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

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

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.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Languages, Design

Keywords programming languages, distributed programming, marshalling, serialisation, abstract types, modules, rebinding, version control, type theory, ML

ICFP'05 September 26–28, 2005, Tallinn, Estonia.

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 [JoC], Nomadic Pict [SWP99], and Ambient primitives [CG98], 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.

Acute has a full definition [SLW$^+$04], covering syntax, typing, compilation, and operational semantics. A prototype implementation is also available [SLW$^+$05], which is efficient enough to run moderate examples but which closely mirrors the structure of the operational semantics. This paper is devoted to an informal presentation of the main design points, with small but executable examples. For details of the many semantic subtleties, and for discussion of further design points, we refer the reader to [SLW$^+$04].

1.1 Acute overview: the main design points

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 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 runtime 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 *runtime 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 [Shi05, SPG03] 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 `module`s and `import`s 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 `module`s and `import`s. 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.

Acute is not intended as a proposal for a full-scale 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 a large part of what is needed to bring the benefits of ML-like languages to the programming of large-scale distributed systems.

1.2 Semantics, Implementation, and Examples

Most of the Acute grammar is standard, a fragment of OCaml. For concreteness we summarise the new constructs in Figure 1. The remainder of the paper explains the main aspects of their meaning and usage (not all details of the grammar will be covered).

Semantics The Acute 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. Module initialisation can involve arbitrary computation. The core 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 (the implementation does some ad-hoc partial inference).

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

Our semantics for rebinding rests on the *redex-time* evaluation strategy, introduced in [BHS$^+$03] 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 [GMZ00] to the entire Acute language (except, to date, the System F constructs). This is technically delicate (and not needed for implementations, which can erase all brackets) but provides useful clarity in a setting where abstraction boundaries may be complex, with abstract types shared between programs.

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

Implementation 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 300 times slower than OCaml bytecode.

T	$::=$	$\mathsf{int} \mid \mathsf{bool} \mid \mathsf{string} \mid \mathsf{unit} \mid \mathsf{char} \mid \mathsf{void} \mid T_1 * .. * T_n \mid T_1 + .. + T_n \mid T \to T' \mid T\ \ \mathsf{list} \mid T\ \ \mathsf{option} \mid T\ \ \mathsf{ref} \mid \mathsf{exn} \mid \mathrm{M}_M.\mathrm{t} \mid$ $t \mid \forall\, t.T \mid \exists\, t.T \mid T\ \ \mathsf{name} \mid T\ \ \mathsf{tie} \mid \mathsf{thread} \mid \mathsf{mutex} \mid \mathsf{cvar} \mid \mathsf{thunkifymode} \mid \mathsf{thunkkey} \mid \mathsf{thunklet} \mid h.\mathrm{t} \mid \mathrm{n}$	
e	$::=$	$\ldots \mid \mathbf{marshal}\ e_1\ e_2\ :\ T \mid \mathbf{unmarshal}\ \ e\ \mathbf{as}\ T \mid$	marshalling
		$\mathbf{fresh}_T \mid \mathbf{cfresh}_T \mid \mathbf{hash}(\mathrm{M}_M.\mathrm{x})_T \mid \mathbf{hash}(T, e_2)_{T'} \mid \mathbf{hash}(T, e_2, e_1)_{T'} \mid \mathrm{n}_T \mid h.\mathrm{x}$	name creation and names
		$\mathbf{namecase}\ \ e_1\ \ \mathbf{with}\ \{t,(x_1,x_2)\}\ \mathbf{when}\ \ x_1 = e \to e_2\ \ \mathbf{otherwise}\ \to e_3 \mid$	namecase
		$\mathbf{create\_thread} \mid \ldots \mid \mathbf{thunkify} \mid [e]^{T}_{eqs} \mid \ldots$	threads, thunkify, coloured brackets

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

Figure 1. Acute syntax: the full type grammar and the main non-standard expression and module forms. Here h is a module name, hash- or fresh-generated, n is a freshly-generated name, t is a module type field external identifier, M_M is a module external/internal identifier pair, and MK is a string constant. The type subscripts are typically inferred. The highlighted forms are only in the semantics, not source programs.

The definition is too large (on the scale of the ML definition 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.

Examples (§10) 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 [SWP99] primitives for communication and mobility, and an implementation of the Ambient primitives [CG98]. These require and use most of the new features.

Relationship to previous work (§11, §12) Acute builds on previous work, in which we introduced new-bound type names for abstract types [Sew01], hash-generated type names [LPSW03], and controlled dynamic rebinding in a lambda-calculus [BHS^{+}03], all in simple variants for for small calculi.

Our contribution here 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.

Other related work is discussed in §11, and we conclude in §12.

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 [TLK96], Obliq [Car95], and JoCaml [JoC]), we believe that *a general-purpose distributed programming language should not have a built-in commitment to any particular means of interaction.*

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

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

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

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

3. Basic type-safe distributed interaction

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

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

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

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

Compiling the two programs and then executing them in parallel results in the second printing 8. This and subsequent examples are executable Acute code. For brevity they use a simple address-less `IO` library, providing primitives `send:string->unit` and `receive:unit->string`[1]. Below we write the parallel exe-

[1] To emphasise that interaction might be via communication or via persistent store, there are two implementations of `IO`, one using TCP and one using file IO; either can be used for these examples.

cution 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. 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. Here 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.[2]

In Acute we focus on the trusted case, with option (a), and the problems of distributed typing, naming, and rebinding it raises. A full language should also support the untrusted case, e.g. with marshalling to ASN.1 or XML, and type- or proof-carrying code for marshalled functions.

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.

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; there is usually no way for an implementation to infer the correct behaviour. We adapt the mechanism proposed in [BHS+03] (from a lambda-calculus setting to an ML-style module language) to indicate which resources should be rebound and which shipped for any marshal operation. An Acute program consists roughly of a sequence of module definitions, interspersed with *marks*, followed by running processes; those module definitions, together with implicit module definitions for standard libraries, are the resources. Marks essentially name the sequence of module definitions preceding them. Marshal operations are each with respect to a mark; the modules below that mark are shipped and references to modules above that mark are rebound, to whatever local definitions may be present at the receiver. The mark `"StdLib"` used in §3 is declared at the end of the standard library; this mark and library are in scope in all examples.

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

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

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

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

[2] 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.

```
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 * overrides the default behaviour, which would constrain rebinding only to identical copies of M1. Marks are simply string constants, not binders subject to alpha equivalence, as they need to be dynamically rebound. For example, if one marshals a function that has an embedded marshal with respect to "StdLib", and then unmarshals and executes it elsewere, one typically wants the embedded marshal to act with respect to the now-local "StdLib".

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

A language with rebinding cannot use a standard call-by-value operational semantics, which substitutes out identifier definitions as it comes to them, as some definitions may need to be rebound later. Two alternative CBV reduction strategies were developed in [BHS$^+$03] in a simple lambda-calculus setting: *redex-time*, in which one instantiates an identifier with its value only when the identifier occurs in redex-position, and *destruct-time* where instantiation occurs even later, when the identifier appears in a context which needs to destruct the outermost structure of the value. 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 runtime. For example:

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

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

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

is roughly the program below.

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

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

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

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

4.5 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.

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.

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 [Ler94] and Harper et al [HL94] do).

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

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

Case 1 For a module such as EvenCounter above that is effect-free (i.e. evaluation of the structure body involves no effects) we can use module *hashes* as global names for abstract types, generalising our earlier work [LPSW03] 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 [Sew01], enables interoperation between programs linked against the same compiled module, while forbidding interoperation between different builds.

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

We let the programmer specify which of the three behaviours is required with a `hash`, `fresh`, or `cfresh` mode in the module definition, writing e.g. `module hash EvenCounter`. In general it would be abstraction-breaking to specify `hash` or `cfresh` for an effect-full module. To prevent this requires some kind of effect analysis, for which we use coarse but simple notions of *valuability*, following [HS00], and of *compile-time valuability*. The mode defaults to the most liberal possible if omitted, and `hash!` and `cfresh!` modes allow valuability to be overridden where necessary.

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

6. Naming: expression names

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

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

T `name`

of names associated with it. Values of these types (like type names) can be generated freshly at runtime, freshly at compile-time, or deterministically by hashing, with expression forms `fresh`, `cfresh`, `hash(M.x)`, `hash(`T,e`)`, and `hash(`T,e,e`)`. We detail these forms below, showing how they support several important scenarios. In each, the basic question is how one establishes a name shared between sender and receiver code such that testing equality of the name ensures the type correctness of communicated values (and hence that there will be no unmarshal failures in the communication library).

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.[3]

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 runtime 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`[4], 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

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

[4] The `%[int]` is an explicit type application, and later `%[]` are placeholders for inferred types.

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

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

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

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

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

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

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

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

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

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

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

The `hash(N.f)` gives a T `name` where T = `int->unit` is the type of `N.f`; the surrounding hash coercion `hash(int,"",_)` constructs an `int name` from this.[5] 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.

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. 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 re-

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

quire 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.

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.

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 [HP05]: in any fully-linked executable there is just a single such `M`, though (with separate compilation) it may be unknown at compile-time which module definition for `M` it will be linked to. In contrast, the possibility of rebinding makes some references *indefinite* — during a single distributed execution, they may be bound to different modules.

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

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

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

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

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

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

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

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

In the case above this hash would be roughly

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

and where one imports a module with an abstract type field

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

the hash h =

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

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

In the `EvenCounter` example, the imported module had no abstract type fields. 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

With changing versions, sometimes one must allow new code to see through the abstraction boundaries of earlier abstract types, either to make new types compatible with old (if their invariants are essentially the same) or to express conversion functions. In [Sew01, LPSW03] we proposed a *strong coercion* `with!` to do this, and Acute includes a variant thereof. By analogy with ML sharing specifications, we allow a module definition to have a *withspec*, a list of equalities between abstract types and representation types from modules constructed earlier (often this will be of previous builds of the same module). 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.

8.3 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 Acute idiom for remote invocation simply makes use of its dynamic rebinding facilities, e.g. by marshalling a thunk mentioning an identifier `N.f`. This involves dynamic subsignature and version checks — much more costly than name equality, but also much more flexible.

9. Mobility, `thunkify`, and local concurrency

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, e.g. to move services to replacement hardware. 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.

Distributed programming also requires support for local concurrency, with threads and constructs for interaction between them. In large programs we expect both shared-memory and message-passing interaction to be required. In Acute we initially provide shared-memory interaction between language-level threads, as in OCaml: 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. (Some forms of message passing, e.g. Join patterns with their multi-way binding construct, would need direct language support.)

Thunkification for a single thread would be close to `call/cc`, but in the concurrent setting there are many possible forms: asynchronous or synchronous, with different atomicity, and with different interactions with naming, blocking system calls, and module initialisation. The choices and our rationale are discussed in [SLW+04]; here we note only that we have an asynchronous `thunkify` that can atomically collect a group of named threads, mutexes, and condition variables. It is a dangerous operation, as one might for example separate a thread from a mutex on which it is blocked, but this seems to be inescapable, arising ultimately from the possibility of disconnection between subcomputations. We expect it to be used to implement libraries that simultaneously provide computation mobility (or checkpointing) and safe distributed interaction mechanisms.

10. 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 §8.3. 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.

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.

11. 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 [Sew01, LPSW03, BHS+03], 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[6], or on version change.

Early work on adding local concurrency to ML resulted in Concurrent ML [Rep99] and the initial Facile, both based on the SML/NJ implementation. Facile was later extended with rich support for distributed execution, including a notion of *location* and computation mobility [TLK96]. Erlang [AVWW96] supports concurrency, messaging and distribution, but without static typing.

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

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

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

Work on Alice [BRS+05, Ros03] 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 supports type-safe, but not abstraction-safe, marshalling of non-functional values in OCaml [FW00].

Both Java and .NET have some versioning support, though neither is integrated with the type system. Java serialisation, used in RMI, includes *serialVersionUID*s for classes of any serialised objects. These default to (roughly) hashes of the method names and types, not including the implementation. Class authors can override them with hashes of previous versions. Linking for Java, and in particular binary compatibility, has been studied by Drossopoulou et al. [DEW99]. The .NET framework supports versioning of *assemblies* [Dot03]. Sharable assemblies must have *strong names*, which include a public key, file hashes, and a *major.minor.build.revision* version. Compile-time assembly references can be modified before use by XML policy files of the application, code publisher, and machine administrator; the semantics is complex [BMED05].

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

As discussed in §3 Acute addresses the case in which complex values must be communicated and the interacting runtimes are not malicious. Much other work applies to the untrusted case, with various forms of proof-carrying code and wire-format ASN.1 and XML typing.

12. Conclusions and future work

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

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

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

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

More fundamentally: we must study more refined low-level linking, for negotiation and for access control (revisiting the linear mark/module structure); subtyping is needed for many version-change scenarios, perhaps with corresponding subhash relations; and the constructs we have presented should be integrated with support for untrusted interaction. Expressing libraries of distributed references with distributed garbage collection is also a challenge. This combination 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, and Alisdair Wren provided helpful comments.

References

[AVWW96] J. Armstrong, R. Virding, C. Wikstrom, and M. Williams. *Concurrent Programming in Erlang*. Prentice Hall, 1996. 2nd ed.

[BCF02] N. Benton, L. Cardelli, and C. Fournet. Modern concurrency abstractions for C$\sharp$. In *Proc. ECOOP, LNCS 2374*, 2002.

[BHS+03] G. Bierman, M. Hicks, P. Sewell, G. Stoyle, and K. Wansbrough. Dynamic rebinding for marshalling and update, with destruct-time λ. In *Proc. ICFP*, 2003.

[BMED05] Alex Buckley, Michelle Murray, Susan Eisenbach, and Sophia Drossopoulou. Flexible bytecode for linking in .NET. In *Proc. BYTECODE 2005*, April 2005.

[BRS+05] D. Le Botlan, A. Rossberg, C. Schulte, G. Smolka, and G. Tack, 2005. `www.ps.uni-sb.de/alice/`.

[Car95] L. Cardelli. A language with distributed scope. In *Proc. 22nd POPL*, pages 286–297, 1995.

[CG98] L. Cardelli and A. D. Gordon. Mobile ambients. In *Proc. FoSSaCS, LNCS 1378*, 1998.

[DEW99] S. Drossopoulou, S. Eisenbach, and D. Wragg. A fragment calculus towards a model of separate compilation, linking and binary compatibility. In *Proc. LICS*, 1999.

[Dot03] Packacking and deploying .Net framework applications (.Net framework tutorials, msdn), 2003.

[FGL+96] C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, and D. Rémy. A calculus of mobile agents. In *Proc. 7th CONCUR, LNCS 1119*, 1996.

[FW00] Jun Furuse and Pierre Weis. Entrées/sorties de valeurs en Caml. In *J. Francophones des Langages Applicatifs*, 2000.

[GMZ00] D. Grossman, G. Morrisett, and S. Zdancewic. Syntactic type abstraction. *ACM TOPLAS*, 22(6):1037–1080, 2000.

[HL94] R. Harper and M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Proc. 21st POPL*, 1994.

[HP05] R. Harper and B. C. Pierce. Design issues in advanced module systems, 2005. Chapter in *Advanced Topics in Types and Programming Languages*, B. C. Pierce, editor.

[HRY04] M. Hennessy, J. Rathke, and N. Yoshida. Safedpi: A language for controlling mobile code. In *Proc. FOSSACS, LNCS 2987*, 2004.

[HS00] R. Harper and C. Stone. A type-theoretic interpretation of standard ML. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*. 2000.

[JoC] JoCaml. `http://pauillac.inria.fr/jocaml/`.

[Ler94] X. Leroy. Manifest types, modules, and separate compilation. In *Proc. 21st POPL*, 1994.

[LPSW03] J. J. Leifer, G. Peskine, P. Sewell, and K. Wansbrough. Global abstraction-safe marshalling with hash types. In *Proc. 8th ICFP*, 2003.

[MCHP04] T. Murphy, K. Crary, R. Harper, and F. Pfenning. A symmetric modal lambda calculus for distributed computing. In *Proc. LICS*, 2004.

[PT00] B. C. Pierce and D. N. Turner. Pict: A programming language based on the pi-calculus. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*. 2000.

[Rep99] J. H. Reppy. *Concurrent Programming in ML*. Cambridge Univ Press, 1999.

[Ros03] A. Rossberg. Generativity and dynamic opacity for abstract types. In *Proc. 5th PPDP*, August 2003.

[Sew01] P. Sewell. Modules, abstract types, and distributed versioning. In *Proc. 28th POPL*, 2001.

[Shi05] M. R. Shinwell. The Fresh Approach: functional programming with names and binders. Technical Report UCAM-CL-TR-618, University of Cambridge, Computer Laboratory, 2005.

[SLW+04] P. Sewell, J. J. Leifer, K. Wansbrough, M. Allen-Williams, Francesco Z. Nardelli, P. Habouzit, and V. Vafeiadis. Acute: High-level programming language design for distributed computation. design rationale and language definition. Technical Report 605, University of Cambridge Computer Laboratory, October 2004. Also published as INRIA RR-5329. 193pp.

[SLW+05] P. Sewell, J. J. Leifer, K. Wansbrough, M. Allen-Williams, F. Zappa Nardelli, P. Habouzit, and V. Vafeiadis. Source release of the Acute system, January 2005. Available from `http://www.cl.cam.ac.uk/users/pes20/acute/`.

[SPG03] M. R. Shinwell, A. M. Pitts, and M. J. Gabbay. FreshML: Programming with binders made simple. In *Proc. 8th ICFP*, pages 263–274, 2003.

[SWP99] P. Sewell, P. T. Wojciechowski, and B. C. Pierce. Location-independent communication for mobile agents: a two-level architecture. In *Internet Programming Languages, LNCS 1686*, pages 1–31, 1999.

[TLK96] B. Thomsen, L. Leth, and T.-M. Kuo. A Facile tutorial. In *CONCUR'96, LNCS 1119*, 1996.

[US01] A. Unyapoth and P. Sewell. Nomadic Pict: Correct communication infrastructure for mobile computation. In *Proc. POPL*, pages 116–127, January 2001.

[Ves] Vesta. `http://www.vestasys.org/`.