T}_{eqs}/x\}$, such that earlier pairs scope over later ones. Observe that ρ is both closed and colour-closed: the environment has no free identifiers, and since every value is enclosed in brackets, it is valid at any colour.

Comment: This definition is not entirely correct as stated — below we allow a recursive closure to contain an environment that includes a pair whose second element is the original closure. This is expressible in our implementation language (FreshOCaml, following OCaml, allows recursive value bindings of the form required) but is not well-formed in the naive set-theoretic model of the semantics below. The definition should be adapted.

The new expression and value forms are as follows:

$$\begin{array}{lll}
e & ::= & \ldots \\
 & & \mathbf{inenv}\ \ \rho\ \ \mathbf{do}\ \ e \\
 & & \mathbf{Clos}(\rho, x_2 : T_2, BC_2, e_1, \textsc{None}) \\
 & & \mathbf{Clos}(\rho, x_2 : T_2, BC_2, e_1, \textsc{Some}(x_1 : T_1, BC_1)) \\
v & ::= & \ldots \text{ except for } \mathbf{function}\ \ x \to e \\
 & & \mathbf{Clos}(\rho, x_2 : T_2, BC_2, e_1, \textsc{None}) \\
 & & \mathbf{Clos}(\rho, x_2 : T_2, BC_2, e_1, \textsc{Some}(x_1 : T_1, BC_1))
\end{array}$$

Note that these new expression and value forms appear only in running programs; that is, $\mathbf{function}\ \ x \to e$ remains a source value (and is allowed to appear as a value in a struct, for example), but is no longer a value in a running program (the corresponding closure is, instead).

Identifier lookup uses the environment (this is the delayed substitution in action). We may incorporate the vacuous bracket optimisation, since every binding in the environment has an outermost bracket:

$$\begin{array}{llll}
x \ \longrightarrow^{\rho}_{eqs} & v^{eqs} & \text{if } \rho(x) = [v^{eqs}]^{T}_{eqs} & \text{identifier lookup, bracket eliminated} \\
x \ \longrightarrow^{\rho}_{eqs} & [v^{eqs'}]^{T}_{eqs'} & \text{if } \rho(x) = [v^{eqs'}]^{T}_{eqs'} \text{ and } eqs' \neq eqs & \text{identifier lookup, bracket required}
\end{array}$$

The normal binding constructs (**match** , **try**) use **inenv** in the obvious way:

$\mathbf{match}\ \ v^{eqs}\ \mathbf{with}\ \ p_1 \to e_1 \vert .. \vert p_n \to e_n$	$\to^{\rho}_{eqs}$	$\mathbf{inenv}\ (\rho + \mathrm{matchsub}_{eqs}(v^{eqs}, p_k))\ \mathbf{do}\ \ e_k$	match success (a)
$\mathbf{match}\ \ v^{eqs}\ \mathbf{with}\ \ p_1 \to e_1 \vert .. \vert p_n \to e_n$	$\to^{\rho}_{eqs}$	$\mathbf{raise}\ \ \textsc{Match\_failure}\ v'$	match failure (b)
try ...	...	...	similarly
everything else is straightforward			

Elimination of **inenv** occurs when evaluation within it is complete:

$\mathbf{inenv}\ \ \rho'\ \ \mathbf{do}\ \ v^{eqs}$	$\to^{\rho}_{eqs}$	v^{eqs}	scope exit

Recall the existing rules for functions:

$[\mathbf{function}\ (x_2 : T_2) \to e]^{T_2' \to T_3'}_{eqs'}$	$\to_{eqs}$	$\mathbf{function}\ (x_2 : T_2') \to [\{[x_2]^{T_2}_{eqs'}/x_2\}e]^{T_3'}_{eqs'}$	bracket pushing through function
$(\mathbf{function}\ (x_2 : T_2) \to e)\ v^{eqs}$	$\to_{eqs}$	$\{[v^{eqs}]^{T_2}_{eqs}/x_2\}e$	function application
$\mathbf{let}\ \ \mathbf{rec}\ \ x_1 : T = \mathbf{function}\ (x_2 : T') \to e_1\ \ \mathbf{in}\ \ e_2$	$\to_{eqs}$		recursive function application

$$\{[\{[\mathbf{let}\ \ \mathbf{rec}\ \ x_1 : T = \mathbf{function}\ (x_2 : T') \to e_1\ \ \mathbf{in}\ \ x_1]^{T}_{eqs}/x_1\}\mathbf{function}\ (x_2 : T') \to e_1]^{T}_{eqs}/x_1\}e_2$$

It may seem that a closure should carry its colour as well as its environment. In fact, however, it shouldn't – just as a function doesn't. Colour change is effected by binding, and we take care in substitution (directly or with environments) to insert sufficient brackets to get this right. Therefore we merely have to get bracket pushing correct for closures.

Thus, a closure carries the function argument, function body, and the environment it was defined in. It also carries a bracket context BC, discussed below, and for recursive closures it carries the name and type of the recursive binder, also discussed below. On application, we reintroduce the environment, and bind the argument. On pushing a bracket through a closure, the environment is untouched (it is colour-closed); the bracket is accumulated in the bracket context.

The rules for closures are as follows (notice that the typing rules imply $T_1 \approx T_2 \to T_3$).

$\mathbf{function}\ (x_2 : T_2) \to e$	$\to^{\rho}_{eqs}$	$\mathbf{Clos}(\rho, x_2 : T_2, \_, e, \textsc{None})$	closure formation
$\mathbf{let}\ \ \mathbf{rec}\ \ x_1 : T_1 = \mathbf{function}\ (x_2 : T_2) \to e_1\ \ \mathbf{in}\ \ e_2$	$\to^{\rho}_{eqs}$	$\mathbf{inenv}\ \ \rho'\ \ \mathbf{do}\ \ e_2$	recursive closure formation
where $\rho' = \rho + \{[\mathbf{Clos}(\rho', x_2 : T_2, \_, e_1, \textsc{Some}(x_1 : T_1, \_))]^{T_1}_{eqs}/x_1\}$			
$\mathbf{Clos}(\rho', x_2 : T_2, BC_2, e, xo)\ v^{eqs}$	$\to^{\rho}_{eqs}$	$\mathbf{inenv}\ (\rho' + \{BC_2.[v^{eqs}]^{T_2}_{eqs}/x_2\})\ \mathbf{do}\ \ e$	closure application
$[\mathbf{Clos}(\rho', x_2 : T_2, BC_2, e, \textsc{None})]^{T_2' \to T_3'}_{eqs'}$	$\to^{\rho}_{eqs}$	$\mathbf{Clos}(\rho', x_2 : T_2', BC_2.[\_]^{T_2}_{eqs'}, [e]^{T_3'}_{eqs'}, \textsc{None})$	bracket pushing through closure
$[\mathbf{Clos}(\rho', x_2 : T_2, BC_2, e, \textsc{Some}(x_1 : T_1, BC_1))]^{T_2' \to T_3'}_{eqs'}$			bracket pushing through recursive closure

$$\to^{\rho}_{eqs} \mathbf{Clos}(\rho', x_2 : T_2', BC_2.[\_]^{T_2}_{eqs'}, [e]^{T_3'}_{eqs'}, \textsc{Some}(x_1 : T_2' \to T_3', BC_1.[\_]^{T_1}_{eqs'}))$$

Notice that **let rec** is similar to **function** , and recursive and non-recursive closures share the same application rule. For flattening purposes, however, we must store the name and type $x_1 : T_1$ of the recursive binder so that we are able to reconstruct the appropriate **let rec** ; otherwise naïve application of the ρ would fail to terminate.

In the original bracket pushing through function rule, we perform a substitution on the bound variable(s). We would like to delay this substitution as well. In order to do this, we simply accumulate a sequence of pushed brackets within the closure, adding them to the environment only at application time. This means that bindings in the environment may now be to values surrounded by arbitrary bracket contexts, rather than values only; administrative reductions may be required in order to reduce these to a value. We do not consider this an important difficulty.

We may perform the vacuous bracket optimisation when appending to bracket contexts in closure application and bracket pushing through closure, as follows:

$\mathrm{maybe\_cons\_bs}_{eqs_0}\ [\_]^{T}_{eqs}\ \_$	$=$	$\_$	if $eqs_0 = eqs$
$\mathrm{maybe\_cons\_bs}_{eqs_0}\ [\_]^{T}_{eqs}\ BC.[\_]^{T'}_{eqs_{00}}$	$=$	$BC.[\_]^{T'}_{eqs_{00}}$	if $eqs_{00} = eqs$
$\mathrm{maybe\_cons\_bs}_{eqs_0}\ [\_]^{T}_{eqs}\ BC$	$=$	$BC.[\_]^{T}_{eqs}$	otherwise

Here we only append the bracket if it differs from the innermost colour of the existing bracket context (or the ambient colour eqs_0 if the bracket context is empty).

To correctly define flattening in the recursive case, we must extend the codomain of ρ to include bracket-context forms $BC.*$ (denoted by a literal $*$), in addition to the usual expressions. Flattening can now be defined as follows:

$$\begin{array}{lcl}
\mathrm{flattenclos}_\rho(x) & = & \mathrm{flattenclos}_\rho\ \rho(x) \quad \text{where } \rho(x) \neq BC.* \\
\mathrm{flattenclos}_\rho(x) & = & BC.x \quad \text{where } \rho(x) = BC.* \\
\mathrm{flattenclos}_\rho(\mathbf{inenv}\ \ \rho'\ \ \mathbf{do}\ \ e) & = & \mathrm{flattenclos}_{\rho'}(e) \\
\mathrm{flattenclos}_\rho(\mathbf{Clos}(\rho', x_2 : T_2, BC_2, e_1, \text{NONE})) & = & \mathbf{function}\ (x_2 : T_2) \rightarrow \mathrm{flattenclos}_{(\rho' + \{BC_2.x_2/x_2\})}(e_1) \\
\multicolumn{3}{l}{\mathrm{flattenclos}_\rho(\mathbf{Clos}(\rho', x_2 : T_2, BC_2, e_1, \text{SOME}(x_1 : T_1, BC_1)))} \\
\multicolumn{3}{l}{\quad = \mathbf{let\ \ rec}\ \ x_1 : T_1 = \mathbf{function}\ (x_2 : T_2) \rightarrow \mathrm{flattenclos}_{(\rho' + \{BC_1.*/x_1, BC_2.*/x_2\})}(e_1)\ \mathbf{in}\ \ x_1} \\
\multicolumn{3}{l}{\text{everything else is just recursive descent}}
\end{array}$$

The new codomain form is used where a identifier must be wrapped once, rather than recursively expanded.

The new constructs may be typed directly. We make use of an auxiliary function, envenv, defined as follows:

$$\begin{array}{lcl}
\mathrm{envenv}(\varnothing) & = & \varnothing \\
\mathrm{envenv}(\{[v^{eqs}]^T_{eqs}/x, \rho'\}) & = & x : T, \mathrm{envenv}(\rho')
\end{array}$$

Then the type rules are as follows:

$$\frac{E_\mathrm{n}, E_0, \mathrm{envenv}(\rho) \vdash_{eqs} e : T}{E_\mathrm{n}, E_0, E \vdash_{eqs} \mathbf{inenv}\ \ \rho\ \ \mathbf{do}\ \ e : T}$$

$$\frac{\begin{array}{l} E_\mathrm{n}, E_0, x_2 : T_2 \vdash_{eqs} BC_2.x_2 : T_2' \\ E_\mathrm{n}, E_0, \mathrm{envenv}(\rho), x_2 : T_2' \vdash_{eqs} e_1 : T_3 \end{array}}{E_\mathrm{n}, E_0, E \vdash_{eqs} \mathbf{Clos}(\rho, x_2 : T_2, BC_2, e_1, \text{NONE}) : T_2 \rightarrow T_3}$$

$$\frac{\begin{array}{l} E_\mathrm{n}, E_0, x_1 : T_1 \vdash_{eqs} BC_1.x_1 : T_1' \\ E_\mathrm{n}, E_0, x_2 : T_2 \vdash_{eqs} BC_2.x_2 : T_2' \\ E_\mathrm{n}, E_0, \mathrm{envenv}(\rho) \vdash_{eqs} x_1 : T_1' \\ E_\mathrm{n}, E_0, \mathrm{envenv}(\rho), x_2 : T_2' \vdash_{eqs} e_1 : T_3 \\ E_\mathrm{n}, E_0 \vdash_{eqs} T_1 \approx T_2 \rightarrow T_3 \end{array}}{E_\mathrm{n}, E_0, E \vdash_{eqs} \mathbf{Clos}(\rho, x_2 : T_2, BC_2, e_1, \text{SOME}(x_1 : T_1, BC_1)) : T_1'}$$

where E_0 is that prefix of the environment which arises from E_const and the enclosing *definitions*.

16.12.2 Type closures

A naïve implementation of polymorphism would perform a substitution for each instance of type application, negating much of the benefit of value closures. We therefore introduce type closures as well.

The environment ρ now contains pairs T/t as well as $[v^{eqs}]^T_{eqs}/x$. The reduction arrow, value closures, and the **inenv** form all remain the same. We add a new form $\mathbf{TClos}(\rho, t, e)$ with the obvious meaning; a type abstraction is no longer a value, and instead reduces to the obvious type closure. **inenv** is used everywhere instead of type substitution. $\mathrm{flattenclos}_{\_}\ \_$ is extended in the obvious way. Brackets and type closures may be commuted freely.

Care must be taken to ensure that whenever a type is used, ρ is taken into account; if the type is taken out of its context (as in reduction of a **marshal** expression for example) it must be flattened first.

Part IV

Communication Infrastructure Example

Here we give the Acute code for the communication infrastructure example outlined in §11. It consists of modules `Tcp_padded`, `Tcp_connection_management`, `Tcp_string_messaging`, `Local_channel`, `Distributed_channel`, `Npi1`, `Npi2`, and `Npi`, followed by two simple clients of the `Npi` library, `npi-recv` and `npi-mig`.

```
(* tcp.ac *)
(* This file contains Tcp_padded, Tcp_connection_management and Tcp_string_messaging modules *)

(* These use the Sockets API and local concurrency - threads and mutexes. *)
(* Both are hash modules, providing abstract types of handles.           *)


includesource "util.ac"

(* ****************************************************************** *)
(* **                                                              ** *)
(* ** Tcp_padded                                                   ** *)
(* **                                                              ** *)
(* ****************************************************************** *)

(* The Tcp_padded module implements a wire-format send and receive for
   arbitrary strings.

   The wire format encoding of a string consists of 21 bytes
   containing an ASCII pretty-print of its length followed by the
   string itself.  This is not efficient(!) but is conveniently
   human-readable.
*)

module hash Tcp_padded :
  sig
    val send : Tcp.fd -> ((Tcp.ip * Tcp.port) option) -> string -> unit
    val recv : Tcp.fd -> string
  end =
  struct
    let send fd ippo data =
      let pad data n =
        let padding =
          String.make ( n - (String.length data)) ' ' in
        (data ^ padding) in
      let data_length = String.length data in
      let data_length_string =
        pad (Pervasives.string_of_int data_length) 21 in
      let rec send_all s =
        let no_options = [] in
let s' = (Tcp.send fd ippo s no_options) in
if   0 = (String.length s') then () else send_all s'
      in
      send_all (data_length_string ^ data)

    let recv fd =
      let rec recv_n_bytes = function n ->
        let no_options = [] in
        let (s,_) = Tcp.recv fd n no_options in
        let _ = IO.print_string ("Tcp_padded.recv got " ^ Pervasives.string_of_int (String.length s)
          ^" bytes; expecting " ^ Pervasives.string_of_int (n - String.length s) ^ " more
n") in
      (*  let _ = IO.print_string ("in particular, Tcp_padded.recv got ---" ^ s ^ "---
n") in*)
        let l = String.length s in
        if l = 0 then (Tcp.close fd; raise (Failure "socket closed by the other party") )
        else if l >= n then s else s ^ (recv_n_bytes (n-l)) in
      let data_length_string = recv_n_bytes 21 in
      let first_space = String.index data_length_string ' ' in
      let data_length_string' = String.sub data_length_string 0 first_space in
      let data_length = Pervasives.int_of_string data_length_string' in
      recv_n_bytes data_length

  end

(* ****************************************************************** *)
(* **                                                              ** *)
(* ** Tcp_connection_management                                    ** *)
(* **                                                              ** *)
(* ****************************************************************** *)

(* The Tcp_connection_management module manages collections of TCP
   connections.

   daemon takes a local address (an Tcp.ip option * Tcp.port option)
   and an incoming-connection-handler function and creates a listening
   socket on that address, spawning a thread that invokes the supplied
   function for any incoming connection and then adds the connection
   to a list.  daemon returns a handle which must be passed in to the
   other functions.  (Using handles rather than module state allows a
   single runtime to have multiple instances with different local
   addresses.)

   establish_to takes a handle and remote address.  If there is
   already a connection to that address it returns its file
   descriptor, otherwise it tries to establish one (and returns the
   new file descriptor).

   disestablish_to takes a handle and remote address, closing and
   removing a connection to that address if one exists.

   connection_failed takes a handle and remote address (one for which
   a connection has failed) and removes it from the stored list.

   shutdown closes and removes all connections and closes the
   listening socket.

   local_addr takes a handle and returns the local address.

*)

(* TODO: Deal more sensibly with TCP errors and the REUSEADDR semantics, here and in the clients *)
(* TODO: Think about efficiency *)
(* TODO: Have shutdown cleanly terminate the associated thread *)
```

```
module hash Tcp_connection_management :
  sig
    type fd = Tcp.fd
    type handle
    val daemon : Tcp.ip option * Tcp.port option ->
      ((Tcp.ip option * Tcp.port)->Tcp.addr->fd -> unit) -> handle
    val establish_to : handle -> Tcp.addr -> fd
    val disestablish_to : handle -> Tcp.addr -> unit
    val shutdown : handle -> unit
    val connection_failed : handle -> Tcp.addr -> unit
    val local_addr : handle -> Tcp.ip option * Tcp.port
  end =
  struct
    type fd = Tcp.fd
    type handle =
        (Tcp.ip option * Tcp.port)                           (* local address             *)
          * fd                                               (* listening socket          *)
          * ((Tcp.ip option*Tcp.port)->Tcp.addr->fd->unit)   (* incoming conn handler   *)
          * mutex name                                       (* current connections mutex *)
          * (Tcp.addr * fd) list ref                         (* current connections       *)


    let daemon (ipo,po) f =
      let conn_mutex = fresh in
      create_mutex conn_mutex;
      Pervasives.print_endline ("Created TCP mutex " ^ name_to_string conn_mutex);
      let conn = ref [] in
      let fd = Tcp.tcp_socket () in
      let _ = Tcp.bind fd ipo po in
      let (ipo,p) = match Tcp.getsockname fd with
        (Some ip, Some p) -> (Some ip, p)
      | (None, Some p) -> (None,p)
      | _ -> raise (Failure "no local port after bind()") in
      let _ = let backlog = 5 in Tcp.listen fd backlog in
      (while true do
        let (fd',(ip',p')) = Tcp.accept fd in
        let p'' = (unmarshal (Tcp_padded.recv fd') as Tcp.port) in
        f (ipo,p) (ip',p'') fd' ;   (* note that f terminates before adding this to conn *)
        Utils.locked_by_stmt conn_mutex
          (function () ->
            conn := ((ip',p''),fd') :: !conn)
      done |||
      (((ipo,p),fd,f,conn_mutex,conn)
      ))

    let establish_to h (ip',p') =
      let ((ipo,p),fd_listen,f,conn_mutex,conn) = h in
      Utils.locked_by_stmt2 %[fd] conn_mutex (function ()->
        try
          List.assoc %[Tcp.addr] %[] (ip',p') !conn
        with
          Not_found ->
            let fd = Tcp.tcp_socket () in
    Tcp.bind fd ipo None;
            Pervasives.print_endline ("Establish connecting to p' = " ^
              Pervasives.string_of_int(Tcp.int_of_port p') );
    Tcp.connect fd ip' Some p';
```

```
            let d = (marshal "StdLib" p : Tcp.port) in
            Pervasives.print_endline ("Establish p = " ^ Pervasives.string_of_int(Tcp.int_of_port p));
            (* Pervasives.print_endline ("Establish string = " ^ d ); *)
            Tcp_padded.send fd None d;
            f (ipo,p) (ip',p') fd;
            conn := ((ip',p'),fd) :: !conn;
    fd
      )

    let disestablish_to h (ip',p') =
      let ((ipo,p),fd_listen,f,conn_mutex,conn) = h in
      Utils.locked_by_stmt conn_mutex (function ()->
        try
          let fd = List.assoc %[] %[] (ip',p') !conn in
          conn := List.remove_assoc %[] %[] (ip',p') !conn;
          Tcp.close fd
        with
          Not_found -> ()
    )

    let shutdown h =
      let ((ipo,p),fd_listen,f,conn_mutex,conn) = h in
      Utils.locked_by_stmt conn_mutex (function ()->
        List.iter %[] (function ((ip,p),fd) -> Tcp.close fd) !conn;
        conn := [];
        Tcp.close fd_listen)

    let connection_failed h (ip',p') =
      let ((ipo,p),fd_listen,f,conn_mutex,conn) = h in
      Utils.locked_by_stmt  conn_mutex (function () ->
        conn := List.remove_assoc %[] %[] (ip',p') !conn )


    let local_addr h =
      let ((ipo,p),fd_listen,f,conn_mutex,conn) = h in
      (ipo,p)

  end

(* ****************************************************************** *)
(* **                                                              ** *)
(* ** Tcp_string_messaging                                         ** *)
(* **                                                              ** *)
(* ****************************************************************** *)

(* The Tcp_string_messaging module provides asynchronous messaging of
   strings to TCP addresses, using Tcp_connection_management.

   daemon takes a local address (an Tcp.ip option * Tcp.port option) and
   a function to handle incoming strings, of type

      (Tcp.ip option * Tcp.port) -> Tcp.addr -> string -> unit

   and creates a Tcp_connection_management.daemon, returning a handle.


   send takes a handle, a remote TCP address and a string, uses
   Tcp_connection_management.establish_to to ensure there is a
   connection, and sends the string (encapsulated in a wire format).


   shutdown takes a handle and shuts down (calling
   Tcp_connection_management.shutdown).


   local_addr takes a handle and returns the local TCP address.


   The wire format is implemented by Tcp_padded.
*)


(* TODO: handle send/recv errors and call connection_failed as required  *)
(* TODO: need more locking to stop different send/recvs interleaving     *)
(* TODO: one might want to pass the handle as another argument to the
         function argument to daemon *)

module hash Tcp_string_messaging :
  sig
    type handle
    val daemon : Tcp.ip option * Tcp.port option ->
      ((Tcp.ip option * Tcp.port) -> Tcp.addr -> string -> unit) -> handle
    val send : handle -> Tcp.addr -> string -> unit
    val shutdown : handle -> unit
    val local_addr : handle -> Tcp.ip option * Tcp.port
  end =
  struct
    type handle = Tcp_connection_management.handle

    let daemon (ipo,po) f =
      let g ipop addr' fd =
        create_thread fresh (function () ->
          while true do
            let data = Tcp_padded.recv fd in
            f ipop addr' data
          done
                          ) ()
      in
      Tcp_connection_management.daemon (ipo,po) g

    let send h (ip,p) data =
       let fd = Tcp_connection_management.establish_to h (ip,p) in
       Tcp_padded.send fd (Some(ip,p)) data

    let shutdown h = Tcp_connection_management.shutdown h

    let local_addr h = Tcp_connection_management.local_addr h
  end




(* **************************************************************** *)
(* **                                                            ** *)
(* ** Local_channel                                                  ** *)
(* **                                                                ** *)
(* ****************************************************************** *)

(* Module Local_channel provides simple typed asynchronous local
   channels.

   Function send : forall t. t name -> t -> unit  sends a message
   on the specified name, returning immediately.

   Function recv : forall t. t name -> (t -> unit) -> unit
   registers a receiver on the specified name, returning immediately.

   As soon as there is both a message and a receiver for a name the
   receiver is applied to the message.  The receiver is then removed.

   The interface uses (T name) as the type of channels carrying values
   of type T.  Exposing the fact that this is a name type allows
   clients to use any of the methods for constructing shared typed
   names that Acute provides.

   One might instead think of using ML-style references as channel
   'names'.  For a local implementation that would be fine, but one
   one marshalled values mentioning channels the whole channel data
   structure would be copied, which is not our desired semantics.

   Internally, the pending messages and receivers on the channels are
   stored in a list of existential packages, of type

      (exists t. t name * (t list ref * (t->unit) list ref)) list

   with the Acute namecase operation used in lookups.

   This is a hash! module.  There is module state: the handle h
   consists of a mutex name and a pointer to the channel data
   structure.  (Here h is exposed, abstractly, in the interface,
   purely to work around the current lack of width subsignaturing.)
   Nonetheless, rebinding to local instances of Local_channel.send and
   Local_channel.recv should just work, so we use the hash! mode to
   give an exact-hash version (and, as part of that workaround, to
   make the abstract type of h hash-generated).

   One might think of passing the handle explicitly as an argument to
   send and recv, doing without module state.  That again would lead
   to the wrong semantics for marshalling values that use this
   library.

  *)

(*   NB: fields marked by (*A*) will be removed from the interface  *)
(*   when width subsignaturing is added                             *)

module hash! Local_channel :
  sig
    type handle       (*A*)
    val h : handle    (*A*)
    val send : forall t. t name -> t -> unit
    val recv : forall t. t name -> (t -> unit) -> unit
  end

=
  struct

    type handle = mutex name * (exists t. t name * (t list ref * (t->unit) list ref)) list ref

    let h = (let n = fresh in create_mutex n; n, ref [] )

    (* A handle consists of a mutex and a reference to a list of
       channel structures.  Each channel structure is an existential
       package containing a name, a reference to a list of pending
       messages and a reference to a list of pending receptors.  We
       maintain the invariant that at most one of those two is
       nonempty.

       Channel structures are added to the list as necessary. At
       present they are never removed; we could remove them when they
       become empty.

       The pending messages and pending receptors are kept with the
       oldest at the heads of the lists.
       *)

    (* Note the use of namecase below *)

    let send = Function t -> fun (cn: t name) (v: t) ->
      let (m,csr) = h in
      Utils.locked_by_stmt m
        (function () ->
          let rec lookup cs' = match cs' with
            [] -> csr :=
              ( {t,(cn,(ref (v::[]),ref []))} as exists t. t name * (t list ref * (t->unit) list ref) )
              :: !csr
          | (c: exists t'. t' name * (t' list ref * (t'->unit) list ref))::cs0 ->
              namecase c with
                {t',(cn',xyz)} when cn'=cn ->
                  let ((msgs: t list ref),rcvrs)=xyz in
                  match !rcvrs with    (* in this branch the typechecker needs to know t=t' *)
                    [] -> msgs := (!msgs @ (v::[]))
                  | rcvr::rcvrs0 -> (
      rcvrs:=rcvrs0;   (* could remove this whole channel if it's become empty*)
      create_thread fresh rcvr v)
                otherwise ->
                  lookup cs0
          in lookup !csr
        )

    let recv = Function t -> fun (cn: t name) (f: t -> unit) ->
      let (m,csr) = h in
      Utils.locked_by_stmt m
        (function () ->
          let rec lookup cs' = match cs' with
            [] -> csr := ({t, (cn,(ref [],ref (f::[])))} as
              exists t. t name * (t list ref * (t->unit) list ref)) :: !csr
          | (c: exists t'. t' name * (t' list ref * (t'->unit) list ref))::cs0 ->
```

```
              namecase c with
                {t,(cn',x)} when cn'=cn ->
                  let ((msgs: t list ref),rcvrs)=x in
                    match !msgs with
                      [] -> rcvrs := !rcvrs @ (f::[])
                    | v::vs0 -> (
                        msgs:=vs0;
                        create_thread %[t] fresh f v)
                otherwise ->
                  lookup cs0
          in lookup !csr
        )

  end


mark "LChan"


(* TODO: Extend with replicated input and with blocking receive. *)




includesource "tcp.ac"
includesource "local_channel.ac"

(* ****************************************************************** *)
(* **                                                              ** *)
(* **  Distributed_channel                                         ** *)
(* **                                                              ** *)
(* ****************************************************************** *)

(* Distributed_channel provides simple typed asynchronous distributed
   channels, above Tcp_string_messaging and Local_channel.

   Function init : Tcp.ip option * Tcp.port option -> unit initialises
   a Tcp_string_messaging daemon with the specified port and IP
   address.

   Function send : forall t. string -> (Tcp.addr * t name) -> t -> unit
   sends a message (marshalled wrt the mark specified) to the
   specified channel at the specified TCP address, returning
   immediately.  It does a case split depending on whether the target
   is local or not, for efficiency.

   Function recv : forall t.  t name -> (t -> unit) -> unit registers
   a receiver on the specified name, returning immediately.

   Function local_addr : unit -> Tcp.ip option * Tcp.port option
   returns the registered local address.

   As soon as there is both a message and a receiver for a name the
   receiver is applied to the message.  The receiver is then removed.
   These are _non-mobile_ distributed channels: the receivers cannot
   be moved from one Tcp.addr to another.  See npi.ac for a mobile
   extension.
   Similarly to Local_channel, this is a hash! module.  The module state
   consists of an ho field, recording the Tcp_string_messaging handle
   in use.  To allow client code to determine the local TCP address
   this is set by the init function (it is stored as an option
   reference and can be set at most once).  Use of the hash! mode
   gives the module an exact-hash version.  Use of module state
   (rather than explicitly-passed handles) ensures the right semantics
   when marshalling client code.

   Internally, the wire format consists of marshalled values of type

     exists t'.t' name * t'

   marshalled with respect to whatever mark is supplied to the send
   function. This mark should usually be at or below the mark "DChan"
   just below the module, so that the Distributed_channel code itself is
   not marshalled.
  *)

(*   NB: fields marked by (*A*) will be removed from the interface *)
(*   when  width subsignaturing is added                           *)

module hash! Distributed_channel :
  sig
    type tf                         (*A*)
    type tho                        (*A*)
    val f : tf                      (*A*)
    val ho : tho                    (*A*)
    val init : Tcp.ip option * Tcp.port option -> unit
    val send : forall t. string -> (Tcp.addr * t name) -> t -> unit
    val recv : forall t. t name -> (t -> unit) -> unit
    val local_addr : unit -> Tcp.ip option * Tcp.port
  end
=
  struct
    type tf = (Tcp.ip option * Tcp.port) -> Tcp.addr -> string -> unit
    type tho = Tcp_string_messaging.handle option ref

    let f ipop_local addr_remote data =
      let {t,x} = unmarshal data as  exists  t'. t' name * t'  in
      let (c,v) = x in
      Pervasives.prerr_endline("Got v: " ^ (marshal "StdLib" (v) : t));
      Local_channel.send %[t] c v

    let ho = ref None

    let init (ipo,po) =
      match !ho with
        Some _ -> raise (Failure "Distributed_channel already initialised")
      | None -> ho := Some (Tcp_string_messaging.daemon (ipo,po) f)

    let send = Function t -> fun mk -> fun (addr,(c: t name)) (v: t) ->
      let h = Utils.the %[] !ho in
      let (ip, port) = addr in
      if (Some ip, port) = Tcp_string_messaging.local_addr h then
         Local_channel.send %[t] c v
      else
(Pervasives.prerr_endline("marshalling");
        let data = marshal mk ({t, (c,v)} as exists t'.t' name * t')
            : exists t'.t' name * t' in
( Pervasives.prerr_endline("sending " ^ data);
        Tcp_string_messaging.send h addr data))
    let recv = Function t -> fun (c: t name) (f: t -> unit) ->
      Local_channel.recv %[t] c f

    let local_addr () = Tcp_string_messaging.local_addr (Utils.the %[] !ho)
   end

mark "DChan"

(* TODO: Extend with replicated input and with blocking receive *)
(* Note that with this code the local-send optimisation will only be
effective if the local daemon IP was set explicitly, not
wildcarded. To deal properly with hosts with multiple interfaces one
should check against getifaddrs. *)


includesource "tcp.ac"

(* ****************************************************************** *)
(* **                                                              ** *)
(* **  Npi, consisting of Npi1 and Npi2                            ** *)
(* **                                                              ** *)
(* ****************************************************************** *)

(* The Npi module manages groups of threads in a single acute process,
   implementing the key primitives of the Nomadic Pict language.

   A thread can either be registered with the Npi module or not.
   If it is registered, it belongs to exactly one group thoughout its
   execution.
   Local communication within a group and inter-group communication
   via typed channels is supported.
   Furthermore, there is a "migrate_group" command, which when called by
   one member of the group, migrates the whole group to a new Tcp address.
   For this to work, the other end also needs to have an initialised
   Npi module running.
   The correct operation of this module depends on the client code not
   using any low-level primitives - thread operations, thunkify, etc.


   Most important functions:

   init : (Tcp.ip option * Tcp.port option) -> unit
      initialise group infrastucture to handle inter-group communication
      and group migrations.

   create_group :   forall t. (t -> unit) -> t -> unit
      create a new group containing one (new) thread.

   create_gthread : forall t. (t -> unit) -> t -> unit
      add a new thread to the current group.
   recv_local : forall t. t name -> t
      receive information from a named typed channel

   send_local : forall t. t name -> t -> unit
      send information to a named typed channel (of current group)

   send_remote : forall t.  string -> (Tcp.addr * group name * t name) -> t  -> unit
      send information to a named typed channel of another group at a
      known Tcp address.

   migrate_group : Tcp.addr -> unit
      migrate current group to a new Tcp address.


   As is Local_channel and Distributed_channel, (T name)s are used for
   channels carrying values of type T, allowing any of the Acute
   methods for establishing shared typed names to be used.

   Internally, migration uses thunkify.  Migration and send_remote
   both use marshal, with a wire format of marshalled values of type

     (group name * (exists t. t name * t)) + migration

   for the message and migration cases, where

     type migration = group name
                    * group
                    * mutex name * cvar name
                    * (thunkkey list -> unit)

   The recv_local and send_local use namecase (as in Local_channel).
   Marshalling of migrations is with respect to the mark "Npi_end" set
   below; marshalling for send_remote is with respect to the supplied
   mark, which should usually be below "Npi_end".  There is some
   delicate use of local concurrency with mutexes and cvars.

  *)

(* NB: fields marked by (*A*) will be removed from the interface *)
(*  when width subsignaturing is added                           *)


(* Note the use of hash! (instead of fresh), as we need to rebind to
this interface on migration with type "group" being compatible *)


module hash! Npi1 :
 sig

   type tf = (Tcp.ip option * Tcp.port) -> Tcp.addr -> string -> unit
   type tho = Tcp_string_messaging.handle option ref

   type channel = (exists t. t name * (t list ref * cvar name))

   type group = thread name list ref          (* threads in group *)
              * mutex name list ref           (* mutexes in group *)
              * cvar name list ref            (* cvars in group *)
```

```
             * channel list ref              (* local channels *)

   type migration = group name
                  * group
                  * (thunkkey list -> unit)

   val groups_mutex : mutex name
   val groups : (group name * group) list ref
   val threadmap : (thread name * group name) list ref
   val ho: tho

 end
 =
 struct

   type tf = (Tcp.ip option * Tcp.port) -> Tcp.addr -> string -> unit
   type tho = Tcp_string_messaging.handle option ref

   type channel = (exists t. t name * (t list ref * cvar name))

   type group = thread name list ref          (* threads in group *)
              * mutex name list ref           (* mutexes in group *)
              * cvar name list ref            (* cvars in group *)
              * channel list ref              (* local channels *)

   (* The group data structure is more generous than its usage:
        it allows also mutexes and condition variables to be associated
        with a group (and be migrated propely).
        At the moment there is no create_gmutex/create_gcvar, although
        their implementation would be trivial.
    *)

   type migration = group name
                  * group
                  * (thunkkey list -> unit)

   let groups_mutex = hash(mutex, "Npi global mutex") %[mutex] (* fresh *)(* global mutex *)


   (* Locking strategy:
      - There is a global mutex ("groups_mutex") at each running acute process.
      - Functions acting on the group data structures are all protected by this
        global lock.
      - When a thread wants to receive a message and there are none in the
        channel, the thread waits on the channel's condition variable.
      - When a new message is sent on empty channel, its condition variable is
        signalled so that a waiting receiver is unblocked.
    NB: This does not in principle guarrantee a FIFO delivery order, but will in
        fact have a FIFO ordering with the current version of Acute as threads
        in a condition variable are stored in a FIFO queue.

      The locking strategy is quite coarse; a more fine-grained scheme would be
      possible, where besides the global lock, a lock per group is also kept.
    *)

   let groups = ref []        (* group name -> group *)
   let threadmap = ref []     (* thread name -> group name *)

   (* threadmap exists to find in which group a thread belongs to
      These maps are simply implemented as linked lists, but a production
      implementation would use a hashtable instead.
      Simillarly the list of channels should really be a hashtable.
    *)

   let ho = ref None

 end

mark "Npi1"

module hash! Npi2 :
 sig

   val find_my_group : unit -> Npi1.group name * Npi1.group

   val gthread_wrapper : forall t.  (t->unit) -> t -> unit
   val create_group : forall t. (t -> unit) -> t -> unit
   val create_gthread : forall t.  (t->unit) -> t -> unit

   val recv_local : forall t. t name -> t

   val my_send_local : forall t. Npi1.group -> t name -> t -> unit
   val send_local : forall t. t name -> t -> unit

   val f : Npi1.tf

   val init : (Tcp.ip option * Tcp.port option) -> unit

   val send_remote : forall t.  string -> (Tcp.addr*Npi1.group name*t name) -> t -> unit

   val migrate_group : Tcp.addr -> unit

   val local_addr : unit -> Tcp.ip option * Tcp.port

 end
 =
 struct

   (* returns which group the calling thread belongs to *)
   let find_my_group () =  Utils.locked_by_stmt2 %[] Npi1.groups_mutex
    (function () ->
     Pervasives.print_endline "In find_my_group lock...";
     let gn =
       try  List.assoc %[] %[] (self ()) !Npi1.threadmap
       with Not_found ->
 raise (Failure "find_my_group:assoc")
     in
     let group_info =
       try List.assoc %[] %[] gn !Npi1.groups
       with Not_found -> raise (Failure "find_my_group:assoc[2]")
     in
     (gn, group_info)
    )
   (* Ensure that thread exits gracefully by unregistering itself from
    * the group data structure.
    *)
   let gthread_wrapper = Function t -> fun (f: t -> unit)  (v: t) ->
     f v
(*     let unregister_my_gthread () =
       let tn = self() in
       let rec remove_me xs = match xs with
         []      -> raise Not_found
       | (x::xs) -> if x = tn then xs
                              else x :: remove_me xs in
       let (gn, (ths, _, _, _)) = find_my_group () in
       Utils.locked_by_stmt Npi1.groups_mutex
         (function () ->
           Npi1.threadmap := List.remove_assoc %[] %[] tn !Npi1.threadmap;
           ths := remove_me !ths
         )
     in
       (try f v
        with e -> (try unregister_my_gthread () with _ -> ()); raise e);
       unregister_my_gthread ()
*)
   (* create a new group *)
   let create_group = Function t -> fun (f: t -> unit) (v : t) ->
     let gn = fresh %[Npi1.group] in
     let tn = fresh %[thread] in
     Utils.locked_by_stmt Npi1.groups_mutex
       (function () ->
          let group_info = (ref (tn::[]), ref [], ref [], ref []) in
          Npi1.groups := (gn, group_info) :: !Npi1.groups;
          Npi1.threadmap := (tn, gn) :: !Npi1.threadmap;
          create_thread tn (gthread_wrapper %[t] f) v )

   (* create a new thread in the current group *)
   let create_gthread = Function t -> fun (f: t -> unit) (v: t) ->
      let (gn, (ths, _, _, _)) = find_my_group () in
      let tn = fresh %[thread] in
      Utils.locked_by_stmt Npi1.groups_mutex
        (function () ->
         Npi1.threadmap := (tn, gn) :: !Npi1.threadmap;
         ths := tn :: !ths;
         create_thread %[t] tn (gthread_wrapper %[t] f) v
        )

   (* receive a value from a local channel, blocking if there is none *)
   let recv_local = Function t -> fun (cn: t name) ->
     let (gn, group_info) = find_my_group () in
      let (_,_,_,csr) = group_info in
      Utils.locked_by_stmt2 %[t] Npi1.groups_mutex
        (function () ->
          let rec lookup cs' = match cs' with
            [] -> let my_cvar = fresh %[cvar] in
                  create_cvar my_cvar;
                  csr := ({t, (cn, (ref [], my_cvar))} as Npi1.channel) :: !csr;
                  wait my_cvar Npi1.groups_mutex;
                  lookup !csr
          | (c: Npi1.channel)::cs0 ->
              namecase c with
                {t,(cn',x)} when cn'=cn ->
                  let ((msgs: t list ref), my_cvar) = x in
                    let rec ww () =
                        match !msgs with
                          []    -> wait my_cvar Npi1.groups_mutex; ww ()
                        | v::vs -> msgs := vs; v
                    in
                        ww ()
                otherwise ->
                  lookup cs0
          in lookup !csr
        )

   let my_send_local = Function t -> fun group_info (cn: t name) (v: t) ->
      let (_,_,_,csr) = group_info in
      Utils.locked_by_stmt Npi1.groups_mutex
        (function () ->
          let rec lookup cs' = match cs' with
            [] -> let my_cvar = fresh %[cvar] in
                  create_cvar my_cvar;
                  csr := ({t,(cn,(ref(v::[]),my_cvar))} as Npi1.channel) :: !csr
          | (c: Npi1.channel)::cs0 ->
              namecase c with
                {t,(cn',x)} when cn'=cn ->
                  let ((msgs: t list ref), my_cvar) = x in
                  (match !msgs with
                     [] -> msgs := v :: !msgs; signal my_cvar
                   | _  -> msgs := v :: !msgs)
                otherwise ->
                  lookup cs0
          in lookup !csr
        )

   let send_local = Function t -> fun (cn: t name) (v: t) ->
     let (gn, group_info) =  find_my_group () in
     my_send_local %[t] group_info cn v


   (* We have a single site daemon listen for messages and migrating things.
       - for messages, it uses the group name to look up in the group data structure
         to find the appropriate (local) channel handle, then use that to propagate
         the message.
       - for migrating things, it'll unthunkify and extend the group data structure.
    *)
   let f ipop_local addr_remote data =
     Utils.locked_by_stmt Npi1.groups_mutex
     (function () ->
      Pervasives.print_endline "npi daemon received something";
      try
        match (unmarshal data) with
          inj 1 %[(Npi1.group name * (exists t. t name * t)) + Npi1.migration] (gn, channel)
           -> (* a normal value *)
            Pervasives.print_endline "npi daemon received a value";
            let group_info = try List.assoc %[] %[] gn !Npi1.groups
                             with Not_found -> raise (Failure
              "Received a value for a group not present at this TCP address")
    in
            let {t, x} = channel in
            let (cn, v) = x in
            send_local %[t] cn v
        | inj 2 %[(Npi1.group name*(exists t. t name*t))+Npi1.migration] (gn,groupinfo,unthunk)
           -> (* a migration *)
            Pervasives.print_endline "npi daemon received a migration";
            let (ths, mtxs, cvs, csr) = groupinfo in

             if List.mem_assoc %[] %[] gn !Npi1.groups then
                 (* NB: this should never occur as group names are only created with
                        fresh %[group] and the only operation involving group names
                        is migration which is linear.
                        This check prevents a type of maliciously forged migrations.
                  *)
                 raise (Failure "A group with this same name is already present at this site")
             else (
               Npi1.groups := (gn, groupinfo) :: !Npi1.groups;
               List.iter %[] (fun tn -> Npi1.threadmap := (tn, gn) :: !Npi1.threadmap) !ths;
               let tks = List.map %[] %[] (fun n -> Thread (n, Blocking)) !ths
                       @ List.map %[] %[] (fun n -> Mutex n) !mtxs
                       @ List.map %[] %[] (fun n -> CVar n)  !cvs
                       @ List.map %[] %[] (fun (p: Npi1.channel) ->
                           let {t,x} = p in let (_,(_,n)) = x in CVar n) !csr
               in
               unthunk tks;
       Pervasives.print_endline("unthunked")
             )
      with e -> Pervasives.print_endline "An exception was raised in the npi daemon";
                raise e
     )

   let init (ipo,po)  =
      create_mutex Npi1.groups_mutex;
      Pervasives.print_endline ("Created NPI mutex " ^ name_to_string Npi1.groups_mutex);
      match !Npi1.ho with
        Some _ -> raise (Failure "Npi already initialised")
      | None -> Npi1.ho := Some (Tcp_string_messaging.daemon (ipo,po) f)

   let send_remote = Function t -> fun mk (addr,gn,cn) (v: t) ->
     let h = Utils.the %[] !Npi1.ho in
     let (ip, port) = addr in
     if (Some ip, port) = Tcp_string_messaging.local_addr h then
     (* note this local-send optimisation will only take effect if the
        IP was set explicitly *)
         let group_info = Utils.locked_by_stmt2 %[] Npi1.groups_mutex (function () ->
           try List.assoc %[] %[] gn !Npi1.groups
           with Not_found -> raise (Failure "send_remote:List.assoc")
         ) in
         my_send_local %[t] group_info cn v
      else
       let channel = {t, (cn, v)} as exists t'.t' name * t' in
        let data = inj 1 %[(Npi1.group name*(exists t.t name*t))+Npi1.migration] (gn, channel) in
        let mar_data = marshal mk data in
        Tcp_string_messaging.send h addr mar_data
```

```
(* Migrate the current group to a new Tcp address.
   All threads except for the calling thread are thunkified with Blocking mode.
   The called thread is blocked with a mutex/cvar. As it is marshalled with
   Interrupting mode, it is woken up at the other end with a Thunkify_EINTR
   exception.
 *)
let migrate_group = fun addr ->
   Pervasives.print_endline("migrate_group: started");
   let (gn, group_info) = find_my_group () in
   Pervasives.print_endline("migrate_group: found my group");
   let (ths, mtxs, cvs, csr) = group_info in
   let my_cv = fresh in
   create_cvar my_cv;

   lock Npi1.groups_mutex;
   (* First remove the group and its threads from the global data structures *)
   Npi1.groups := List.remove_assoc %[] %[] gn !Npi1.groups;   (* remove gn -> group_info mapping *)
   List.iter %[]
     (fun tn -> Npi1.threadmap := List.remove_assoc %[] %[] tn !Npi1.threadmap)
     !ths;                                          (* remove tn -> gn mapping *)
   Pervasives.print_endline("migrate_group: removed gn,tn data");
   let initiating_thread_name = self() in
   (* make new thread to perform thunkify, otherwise will thunkify self *)
   create_thread fresh
     (function () ->
       Pervasives.print_endline("migrate_group: thunkify thread started");
        Utils.locked_by_stmt Npi1.groups_mutex
        (function () ->
         Pervasives.print_endline("migrate_group: thunkify thread got lock");
         let get_tmode tn =
            if compare_name tn initiating_thread_name = 0 then
                 Interrupting
            else Blocking in
         let tks = List.map %[] %[] (fun n -> Thread (n, get_tmode n)) !ths
                 @ List.map %[] %[] (fun n -> Mutex n) !mtxs
                 @ List.map %[] %[] (fun n -> CVar n)  !cvs
                 @ List.map %[] %[] (fun (p: Npi1.channel) ->
                     let {t,x} = p in let (_,(_,n)) = x in CVar n) !csr
         in
         Pervasives.print_endline("migrate_group: thunkify thread going to thunkify");
         let thunked = thunkify tks in
         Pervasives.print_endline("migrate_group: thunkify thread done thunkify");
         let data = inj 2 %[(Npi1.group name * (exists t. t name *
            t)) + Npi1.migration] (gn, group_info, thunked) in
         let mar_data = marshal "Npi_end" data in
         Pervasives.print_endline("migrate_group: going to send marshalled: ... "
           (* ^ mar_data *) );
         let h = Utils.the %[] !Npi1.ho in
         Tcp_string_messaging.send h addr mar_data
        )
     ) ();
    (* must block thread initiating migration, until thunkify has completed *)
    try
      wait my_cv Npi1.groups_mutex    (* Block here - thunkify will cause Thunkify_EINTR *)
    with Thunkify_EINTR -> () (* Migration completed -- we can now continue execution *)

 let local_addr () = Tcp_string_messaging.local_addr (Utils.the %[] !Npi1.ho)
```

```
  end

mark "Npi2"




module hash! Npi :
 sig

   type group

   val create_group : forall t. (t -> unit) -> t -> unit
   val create_gthread : forall t.  (t->unit) -> t -> unit

   val recv_local : forall t. t name -> t
   val send_local : forall t. t name -> t -> unit

   val init : (Tcp.ip option * Tcp.port option) -> unit

   val send_remote : forall t.  string -> (Tcp.addr * group name * t name) -> t  -> unit
   val migrate_group : Tcp.addr -> unit

   val local_addr : unit -> Tcp.ip option * Tcp.port

 end
 =
 struct

   type group = Npi1.group

   let create_group   = Npi2.create_group
   let create_gthread = Npi2.create_gthread
   let recv_local     = Npi2.recv_local
   let send_local     = Npi2.send_local
   let init           = Npi2.init
   let send_remote    = Npi2.send_remote
   let migrate_group  = Npi2.migrate_group
   let local_addr     = Npi2.local_addr

  end

mark "Npi_end"



(* ****************************************************************** *)
(* **                                                              ** *)
(* ** npi-recv client                                              ** *)
(* **                                                              ** *)
(* ****************************************************************** *)

(* example npi client, initialising an npi daemon *)


includesource "npi.ac"
```

```
let addr  (ip, port) = (Tcp.ip_of_string ip, Tcp.port_of_int port) in

let _ = Npi.init (Some(Tcp.ip_of_string "127.0.0.1"),
                  Some(Tcp.port_of_int 6401)) in

Pervasives.prerr_endline("npi-recv done initialising")

(* ****************************************************************** *)
(* **                                                              ** *)
(* ** npi-mig client                                               ** *)
(* **                                                              ** *)
(* ****************************************************************** *)

(* example npi client, migrating an npi group there and back *)

includesource "npi.ac"

let addr  (ip, port) = (Tcp.ip_of_string ip, Tcp.port_of_int port) in

let _ = Npi.init (Some(Tcp.ip_of_string "127.0.0.1"),
                  Some(Tcp.port_of_int 6400)) in

Pervasives.prerr_endline("npi-mig done initialising");

let _ = Npi.create_group %[]
    (fun () ->
      Pervasives.prerr_endline("group created");
      Npi.migrate_group (addr ("127.0.0.1", 6401));
      Pervasives.prerr_endline("group migrated");
      Npi.migrate_group (addr ("127.0.0.1", 6400));
      Pervasives.prerr_endline("group migrated back")
    ) ()  in

()
```

Part V
Implementation

17 Overview

The implementation is written in FreshOCaml [SPG03], currently around 25 000 lines of code. It has been developed together with the language definition. By and large the definition has led, with extensions and changes to the definition being followed by implementation work to match. This exposed many ambiguities and errors in the semantics. In a few cases the implementation led, with changes propagated back into the definition afterwards. An automated testing framework helps ensure the two are in sync, with tests of compilation and execution that can be re-run automatically.

The main priority for the implementation was to be rather close to the semantics, to make it easy to change as the definition changed, and easy to have reasonable confidence that the two agree, while being efficient enough to run moderate examples. The runtime is essentially an interpreter over the abstract syntax, finding redexes and performing reduction steps as in the semantics. For efficiency it uses closures (as described in §16.12) and represents terms as pairs of an explicit evaluation context and the enclosed term (roughly as in [Rém02, §1.3.1, Ex. 1]) to avoid having to re-traverse the whole term when finding redexes. Marshalled values $\mathbf{marshalled}(E_{\mathrm{n}},\ E_s,\ s,\ \mathit{definitions},\ e,\ T)$ are represented simply by a pretty-print of their abstract syntax. Numeric hashes use a hash function applied to a pretty-print of their body; it is thus important for this pretty-print to be canonical, choosing bound identifiers appropriately. Acute threads are reduced in turn, round-robin. A pool of OS threads is maintained for making blocking system calls. A `genlib` tool makes it easy to import (restricted versions of) OCaml libraries, taking OCaml `.mli` interface files and generating embeddings and projections between the OCaml and internal Acute representations. It does not support higher-order functions, which would be challenging in the presence of concurrency.

To give a *very* crude idea of performance, the initialisation phase of the `blockhead.ac` game performs about 220000 steps (roughly corresponding to reduction steps) in 4.5 seconds, without runtime typechecking and with the vacuous bracket optimisation. The naive Fibonacci function of 25

```
let rec fib:int->int = function (x:int) ->
  if x <=2 then
    1
  else
    (fib (x-1)) + (fib(x-2))
in
let x = fib 25
```

involves about 1.6 million steps and takes 18 seconds, again without runtime typechecking and with vacuous bracket optimisation. Running the same code in the OCaml toplevel takes 0.0056 seconds, so the Acute implementation is around 3000 times slower. Turning on runtime typechecking in Acute (and using `definitions_lib_small.ac`) for Fibonacci of 15 takes the execution time from 0.16 seconds to 495 seconds (11000 steps), a slowdown of another factor of 3000. These figures are all for a 3.20GHz Pentium 4. In practice this level of performance has been reasonable for the examples we have considered to date. The blockhead and minesweeper games are playable, and three sample communication infrastructures, based on Nomadic Pict, Distributed Join Calculus, and Ambients, all execute tolerably well. Runtime typechecking, while it would be good to have feasible for these larger examples, in fact is mostly useful for more focussed test cases, for which one wishes to observe the individual reduction steps in any case.

18 Command line options

```
acute <options> <filename>
  where
the <level>s are:
  0 none
  1 expression
  2 expression, store
  3 expression, store, userdefns
  4 expression, store, userdefns, libdefns

and options are:
  -definitionslib <filename>     semantic: Read the standard definitions from <filename>
(default: definitions_lib_small.ac
 but use definitions_lib.ac for full set)
  -nodefinitionslib              semantic: No standard definitions
  -o <filename>                  phase: Output to <filename> (default: <stdout>)
  -df <filename>                 phase: Print final state dump to to <filename> (default: <stdout>)
  -err <filename>                phase: Print debug output to <filename> (default: <stderr>)
  -writefinal <filename>         phase: Pretty print result to <filename> (default: <stdout>)
  -checkfinal <filename>         phase: Check result against contents of <filename> (default: None)
  -emitobjectfile <filename>     phase: Emit compiled (object) code after compilation
  -emitsourcefile <filename>     phase: Emit source code after compilation
  -debugs <class>[,<class>..]    output: Which classes of debug output to display
(default: default,flattenclos,desugar,tcopt,mkhash,lexer,evalstep,marshal,hashify,tcquant,linkok,namecase,nameval
  -dumpstepinterval <n>          output: Print the configuration (at dumptrace level) every <n> steps
  -dumpfrom <n>                  output: Only print the configuration (at dumptrace level) after <n> steps
  -printstepinterval <n>         output: Print the reduction step count every <n> steps
  -noprintstepinterval           output: Do not print the reduction step count
  -production                    rttc: Set options used for a production implementation
 <norttc><nomttc><notypecheckcompiled><lithash><nolinkok_sig_typecheck><hack_optimise>
  -noproduction                  rttc: Set options used for a non-production implementation
 <rttc><mttc><typecheckcompiled><nolithash><linkok_sig_typecheck><nohack_optimise>
  -tcdepth <depth>               (4) output: Context depth for typechecking errors
  -dumpparse <level> (0-4)       (0) output: Dump result of parse
  -dumppreinf <level> (0-4)      (0) output: Dump input to inference
  -dumppostinf <level> (0-4)     (0) output: Dump output of inference
  -dumpdesugared <level> (0-4)   (0) output: Dump output of desugaring
  -dumpcompiled <level> (0-4)    (3) output: Dump output of compilation
  -dumptrace <level> (0-4)       (1) output: Dump traced execution steps
  -dumpfinal <level> (0-4)       (1) output: Dump final state (if no type failure)
  -dumptypefail <level> (0-4)    (3) output: Dump on type failure (or unmarshalfail)
  -[no]showpasses                (*) output: Show names of compilation passes
  -[no]showtimes                 (*) output: Show time taken per pass
  -[no]showprogress              ( ) output: Show progress during type inference
  -[no]showlocs                  ( ) output: Show locations in dump output
  -[no]showtrailer               (*) output: Show trailer information (e.g., hash values) when printing
  -[no]suffixall                 ( ) output: Always suffix names, even when unshadowed
  -[no]shownames                 ( ) output: Show internal representation of bound names
  -[no]globalhashmap             (*) output: Use a common map for abbreviating hashes and abstract names
  -[no]show_options              ( ) output: Show the command line used, including default options
  -[no]showtcenv                 ( ) output: Show environment in typecheck errors
  -[no]emitobject                ( ) output: Emit compiled (object) code after compilation
  -[no]printenv                  ( ) output: Print runtime environments
  -[no]printenvbodies            ( ) output: Print runtime environment bodies (RHSs)
  -[no]printclos                 ( ) output: Print closures as closures (rather than expanding)
  -[no]printerrordeath           (*) output: Print error message when a thread exits with an exception
  -[no]printcleandeath          ( ) output: Print message when a thread exits cleanly
  -[no]debug                    ( ) output: Generate debug output (on stderr)
  -[no]showfocussing            ( ) output: Show focussing process in dumptrace
  -[no]dumptex                  ( ) output: Dump in tex format
  -[no]dumphuman                ( ) output: Dump for humans (no type annotations)
  -[no]dumpall                  ( ) output: Don't ever abbreviate traces to ...
  -[no]parsetest                ( ) phase: Parser - pretty printer identity test
  -[no]desugar                  (*) phase: Desugar
  -[no]compile                  (*) phase: Compile
  -[no]typecheckcompiled        (*) phase: Typecheck the compiled program
  -[no]run                      (*) phase: Run program
  -[no]lithash                  ( ) rttc: Emit literal 0#123ABC hashes in certain places
  -[no]rttc                     (*) rttc: Do runtime typechecking
  -[no]mttc                     (*) rttc: Do unmarshaltime typechecking
  -[no]terminate_on_tc          (*) rttc: Terminate if typecheckcompiled or rttc is on and fails
  -[no]default                  (*) semantic: Default underspecified types to unit
  -[no]disable_import_typecheck  ( ) semantic: Disable typechecking of import links
  -[no]disable_eqsok_typecheck   ( ) semantic: Disable typechecking of |- eqs ok
  -[no]internal_weqs            (*) semantic: Allow use of with! equations inside modules
 (not just at boundary)
  -[no]linkok_sig_typecheck     (*) semantic: Do full subsignature typecheck in linkok
 (not just syntactic check)
  -[no]hack_optimise            (*) semantic: Perform vacuous-bracket optimisation
  -[no]really_hack_optimise     ( ) semantic: Erase all brackets
  -[no]abstract_existentials    (*) semantic: Dynamically-abstract existentials
  -[no]nonunitthread            ( ) semantic: Threads do not have to evaluate to unit
  -[no]marshaltex               ( ) semantic: Marshal in tex format (cannot be unmarshalled)
  -help   Display this list of options
  --help  Display this list of options
```

19 Concrete user source grammar

This is the concrete source grammar, automatically extracted from the implementation `ocamlyacc` source.

core_type	::=	core_type_pri
compilation_unit_definition	::=	[source_definition | `includesource` STRING | `includecompiled` STRING]
compilation_unit_definitions	::=	{ compilation_unit_definition semisemis }
nameenv	::=	{ (} | nameenv_non_empty })
nameenv_non_empty	::=	[{ nameenv_entry , } nameenv_entry]
nameenv_entry	::=	[ABSTRNAME : (`nmodule` modname_extern hmodule_body | `nimport` modname_extern himport_body | `Type` | core_type_pri)]
definitions	::=	{ definition }
optional_mode	::=	[`hash` | `hash!` | `cfresh!` | `cfresh` | `fresh`]
definition	::=	[`cmodule` modname_binder cmodule_body | `cimport` modname_binder cimport_body | `module` `fresh` modname_binder module_body | `import` `fresh` modname_binder import_body | `mark` STRING]
source_definition	::=	[`module` optional_mode modname_binder module_body | `amodule` modname_binder amodule_body | `import` optional_mode modname_binder import_body | `mark` STRING]
module_body	::=	: module_type version_opt = module_expr withspec_opt
valuability	::=	`valuable`
	|	`cvaluable`
	|	`nonvaluable`
valuabilities	::=	(valuability , valuability)
cmodule_body	::=	hash : eqs module_type valuabilities module_type version_val = module_expr
hmodule_body	::=	: eqs module_type version_nonopt = module_expr
amodule_body	::=	: module_type = modname_use
import_body	::=	: module_type version_constraint_opt likespec resolvespec_opt moo_module_opt
cimport_body	::=	hash : module_type valuabilities module_type version_constraint_val likestr resolvespec_nonopt moo_module
himport_body	::=	: module_type version_constraint_nonopt likestr
hash	::=	`hash` (`hmodule` modname_extern hmodule_body)
	|	`hash` (`himport` modname_extern himport_body)
	|	LITHASH
	|	ABSTRNAME
hash_or_modname_dot_ident	::=	[(hash | modname_use) . ident_extern]
name_value	::=	[`name_value` ((`hash` (hash . ident_extern) app_ty) | `hash` (core_type_pri , STRING)) | `hash` (core_type_pri , STRING , name_value)) | ABSTRNAME app_ty))]
eqs	::=	{ (} | eqs_body_non_empty })
eqs_body_non_empty	::=	[eqs_body_item [, eqs_body_non_empty]]
eqs_body_item	::=	[(hash | modname_use) . typname_extern = core_type_pri]
version_opt	::=	[`version` version]
version_val	::=	`version` version
version_nonopt	::=	`version` version
version_constraint_val	::=	`version` version_constraint
version_constraint_nonopt	::=	`version` version_constraint
version_constraint_opt	::=	[`version` version_constraint]

```
withspec_opt                           ::=  [ with! weqs ]
weqs_single                            ::=  modname_use . typname_extern = core_type_pri
weqs                                   ::=  weqs_rev
weqs_rev                               ::=  [ { weqs_single , } weqs_single ]
likespec                               ::=  [ like modname_use | likestr ]
likestr                                ::=  [ like struct structure end ]
resolvespec_nonopt                     ::=  [ by resolvespec_non_empty ]
resolvespec_opt                        ::=  [ resolvespec_nonopt ]
moo_module_opt                         ::=  [ [ moo_module ] ]
moo_module                             ::=  [ = ( unlinked | modname_use ) ]
module_expr                            ::=  struct structure end
module_type                            ::=  sig signature end
structure_items                        ::=  [ structure_item ( ;; structure_items | structure_items ) ]
structure                              ::=  [ structure_item ( ;; structure_items | structure_items ) ]
structure_item                         ::=  let ident_binder = typed_expr
                                         |  let ident_binder non_empty_pattern_list = typed_expr
                                         |  type typname_binder = core_type_pri
signature_items                        ::=  [ signature_item ( ;; signature_items | signature_items ) ]
signature                              ::=  [ signature_item ( ;; signature_items | signature_items ) ]
signature_item                         ::=  val ident_binder : core_type_pri
                                         |  type typname_binder
                                         |  type typname_binder = core_type_pri
                                         |  type typname_binder : kind
marshalled_body                        ::=  marshalled_nameenv_opt , { definitions } , { loctyp_list } , {
                                            store } , simple_expr , core_type_pri
marshalled_nameenv_opt                 ::=  _
                                         |  nameenv
marshalled_value_pri                   ::=  marshalled ( marshalled_body )
store                                  ::=  [ store_non_empty ]
store_non_empty                        ::=  [ { store_item , } store_item ]
store_item                             ::=  ( location := expr )
hash_in_version                        ::=  hash ( hmodule modname_extern hmodule_body )
                                         |  hash ( himport modname_extern himport_body )
                                         |  LITHASH
                                         |  ABSTRNAME
version_literal                        ::=  INT
                                         |  hash_in_version
version                                ::=  atomic_version [ version_dotted_suffix ]
version_dotted_suffix                  ::=  { . atomic_version } . atomic_version
atomic_version                         ::=  myname
                                         |  version_literal
atomic_hash_version_constraint         ::=  [ modname_use | hash_in_version ]
atomic_version_constraint              ::=  [ atomic_hash_version_constraint | INT ]
atomic_version_constraints_non_empty   ::=  [ { atomic_version_constraint . } atomic_version_constraint ]
tail_version_constraint                ::=  atomic_version_constraint
                                         |  INT - INT
                                         |  - INT
                                         |  INT -
                                         |  *
version_constraint                     ::=  [ name = atomic_hash_version_constraint | tail_version_constraint
                                            | atomic_version_constraints_non_empty . tail_version_constraint
                                            ]
resolvespec_non_empty                  ::=  [ resolvespec_item [ , resolvespec_non_empty ] ]
```

resolvespec_item	::=	Static_Link
	|	Here_Already
	|	STRING
seq_expr	::=	[expr [(; | |||) seq_expr]]
expr	::=	simple_expr
	|	simple_expr simple_expr_or_app_ty_list
	|	let pattern = typed_expr in seq_expr
	|	let ident_internal_binder non_empty_pattern_list = typed_expr in seq_expr
	|	let rec ident_internal_binder non_empty_pattern_list = typed_expr in seq_expr
	|	let rec ident_internal_binder optional_colon_core_type_pri = function mtch_when_sugary in seq_expr
	|	match seq_expr with mtch
	|	function mtch_when_sugary
	|	fun non_empty_pattern_list -> seq_expr
	|	try seq_expr with mtch
	|	ref opt_ty simple_expr
	|	ref opt_ty
	|	raise simple_expr
	|	if seq_expr then expr else expr
	|	while seq_expr do seq_expr done
	|	expr :: expr
	|	expr && expr
	|	expr || expr
	|	expr := opt_ty expr
	|	expr = opt_ty expr
	|	expr @ opt_ty expr
	|	expr + expr
	|	expr - expr
	|	expr * expr
	|	expr > expr
	|	expr < expr
	|	expr INFIXOP0 expr
	|	expr INFIXOP1 expr
	|	expr INFIXOP2 expr
	|	expr INFIXOP3 expr
	|	expr INFIXOP4 expr
	|	expr freshfor expr
	|	- expr
	|	Function typname_internal_binder -> seq_expr
	|	let { typname_internal_binder , ident_internal_binder } = typed_expr in seq_expr
	|	namecase expr with { typname_internal_binder , (ident_internal_binder , ident_internal_binder) } when ident_use = expr -> expr otherwise -> expr
typed_expr	::=	seq_expr
	|	seq_expr : loc_core_type
	|	seq_expr as loc_core_type
	|	typed_expr1
	|	seq_expr ; typed_expr1
	|	seq_expr ||| typed_expr1
typed_expr1	::=	{ core_type_pri , expr } as core_type_pri

```
simple_expr                      ::=  { constr0 | ident_use | econst_use | modname_use . ident_extern |
                                      hash . ident_extern | location | ( typed_expr ) | (
                                      expr_comma_list ) | ! opt_ty simple_expr | constr1 simple_expr |
                                      standalone_infixop | fresh opt_ty | cfresh opt_ty | hash (
                                      hash_or_modname_dot_ident ) app_ty | hash ( core_type_pri ,
                                      expr ) app_ty | hash ( core_type_pri , expr , expr ) app_ty |
                                      name_value | swap expr and expr in simple_expr | support
                                      opt_ty simple_expr | modname_use @ ident_extern | name_of_tie
                                      simple_expr | val_of_tie simple_expr | PREFIXOP |
                                      PREFIXOP_TYP opt_ty | marshal simple_expr simple_expr |
                                      unmarshal }
simple_expr_or_app_ty_list       ::=  simple_expr
                                   |  app_ty
                                   |  simple_expr simple_expr_or_app_ty_list
                                   |  app_ty simple_expr_or_app_ty_list
opt_ty                           ::=  [ [ %[ core_type_pri ] ] ]
app_ty                           ::=  [ %[ ( core_type_pri ] | ] ) ]
optional_colon_core_type_pri     ::=  [ [ : core_type_pri ] ]
location                         ::=  {< INT >}
loctyp_list                      ::=  [ loctyp_list_non_empty ]
loctyp_list_non_empty            ::=  { loctyp_pair , } loctyp_pair
loctyp_pair                      ::=  ( location : core_type_pri )
mtch                             ::=  [ [ | ] match_cases ]
mtch_when_sugary                 ::=  [ mtch | ( ident_internal_binder : core_type_pri ) match_action ]
match_cases                      ::=  pattern_match_action { | pattern_match_action }
pattern_match_action             ::=  pattern match_action
match_action                     ::=  -> seq_expr
expr_comma_list                  ::=  ( expr_comma_list | expr ) , expr
standalone_infixop               ::=  ( ( standalone_infixopstr ) | && ) | || ) | ! opt_ty ) | = opt_ty ) |
                                      := opt_ty ) | @ opt_ty ) )
standalone_infixopstr            ::=  [ + | - | * | < | > | INFIXOP0 | INFIXOP1 | INFIXOP2 |
                                      INFIXOP3 | INFIXOP4 ]
pattern                          ::=  pattern_pri
pattern_pri                      ::=  simple_pattern
                                   |  constr1 simple_pattern
                                   |  pattern_pri : : pattern_pri
simple_pattern                   ::=  ident_internal_binder
                                   |  _
                                   |  constr0
                                   |  - INT
                                   |  ( pattern_pri )
                                   |  ( pattern_pri : core_type_pri )
                                   |  ( pattern_comma_list )
pattern_comma_list               ::=  ( pattern_comma_list | pattern_pri ) , pattern_pri
non_empty_pattern_list           ::=  non_empty_rev_pattern_list
non_empty_rev_pattern_list       ::=  { pattern_pri } pattern_pri
kind                             ::=  Type
                                   |  Eq ( core_type_pri )
loc_core_type                    ::=  core_type_pri
core_type_pri                    ::=  fun_core_type
                                   |  forall typname_internal_binder . core_type_pri
                                   |  exists typname_internal_binder . core_type_pri
fun_core_type                    ::=  tup_core_type { -> tup_core_type }
```

```
simple_core_type           ::=  ( core_type_pri )
                              |  typname_constr_use0
                              |  modname_use . typname_extern
                              |  hash . typname_extern
                              |  simple_core_type ref
                              |  simple_core_type name
                              |  simple_core_type typname_constr_use1
tup_core_type              ::=  simple_core_type [ * core_type_list_tuple | + core_type_list_sum ]
core_type_list_tuple       ::=  simple_core_type { * simple_core_type }
core_type_list_sum         ::=  simple_core_type { + simple_core_type }
constr0                    ::=  [ ] opt_ty
                              |  None opt_ty
                              |  baseconstr0
baseconstr0                ::=  ( )
                              |  INT
                              |  false
                              |  true
                              |  CHAR
                              |  STRING
                              |  BASECON0
constr1                    ::=  inj INT app_ty
                              |  Some
                              |  tiecon
                              |  NODE
                              |  BASECON1
ident_use                  ::=  LIDENT
econst_use                 ::=  ECONST
ident_binder               ::=  LIDENT
ident_internal_binder      ::=  LIDENT
ident_extern               ::=  LIDENT
typname_constr_use0        ::=  LIDENT
typname_constr_use1        ::=  LIDENT
typname_binder             ::=  LIDENT
typname_extern             ::=  LIDENT
typname_internal_binder    ::=  LIDENT
modname_use                ::=  [ UIDENT ]
modname_binder             ::=  UIDENT
modname_extern             ::=  UIDENT
semisemis                  ::=  [ [ semisemis_plus ] ]
semisemis_plus             ::=  [ { ;; } ;; ]
```

20 Concrete compiled-form grammar

This is the concrete compiled-form grammar, automatically extracted from the implementation `ocamlyacc` source.

core_type	::=	core_type_pri
compilation_unit_definition	::=	[source_definition | `includesource` STRING | `includecompiled` STRING]
compilation_unit_definitions	::=	{ compilation_unit_definition semisemis }
nameenv	::=	{ (} | nameenv_non_empty })
nameenv_non_empty	::=	[{ nameenv_entry , } nameenv_entry]
nameenv_entry	::=	[ABSTRNAME : (`nmodule` modname_extern hmodule_body | `nimport` modname_extern himport_body | `Type` | core_type_pri)]
definitions	::=	{ definition }
optional_mode	::=	[`hash` | `hash!` | `cfresh!` | `cfresh` | `fresh`]
definition	::=	[`cmodule` modname_binder cmodule_body | `cimport` modname_binder cimport_body | `module` `fresh` modname_binder module_body | `import` `fresh` modname_binder import_body | `mark` STRING]
source_definition	::=	[`module` optional_mode modname_binder module_body | `amodule` modname_binder amodule_body | `import` optional_mode modname_binder import_body | `mark` STRING]
module_body	::=	: module_type version_opt = module_expr withspec_opt
valuability	::=	`valuable`
	|	`cvaluable`
	|	`nonvaluable`
valuabilities	::=	(valuability , valuability)
cmodule_body	::=	hash : eqs module_type valuabilities module_type version_val = module_expr
hmodule_body	::=	: eqs module_type version_nonopt = module_expr
amodule_body	::=	: module_type = modname_use
import_body	::=	: module_type version_constraint_opt likespec resolvespec_opt moo_module_opt
cimport_body	::=	hash : module_type valuabilities module_type version_constraint_val likestr resolvespec_nonopt moo_module
himport_body	::=	: module_type version_constraint_nonopt likestr
hash	::=	`hash` (`hmodule` modname_extern hmodule_body)
	|	`hash` (`himport` modname_extern himport_body)
	|	LITHASH
	|	ABSTRNAME
hash_or_modname_dot_ident	::=	[(hash | modname_use) . ident_extern]
name_value	::=	[`name_value` ((`hash` (hash . ident_extern) app_ty) | `hash` (core_type_pri , STRING)) | `hash` (core_type_pri , STRING , name_value)) | ABSTRNAME app_ty))]
eqs	::=	{ (} | eqs_body_non_empty })
eqs_body_non_empty	::=	[eqs_body_item [, eqs_body_non_empty]]
eqs_body_item	::=	[(hash | modname_use) . typname_extern = core_type_pri]
version_opt	::=	[`version` version]
version_val	::=	`version` version
version_nonopt	::=	`version` version
version_constraint_val	::=	`version` version_constraint
version_constraint_nonopt	::=	`version` version_constraint
version_constraint_opt	::=	[`version` version_constraint]

withspec_opt	::=	[with! weqs]
weqs_single	::=	modname_use . typname_extern = core_type_pri
weqs	::=	weqs_rev
weqs_rev	::=	[{ weqs_single , } weqs_single]
likespec	::=	[like modname_use | likestr]
likestr	::=	[like struct structure end]
resolvespec_nonopt	::=	[by resolvespec_non_empty]
resolvespec_opt	::=	[resolvespec_nonopt]
moo_module_opt	::=	[moo_module]
moo_module	::=	[= (unlinked | modname_use)]
module_expr	::=	struct structure end
module_type	::=	sig signature end
structure_items	::=	{ structure_item }
structure	::=	[structure_item structure_items]
structure_item	::=	let ident_binder = typed_expr
	|	type typname_binder = core_type_pri
signature_items	::=	{ signature_item }
signature	::=	[signature_item signature_items]
signature_item	::=	val ident_binder : core_type_pri
	|	type typname_binder
	|	type typname_binder = core_type_pri
	|	type typname_binder : kind
marshalled_body	::=	marshalled_nameenv_opt , { definitions } , { loctyp_list } , { store } , simple_expr , core_type_pri
marshalled_nameenv_opt	::=	_
	|	nameenv
marshalled_value_pri	::=	marshalled (marshalled_body)
store	::=	[store_non_empty]
store_non_empty	::=	[{ store_item , } store_item]
store_item	::=	(location := expr)
hash_in_version	::=	hash (hmodule modname_extern hmodule_body)
	|	hash (himport modname_extern himport_body)
	|	LITHASH
	|	ABSTRNAME
version_literal	::=	INT
	|	hash_in_version
version	::=	atomic_version [version_dotted_suffix]
version_dotted_suffix	::=	{ . atomic_version } . atomic_version
atomic_version	::=	myname
	|	version_literal
atomic_hash_version_constraint	::=	[modname_use | hash_in_version]
atomic_version_constraint	::=	[atomic_hash_version_constraint | INT]
atomic_version_constraints_non_empty	::=	[{ atomic_version_constraint . } atomic_version_constraint]
tail_version_constraint	::=	atomic_version_constraint
	|	INT - INT
	|	- INT
	|	INT -
	|	*
version_constraint	::=	[name = atomic_hash_version_constraint | tail_version_constraint | atomic_version_constraints_non_empty . tail_version_constraint]
resolvespec_non_empty	::=	[resolvespec_item [, resolvespec_non_empty]]
resolvespec_item	::=	Static_Link

```
                      |   Here_Already
                      |   STRING
seq_expr             ::=  [ expr [ ( ; | ||| ) seq_expr ] ]
expr                 ::=  simple_expr
                      |   simple_expr simple_expr_or_app_ty_list
                      |   let rec ident_internal_binder optional_colon_core_type_pri =
                          function mtch_when_sugary in seq_expr
                      |   match seq_expr with mtch
                      |   function mtch_when_sugary
                      |   try seq_expr with mtch
                      |   ref opt_ty simple_expr
                      |   raise simple_expr
                      |   if seq_expr then expr else expr
                      |   while seq_expr do seq_expr done
                      |   expr :: expr
                      |   expr && expr
                      |   expr || expr
                      |   expr := opt_ty expr
                      |   expr = opt_ty expr
                      |   expr @ opt_ty expr
                      |   expr + expr
                      |   expr - expr
                      |   expr * expr
                      |   expr > expr
                      |   expr < expr
                      |   expr INFIXOP0 expr
                      |   expr INFIXOP1 expr
                      |   expr INFIXOP2 expr
                      |   expr INFIXOP3 expr
                      |   expr INFIXOP4 expr
                      |   expr freshfor expr
                      |   - expr
                      |   Function typname_internal_binder -> seq_expr
                      |   let { typname_internal_binder , ident_internal_binder } =
                          typed_expr in seq_expr
                      |   namecase expr with { typname_internal_binder , (
                          ident_internal_binder , ident_internal_binder ) } when ident_use =
                          expr -> expr otherwise -> expr
typed_expr           ::=  seq_expr
                      |   seq_expr : loc_core_type
                      |   seq_expr as loc_core_type
                      |   typed_expr1
                      |   seq_expr ; typed_expr1
                      |   seq_expr ||| typed_expr1
typed_expr1          ::=  { core_type_pri , expr } as core_type_pri
```

simple_expr	::=	{ constr0 | ident_use | econst_use | modname_use . ident_extern | hash . ident_extern | location | (typed_expr) | (expr_comma_list) | ! opt_ty simple_expr | constr1 simple_expr | standalone_infixop | `fresh` opt_ty | `cfresh` opt_ty | `hash` (hash_or_modname_dot_ident) app_ty | `hash` (core_type_pri , expr) app_ty | `hash` (core_type_pri , expr , expr) app_ty | name_value | `swap` expr `and` expr `in` simple_expr | `support` opt_ty simple_expr | modname_use @ ident_extern | `name_of_tie` simple_expr | `val_of_tie` simple_expr | PREFIXOP | PREFIXOP_TYP opt_ty | [expr]_[}^{ core_type_pri } | [expr]_[eqs_body_non_empty }^{ core_type_pri } | `marshal` simple_expr simple_expr | `marshalz` STRING simple_expr | `unmarshal` }
simple_expr_or_app_ty_list	::=	simple_expr
	|	app_ty
	|	simple_expr simple_expr_or_app_ty_list
	|	app_ty simple_expr_or_app_ty_list
opt_ty	::=	[%[core_type_pri]]
app_ty	::=	[%[core_type_pri]]
optional_colon_core_type_pri	::=	[: core_type_pri]
location	::=	{< INT >}
loctyp_list	::=	[loctyp_list_non_empty]
loctyp_list_non_empty	::=	{ loctyp_pair , } loctyp_pair
loctyp_pair	::=	(location : core_type_pri)
mtch	::=	[match_cases]
mtch_when_sugary	::=	[mtch | (ident_internal_binder : core_type_pri) match_action]
match_cases	::=	pattern_match_action { | pattern_match_action }
pattern_match_action	::=	pattern match_action
match_action	::=	-> seq_expr
expr_comma_list	::=	(expr_comma_list | expr) , expr
standalone_infixop	::=	((standalone_infixopstr) | &&) | ||) | ! opt_ty) | = opt_ty) | := opt_ty) | @ opt_ty))
standalone_infixopstr	::=	[+ | - | * | < | > | INFIXOP0 | INFIXOP1 | INFIXOP2 | INFIXOP3 | INFIXOP4]
pattern	::=	pattern_pri
pattern_pri	::=	simple_pattern
	|	constr1 simple_pattern
	|	pattern_pri :: pattern_pri
simple_pattern	::=	ident_internal_binder
	|	_
	|	constr0
	|	- INT
	|	(pattern_pri)
	|	(pattern_pri : core_type_pri)
	|	(pattern_comma_list)
pattern_comma_list	::=	(pattern_comma_list | pattern_pri) , pattern_pri
kind	::=	`Type`
	|	`Eq` (core_type_pri)
loc_core_type	::=	core_type_pri
core_type_pri	::=	fun_core_type
	|	`forall` typname_internal_binder . core_type_pri
	|	`exists` typname_internal_binder . core_type_pri
fun_core_type	::=	tup_core_type { -> tup_core_type }

simple_core_type	::=	(core_type_pri)
	|	typname_constr_use0
	|	modname_use . typname_extern
	|	hash . typname_extern
	|	simple_core_type ref
	|	simple_core_type name
	|	simple_core_type typname_constr_use1
tup_core_type	::=	simple_core_type [* core_type_list_tuple | + core_type_list_sum]
core_type_list_tuple	::=	simple_core_type { * simple_core_type }
core_type_list_sum	::=	simple_core_type { + simple_core_type }
constr0	::=	[ ] opt_ty
	|	None opt_ty
	|	baseconstr0
baseconstr0	::=	( )
	|	INT
	|	false
	|	true
	|	CHAR
	|	STRING
	|	BASECON0
constr1	::=	inj INT app_ty
	|	Some
	|	tiecon
	|	NODE
	|	BASECON1
ident_use	::=	LIDENT
econst_use	::=	ECONST
ident_binder	::=	[LIDENT [LIDENT]]
ident_internal_binder	::=	LIDENT
ident_extern	::=	LIDENT
typname_constr_use0	::=	LIDENT
typname_constr_use1	::=	LIDENT
typname_binder	::=	[LIDENT [LIDENT]]
typname_extern	::=	LIDENT
typname_internal_binder	::=	LIDENT
modname_use	::=	[UIDENT [UIDENT]]
modname_binder	::=	[UIDENT [UIDENT]]
modname_extern	::=	UIDENT
semisemis	::=	[[semisemis_plus]]
semisemis_plus	::=	[{ ;; } ;;]

21 Library interfaces

The following libraries are semi-automatically imported from OCaml – see the OCaml documentation for their semantics. For the moment, for historical reasons, the types are mostly concretized. They are subject to frequent change.

```
(* Automatically generated by genlib.ml. Do not edit directly! *)

module hash!Pervasives : sig
  val min : int -> int -> int
  val max : int -> int -> int
  val not : bool -> bool
  val abs : int -> int
  val lnot : int -> int
  val int_of_char : char -> int
  val char_of_int : int -> char
  val string_of_bool : bool -> string
  val bool_of_string : string -> bool
  val string_of_int : int -> string
  val int_of_string : string -> int
  val print_char : char -> unit
  val print_string : string -> unit
  val print_int : int -> unit
  val print_endline : string -> unit
  val print_newline : unit -> unit
  val prerr_char : char -> unit
  val prerr_string : string -> unit
  val prerr_int : int -> unit
  val prerr_endline : string -> unit
  val prerr_newline : unit -> unit
  val read_line : unit -> string
  val read_int : unit -> int
end

module hash!Agraphics : sig
  val open_graph : string -> unit
  val close_graph : unit -> unit
  val set_window_title : string -> unit
  val clear_graph : unit -> unit
  val size_x : unit -> int
  val size_y : unit -> int
  val rgb : int -> int -> int -> int
  val set_color : int -> unit
  val background : unit -> int
  val foreground : unit -> int
  val black : unit -> int
  val white : unit -> int
  val red : unit -> int
  val green : unit -> int
  val blue : unit -> int
  val yellow : unit -> int
  val cyan : unit -> int
  val magenta : unit -> int
  val plot : int -> int -> unit
  val plots : (int * int) list -> unit
  val point_color : int -> int -> int
  val moveto : int -> int -> unit
  val rmoveto : int -> int -> unit
  val current_x : unit -> int
  val current_y : unit -> int
  val current_point : unit -> int * int
  val lineto : int -> int -> unit
  val rlineto : int -> int -> unit
  val curveto : int * int -> int * int -> int * int -> unit
  val draw_rect : int -> int -> int -> int -> unit
  val draw_poly_line : (int * int) list -> unit
  val draw_poly : (int * int) list -> unit
  val draw_segments : (int * int * int * int) list -> unit
  val draw_arc : int -> int -> int -> int -> int -> int -> unit
  val draw_ellipse : int -> int -> int -> int -> unit
  val draw_circle : int -> int -> int -> unit
  val set_line_width : int -> unit
  val draw_char : char -> unit
  val draw_string : string -> unit
  val set_font : string -> unit
  val set_text_size : int -> unit
  val text_size : string -> int * int
  val fill_rect : int -> int -> int -> int -> unit
  val fill_poly : (int * int) list -> unit
  val fill_arc : int -> int -> int -> int -> int -> int -> unit
  val fill_ellipse : int -> int -> int -> int -> unit
  val fill_circle : int -> int -> int -> unit
  val transp : unit -> int
  val wait_next_event : int list -> int * int * bool * bool * char
  val mouse_pos : unit -> int * int
  val button_down : unit -> bool
  val read_key : unit -> char
  val key_pressed : unit -> bool
  val sound : int -> int -> unit
  val auto_synchronize : bool -> unit
  val synchronize : unit -> unit
  val display_mode : bool -> unit
  val remember_mode : bool -> unit
end

module hash!Char : sig
  val code : char -> int
  val chr : int -> char
  val escaped : char -> string
  val lowercase : char -> char
  val uppercase : char -> char
  val compare : char -> char -> int
  val unsafe_chr : int -> char
end

module hash!String : sig
  val length : string -> int
  val get : string -> int -> char
  val create : int -> string
  val make : int -> char -> string
  val copy : string -> string
  val sub : string -> int -> int -> string
  val concat : string -> string list -> string
  val escaped : string -> string
  val index : string -> char -> int
  val rindex : string -> char -> int
  val index_from : string -> int -> char -> int
  val rindex_from : string -> int -> char -> int
  val contains : string -> char -> bool
  val contains_from : string -> int -> char -> bool
  val rcontains_from : string -> int -> char -> bool
  val uppercase : string -> string
  val lowercase : string -> string
  val capitalize : string -> string
  val uncapitalize : string -> string
  val compare : string -> string -> int
end

module hash!Sys : sig
  val file_exists : string -> bool
  val remove : string -> unit
  val rename : string -> string -> unit
  val getenv : string -> string
  val command : string -> int
  val chdir : string -> unit
  val getcwd : unit -> string
  val catch_break : bool -> unit
end

module hash!Tcp : sig
  type fd : Type
  type ip : Type
  type port : Type
  type addr : Eq(ip * port)
  type netmask : Type
  type ifid : Type
  type msgbflag : Eq(int)
  type sock_type : Eq(int)
  val ip_of_string : string -> ip
  val string_of_ip : ip -> string
  val port_of_int : int -> port
  val int_of_port : port -> int
  val fd_of_int_private : int -> fd
  val int_of_fd : fd -> int
  val ifid_of_string2 : string -> ifid
  val string_of_ifid2 : ifid -> string
  val netmask_of_int2 : int -> netmask
  val int_of_netmask2 : netmask -> int
  val accept : fd -> fd * (ip * port)
  val bind : fd -> ip option -> port option -> unit
  val close : fd -> unit
  val connect : fd -> ip -> port option -> unit
  val dup : fd -> fd
  val dupfd : fd -> int -> fd
  val getifaddrs2 : unit -> (ifid * ip * ip list * netmask) list
  val getsockname : fd -> ip option * port option
  val getpeername : fd -> ip * port
  val getsockerr : fd -> unit
  val getsocklistening : fd -> bool
  val listen : fd -> int -> unit
  val pselect2 : fd list -> fd list -> fd list -> (int * int) option -> fd list *  (fd list * fd list)
  val recv : fd -> int -> msgbflag list -> string *  ((ip option * port option) * bool) option
  val send : fd -> (ip * port) option -> string -> msgbflag list -> string
  val shutdown : fd -> bool -> bool -> unit
  val sockatmark : fd -> bool
  val socket : int -> fd
  val tcp_socket : unit -> fd
  val udp_socket : unit -> fd
end

module hash!Persist : sig
  val write : string -> unit
  val read : unit -> string
  val write2 : string -> unit
  val read2 : unit -> string
end

module hash!Digest : sig
  val string : string -> string
  val substring : string -> int -> int -> string
  val file : string -> string
  val to_hex : string -> string
end

module hash!Filename : sig
  val concat : string -> string -> string
  val is_relative : string -> bool
  val is_implicit : string -> bool
  val check_suffix : string -> string -> bool
  val chop_suffix : string -> string -> string
  val chop_extension : string -> string
  val basename : string -> string
  val dirname : string -> string
  val temp_file : string -> string -> string
  val quote : string -> string
end

module hash!Unix : sig
  val sleep : int -> unit
end
```

22 The IO module

For writing concise examples we use either the persistent store `IO` module or the following network `IO` module, which implements `send` and `receive` with loopback TCP and provides brief aliases for `Pervasives.print_string` and `Pervasives.print_int`.

```
module IO :
  sig
    val print_int : int->unit
    val print_string : string -> unit
    val print_newline : unit -> unit
    val send : string -> unit
    val receive : unit -> string
  end =
  struct
    let print_int = function x ->
      Pervasives.print_int x

    let print_string  = function s ->
      Pervasives.print_string s

    let print_newline  = function () ->
      Pervasives.print_newline ()

    let send = function data ->
      let fdesc = Tcp.tcp_socket () in
      let _ = Tcp.connect fdesc (Tcp.ip_of_string "127.0.0.1") (Some (Tcp.port_of_int 6666)) in
      let pad =
        function s ->
          function n ->
            let padding =
              String.make ( n - (String.length s)) ' ' in
            (s ^ padding) in
      let data_length = String.length data in
      let data_length_string =
        pad (Pervasives.string_of_int data_length) 21 in
      let rec send_all = function s ->
        let no_options = [] in
        let s' = (Tcp.send fdesc None s no_options) in
        if   0 = (String.length s') then () else send_all s' in
      send_all (data_length_string ^ data);
      Tcp.close fdesc

    let receive = function () ->
      let fdesc = Tcp.tcp_socket () in
      let _ = Tcp.bind  fdesc (Some (Tcp.ip_of_string "127.0.0.1"))
        (Some (Tcp.port_of_int 6666)) in
      let _ = Tcp.listen fdesc 5 in
      let (fdesc2,(_,_))= Tcp.accept fdesc in
      let rec recv_n_bytes = function n ->
        let no_options = [] in
        let (s, _) = Tcp.recv fdesc2 n no_options in
        let l = String.length s in
        if  l >= n then s else s ^ (recv_n_bytes (n-l)) in
      let data_length_string = recv_n_bytes 21 in
      let first_space = String.index data_length_string ' ' in
      let data_length_string' = String.sub data_length_string 0 first_space in
      let data_length = Pervasives.int_of_string data_length_string' in
      let data = recv_n_bytes data_length in
      Tcp.close fdesc;
      Tcp.close fdesc2;
      data
  end
```

Appendix

This appendix gives most of the Acute syntax for reference. This is the fully type-annotated source language, including sugared forms, together with other non-source constructs that are needed to express the semantics. The implementation can infer many of the type annotations, and the *mode*, *withspec*, *likespec*, *vce*, *vne*, and *resolvespec* annotations on **module** and **import** default to reasonable values if omitted. The internal parts M, t and x of identifiers M_M, t_t and x_x are inferred by scope resolution.

Novel source features are highlighted in green and novel non-source constructs are highlighted in yellow.

Abstract names n **Store locations** l **Standard library constants** (with arity) x^n

Kinds

K ::= Type | EQ(T)

Types

T ::= int | bool | string | unit | char | void | $T_1 * .. * T_n$ | $T_1 + .. + T_n$ | $T \to T'$ | T list | T option | T ref | exn | $\mathrm{M}_M.\mathrm{t}$ |
t | $\forall\, t.T$ | $\exists\, t.T$ | T name | T tie | thread | mutex | cvar | thunkifymode | thunkkey | thunklet | h.t | n

Constructors C_0 ::= ... C_1 ::= ...

Operators

op ::= **ref**$_T$ | $(=_T)$ | $(<)$ | $(\leq)$ | $(>)$ | $(\geq)$ | **mod** | **land** | **lor** | **lxor** | **lsl** | **lsr** | **asr** | $(+)$ | $(-)$ | $(*)$ | $(/)$ | $-$ | $(@_T)$ | $(\hat{})$ |
create_thread$_T$ | **self** | **kill** | **create_mutex** | **lock** | **try_lock** | **unlock** | **create_cvar** | **wait** | **signal** |
broadcast | **exit** $_T$ | **compare_name**$_T$ | **thunkify** | **unthunkify**

Expressions

e ::= C_0 | C_1 e | e_1 :: e_2 | $(e_1, .., e_n)$ | **function** $mtch$ | **fun** $mtch$ | l | op^n e_1 ... e_n | x^n e_1 ... e_n | x | $\mathrm{M}_M.\mathrm{x}$ |
if e_1 **then** e_2 **else** e_3 | **while** e_1 **do** e_2 **done** | e_1 && e_2 | e_1 || e_2 | e_1 **;** e_2 | e_1 e_2 | **!**$_T e$ | e_1 **:=**$_T$ e_2 |
match e **with** $mtch$ | **let** $p = e'$ **in** e'' | **let** $x : T\ p_1..p_n = e'$ **in** e'' | **let rec** $x : T$ = **function** $mtch$ **in** e |
let rec $x : T\ p_1..p_n = e'$ **in** e'' | **raise** e | **try** e **with** $mtch$ | $\Lambda\, t \to e$ | e T | $\{T, e\}$ **as** T' | **let** $\{t, x\} = e_1$ **in** e_2 |
marshal e_1 e_2 : T | **unmarshal** e **as** T |
fresh$_T$ | **cfresh**$_T$ | **hash**$(X.\mathrm{x})_T$ | **hash**$(T, e_2)_{T'}$ | **hash**$(T, e_2, e_1)_{T'}$ |
swap e_1 **and** e_2 **in** e_3 | e_1 **freshfor** e_2 | **support**$_T$ e | M_M@x | **name_of_tie** e | **val_of_tie** e |
namecase e_1 **with** $\{t, (x_1, x_2)\}$ **when** $x_1 = e \to e_2$ **otherwise** $\to e_3$ | $e_1 ||| e_2$ |
n_T | h.x | e_1 **:=**$'_T$ e_2 | **marshalz** $\underline{s}$ e : T | **RET**$_T$ | **SLOWRET**$_T$ | **TERM** | **op**$(op^n)^n$ e_1 .. e_n |
op$(x^n)^n$ e_1 .. e_n | $[e]^T_{eqs}$ | **resolve**$(\mathrm{M}_M.\mathrm{x}, \mathrm{M}'_{M'}, resolvespec)$ | **resolve_blocked**$(\mathrm{M}_M.\mathrm{x}, \mathrm{M}'_{M'}, resolvespec)$

Matches and Patterns

$mtch$ ::= $p \to e$ | $(p \to e \mid mtch)$
p ::= $(\_ : T)$ | $(x : T)$ | C_0 | C_1 p | p_1 :: p_2 | $(p_1, .., p_n)$ | $(p : T)$

Signatures and Structures

sig ::= empty | **val** $\mathrm{x}_x : T$ sig | **type** $\mathrm{t}_t : K$ sig Sig ::= **sig** sig **end**
str ::= empty | **let** $\mathrm{x}_x : T\ p_1..p_n = e$ str | **type** $\mathrm{t}_t = T$ str Str ::= **struct** str **end**

Version and version constraint expressions

$avne$	::=	$\underline{n}$ | $\underline{N}$ | h | **myname**	$avce$	::=	$ahvce$ | $\underline{n}$
vne	::=	$avne$ | $avne.vne$	$dvce$	::=	$avce$ | $\underline{n}$–$\underline{n}'$ | –$\underline{n}$ | $\underline{n}$– | $*$ | $avce.dvce$
$ahvce$	::=	$\underline{N}$ | h | M_M	vce	::=	$dvce$ | **name** = $ahvce$

Source definitions and Compilation Units

$$\begin{array}{rcl}
\mathit{sourcedefinition} & ::= & \textbf{module}\ \mathit{mode}\ \mathrm{M}_M : \mathit{Sig}\ \ \textbf{version}\ \ \mathit{vne} = \mathit{Str}\ \mathit{withspec} \\
 & & \textbf{import}\ \mathit{mode}\ \mathrm{M}_M : \mathit{Sig}\ \ \textbf{version}\ \ \mathit{vce}\ \mathit{likespec}\ \ \textbf{by}\ \ \mathit{resolvespec} = \mathit{Mo} \\
 & & \textbf{mark}\ \ \mathrm{MK} \\
 & & \textbf{module}\ \mathrm{M}_M : \mathit{Sig} = \mathrm{M'}_{M'} \\
 & & \\
\mathit{mode} & ::= & \textbf{hash} \mid \textbf{cfresh} \mid \textbf{fresh} \mid \textbf{hash!} \mid \textbf{cfresh!} \\
\mathit{withspec} & ::= & \mathrm{empty} \mid \textbf{with}\ !\mathit{eqs} \\
\mathit{likespec} & ::= & \mathrm{empty} \mid \textbf{like}\ \ \mathrm{M}_M \mid \textbf{like}\ \ \mathit{Str} \\
\mathit{resolvespec} & ::= & \mathrm{empty} \mid \textsc{Static\_Link}, \mathit{resolvespec} \mid \textsc{Here\_Already}, \mathit{resolvespec} \mid \mathit{URI}, \mathit{resolvespec} \\
\mathit{Mo} & ::= & \mathrm{M}_M \mid \textsc{unlinked} \\
 & & \\
\mathit{compilationunit} & ::= & \mathrm{empty} \mid e \mid \mathit{sourcedefinition}\ \textbf{;;}\ \mathit{compilationunit} \mid \\
 & & \textbf{includesource}\ \ \mathit{sourcefilename}\ \textbf{;;}\ \mathit{compilationunit} \mid \\
 & & \textbf{includecompiled}\ \ \mathit{compiledfilename}\ \textbf{;;}\ \mathit{compilationunit}
\end{array}$$

Compiled Definitions and Compiled Units

$$\begin{array}{rcl}
\mathit{definition} & ::= & \textbf{cmodule}\ldots \mid \textbf{cimport}\ldots \mid \textbf{module}\ \ \textbf{fresh}\ldots \mid \textbf{import}\ \ \textbf{fresh}\ldots \mid \textbf{mark}\ \ \mathrm{MK} \\
\mathit{compiledunit} & ::= & \mathrm{empty} \mid e \mid \mathit{definition}\ \textbf{;;}\ \mathit{compiledunit}
\end{array}$$

Marshalled value contents (marshalled values are strings that unmarshal to these)

$$\mathit{mv} \quad ::= \quad \textbf{marshalled}(E_{\mathrm{n}},\ E_s,\ s,\ \mathit{definitions},\ e,\ T)$$

Module names (hashes and abstract names)

$$\begin{array}{rcl}
h & ::= & \textbf{hash}(\textbf{hmodule}_{\mathit{eqs}}\ \mathrm{M} : \mathit{Sig}_0\ \ \textbf{version}\ \ \mathit{vne} = \mathit{Str}) \mid \textbf{hash}(\textbf{himport}\ \mathrm{M}\ : \mathit{Sig}_0\ \ \textbf{version}\ \ \mathit{vc}\ \ \textbf{like}\ \ \mathit{Str}) \mid \mathrm{n} \\
X & ::= & \mathrm{M}_M \mid h
\end{array}$$

Expression name values

$$\mathbf{n} \quad ::= \quad \mathrm{n}_T \mid \textbf{hash}(h.\mathrm{x})_T \mid \textbf{hash}(T', \underline{s})_T \mid \textbf{hash}(T', \underline{s}, \mathbf{n})_T$$

(In the implementation all h and $\mathbf{n}$ forms can be represented by a long bitstring taken from $\mathbb{H}$, ranged over by $\underline{N}$.)

Type equation sets (the M_M forms occur in the source language)

$$\mathit{eqs} \quad ::= \quad \varnothing \mid \mathit{eqs}, X.\mathrm{t} \approx T$$

Type Environments (for identifiers and store locations — not required at run-time in the implementation)

$$E \quad ::= \quad \mathrm{empty} \mid E, x : T \mid E, l : T\ \ \mathsf{ref} \mid E, t : K \mid E, \mathrm{M}_M : \mathit{Sig}$$

Type Environments (for global names — not required in the implementation)

$$\begin{array}{l}
E_{\mathrm{n}} ::= \mathrm{empty} \mid E_{\mathrm{n}}, \mathrm{n} : \textbf{nmodule}_{\mathit{eqs}}\ \mathrm{M} : \mathit{Sig}_0\ \ \textbf{version}\ \ \mathit{vne} = \mathit{Str} \mid E_{\mathrm{n}}, \mathrm{n} : \textbf{nimport}\ \mathrm{M} : \mathit{Sig}_0\ \ \textbf{version}\ \ \mathit{vc}\ \ \textbf{like}\ \ \mathit{Str} \mid \\
\qquad E_{\mathrm{n}}, \mathrm{n} : \textsc{Type} \mid E_{\mathrm{n}}, \mathrm{n} : T\ \ \mathsf{name}
\end{array}$$

Processes

$$P \quad ::= \quad 0 \mid (P_1 \mid P_2) \mid \mathbf{n} : \mathit{definitions}\ e \mid \mathbf{n} : \mathrm{MX}(\underline{b}) \mid \mathbf{n} : \mathrm{CV}$$

Single-Machine Configurations

$$\mathit{config} \quad ::= \quad E_{\mathrm{n}}\ \textbf{;}\ \langle E_s,\ s,\ \mathit{definitions},\ P\rangle$$

References

[Ali03] The Alice project, 2003. `http://www.ps.uni-sb.de/alice/`.

[AVWW96] J. Armstrong, R. Virding, C. Wikstrom, and M. Williams. *Concurrent Programming in Erlang*. Prentice Hall, 1996. 2nd ed.

[BCF02] N. Benton, L. Cardelli, and C. Fournet. Modern concurrency abstractions for C$^\sharp$. In *Proc. ECOOP, LNCS 2374*, 2002.

[BHS$^+$03] G. Bierman, M. Hicks, P. Sewell, G. Stoyle, and K. Wansbrough. Dynamic rebinding for marshalling and update, with destruct-time λ. In *Proc. ICFP*, 2003.

[Bou03] Gérard Boudol. ULM: A core programming model for global computing. Draft, 2003.

[Car95] L. Cardelli. A language with distributed scope. In *Proc. 22nd POPL*, pages 286–297, 1995.

[DEW99] S. Drossopoulou, S. Eisenbach, and D. Wragg. A fragment calculus towards a model of separate compilation, linking and binary compatibility. In *Proc. LICS*, pages 147–156, 1999.

[dot03] Packacking and deploying .net framework applications (.net framework tutorials), 2003. `http://msdn/microsoft.com/library/default.asp?url=/library/en-us/dnanchor/html/netdevanchor.asp`.

[Fes01] Fabrice Le Fessant. Detecting distributed cycles of garbage in large-scale systems. In *Proc. Principles of Distributed Computing(PODC)*, 2001.

[FGL$^+$96] C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, and D. Rémy. A calculus of mobile agents. In *Proc. 7th CONCUR, LNCS 1119*, 1996.

[GMZ00] D. Grossman, G. Morrisett, and S. Zdancewic. Syntactic type abstraction. *ACM TOPLAS*, 22(6):1037–1080, 2000.

[HL94] R. Harper and M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Proc. 21st POPL*, 1994.

[HP] R. Harper and B. C. Pierce. Design issues in advanced module systems. Chapter in *Advanced Topics in Types and Programming Languages*, B. C. Pierce, editor. To appear.

[HP99] Haruo Hosoya and Benjamin C. Pierce. How good is local type inference? Technical Report MS-CIS-99-17, University of Pennsylvania, June 1999.

[HRY04] M. Hennessy, J. Rathke, and N. Yoshida. Safedpi: A language for controlling mobile code. In *Proc. FOSSACS, LNCS 2987*, 2004.

[HS00] R. Harper and C. Stone. A type-theoretic interpretation of standard ML. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*. 2000.

[JoC] JoCaml. `http://pauillac.inria.fr/jocaml/`.

[LBR03] Didier Le Botlan and Didier Rémy. MLF: Raising ML to the power of System-F. In *Proceedings of the International Conference on Functional Programming (ICFP 2003), Uppsala, Sweden*, pages 27–38. ACM Press, aug 2003.

[Ler94] X. Leroy. Manifest types, modules, and separate compilation. In *Proc. 21st POPL*, 1994.

[LPSW03] J. J. Leifer, G. Peskine, P. Sewell, and K. Wansbrough. Global abstraction-safe marshalling with hash types. In *Proc. 8th ICFP*, 2003.

[MCHP04] T. Murphy, K. Crary, R. Harper, and F. Pfenning. A symmetric modal lambda calculus for distributed computing. In *Proc. LICS*, 2004.

[OK93] Atsushi Ohori and Kazuhiko Kato. Semantics for communication primitives in a polymorphic language. In *Proc. POPL*, pages 99–112, 1993.

[OZZ01] Martin Odersky, Christoph Zenger, and Matthias Zenger. Colored local type inference. *ACM SIGPLAN Notices*, 36(3):41–53, March 2001.

[PT98] Benjamin C. Pierce and David N. Turner. Local type inference. 1998. Full version in *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(1), January 2000, pp. 1–44.

[PT00] B. C. Pierce and D. N. Turner. Pict: A programming language based on the pi-calculus. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*. 2000.

[Rém02] Didier Rémy. Using, understanding, and unraveling the ocaml language. In Gilles Barthe, editor, *Applied Semantics. Advanced Lectures. LNCS 2395*, pages 413–537. 2002.

[Rep99] J. H. Reppy. *Concurrent Programming in ML*. Cambridge Univ Press, 1999.

[Ros03] A. Rossberg. Generativity and dynamic opacity for abstract types. In *Proc. 5th PPDP*, August 2003.

[Sew00] Peter Sewell. Applied π – a brief tutorial. Technical Report 498, Computer Laboratory, University of Cambridge, August 2000. An extract appeared as Chapter 9, Formal Methods for Distributed Processing, A Survey of Object Oriented Approaches.

[Sew01] P. Sewell. Modules, abstract types, and distributed versioning. In *Proc. 28th POPL*, 2001.

[SLW$^+$] Peter Sewell, James J. Leifer, Keith Wansbrough, Mair Allen-Williams, Francesco Zappa Nardelli, Pierre Habouzit, and Viktor Vafeiadis. Acute: high-level programming language design for distributed computation. Draft available `http://www.cl.cam.ac.uk/users/pes20/acute`.

[SPG03] M. R. Shinwell, A. M. Pitts, and M. J. Gabbay. FreshML: Programming with binders made simple. In *Proc. 8th ICFP*, pages 263–274, 2003.

[SWP99] P. Sewell, P. T. Wojciechowski, and B. C. Pierce. Location-independent communication for mobile agents: a two-level architecture. In *Internet Programming Languages, LNCS 1686*, pages 1–31, 1999.

[SY97] T. Sekiguchi and A. Yonezawa. A calculus with code mobility. In *Proc. 2nd FMOODS*, pages 21–36, 1997.

[TLK96] Bent Thomsen, Lone Leth, and Tsung-Min Kuo. A Facile tutorial. In *CONCUR'96, LNCS 1119*, 1996.

[US01] A. Unyapoth and P. Sewell. Nomadic Pict: Correct communication infrastructure for mobile computation. In *Proc. POPL*, pages 116–127, January 2001.

[ves] Vesta. `http://www.vestasys.org/`.

[Wei02] Stephanie Weirich. *Programming With Types*. PhD thesis, Cornell University, August 2002.

Index

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot-St-Martin (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399