Acute: High-level programming language design for distributed computation

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

Abstract

Existing languages provide good support for typeful programming of standalone programs. In a distributed system, however, there may be interaction between multiple instances of many distinct programs, sharing some (but not necessarily all) of their module structure, and with some instances rebuilt with new versions of certain modules as time goes on. In this paper we discuss programming-language support for such systems, focussing on their typing and naming issues.

We describe an experimental language, Acute, which extends an ML core to support distributed development, deployment, and execution, allowing type-safe interaction between separately built programs. The main features are: (1) type-safe marshalling of arbitrary values; (2) type names that are generated (freshly and by hashing) to ensure that type equality tests suffice to protect the invariants of abstract types, across the entire distributed system; (3) expression-level names generated to ensure that name equality tests suffice for type safety of associated values, e.g. values carried on named channels; (4) controlled dynamic rebinding of marshalled values to local resources; and (5) thunkification of threads and mutexes to support computation mobility.

These features are a large part of what is needed for typeful distributed programming. They are a relatively lightweight extension of ML, should be efficiently implementable, and are 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. This disentangles the language runtime from communication intricacies. This paper highlights the main design choices in Acute. It is supported by a full language definition (of typing, compilation, and operational semantics), by a prototype implementation, and by example distribution libraries.

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, distributed systems cannot in general be deployed or updated atomically. They are not necessarily 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, especially without type safety. Existing programming languages, such as ML, Haskell, Java and C$^\sharp$, provide good support for local computation, with rich type structures and (mostly) static type-safety guarantees. When it comes to distributed computation, however, they fall short, with little support for its many system-development challenges.

In this work we seek to remedy this lack, concentrating on what must be added to ML-like (typed, call-by-value, higher-order) languages to support typed distributed programming. We have defined and implemented a programming language, Acute, which extends an OCaml core with features for type-safe marshalling and naming in the distributed setting. Our extensions are lightweight changes to ML, but suffice to enable sophisticated distributed infrastructure, e.g. substantial parts of JoCaml (Conchon & Fessant, 1999), Nomadic Pict (Sewell *et al.*, 1999), and Ambient primitives (Cardelli & Gordon, 1998), to be programmed as simple libraries. Acute's support for interaction *between* programs goes well beyond previous work, allowing type-safe interaction between different runtime instances, different builds, and different versions of programs, whilst respecting modular structure and type abstraction boundaries in each interacting partner. In a distributed system it will often be impossible to detect all type errors statically, but it is not necessary to be completely dynamic — errors should be detected as early as possible in the development, deployment, and execution process. We show how this can be done.

The main part of this paper, §2–9, is devoted to an informal presentation of the main design issues, which we introduce briefly in the remainder of this section. It uses small but executable examples to discuss these from the programmer's point of view. Acute has a full definition (Sewell *et al.*, 2004), covering syntax, typing, compilation, and operational semantics, and a prototype implementation is also available (Sewell *et al.*, 2005a). This closely mirrors the structure of the operational semantics; it is efficient enough to run moderate examples. The semantics and implementation are outlined in §12 and §13 respectively. The definition and implementation have both been essential to resolve the many semantic subtleties introduced by the synthesis of the various features.

We demonstrate that Acute does indeed support typeful distributed programs with various examples (§11), including distributed communication infrastructure libraries, and in §14 and §15 we describe related and future work and conclude.

Starting point The starting point for Acute is a conventional ML-like language. The Acute core language 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. 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 to keep the language relatively 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 and is addressed in recent work by Peskine (2007)). Some more substantial extensions are discussed in the Conclusion. To avoid syntax debate we fix on that of OCaml. Most of the Acute grammar is shown in Appendix A, with the novel forms highlighted.

Type-safe marshalling (§2, §3) Our basic addition to ML is type-safe marshalling: constructs to marshal arbitrary values to byte-strings, with a type equality check at unmarshal-time guaranteeing safety. We argue that this is the right level of abstraction for a general-purpose distributed language, allowing complex communication infrastructure algorithms to be coded (type-safely) as libraries, above the standard byte-string network and persistent store APIs, rather than built in to the language runtime. We recall the different design choices for trusted and untrusted interaction.

Dynamic linking and rebinding (§4) When marshalling and unmarshalling code values, e.g. to communicate ML functions between machines, it may be necessary to *dynamically rebind* them to local resources at their destination. Similarly, one may need to *dynamically link* modules. There are many questions here: how to specify which resources should be shipped with a marshalled value and which should be dynamically rebound; what evaluation strategy to use; when rebinding takes effect; and what to rebind to. In this section our aim is to articulate the design space; for Acute we make interim choices which suffice to bring out the typing and versioning issues involved in rebinding while keeping the language simple. A running Acute program consists roughly of a sequence of `module` definitions (of ML structures), `import`s of modules with specified signatures, which may or may not be linked, and `mark`s which indicate where rebinding can take effect; together with running processes and a shared store.

Type names (§5) Type-safe marshalling demands a run time notion of *type identity* that makes sense across multiple versions of differing programs. For concrete types this is conceptually straightforward — for example, one can check the equality between type `int` from one program instance and type `int` from another. For abstract types more care is necessary. Static type systems for ML modules involve non-trivial theories of type equality based on the source-code names of abstract types (e.g. `M.t`), but these are only meaningful within a single program. We generate globally meaningful *run-time type names* for abstract types in three ways: by *hashing* module definitions, taking their dependencies into account; or *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 for modules with effect-full initialisation. In all three cases the way in which names are generated ensures that type name equality tests suffice to protect the invariants of abstract types.

Expression-level names (§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 ways in which expression-level names are generated ensure that name equality tests suffice to guarantee that any associated values (e.g. any values passed on named channels) have the right types. The polytypic `support` and `swap` operations of Shinwell, Pitts, and Gabbay's FreshOCaml (Shinwell, 2005; Shinwell *et al.*, 2003) are included to support swizzling of local names during communication.

Versions and version constraints (§7, §8) 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, simply 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.

There is a 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.

Local concurrency and thunkification (§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* (loosely analogous to `call/cc`), allowing a collection of threads (and mutexes and condition variables) to be atomically captured as a thunk that can then be marshalled and communicated or stored; this enables various constructs for mobility and checkpointing to be coded up.

Polymorphism (§10) Acute 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. For writing typed communication libraries we need to compare names of different name types, with the 'true' branch typed under the assumption that these are the same; we add a **namecase** operation that combines this with existential unpacking.

Examples (§11) We demonstrate that Acute does indeed support typeful distributed programs with several medium-scale examples, all written as libraries in Acute above the byte-string TCP Sockets API: a typed distributed channel library, an implementation of the Nomadic Pict (Sewell *et al.*, 1999) primitives for communication and mobility, and an implementation of the Ambient primitives (Cardelli & Gordon, 1998). These require and use most of the new features.

Semantics (§12) The main parts of the Acute definition are a type system, a definition of compilation, and a small-step operational semantics. The static type system for source programs is based on an OCaml core and a second-class module system, with singleton kinds for expressing abstract and manifest type fields in modules.

The definition of compilation describes how global type-level and expression-level names are constructed, including the details of hash bodies.

The operational semantics for rebinding rests on our *redex-time* evaluation strategy (Bierman *et al.*, 2003) for simply typed λ-calculus and here adapted to a second-class module system: to express rebinding the semantics must preserve the module structure throughout computation instead of substituting it away.

The semantics also preserves abstraction boundaries throughout computation, with a generalisation of the *coloured brackets* of Grossman et al. (2000) 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. This too can be erased in implementations, which can implement hashes and fresh names with literal bit-strings (e.g. 128-bit MD5 or 160-bit SHA1 hashes, and pseudo-random numbers), but is needed to state type preservation and progress properties. As we discuss later, the abstraction-preserving semantics makes these rather stronger than usual.

Implementation (§13) 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 in structure to the semantics. The runtime interprets an intermediate language which is essentially the abstract syntax extended with closures. Performance is roughly 3000 times slower than OCaml bytecode.

The definition is too large (on the scale of the ML definition (Milner *et al.*, 1990) rather than an idealised λ-calculus) to make proofs of soundness properties feasible with the available resources and tools. To increase confidence in both semantics and implementation, therefore, our implementation is designed to optionally type-check the entire configuration after each reduction step. This has been extremely useful, identifying delicate issues in both the semantics and the code.

Relationship to previous work Acute builds on previous work, in which we introduced new-bound type names for abstract types (Sewell, 2001), hash-generated type names (Leifer *et al.*, 2003a), and controlled dynamic rebinding in a lambda-calculus (Bierman *et al.*, 2003), all in simple variants for for small calculi.

Our contribution in this paper 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. The main new technical innovations 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.

This paper is a revised and extended version of Sewell et al. (2005b) and Part I of Sewell et al. (2004). With respect to the latter technical report, §12 outlining the semantics is entirely new, and there are various other local changes. The main changes with respect to the former paper 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;
- addition of §10 on polymorphism and `namecase`;
- addition of §12 outlining key aspects of the semantic definition; and
- addition of §13 describing the implementation.

2 Distributed abstractions: language vs libraries

A fundamental question for a distributed language is what communication support 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 (Thomsen *et al.*, 1996), Obliq (Cardelli, 1995), and JoCaml (Conchon & Fessant, 1999)), 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 distributed abstractions as 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 primitives `send:string->unit` and `receive:unit->string`. To emphasise that interaction might be via communication or via persistent store, 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 two separately built programs vertically, separated by a dash —.

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 as it receives a `string` which it uses as an `int`. 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, but 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. Note, though, that this option is not always available: 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.(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.)

In Acute we focus on the trusted case, with option (a), and the problems of distributed typing, naming, and rebinding it raises. We aim to protect against accidental programming and configuration error, not against malice. A full language should also support the untrusted case, perhaps with type- or proof-carrying code for marshalled functions.

One goal for Acute is to make it possible to write high-level distributed infrastructure (middleware) in a high-level language. It would be a very interesting exercise to provide as much as possible of the functionality of a middleware framework such as CORBA in an Acute library. Note, though, that the main focus of Acute is rather different from that of the core of CORBA (the IDL and IIOP). CORBA provides globally defined scalar types and arrays and records thereof, along with marshalling and unmarshalling functions in many languages. Its goal is thus to transport *concrete* data between programs written in a *heterogenous* collection of languages. Acute, by contrast, only supports communication between *homogeneous* runtimes (all executing code compiled by Acute), but has much more ambitious support for the *content* of communicated data, including values of abstract types, fragments of executable code (modules), etc., and less heavyweight machinery. However, we would hope that much of the higher-level functionality of CORBA (name and trader services, messaging, fault-tolerance, etc.) could be elegantly written in an Acute-like language.

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 module 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 ASN.1 or 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.

Marshalling can be done on values of any type, including polymorphic values (Acute has System F style explicit polymorphism). Elsewhere we discuss in depth the issues involved in handling implicit polymorphism, in our work on HashCaml (Billings *et al.*, 2006).

Rather than a polymorphic `marshal`, that can be used uniformly on values of arbitrary types, one could provide machinery for user-defined marshalling functions, integrating marshalling with datastructure traversal. In Acute we factor the two out.

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 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 (2–3) (to be rebound) and (4) (to be shipped) must be specified by the programmer at the sender site; there is usually no way for an implementation to infer the correct behaviour. We adapt the mechanism proposed by Bierman et al. (2003) (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; 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, with the `6` taken from the `M2` module in the second program and the `3` taken from a copy of the `M1` module shipped with the marshalled value.

```
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 (avoiding “DLL hell”) — 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 use 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 `M1` module definition earlier in the example). The `version *` allows rebinding to any version of `M1`. This overrides the default behaviour, which would permit rebinding only to identical copies of the local `M1`. Marks are simply string constants, not binders subject to alpha equivalence, as they themselves 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. We developed two alternative CBV reduction strategies, in (Bierman *et al.*, 2003), 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 occurs even later, when the identifier appears in a context which needs to destruct the outermost structure of the value. Both of these are, in the absence of marshalling, observationally equivalent to call-by-value reduction (Stoyle, 2006). The destruct-time semantics permits more rebinding, but is also rather complex. We therefore 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 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 run time. 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, with an unlinked `import` of `M`.

```
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.

4.5 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. 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 perhaps be encoded (with exceptions and affine continuations, or by encoding imports as option references) but here we focus on the user language.

4.6 The structure of marks and modules

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

This linear structure suffices as a setting to explore the typing and naming issues in the remainder of the paper, but it is probably not ideal. For example, one might want cyclic linking (involving the complexities of recursive modules or mixins); or support for two endpoints to negotiate about what modules are already shared and what need to be shipped; or explicit control over what must *not* be shipped, e.g. due to license restrictions or security concerns. We leave these for future work.

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
```

In this grammar, *e* is an expression; `includesource` specifies a source file to statically include at the program point where it appears; and `includecompiled` does the same but for object files.

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; its 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. Further, if one has a nested marshalling, i.e. marshalled functions that marshal functions, in combination with a non-trivial mark structure, then the linear order will often not be satisfactory. We leave these issues for future work.

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 ))
```

(The `%[int]` is an explicit System F type application; later we will also use `%[]` as place-holders for inferred types.) 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 (Sekiguchi & Yonezawa, 1997; Boudol, 2003).

Some applications would demand distributed references together with distributed garbage collection (as JoCaml provides (Fessant, 2001)). 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 nested *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 with respect to 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 (1994) and Harper and Lillibridge (1994) do).

For distributed programming we need a notion of type equality that makes sense at run time 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 (Leifer *et al.*, 2003a) to dependent-record modules. 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 but overlapping 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 (Sewell, 2001), 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 (Harper & Stone, 2000), 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 (Pierce & Turner, 2000) and Nomadic Pict (Sewell *et al.*, 1999; Unyapoth & Sewell, 2001) 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.

The constructs described in this section provide, we believe, a good level of abstraction safety. In Acute, abstract types are not compatible (either statically or dynamically) with their representation types, and we give the programmer enough control to tune the dynamic type equality for various scenarios. It seems impossible, however, to prevent *all* information about the implementation of an abstract type leaking out. Most obviously, an equality comparison on the byte sequences of marshalled values of two implementations of an abstract type could reveal that they they have different representations.

Additionally, there are several points at which it seems that programmers writing distributed multi-version programs involving abstract types would need to actively break abstractions (see §8.2) or to indirectly break abstraction by overriding valuability checks (see §8.3). Dynamic rebinding to modules that provide abstract types intrinsically requires some representation information (see the *likespec*s of §8.1), and the polytypic name operations of §6.3 are also intrinsically abstraction-breaking (though the usefulness of these may be debatable).

Accordingly, abstract types in Acute, as in other languages such as OCaml, should be viewed as mechanisms to reduce a class of accidental programmer errors, not as providing guarantees of parametricity.

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, 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 run time, 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 explain 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 (and hence that there will be no unmarshal failures in the communication library).

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 run-time 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*. (Acute does not 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`.)

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).

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

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

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

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

These generate 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 run time. 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. (Such coercions support `Chan.c` type constructors too, e.g. to construct an `int Chan.c name` from an `(int->unit) name`.) 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`.

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 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 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 version and constraint languages — in a particular distributed code environment, one may want a particular local choice for these. 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 (see §8.2) 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 (Harper & Pierce, 2005): 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 run-time 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, and indeed is rebound to a plus-three function.

```
 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. In cases where there are such, 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 `imports` 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.

We proposed (Sewell, 2001; Leifer *et al.*, 2003a) 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 `ints`, 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 Leifer et al. (2003a).

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 run time 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, delivering an incoming message by demultiplexing on a name it contains. 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 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. 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 is more generally 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 whether 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 (Sewell, 2000) 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. Our examples demonstrate 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 optimised version, 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, as expressed through the sum type `thunkifymode` defined below.

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. A `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`, and `thunkkey`. The first three types are empty (phantom types); they are introduced to form types `thread name`, `mutex name`, and `cvar name`. A `thunkifymode` is either `Interrupting` or `Blocking`; and 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    create_cvar : cvar name->unit
lock : mutex name->unit            wait : cvar name->mutex name->unit
try_lock : mutex name->bool        signal : cvar name->unit
unlock : mutex 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 `thunkkey`s 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 then thunkifies, marshals, and sends thread `t1`; the second then receives that and applies it, creating a freshly named thread (and mutex) locally that prints `5 6 7 8`.

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. (An alternate approach to universals and existentials, which we do not consider here, is to add first-class modules to the language (Peskine, 2007).)

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* (Pierce & Turner, 1998; Hosoya & Pierce, 1999) and *coloured local type inference* (Odersky *et al.*, 2001). Without subtyping, the MLF of Le Botlan and Rémy (2003) allows full System F but can infer types for all ML-typable programs.

Secondly, the interaction between subtyping and hash types requires further work — for instance, using a subhash order derived from subtype and subversion relationships, which is dynamically propagated (Deniélou & Leifer, 2006).

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 run-time 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 run time. Alternatively, if one keeps the structure it would be possible to add some form of run-time type analysis (Weirich, 2002) 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 run-time 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

We have written three example distributed communication libraries in Acute: a distributed message-passing library; an implementation of the Nomadic Pict constructs for migration of mobile computations and communication between them; and an implementation of the Ambient calculus primitives. There are also two games that mostly exercise local computation, `blockhead` and `minesweeper`; the latter using marshalling to save and restore the game state. The distributed message-passing library shows how many 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; a `namecase` operation is used to unpack existential name/value packages, allowing a new type equality to be used in the 'true' branch of a name equality test. 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. Finally, a simple `RFI` module implements remote function invocation above distributed channels.

Clients of this library 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`.

The Nomadic Pict library 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).

The Ambient library implements the mobility primitives of the Ambient calculus. An ambient is a collection of running threads and resources (including other ambients) that migrates as a unit: mobility amounts to restructuring the nesting tree of the ambients. In a distributed world, this nested structure is shared among different runtimes. Interactions between ambients in the same run-time are resolved using local concurrency, mutexes and cvars. Interaction between remote machines may cause an ambient to migrate to another runtime: this is implemented using thunkification and marshalling, on top of the `TCP_string_messaging` library.

It is worth noting that these libraries provide coherent abstractions above the combined low-level concurrency and thunkification features of the language, and clients of the libraries should not also directly use those low-level features. For example, the Nomadic Pict library provides an API including message-passing concurrency and thread creation within a group. The implementation of this thread creation function invokes the low-level `create_thread` but also updates metadata associated with the group, all protected by locking calls, which direct, client usage of the low-level `create_thread` would not do. Similarly, clients should not directly use mutex operations or thunkification. Language support for enforcing such constraints is an interesting problem for future work on module systems.

Each of these libraries is around 1000 lines of Acute code, including comments and utility functions. They took a few days or weeks to write, in sharp contrast to the many months required for the original Nomadic Pict implementation. Much of the remaining complexity is related to local concurrency and locking. The distributed aspects were rather straightforward, with the Acute rebinding semantics used to ensure that communicated code is correctly rebound to the local state of the libraries at the receiver.

The code for these examples is available on the Acute web page (Acute team, n.d.). A related (but simpler) example is given in our later HashCaml paper (Billings *et al.*, 2006).

12 Semantics

The Acute definition, Part III of (Sewell *et al.*, 2004), defines the language syntax, type system, typed desugaring, compilation (abstractly, dealing with issues such as the compile-time construction of type names), operational semantics, and the errors that can arise during compilation and execution.

The definition is written in rigorous but informal (not machine-processed) mathematics, and we state precise type preservation and progress conjectures. We have not attempted to prove these results. Any proof on this scale is a daunting prospect, be it informal or machine-checked — the definition alone is around 80 pages, roughly the size of that of SML (Milner *et al.*, 1990). Indeed, we know of few other high-level languages of similar size with a complete and rigorous definition, and none with fully proven metatheory.[1] Further, the value of informal proofs at this scale is questionable: one might well discover problems in the semantics by carrying out such proof, but it would be hard to have confidence that the proof did not contain errors. Ideally, then, we would have both a machine-processed definition and machine-checked proofs of soundness, but at present our (partial) confidence in the soundness of the definition is based only on a combination of weaker evidence: metatheoretic proofs about small calculi with some of the key features (Sewell, 2001; Bierman *et al.*, 2003; Leifer *et al.*, 2003a), and experience with the implementation (which closely follows the semantics and which can optionally re-typecheck each configuration that is reached).

In this section we give an overview of the key novel features in the definition, illustrated with selected rules and examples.

12.1 Type and term names

The main issue dealt with in the Acute semantics is that of run-time type names. In SML and OCaml types can be erased at run time. In Acute, on the other hand, some run-time representation of types is needed, both at `marshal` points (to include with the marshalled value), and at `unmarshal` points (to do a type equality check between the expected type and that of the marshalled value). For a language with simple types these run-time representations would be conceptually straightforward, isomorphic to the type expressions that

[1] There is ongoing work on Twelf formalisation of a Harper-Stone semantics for SML, with metatheory for an internal language (Lee *et al.*, 2007), and several authors have formalised large foundational Proof-Carrying Code and Typed Assembly Language systems.

occur in source programs. (One might choose to use cryptographic hashes of those instead, gaining performance at the cost of a small probability of error.) In Acute, however, we have abstract types, just as SML and OCaml do. In the absence of marshalling, abstraction is enforced by a combination of statically scoped type names and a type system based on singleton kinds. With marshalling, we need to be able to compare abstract types that may be from partially or completely different programs, so we need run-time representations for these types that make sense globally, not just in some particular scope.

Singleton-kind type system Much of the Acute type system for source-language programs is standard, following the manifest-type system of (Leroy, 1994) and the singleton-kind system of (Harper & Lillibridge, 1994). The type system is over fully type-annotated syntax – our implementation includes a partial type inference algorithm, but that is not formally defined. There is a core language (with System F style polymorphism) and a second-class module language of named structure definitions. Modules can contain type fields which can either be abstract or manifest. For example, the concrete source-language program

```
module EvenCounter'
: sig                                             = struct
    type t          (* abstract *)                  type t=int
    type t'=string  (* concrete/manifest *)         type t'=string
    val start:t                                     let start = 0
    val get:t->int                                  let get = fun (x:int)->x
  end                                             end
```

is formally treated as a definition with a signature

```
module EvenCounter'
: sig
    type t  : Type
    type t' : Eq(string)
    val start:t
    val get:t->int
  end
```

in which both type fields have kind assumptions. The `t` field is simply of kind `Type`, revealing no information, whereas the `t'` field is of kind `Eq(string)` — the kind of all types that are provably equal to `string` — revealing its implementation. The type system uses *paths* `M.t` to name abstract types, with static *selfification* rules that permit the signature of a module identifier (but not of an arbitrary structure) to be strengthened with an equality to the relevant path, as below.

```
module EvenCounter'
: sig
    type t  : Eq(Evencounter'.t)
    type t' : Eq(string)
    val start:t
    val get:t->int
  end
```

Hashes and names in the type grammar Our run-time type representations for abstract types are a dynamic analogue of these paths, and they are introduced by dynamic analogues of the selfification type rules. Hashes and abstract names appear in the type grammar

$$\begin{array}{lll} T & ::= & \ldots \\ & & \mathrm{M}_M.\mathrm{t} \quad \text{t field of module } \mathrm{M}_M \\ & & h.\mathrm{t} \quad \text{t field of hash or name } h \end{array}$$

where h is either a module or import hash, or a pure name:

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

Compilation and module initialisation replace paths $\mathrm{M}_M.\mathrm{t}$ by the $h.\mathrm{t}$ form, which is not permitted in source programs. (Here M is a module external identifier, which does not alpha-vary, and M is an alpha-varying internal identifier.)

As the grammar above shows, the semantics preserves the internal structure of hashes, with the $\mathbf{hash}(\ldots)$ in the semantics treated as a formal constructor. This is needed to state type preservation, to define typing for the expressions that can arise at run time. However, we take care to ensure that an implementation can use numeric hashes, without depending on their internal structure. The current Acute implementation can do either, using the internal structure only when the optional per-step typechecking is enabled.

The typing rules for hashes and names are similar to those for module identifiers. For example, the two rules for type formation are below.

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

(we return later to the role of the *eqs* subscripts). The rule for $h.\mathrm{t}$ has an additional premise that permits use of $h.\mathrm{t}$ only when t really is abstract in the signature of h. The assumption $E \vdash_{\mathit{eqs}} h : \mathit{Sig}$ indirectly ensures that h is well-formed, for which the module-hash rule is below.

$$\frac{\begin{array}{l} h = \mathbf{hash}(\mathbf{hmodule}_{\mathit{eqs}}\ \mathrm{M} : \mathit{Sig}_0\ \ \mathbf{version}\ \ \mathit{vne} = \mathit{Str}) \\ E_{\mathrm{n}}, E_{\mathrm{const}} \vdash_{\mathit{eqs}} \mathit{Str} : \mathit{Sig}_0 \\ \vdash \mathit{Str}\ \ \mathbf{flat} \\ \vdash \mathit{Sig}_0\ \ \mathbf{flat} \end{array}}{E_{\mathrm{n}} \vdash h\ \ \mathbf{ok}}$$

This checks that the hashed structure Str matches the signature Sig_0, but in a globally meaningful type environment $E_{\mathrm{n}}, E_{\mathrm{const}}$ rather than the type environment of any particular program context. The E_{const} is a fixed type environment of special constants for the standard library. The E_{n} is a global environment of assumptions on the pure names that have been created so far, of the form below. It associates names n (taken from a fixed infinite set) with types (for term-level names), kind TYPE (for type names of opened existentials), or module/import data (for **fresh** and **cfresh** modules and imports).

$$\begin{array}{lll} E_{\mathrm{n}} & ::= & \text{empty} \\ & & E_{\mathrm{n}}, \mathrm{n} : T \quad \mathsf{name} \\ & & E_{\mathrm{n}}, \mathrm{n} : \mathrm{TYPE} \\ & & E_{\mathrm{n}}, \mathrm{n} : \mathbf{nmodule}_{eqs}\ \mathrm{M} : Sig_0\ \ \mathbf{version}\ \ vne = Str \\ & & E_{\mathrm{n}}, \mathrm{n} : \mathbf{nimport}\ \mathrm{M} : Sig_0\ \ \mathbf{version}\ \ vc\ \ \mathbf{like}\ \ Str \end{array}$$

We construct hashes ‘up to provable type equality’. Part of this is captured by the **flat** premises of the type rule above, which ensure that the structure is normalised w.r.t. any type definitions $\mathrm{t}_t = T$ within it (by substituting T for t), and similarly for manifest type assumptions in the signature.

There is a further subtlety concerning hashes that might be pasted into version expressions and constraints by the programmer. For these, the system cannot ensure that they are hashes of well-formed modules, but in this context soundness does not depend on that, so they can be treated semantically simply as numeric constants.

Construction of module hashes and names Hashes and names h are constructed per-module (and per-import), rather than per-abstract-type. How and when they are constructed depends on whether the module (or import) is annotated **hash**, **cfresh** or **fresh** (which will generally depend on whether it is valuable, cvaluable or non-valuable):

- **hash**: compute module hash h at compile time
- **cfresh**: generate module name h fresh at compile time
- **fresh**: generate module name h fresh at module initialisation time

The Acute compilation semantics specifies how the first two are done, of which the most interesting is the **hash** case. In outline:

1. 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).
2. any *withspec* is checked, and the resulting set of type equations, normalised, is recorded.
3. the hash of this module is constructed, first replacing any other-module expression dependencies $\mathrm{M}'_{M'}.\mathrm{x}$ by the corresponding $h'.\mathrm{x}$.
4. that hash is used to selfify the remaining abstract type fields of the signature, replacing **type** $\mathrm{t}_t : \mathrm{TYPE}$ by **type** $\mathrm{t}_t : \mathrm{EQ}(h.\mathrm{t})$.
5. the version number expression of the module is evaluated, replacing **myname** by the hash h. This must be done after calculation of the hash as otherwise recursive hashes would be needed.

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 (computable from Sig_0 and h), vn is the version number, and Str is the normalised structure. Syntactic equality on normalised types corresponds to provable static type equality.

Compilation: simple hash modules For example, the EvenCounter example from §5 is compiled to the cmodule below (generated by our implementation).

```
cmodule EvenCounter[M0] h0_EvenCounter : {}       (* id, name, eqs *)
  sig type t[t0] : Type                           (* abstract sig *)
      val start[start0] : t0
      val get[get0] : t0 -> int
      val up[up0] : t0 -> t0
  end (valuable, valuable)                        (* valuability *)
  sig type t[t0] : Eq(h0_EvenCounter.t)           (* selfified sig *)
      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                      (* version *)
= struct                                          (* 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
```

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 the external module identifier, signature, version expression, and structure. This hash is taken up to alpha equivalence by choosing canonical strings for bound identifiers (the semantics is up to alpha throughout, so there the formal **hash**(...) constructor is applied to an alpha equivalence class) 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 numeric hash forms are shown. The compiled `cmodule EvenCounter` has two signatures, one in which source abstract types are still abstract (used for type-checking later modules) 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` (used for type normalisation of later modules). The version of the compiled module has defaulted to its hash-generated name.

Compilation: hash module dependencies The result of compiling modules `M` and `EvenCounter` from §8 is below, showing how the construction of hashes captures any semantic dependencies between the modules. 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
cmodule EvenCounter[M₀] h1_EvenCounter : {}
  sig type t[t₀] : Type    [...]
      val up[up₀] : t₀ -> t₀
  end (valuable, valuable)
  sig type t[t₀] : Eq(h1_EvenCounter.t)      [...]
      val up[up₀] : h1_EvenCounter.t -> h1_EvenCounter.t
  end
  version h1_EvenCounter
= struct type t[t₀] = int        [...]
         let up[up₀] = function (x₀ : int) -> M[M₀].f x₀
  end
```

where

```
h0_M =
  hash(hmodule M : {}
    sig val f[f₀] : int -> int end
  version myname
  = struct
     let f[f₀] = function (x₀ : int) -> x₀ + 2
  end)
  = 0#FBCF6A65CCD4F06635C5188503EA9B72
```

and

```
h1_EvenCounter =
  hash(hmodule EvenCounter : {}
  sig type t[t₀] : Type        [...]
      val up[up₀] : t₀ -> t₀
  end version myname
  = struct
    type t[t₀] = int    [...]
    let up[up₀] = function (x₀ : int) -> h0_M.f x₀
  end)
  = 0#F5EF4DE7D2DCB9E8D56EE8AAD19AE3E9
```

Compilation: fresh and cfresh modules, and imports In the **cfresh** case compilation constructs an h for the module randomly instead of by hashing, but is otherwise similar. In the **fresh** case the h for the module is constructed randomly at the start of its execution, whereupon it can be used to selfify and normalise types just as in compilation.

Imports are treated broadly similarly, with a *likespec* rather than a *withspec*, resulting in a compiled form

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

Term names Supporting the various term-name cases of §6 is basically straightforward: the expression grammar for executing programs (though not for source programs) permits $\mathbf{hash}(h.\mathrm{x})_{T'}$, for a name created based on a module value field; n_T, for a pure name; and the various other hash forms. (There are complications w.r.t. coloured brackets for names which we do not go in to here.)

12.2 Run-time configurations

The operational semantics is defined as a labelled transition system over configurations of a single machine. These have the form

$$E_{\mathrm{n}}\ \mathbf{;}\ \langle E_s,\ s,\ \mathit{definitions},\ P\rangle$$

where E_{n} is a global type environment for fresh names, s and E_s are the store and its typing environment, *definitions* is a list of module values, imports, and marks, and P is a multiset of named running threads ($\mathbf{n} : \mathit{definitions}\ \ e$), mutexes ($\mathbf{n} : \mathrm{MX}(\underline{b})$) and condition variables ($\mathbf{n} : \mathrm{CV}$). The type environments E_{n} and E_s are not required in a production implementation. The E_{n} is not regarded as binding in the configuration body (in contrast to π-calculus new-binders) to avoid the need to consider alpha conversion of names occuring within hashes and marshalled values.

Compiling a program generates a configuration $E_{\mathrm{n}}\ \ \mathbf{;}\ \ \langle\mathrm{empty}, \varnothing, \mathrm{empty}, \mathbf{n} : \mathit{definitions}\ \ e\rangle$, where E_{n} contains **cfresh** names created during compilation, and $\mathbf{n}$ is the name of the initial thread. When this is executed the module definitions in *definitions* are initialised sequentially, producing module values which are moved to the global definitions of the configuration. Only when *definitions* is empty is the expression e executed. New threads might be created during module initialisation or during execution of e; in either case they are created without per-thread module definitions, but must be in the scope of the previously initialised global definitions.

Labels on transitions are used to record OS calls and returns, and also requests for code at a URI (needed when evaluating resolve specs). The operational semantics is defined using various classes of evaluation context and reduction (or labelled transition) axioms.

12.3 Module field instantiation

By keeping the global module definitions (rather that substituting them away) we can perform redex-time instantiation as in §4.3. The simplest case is that of a module field reference $\mathrm{M}_M.\mathrm{x}$ that is in redex position, with a module value for M_M

$$\mathit{definition} = \mathbf{cmodule}_{h;\mathit{eqs}_0;\mathit{Sig}_0}\ \mathit{vubs}\ \mathrm{M}_M : \mathit{Sig}_1 = \mathit{Str}$$

in the global $\mathit{definitions}$ (suppose also that no type abstraction is involved). There will be a value field $(\mathbf{let}\ \ \mathrm{x}_x = v^{\varnothing}) \in\ \mathit{Str}$ and we can simply instantiate $\mathrm{M}_M.\mathrm{x}$ by $v^{\varnothing}$. More generally, M_M may be bound indirectly, via a chain of linked imports.

Otherwise, M_M may be bound to an unlinked import (again, possibly via a chain of linked imports) and the $\mathit{resolvespec}$ of that import must be examined. This may involve local linking or attempts to fetch new definitions from URIs; in either case a linkok check is performed between the import and any module (or import) it might be linkable to, involving checks that (1) the external identifiers match; (2) the interfaces match (we check a syntactic subsignature relation that coincides with the full relation on flattened signatures); (3) the versions match; and (4) the representations of types mentioned in the import's $\mathit{likespec}$ match.

12.4 Marshalling

Marshalled values are strings that represent 5-tuples

$$\mathbf{marshalled}(E_\mathrm{n},\ E_s,\ s,\ \mathit{definitions},\ v^{\varnothing},\ T)$$

where $v^{\varnothing}$ is the value itself, T is its (normalised) type, for use in the unmarshal type equality check, $\mathit{definitions}$ is a sequence of module definitions and imports, s and E_s are a location-closed store and store typing that are reachable from locations in $v^{\varnothing}$ and $\mathit{definitions}$, and E_n is the fragment of the global name type environment needed for the other components. The semantics does not specify in detail how these are represented as strings; it is simply parameterised on a $\mathrm{raw\_unmarshal}$ function from strings to such 5-tuples that includes all marshalled values in its range. A production implementation would not need E_n or E_s.

The interesting question in defining the marshalling semantics is what $\mathit{definitions}$ are shipped. Broadly, for a $\mathbf{marshal}$"MK"$(v^{\varnothing})\,T$ with respect to a mark "MK", either there is no $\mathbf{mark}$ "MK" in the global $\mathit{definitions}$ (in which case an exception is raised) or we have some

$$\mathit{definitions} = \mathit{definitions}_1\ \mathbf{;;}\ \mathbf{mark}\ \text{“MK”}\ \mathbf{;;}\ \mathit{definitions}_2$$

and $\mathbf{mark}$ "MK" $\notin\ \mathit{definitions}_2$. We prune $\mathit{definitions}_2$, omitting any modules that are not needed, but including all marks, to give $\mathit{definitions}_2'$. We also calculate which modules from $\mathit{definitions}_1$ are refered to by these or by the value $v^{\varnothing}$ (taking care that these can also contain store locations and store locations can contain functions refering to modules). For each of these modules (which must be in $\mathit{definitions}_1$) we construct an import; the final marshalled $\mathit{definitions}'$ is $\mathrm{makeimports}(\mathit{definitions}_1)\ \mathbf{;;}\ \mathit{definitions}_2'$. The import constructed for a structure is a default form with a signature taken from that of the structure and an exact-name version constraint. The import constructed for an import is essentially an unlinked copy of the original import.

For example, recall the program from §4.2 that marshals a thunk referring to one module above a mark and one below.

```
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)
```

The marshalled value 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[M0].y, M2[M0].z)),
  unit -> int * int)
```

When a value of a type involving an abstract type is marshalled, type normalisation ensures that a type involving a hash or name is included in the marshalled value, e.g. for the value of type `unit->M1.t` marshalled below the run-time type in the marshalled value is `unit -> h0_M1.t`.

```
module M1:sig type t val y:t end = struct type t=int let y=6 end
mark "MK"
marshal "MK" (fun ()-> M1.y : unit->M1.t )
```

Here that value also refers to the code of `M1`, which is defined above the mark `"MK"` referred to in the `marshal`, so an import of that module is also generated and included in the marshalled value. In this case the import has a nontrivial *likespec*.

```
marshalled ({}, {
  cimport M1[M0] h0_M1 :
   sig type t[t] : Type val y[x] : t end (valuable, valuable)
   sig type t[t] : Eq(h0_M1.t) val y[x] : h0_M1.t end
   version name = h0_M1
   like struct type t[t] = int end
   by Here_Already = unlinked }, {}, {},
  (function (x : unit) -> match x with () -> M1[M₀].y),
  unit -> h0_M1.t)
```

12.5 Coloured brackets

Most calculi and languages with type abstraction (existentials or ML-style modules) either have no operational semantics or have reduction rules that forget abstraction boundaries, e.g. with this rule for opening an existential package

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

or analogous rules for modules that replace abstract type paths by their representation.

This style of semantics suffices for soundness, ensuring that values are not used during execution at inappropriate concrete types (e.g. that integers are not used as functions). However, there is a stronger property that interests us, namely abstraction preservation: for example, that values of type `EvenCounter.t` (see §5) are not manipulated at run time except by code from the `EvenCounter` module. While abstraction preservation does indeed hold for the executions of well-typed source programs thanks to static typing and scoping rules, this cannot be seen by looking that the execution states once the reductions have deleted all abstraction.

The Acute design involves many subtle issues relating to abstraction boundaries — when a value marshalled within one abstraction boundary can be unmarshalled in another, the semantics of *withspec* and *likespec*, etc.. Accordingly, to establish greater confidence in the internal coherence of the semantics we arrange to preserve abstraction boundaries throughout execution. Building on our previous work (Leifer *et al.*, 2003a), which in turn drew on (Grossman *et al.*, 2000), we use *coloured brackets* to delimit subexpressions in which sets eqs of type equalities between abstract types and their representations can be used. Additionally, most type judgements, and the operational relations, are stated with respect to such sets of equalities eqs, reflecting which abstract types may be considered transparent.

Coloured bracket expressions take the form $[e]^T_{eqs}$, where the type equations, generated by the grammar,

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

record 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 $[e]^T_{eqs}$ is of type T; inside, the type equations eqs can be used in typechecking e, as formalised by the associated type rule:

$$\frac{\begin{array}{l} E \vdash eqs\ \ \textbf{ok} \\ E \vdash_{\varnothing}\ T : \textsc{Type} \\ E \vdash_{eqs'}\ e : T \end{array}}{E \vdash_{eqs}\ [e]^T_{eqs'} : T}$$

Note that brackets are not additive — the inner expression is typed with respect to eqs', not the union of the ambient eqs and the eqs': thus the type equalities available when evaluating code from a module are determined by the module itself, not the chain of modules traversed on the call stack.

Brackets and non-empty equation sets do not occur in user source language programs, and brackets could be erased in a production implementation. (Indeed, maintaining them at run time would likely be very expensive, and even in our prototype implementation we added a 'vacuous-bracket' optimisation that greedily suppresses semantically superfluous brackets rather than have them be eliminated lazily, as defined in Sec. 16.11 of (Sewell *et al.*, 2004).) In the semantics, brackets are introduced primarily when instantiating a module field reference $\mathrm{M}_M.\mathrm{x}$ from a module M_M that introduced some abstract types, as we illustrate in the following example.

Consider the `EvenCounter` of §5, with fields `start : EvenCounter.t` and `get : EvenCounter.t->int`:

```
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
    [...]                                  [...]
  end                                    end
```

Expressions `EvenCounter.start` and `EvenCounter.get` will be instantiated, when they appear in redex-position, to $[\mathtt{0}]^{h.\mathtt{t}}_{h.\mathtt{t=int}}$ and $[\mathtt{fun\ (x:int)\text{->}x}]^{h.\mathtt{t\text{->}int}}_{h.\mathtt{t=int}}$ respectively, where $h = \mathit{h0\_EvenCounter}.\mathtt{t}$ is the hash-generated module name of `EvenCounter` as in §12.1.

The behaviour of brackets could be expressed either with a structural congruence or with reductions. The former might be conceptually clearer, but the latter is easier to implement, and so we adopt it to simplify our prototype implementation (in which we do keep brackets at run time, to support the optional run-time type checking of reachable configurations). A further disadvantage of structural congruence is that it would complicate progress and determinacy proofs by adding that obligation to show confluence with respect to the directed reductions, something we were pleased to avoid in Leifer et al. (2003b).

Bracket reduction rules push brackets through values in cases where the outermost value structure and the outermost type structure of the bracket type coincide, e.g.

$$
\begin{array}{lll}
[v_1^{eqs'} :: v_2^{eqs'}]^{T\ \mathsf{list}}_{eqs'} & \longrightarrow_{eqs} & [v_1^{eqs'}]^{T}_{eqs'} :: [v_2^{eqs'}]^{T\ \mathsf{list}}_{eqs'} \\
[(v_1^{eqs'}, .., v_n^{eqs'})]^{T_1 * .. * T_n}_{eqs'} & \longrightarrow_{eqs} & ([v_1^{eqs'}]^{T_1}_{eqs'}, .., [v_n^{eqs'}]^{T_n}_{eqs'}) \quad n \geq 2 \\
[\mathrm{C}_n\ v_1^{eqs'}\ ...\ v_n^{eqs'}]^{T_0}_{eqs'} & \longrightarrow_{eqs} & \mathrm{C}_n\ [v_1^{eqs'}]^{T_1}_{eqs'}\ ...\ [v_n^{eqs'}]^{T_n}_{eqs'} \\
 & & \text{for other contructors } \mathrm{C}_n : T_1 \to ... \to T_n \to T_0
\end{array}
$$

As reduction can occur inside brackets, we index the reduction relation by a colour eqs (as seen above), which represents the equations provided by the innermost coloured bracket in the surrounding evaluation context. The bracket reduction rules depend on the ambient equations, as we see below, and so the notion of value is also dependent on a set of equations: $v^{eqs'}$ ranges over the expressions that are values at eqs', and $v^{\varnothing}$ ranges over expressions that are values at the empty equation set.

Bracket type revelation permits an abstract type that is transparent both inside and outside coloured brackets to be replaced by its concrete representation:

$$[v^{eqs'}]^{h.\mathrm{t}}_{eqs'} \quad \longrightarrow_{eqs} \quad [v^{eqs'}]^{T}_{eqs'} \qquad (h.\mathrm{t} \approx T) \in \; eqs \; \wedge \; h.\mathrm{t} \; \in \mathrm{dom}(eqs')$$

while bracket elimination removes redundant nested brackets

$$[[v^{eqs''}]^{h.\mathrm{t}}_{eqs''}]^{h.\mathrm{t}}_{eqs'} \quad \longrightarrow_{eqs} \quad [v^{eqs''}]^{h.\mathrm{t}}_{eqs''} \qquad h.\mathrm{t} \; \notin \mathrm{dom}(eqs')$$

(The side condition ensures that this rule does not form a critical pair with others, in particular the revelation rule just above it.)

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:

$$[\textbf{function}\ (x : T) \rightarrow e]^{T' \rightarrow T''}_{eqs'} \quad \longrightarrow_{eqs} \quad \textbf{function}\ (x : T') \rightarrow [\{[x]^{T}_{eqs'}/x\}e]^{T''}_{eqs'}$$

In order to understand the extra brackets added in the substitution, consider any type derivation of the LHS. The binder would be placed in the environment as $x : T$. On the RHS, it appears as $x : T'$, thus breaking type preservation if x were to be used in a subexpression of e for which T and T' were not equivalent. The brackets in the substition prevent this by giving $[x]^{T}_{eqs'}$ the type T, since T and T' *are* indeed equivalent in eqs'. (Our colleague Gilles Peskine has proposed a different strategy (Peskine, 2007), involving adding colours to the bindings in type environments; at the expense of some added complexity in the typing judgements, he can simplify some of the reduction rules, in particular function application, for which he can dispense with the extra brackets present in our system.)

In the reduction axiom for function application

$$(\textbf{function}\ (x : T) \rightarrow e)\ v^{eqs} \quad \longrightarrow_{eqs} \quad \{[v^{eqs}]^{T}_{eqs}/x\}e$$

(and similarly for recursive functions) the value v^{eqs} is well-typed in eqs but not necessarily in other colour contexts where x is used in e, so v^{eqs} is protected by brackets in the substitution on the RHS.

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.

Example Below we show an example reduction sequence for the expression `EvenCounter.get EvenCounter.start`. This is a top-level reduction sequence, with reduction steps at the empty equation set $\varnothing$, but the derivations of several reductions involve reductions at $\{$`h.t = int`$\}$ where h is the hash of `EvenCounter`.

`EvenCounter[M`$_0$`].get EvenCounter[M`$_0$`].start`

$\longrightarrow_{\varnothing}$ instantiate `EvenCounter[M`$_0$`].get`
`[(function (x : int) -> x) ]`$^{h.\texttt{t -> int}}_{\{h.\texttt{t = int}\}}$ `EvenCounter[M`$_0$`].start`

$\longrightarrow_{\varnothing}$ push brackets through lambda

$$\texttt{(function (x:}h\texttt{.t) -> } [[\texttt{x}]^{\texttt{int}}_{\{h\texttt{.t = int}\}}]^{\texttt{int}}_{\{h\texttt{.t = int}\}}\texttt{) EvenCounter[M}_0\texttt{].start}$$

$\longrightarrow_{\varnothing}$ instantiate $\texttt{EvenCounter[M}_0\texttt{].start}$

$$\texttt{(function (x:}h\texttt{.t) -> } [[\texttt{x}]^{\texttt{int}}_{\{h\texttt{.t = int}\}}]^{\texttt{int}}_{\{h\texttt{.t = int}\}}\texttt{) } [\texttt{0}]^{h\texttt{.t}}_{\{h\texttt{.t = int}\}}$$

$\longrightarrow_{\varnothing}$ substitute the value into the body of the function, rebracketing it to preserve its colour

$$\{\ [\ [\ \texttt{0}\ ]^{h\texttt{.t}}_{\{h\texttt{.t = int}\}}\ ]^{h\texttt{.t}}_{\varnothing}\ /\ \texttt{x}\ \}\quad [[\texttt{x}]^{\texttt{int}}_{\{h\texttt{.t = int}\}}]^{\texttt{int}}_{\{h\texttt{.t = int}\}}$$

= substitution

$$[[\quad [\ [\ \texttt{0}\ ]^{h\texttt{.t}}_{\{h\texttt{.t = int}\}}\ ]^{h\texttt{.t}}_{\varnothing}\quad]^{\texttt{int}}_{\{h\texttt{.t = int}\}}]^{\texttt{int}}_{\{h\texttt{.t = int}\}}$$

$\longrightarrow_{\varnothing}$ bracket elimination for the inner two brackets

$$[[\quad [\ \texttt{0}\ ]^{h\texttt{.t}}_{\{h\texttt{.t = int}\}}\quad]^{\texttt{int}}_{\{h\texttt{.t = int}\}}]^{\texttt{int}}_{\{h\texttt{.t = int}\}}$$

$\longrightarrow_{\varnothing}$ bracket type revelation for the inner brackets

$$[[\quad [\ \texttt{0}\ ]^{\texttt{int}}_{\{h\texttt{.t = int}\}}\quad]^{\texttt{int}}_{\{h\texttt{.t = int}\}}]^{\texttt{int}}_{\{h\texttt{.t = int}\}}$$

$\longrightarrow_{\varnothing}$ bracket pushing for the inner brackets through the constructor 0

$$[[\quad \texttt{0}\qquad]^{\texttt{int}}_{\{h\texttt{.t = int}\}}]^{\texttt{int}}_{\{h\texttt{.t = int}\}}$$

$\longrightarrow_{\varnothing}$ bracket pushing for the inner brackets through the constructor 0

$$[\quad \texttt{0}\qquad]^{\texttt{int}}_{\{h\texttt{.t = int}\}}$$

$\longrightarrow_{\varnothing}$ bracket pushing for the inner brackets through the constructor 0

$$\texttt{0}$$

Store- and name-related bracket dynamics Bracket handling for store and name operations 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 Grossman et al. (c.f. §4.2 of (Grossman *et al.*, 2000)).

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:

$$\begin{array}{lll} !_T\ [v^{eqs'}]^{T'\ \mathsf{ref}}_{eqs'} & \longrightarrow_{eqs} & [!_{T'}\ v^{eqs'}]^{T'}_{eqs'} \\ [v'^{eqs'}]^{T'\ \mathsf{ref}}_{eqs'}\ \mathbf{:=}_T\ v^{eqs} & \longrightarrow_{eqs} & [v'^{eqs'}\ \mathbf{:=}_{T'}\ [v^{eqs}]^{T'}_{eqs}]^{\mathsf{unit}}_{eqs'} \end{array}$$

When bracket pulling through $!_T$ it may not be immediately obvious why the bracket on the RHS is at T' and not T. The rule as written 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 with respect to the empty equation set ($v^{\varnothing}$). When we have exposed a raw location, $!_T$ can dereference it:

$$E_{\mathrm{n}}\ \mathbf{;}\ \langle E_s,\ (s, l \mapsto v^{\varnothing}),\ \mathit{definitions},\ !_T\ l\rangle \quad \longrightarrow_{eqs} \quad E_{\mathrm{n}}\ \mathbf{;}\ \langle E_s,\ (s, l \mapsto v^{\varnothing}),\ \mathit{definitions},\ v^{\varnothing}\rangle$$

(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 $E_{\mathrm{n}}, E_s \vdash_{eqs} v^{\varnothing} : T_0$, hence $E_{\mathrm{n}}, E_s \vdash_{eqs} v^{\varnothing} : T$ as desired.)

For assignment, when we have exposed a raw location, $:=_T$ prepares the value to be put in the store by wrapping it in ambient-equation-set brackets; when that becomes a value with respect to $\varnothing$, perhaps involving several bracket reductions, we can install it in the store:

$$\begin{array}{ll} l \mathrel{\mathbf{:=}_T} v^{eqs} & \longrightarrow_{eqs} l \mathrel{\mathbf{:=}'_T} [v^{eqs}]^{T}_{eqs} \\ E_{\mathrm{n}}\ \mathbf{;}\ \langle E_s,\ (s, l \mapsto v'^{\varnothing}),\ \mathit{definitions},\ l \mathrel{\mathbf{:=}'_T} v^{\varnothing}\rangle & \longrightarrow_{eqs} E_{\mathrm{n}}\ \mathbf{;}\ \langle E_s,\ (s, l \mapsto v^{\varnothing}),\ \mathit{definitions},\ ()\rangle \end{array}$$

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. It is unclear whether this is truly satisfactory, but in any case there is a basic tension: the polytypic name operations can intrinsically be used to partially see through some abstraction boundaries.

Type normalisation An abstraction-preserving semantics sheds light on type normalisation and marshalling within abstraction boundaries (c.f. §8.5). In any given type environment E and colour eqs, each semantic type may be represented by any member of an equivalence class of syntactic types defined by the provable-type-equivalence relation $E \vdash_{eqs} T \approx T'$. Our 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 $\{X.t \rightsquigarrow T | (X.t \approx T) \in eqs\}$, $M.t \rightsquigarrow T | M : \mathit{Sig} \in E \wedge t : \mathrm{EQ}(T) \in \mathit{Sig}$, and $t \rightsquigarrow T | t : \mathrm{EQ}(T) \in E$. This is important because in certain circumstances the syntactic representative chosen for a semantic type is significant: especially, our marshal/unmarshal type equality check is a check of equality of normalised types, not of provable equality, so that type equations and brackets need not be maintained at run time.

Reflections on brackets The bracket machinery required to make the semantics abstraction-preserving was non-trivial, so one may ask whether the benefits were worth the complexity. On the whole we believe they were: the semantics makes clear, at all syntactic points in all configurations reachable by reduction, what type equations are in scope. Further, the type preservation property for an abstraction-preserving semantics is a much stronger test that it is internally coherent than it would be for an abstraction-erasing semantics. This (as realised by our run-time configuration typechecking) brought several misconceptions to light during development of the language.

13 Implementation

The implementation is written in FreshOCaml (Shinwell *et al.*, 2003), currently around 25 000 lines of code (we later also ported it to OCaml). 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 helped 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 and represents terms as pairs of an explicit evaluation context and the enclosed term (roughly as in §1.3.1, Ex. 1 of Rémy (2002)) 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 run-time typechecking and with the vacuous bracket optimisation. The naive Fibonacci function of 25 involves about 1.6 million steps and takes 18 seconds, again without run-time 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 run-time 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. While it would be good if run-time typechecking were feasible for these larger examples, it is in fact mostly useful for more focussed test cases — for which one wishes to observe the individual reduction steps in any case.

14 Related work

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 our earlier papers (Sewell, 2001; Leifer *et al.*, 2003a; Bierman *et al.*, 2003), confining our attention here to some of the most relevant distributed programming language developments. Many 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, or on version change. (Several languages, including JoCaml and Nomadic Pict, have ad-hoc 'traders' for establishing initial connections between programs.)

Early work on adding local concurrency to ML resulted in Concurrent ML (Reppy, 1999) 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 (Thomsen *et al.*, 1996). dML (Ohori & Kato, 1993) was another distributed extension of ML, implementable by translation into remote procedure calls without requiring communication at higher types. Erlang (Armstrong *et al.*, 1996) supports concurrency, messaging and distribution, but without static typing.

The Pict experiment (Pierce & Turner, 2000) 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 (Fournet *et al.*, 1996) and subsequent JoCaml implementation (Conchon & Fessant, 1999) 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$^\sharp$ (Benton *et al.*, 2002) 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 (Cardelli, 1995), 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 (Sewell *et al.*, 1999; Unyapoth & Sewell, 2001) 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 (Hennessy *et al.*, 2004) and with modal types (Murphy *et al.*, 2004). Even where this is possible, however, one must still deal with low-level network failure.

Work on Alice (Rossberg *et al.*, 2006; Rossberg, 2003) is perhaps closest to ours, with ML modules, support for marshalling ('pickling') arbitrary values, and run-time fresh generation of abstract type names, but without rebinding, our distributed type and term naming, or version control. Furuse and Weis support type-safe, but not abstraction-safe, marshalling of non-functional values in OCaml (Furuse & Weis, 2000).

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. (1999). The .NET framework supports versioning of *assemblies* (Dot03, 2003). 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 (Buckley *et al.*, 2005).

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 (Heydon *et al.*, 2006) 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 ASN.1 and XML typing.

15 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 not intended as a proposal for a production language, but rather a vehicle for experimentation and a starting point for debate — several necessary but relatively straightforward features have been omitted, and substantial problems remain for future work (especially some of the questions of §4). Nonetheless, we believe that our examples demonstrate that the combination of the above features is much of what is needed to bring the benefits of ML-like languages to the programming of large-scale distributed systems, supporting typed, higher-order, 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 run-time 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, whereupon it is dereferenced. Since field references inside a thunk remain as pointers, they could easily be rebound with only modest changes to the runtime. A preliminary experiment has confirmed that this is feasible (Billings, 2005). This involved adapting the OCaml bytecode runtime to support marshalling of closures and simple rebinding, and replacing the Acute runtime by a simple compiler from a fragment of Acute to this bytecode. Performance for local computation was roughly 2–3 times slower than OCaml bytecode, and there is doubtless much scope for optimisation. Of course compile-time inlining optimisations between parts of code separated by a mark would no longer be straightforward. Our more recent work on HashCaml (Billings *et al.*, 2006) integrated core features of Acute into the OCaml implementation, with encouraging results.

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 a language with lower-level linking might well be simpler than that presented here, factoring out the algorithmic issues of $\mathit{resolvespec}$s, for example.
- Subtyping is needed for many version-change scenarios, perhaps with corresponding subhash relations (Deniélou & Leifer, 2006). 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, explicit automata, and software transactional memory; 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), EPSRC grants GRN24872 and GRT11715, EC FET-GC project IST-2001-33234 PEPITO, and APPSEM 2. We thank Vilhelm Sjöberg and Christian Steinrücken for their early implementation work and Gilles Peskine for discussions on coloured brackets. Andrew Appel, Matthew Fairbairn, Jean-Jacques Lévy, Luc Maranget, Mark Shinwell, Gilles Peskine, Francois Pottier, Alisdair Wren, and the anonymous referees provided helpful comments.

A Acute syntax summary

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 $\boxed{\text{highlighted in a green frame and light background}}$ and novel non-source constructs are $\boxed{\text{highlighted in a red frame and dark background}}$.

Abstract names $\boxed{\mathrm{n}}$ **Store locations** l **Standard library constants** (with arity) x^n

Kinds

$$K \quad ::= \quad \textsc{Type} | \mathrm{EQ}(T)$$

Types

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

Constructors $\mathrm{C}_0 ::= \ldots \qquad \mathrm{C}_1 ::= \ldots$

Expressions

$$\begin{aligned}e ::= {} & \mathrm{C}_0|\mathrm{C}_1\ e|e_1 :: e_2|(e_1,..,e_n)|\textbf{function}\ \ \mathit{mtch}|\textbf{fun}\ \ \mathit{mtch}|l|op^n\ e_1 \ldots e_n|x^n\ e_1 \ldots e_n|\\ & x|\mathrm{M}_M.\mathrm{x}|\textbf{if}\ \ e_1\ \ \textbf{then}\ \ e_2\ \ \textbf{else}\ \ e_3|\textbf{while}\ \ e_1\ \ \textbf{do}\ \ e_2\ \ \textbf{done}|e_1\ \&\&\ e_2|e_1\ ||\ e_2|e_1\ \textbf{;}\ e_2|\\ & e_1\ e_2|!_T\,e|e_1\ \textbf{:=}_T\ e_2|\textbf{match}\ \ e\ \ \textbf{with}\ \ \mathit{mtch}|\textbf{let}\ \ p = e'\ \ \textbf{in}\ \ e''|\\ & \textbf{let}\ \ x : T\ p_1..p_n = e'\ \ \textbf{in}\ \ e''|\textbf{let}\ \ \textbf{rec}\ \ x : T = \textbf{function}\ \ \mathit{mtch}\ \ \textbf{in}\ \ e|\\ & \textbf{let}\ \ \textbf{rec}\ \ x : T\ p_1..p_n = e'\ \ \textbf{in}\ \ e''|\textbf{raise}\ \ e|\textbf{try}\ \ e\ \ \textbf{with}\ \ \mathit{mtch}|\\ & \Lambda\ t \to e|e\ \ T|\{T, e\}\ \textbf{as}\ \ T'|\textbf{let}\ \{t, x\} = e_1\ \ \textbf{in}\ \ e_2|\\ & \boxed{\textbf{marshal}\ e_1\ e_2\ :\ T|\textbf{unmarshal}\ \ e\ \textbf{as}\ T|}\\ & \boxed{\textbf{fresh}_T|\textbf{cfresh}_T|\textbf{hash}(X.\mathrm{x})_T|\textbf{hash}(T, e_2)_{T'}|\textbf{hash}(T, e_2, e_1)_{T'}|}\\ & \boxed{\textbf{swap}\ \ e_1\ \ \textbf{and}\ \ e_2\ \ \textbf{in}\ \ e_3|e_1\ \ \textbf{freshfor}\ \ e_2|\textbf{support}_T\,e|}\\ & \boxed{\mathrm{M}_M@\mathrm{x}|\textbf{name\_of\_tie}\ e|\textbf{val\_of\_tie}\ e|}\\ & \boxed{\textbf{namecase}\ \ e_1\ \ \textbf{with}\ \{t, (x_1, x_2)\}\ \textbf{when}\ \ x_1 = e \to e_2\ \ \textbf{otherwise}\ \to e_3|e_1|||e_2|}\\ & \boxed{\mathrm{n}_T|h.\mathrm{x}|e_1\ \textbf{:=}'_T\ e_2|\textbf{marshalz}\ \underline{s}\ e\ :\ T|\textbf{RET}_T|\textbf{SLOWRET}_T|\textbf{TERM}|}\\ & \boxed{\textbf{op}(op^n)^n\ e_1\ ..\ e_n|\textbf{op}(x^n)^n\ e_1\ ..\ e_n|[e]^T_{eqs}|}\\ & \boxed{\textbf{resolve}(\mathrm{M}_M.\mathrm{x}, \mathrm{M}'_{M'}, \mathit{resolvespec})|\textbf{resolve\_blocked}(\mathrm{M}_M.\mathrm{x}, \mathrm{M}'_{M'}, \mathit{resolvespec})}\end{aligned}$$

Operators

$$\begin{array}{lll} op & ::= & \mathbf{ref}_T|(=_T)|(<)|(\leq)|(>)|(\geq)|(+)|(-)|(*)|(/)|-|(@_T)|(\hat{})| \\ & & \mathbf{mod}|\mathbf{land}|\mathbf{lor}|\mathbf{lxor}|\mathbf{lsl}|\mathbf{lsr}|\mathbf{asr}| \\ & & \mathbf{create\_thread}_T|\mathbf{self}|\mathbf{kill}|\mathbf{create\_mutex}|\mathbf{lock}|\mathbf{try\_lock}|\mathbf{unlock}| \\ & & \mathbf{create\_cvar}|\mathbf{wait}|\mathbf{signal}|\mathbf{broadcast}|\mathbf{exit}\ _T| \\ & & \boxed{\mathbf{compare\_name}_T|\mathbf{thunkify}|}\ \boxed{\mathbf{unthunkify}} \end{array}$$

Matches and Patterns

$$\begin{array}{rll} mtch & ::= & p \to e|(p \to e|mtch) \\ p & ::= & (\_ : T)|(x : T)|\mathrm{C}_0|\mathrm{C}_1\ p|p_1 :: p_2|(p_1, .., p_n)|(p : T) \end{array}$$

Signatures and Structures

$$\begin{array}{ll} sig ::= \text{empty}\ |\mathbf{val}\ \ \mathrm{x}_x : T\ sig|\mathbf{type}\ \ \mathrm{t}_t : K\ sig & Sig ::= \mathbf{sig}\ \ sig\ \ \mathbf{end} \\ str ::= \text{empty}\ |\mathbf{let}\ \ \mathrm{x}_x : T\ p_1..p_n = e\ str|\mathbf{type}\ \ \mathrm{t}_t = T\ str & Str ::= \mathbf{struct}\ \ str\ \ \mathbf{end} \end{array}$$

Version and version constraint expressions

$$\begin{array}{lll|lll} avne & ::= & \underline{n}|\underline{N}|h|\mathbf{myname} & avce & ::= & ahvce|\underline{n} \\ vne & ::= & avne|avne.vne & dvce & ::= & avce|\underline{n}\text{–}\underline{n}'|\text{–}\underline{n}|\underline{n}\text{–}|*|avce.dvce \\ ahvce & ::= & \underline{N}|h|\mathrm{M}_M & vce & ::= & dvce|\mathbf{name} = ahvce \end{array}$$

Source definitions and Compilation Units

$$\begin{array}{rl} sourcedefinition ::= & \mathbf{module}\ mode\ \mathrm{M}_M : Sig\ \ \mathbf{version}\ \ vne = Str\ withspec \\ & \mathbf{import}\ mode\ \mathrm{M}_M : Sig\ \ \mathbf{version}\ \ vce\ likespec\ \ \mathbf{by}\ \ resolvespec = Mo \\ & \mathbf{mark}\ \ \mathrm{MK} \\ & \mathbf{module}\ \mathrm{M}_M : Sig = \mathrm{M'}_{M'} \\ \\ mode ::= & \mathbf{hash}|\mathbf{cfresh}|\mathbf{fresh}|\mathbf{hash!}|\mathbf{cfresh!} \\ withspec ::= & \text{empty}\ |\ \mathbf{with}\ !eqs \\ likespec ::= & \text{empty}\ |\ \mathbf{like}\ \ \mathrm{M}_M|\ \mathbf{like}\ \ Str \\ resolvespec ::= & \text{empty}\ | \\ & \textsc{Static\_Link}, resolvespec| \\ & \textsc{Here\_Already}, resolvespec| \\ & URI, resolvespec \\ Mo ::= & \mathrm{M}_M|\textsc{unlinked} \\ \\ compilationunit ::= & \text{empty}\ |e|sourcedefinition\ \mathbf{;;}\ compilationunit| \\ & \mathbf{includesource}\ \ sourcefilename\ \mathbf{;;}\ compilationunit| \\ & \mathbf{includecompiled}\ \ compiledfilename\ \mathbf{;;}\ compilationunit \end{array}$$

Compiled Definitions and Compiled Units

$$\begin{array}{rl} definition ::= & \mathbf{cmodule}...|\mathbf{cimport}...|\mathbf{module}\ \ \mathbf{fresh}...|\mathbf{import}\ \ \mathbf{fresh}...|\mathbf{mark}\ \ \mathrm{MK} \\ compiledunit ::= & \text{empty}\ |e|definition\ \mathbf{;;}\ compiledunit \end{array}$$

Marshalled value contents (marshalled values are strings that unmarshal to these)

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

Module names (hashes and abstract names)

$$\begin{array}{lll} h & ::= & \mathbf{hash}(\mathbf{hmodule}_{eqs}\ \mathrm{M} : Sig_0\ \ \mathbf{version}\ \ vne = Str)| \\ & & \mathbf{hash}(\mathbf{himport}\ \mathrm{M}\ : Sig_0\ \ \mathbf{version}\ \ vc\ \ \mathbf{like}\ \ Str)| \\ & & \mathrm{n} \\ X & ::= & \mathrm{M}_M | h \end{array}$$

Expression name values

$$\mathbf{n} \quad ::= \quad \mathrm{n}_T|\mathbf{hash}(h.\mathrm{x})_T|\mathbf{hash}(T', \underline{s})_T|\mathbf{hash}(T', \underline{s}, \mathbf{n})_T$$

(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)

$$eqs \quad ::= \quad \varnothing | eqs, X.\mathrm{t} \approx T$$

Type Environments (for identifiers and store locations — not required at run time in the implementation)

$$E \quad ::= \quad \text{empty}\ |E, x : T|E, l : T\ \ \mathsf{ref}|E, t : K|E, \mathrm{M}_M : Sig$$

Type Environments (for global names — not required in the implementation)

$$\begin{array}{l} E_\mathrm{n} ::= \text{empty}\ |E_\mathrm{n}, \mathrm{n} : \mathrm{TYPE}|E_\mathrm{n}, \mathrm{n} : T\ \ \mathsf{name}| \\ \qquad E_\mathrm{n}, \mathrm{n} : \mathbf{nmodule}_{eqs}\ \mathrm{M} : Sig_0\ \ \mathbf{version}\ \ vne = Str| \\ \qquad E_\mathrm{n}, \mathrm{n} : \mathbf{nimport}\ \mathrm{M} : Sig_0\ \ \mathbf{version}\ \ vc\ \ \mathbf{like}\ \ Str \end{array}$$

Processes

$$P \quad ::= \quad 0|(P_1|P_2)|\mathbf{n} : definitions\ e|\mathbf{n} : \mathrm{MX}(\underline{b})|\mathbf{n} : \mathrm{CV}$$

Single-Machine Configurations

$$config \quad ::= \quad E_\mathrm{n}\ \mathbf{;}\ \langle E_s,\ s,\ definitions,\ P\rangle$$

References

Acute team, The. *Acute*. `http://www.cl.cam.ac.uk/users/pes20/acute`.

Armstrong, J., Virding, R., Wikstrom, C., & Williams, M. (1996). *Concurrent Programming in Erlang*. Prentice Hall. 2nd ed.

Benton, N., Cardelli, L., & Fournet, C. (2002). Modern concurrency abstractions for C$^\sharp$. *Proc. ECOOP, LNCS 2374*.

Bierman, G., Hicks, M., Sewell, P., Stoyle, G., & Wansbrough, K. (2003). Dynamic rebinding for marshalling and update, with destruct-time λ. *Proc. ICFP*.

Billings, John. (2005). *A bytecode compiler for Acute*. Computer Science Tripos Part II Dissertation, University of Cambridge.

Billings, John, Sewell, Peter, Shinwell, Mark, & sa, Rok Strniš 2006 (Sept.). Type-safe distributed programming for OCaml. *Pages 20–31 of: Proc. ML'06, 2006 ACM SIGPLAN workshop on ML*.

Boudol, Gérard. (2003). *ULM: A core programming model for global computing*. Draft.

Buckley, Alex, Murray, Michelle, Eisenbach, Susan, & Drossopoulou, Sophia. 2005 (April). Flexible bytecode for linking in .NET. *Proc. BYTECODE 2005*.

Cardelli, L. (1995). A language with distributed scope. *Pages 286–297 of: Proc. 22nd POPL*.

Cardelli, L., & Gordon, A. D. (1998). Mobile ambients. *Proc. FOSSACS, LNCS 1378*.

Conchon, Sylvain, & Fessant, Fabrice Le. (1999). Jocaml: Mobile agents for Objective-Caml. *Pages 22–29 of: Proc. ASA/MA*.

Deniélou, Pierre-Malo, & Leifer, James J. (2006). Abstraction preservation and subtyping in distributed languages. *Proc. 11th ICFP*.

Dot03. (2003). *Packacking and deploying .Net framework applications (.Net framework tutorials, MSDN)*.

Drossopoulou, S., Eisenbach, S., & Wragg, D. (1999). A fragment calculus towards a model of separate compilation, linking and binary compatibility. *Proc. LICS*.

Fessant, Fabrice Le. (2001). Detecting distributed cycles of garbage in large-scale systems. *Proc. Principles of Distributed Computing (PODC)*.

Fournet, C., Gonthier, G., Lévy, J.-J., Maranget, L., & Rémy, D. (1996). A calculus of mobile agents. *Proc. 7th CONCUR, LNCS 1119*.

Furuse, Jun, & Weis, Pierre. (2000). Entrées/sorties de valeurs en Caml. *J. francophones des langages applicatifs*.

Grossman, D., Morrisett, G., & Zdancewic, S. (2000). Syntactic type abstraction. *ACM TOPLAS*, **22**(6), 1037–1080.

Harper, R., & Lillibridge, M. (1994). A type-theoretic approach to higher-order modules with sharing. *Proc. 21st POPL*.

Harper, R., & Pierce, B. C. (2005). *Design issues in advanced module systems*. Chapter in *Advanced Topics in Types and Programming Languages*, B. C. Pierce, editor.

Harper, R., & Stone, C. (2000). A type-theoretic interpretation of standard ML. *Proof, language and interaction: Essays in honour of Robin Milner*.

Hennessy, M., Rathke, J., & Yoshida, N. (2004). Safedpi: A language for controlling mobile code. *Proc. FOSSACS, LNCS 2987*.

Heydon, Allan, Levin, Roy, Mann, Timothy, & Yu, Yuan. (2006). *Software configuration management using Vesta*. Springer-Verlag.

Hosoya, Haruo, & Pierce, Benjamin C. 1999 (June). *How good is local type inference?* Tech. rept. MS-CIS-99-17. University of Pennsylvania.

Le Botlan, Didier, & Rémy, Didier. (2003). MLF: Raising ML to the power of System-F. *Pages 27–38 of: Proceedings of the International Conference on Functional Programming (ICFP 2003), Uppsala, Sweden*. ACM Press.

Lee, Daniel K., Crary, Karl, & Harper, Robert. 2007 (Jan.). Towards a mechanized metatheory of Standard ML. *Proc. 34th ACM SIGPLAN-SIGACT symposium on principles of programming languages*.

Leifer, J. J., Peskine, G., Sewell, P., & Wansbrough, K. (2003a). Global abstraction-safe marshalling with hash types. *Proc. 8th ICFP*.

Leifer, James J., Peskine, Gilles, Sewell, Peter, & Wansbrough, Keith. (2003b). *Global abstraction-safe marshalling with hash types*. Tech. rept. RR-4851. INRIA Rocquencourt. Available from `http://moscova.inria.fr/~leifer/research.html`. Also published as UCAM-CL-TR-569.

Leroy, X. (1994). Manifest types, modules, and separate compilation. *Proc. 21st POPL*.

Milner, R., Tofte, M., & Harper, R. (1990). *The definition of Standard ML*. MIT Press.

Murphy, T., Crary, K., Harper, R., & Pfenning, F. (2004). A symmetric modal lambda calculus for distributed computing. *Proc. LICS*.

Odersky, Martin, Zenger, Christoph, & Zenger, Matthias. (2001). Colored local type inference. *ACM SIGPLAN Notices*, **36**(3), 41–53.

Ohori, Atsushi, & Kato, Kazuhiko. (1993). Semantics for communication primitives in a polymorphic language. *Pages 99–112 of: Proc. POPL*.

Peskine, Gilles. (2007). *HOS: un calcul de modules adapté aux environnements répartis*. Ph.D. thesis. Forthcoming.

Pierce, B. C., & Turner, D. N. (2000). Pict: A programming language based on the pi-calculus. *Proof, language and interaction: Essays in honour of Robin Milner*.

Pierce, Benjamin C., & Turner, David N. (1998). Local type inference. *Proc. POPL*. Full version in *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(1), January 2000, pp. 1–44.

Rémy, Didier. (2002). Using, understanding, and unraveling the ocaml language. *Pages 413–537 of:* Barthe, Gilles (ed), *Applied Semantics. Advanced Lectures. LNCS 2395*.

Reppy, J. H. (1999). *Concurrent programming in ML*. Cambridge University Press.

Rossberg, A. 2003 (Aug.). Generativity and dynamic opacity for abstract types. *Proc. 5th PPDP*.

Rossberg, Andreas, Botlan, Didier Le, Tack, Guido, Brunklaus, Thorsten, & Smolka, Gert. (2006). *Alice through the looking glass*. Trends in Functional Programming, vol. 5. Munich, Germany: Intellect Books, Bristol, UK. Pages 79–96.

Sekiguchi, T., & Yonezawa, A. (1997). A calculus with code mobility. *Pages 21–36 of: Proc. 2nd FMOODS*.

Sewell, P. (2001). Modules, abstract types, and distributed versioning. *Proc. 28th POPL*.

Sewell, P., Wojciechowski, P. T., & Pierce, B. C. (1999). Location-independent communication for mobile agents: a two-level architecture. *Pages 1–31 of: Internet programming languages, LNCS 1686*.

Sewell, P., Leifer, J. J., Wansbrough, K., Allen-Williams, M., Z. Nardelli, Francesco, Habouzit, P., & Vafeiadis, V. 2004 (Oct.). *Acute: High-level programming language design for distributed computation. Design rationale and language definition*. Tech. rept. 605. University of Cambridge Computer Laboratory. Also published as INRIA RR-5329. 193pp.

Sewell, P., Leifer, J. J., Wansbrough, K., Allen-Williams, M., Zappa Nardelli, F., Habouzit, P., & Vafeiadis, V. 2005a (Jan.). *Source release of the Acute system*. Available from `http://www.cl.cam.ac.uk/users/pes20/acute/`.

Sewell, Peter. 2000 (Aug.). *Applied π – a brief tutorial*. Tech. rept. 498. Computer Laboratory, University of Cambridge. An extract appeared as Chapter 9, Formal Methods for Distributed Processing, A Survey of Object Oriented Approaches.

Sewell, Peter, Leifer, James J., Wansbrough, Keith, Zappa Nardelli, Francesco, Allen-Williams, Mair, Habouzit, Pierre, & Vafeiadis, Viktor. 2005b (Sept.). Acute: High-level programming language design for distributed computation. *Proceedings of ICFP 2005: International Conference on Functional Programming (Tallinn)*.

Shinwell, M. R. (2005). *The Fresh Approach: functional programming with names and binders*. Tech. rept. UCAM-CL-TR-618. University of Cambridge, Computer Laboratory.

Shinwell, M. R., Pitts, A. M., & Gabbay, M. J. (2003). FreshML: Programming with binders made simple. *Pages 263–274 of: Proc. 8th ICFP*.

Stoyle, Gareth. (2006). *A theory of dynamic software updates*. Ph.D. thesis, University of Cambridge.

Thomsen, B., Leth, L., & Kuo, T.-M. (1996). A Facile tutorial. *CONCUR'96, LNCS 1119*.

Unyapoth, A., & Sewell, P. 2001 (Jan.). Nomadic Pict: Correct communication infrastructure for mobile computation. *Pages 116–127 of: Proc. POPL*.

Weirich, Stephanie. 2002 (Aug.). *Programming with types*. Ph.D. thesis, Cornell University.