

# Acute: high-level programming language design for distributed computation

Peter Sewell\* James J. Leifer<sup>†</sup> Keith Wansbrough\* Mair Allen-Williams\*  
Francesco Zappa Nardelli<sup>†,\*</sup> Pierre Habouzit<sup>†</sup> Viktor Vafeiadis\*

\*University of Cambridge <sup>†</sup>INRIA Rocquencourt

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

## Abstract

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

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

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

## 1 Introduction

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

the intrinsic concurrency and nondeterminism of distributed systems, and the complexity of the underlying network layers, makes them particularly hard to understand and debug.

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

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

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

This paper is devoted to an informal presentation of the main design points, omitting details of the semantics. It is supported, however, by a full definition of Acute [Acu], covering syntax, typing, compilation, and operational semantics, and by a prototype implementation, efficient enough to run moderate examples while remaining close to the semantics. Both have been essential: the synthesis of the various features has involved many semantic subtleties. The definition is too large (on the scale of the ML definition rather than an idealised  $\lambda$ -calculus) to make proofs of the properties feasible with the available resources and tools. To increase confidence in both semantics and implementation, therefore, our

implementation can optionally type-check the entire configuration after each reduction step.

**Outline** The paper is structured as follows, with §2–9 discussing the main design points.

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

§4: We introduce *dynamic linking* and *rebinding* to local resources in the setting of a language with an ML-like second-class module system. There are many questions here: of how to specify which resources should be shipped with a marshalled value and which dynamically rebound; what evaluation strategy to use; when rebinding takes effect; and what to rebound to. In this Section our aim is to expose the design choices rather than identify definitive solutions. It is a necessary preliminary to our work in §§5–10. For Acute we make interim choices, reasonably simple and sufficient to bring out the typing and versioning issues involved in rebinding, which here is at the granularity of module identifiers. A running Acute program consists roughly of a sequence of module definitions (of ML structures), imports of modules with specified signatures, which may or may not be linked, and marks which indicate where rebinding can take effect; together with running processes and a shared store.

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

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

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

code consumers or producers have control.

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

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

§10: We demonstrate that Acute does indeed support typeful distributed programs with an example distributed communication infrastructure library.

Finally, in §11 and §12 we describe related work and conclude. An optional appendix summarises most of the Acute syntax.

**Semantics and Implementation** The definition of compilation describes how global type- and expression-level names are constructed. Unusually, the semantics preserves the module structure throughout computation, instead of substituting it away; this is needed to express rebinding. Abstraction boundaries are also preserved, with a generalisation of the *coloured brackets* of Grossman et al [GMZ00] to the entire Acute language. This is technically delicate (and brackets can be erased in implementations) but provides useful clarity in a setting where abstraction boundaries may be complex, with abstract types shared between programs.

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

The Acute implementation is written in FreshOCaml, as a meta-experiment in using the Fresh features for a medium-scale program (some 20 000 lines). It is a prototype: designed to be efficient enough to run moderate examples while remaining rather close to the semantics. The runtime interprets an intermediate language which is essentially the abstract syntax extended with closures.

**Syntax** For concreteness we summarise the most interesting constructs of Acute for types, expressions, and definitions. (It includes also the standard types  $T \rightarrow T' \mid T \text{ ref} \mid \text{exn} \mid M_M.t \mid t \mid \forall t. T \mid \exists t. T$  and others.) The **highlighted** forms do not occur in source programs.

Here  $h$  is a module name, hash- or freshly-generated;  $n$  is a freshly-generated name, and  $[e]_{eqs}^T$  is a coloured bracket. The other constructs are explained later.

```

T ::= ... | T name | thread | h.t | n

e ::= ... | marshal e1 e2 : T | unmarshal e as T |
freshT | cfreshT | hash(MM.x)T | hash(T, e2)T' |
swap e1 and e2 in e3 | supportT e | [e]Teqs
```

```

sourcedefinition ::=
module mode MM : Sig version vne = Str withspec |
import mode MM : Sig version vce likespec
    by resolvespec = Mo |
mark MK
```

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

## 2 Distributed abstractions: language vs libraries

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

The reason for this is essentially the complexity of the distributed environment: system designers must deal with partial failure, attack, and mobility — of code, of devices, and of running computations. This complexity demands a great variety of communication and persistent store abstractions, with varying performance, security, and robustness properties. At one extreme there are systems with tightly-coupled computation over a reliable network in a single trust domain. Here it might be appropriate to use a distributed shared memory abstraction, implemented above

TCP. At another extreme, interaction may be intrinsically asynchronous between mutually-untrusting runtimes, e.g. with cryptographic certificates communicated via portable persistent storage devices (smartcards or memory sticks), between machines that have no network connection. In between, there are systems that require asynchronous messaging or RMI but, depending on the network firewall structure, tunnel this over a variety of network protocols.

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

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

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

## 3 Basic type-safe distributed interaction

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

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

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

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

Compiling the two programs and then executing them in parallel results in the second printing 8. This and most subsequent examples are executable Acute code. For brevity they use a simple address-less IO library, providing communication primitives `send:string->unit` and `receive:unit->string`. (There are two implementations of IO, using TCP and file IO.) Below we write the parallel execution of the two separately-built programs p1 and p2 separated by a —.

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

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

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

should raise an exception. Allowing interaction via an

untyped medium inevitably means that some dynamic errors are possible, but they should be restricted to clearly-identifiable program points, and detected as early as possible. Here we should do that type check at unmarshal-time. (In some scenarios one may be able to exclude such errors at compile-time, e.g. when communicating on a typed channel; we return to this in §6.)

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

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

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

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

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

We do not discuss the design of the concrete wire format for marshalled values — the Acute semantics presupposes just a partial `raw_unmarshal` function from strings to abstract syntax of configurations, including definitions and store fragments; the prototype implementation simply uses canonical pretty-prints of abstract syntax.

## 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

<sup>1</sup>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.

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

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

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

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

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

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

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

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

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

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

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

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

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

An import introduces a module identifier (the left M1) with a signature; it may or may not be linked to an earlier module or import (this one is, to the earlier M1). The *version \** overrides the default behaviour, which would constrain rebinding only to identical copies of M1. Marks are simply string constants, not binders subject to alpha equivalence, as they need to be dynamically rebound. For example, if one marshals a function that has an embedded marshal with respect to "StdLib", and then unmarshals and executes it elsewhere, 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<sup>+</sup>03] in a simple lambda-calculus setting: *redex-time*, in which one instantiates an identifier with its value only when the identifier occurs in redex-position, and *destruct-time* where instantiation may occur even later. Here, to make the semantics as intuitive as possible, we use the redex-time strategy for module references (local expression reduction remains standard CBV).

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

```

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

```

#### 4.4 The structure of marks and modules

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

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

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

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

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

#### 4.5 Controlling when rebinding happens

We have to choose whether or not to allow execution of partial programs, which are those in which some imports are not linked to any earlier module definition (or import). Partial

programs can arise in two ways. First, they can be written as such, as in conventional programs that use dynamic linking, where a library is omitted from the statically-linked code, to be discovered and loaded at runtime. For example:

```

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

```

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

```

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

```

is roughly the program below.

```

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

```

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

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

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

#### 4.6 Controlling what to rebind to

*How* to look for such an *M* is specified by a *resolvespec* that can (optionally) be included in the import. By default it will be looked for just in the running program, in the sequence of modules defined above the import. Sometimes, though, one may wish to search in the local filesystem (e.g. for conventional shared-object names such as `libc.so.6`), or even at a web URI. In Acute we make an ad-hoc choice of a simple *resolvespec* language: a *resolvespec* is a finite list of

*atomic resolvespecs*, each of which is either `Static_Link`, `Here_Already` or a URI. Lookup attempts proceed down the list, with `Static_Link` indicating the import should already be linked, `Here_Already` prompting a search for a suitable module (with the right name, signature and version) in the running program, and a URI prompting a file to be fetched and examined for the presence of a suitable module.

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

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

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

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

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

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

## 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
  type t = int
end
```

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

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

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

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

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

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

```
hash(
  module EvenCounter
  : sig
    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
)
```

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

```

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

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

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

We let the programmer specify which of the three behaviours is required with a `hash`, `fresh`, or `cfresh` keyword in the module definition, writing e.g. `module hash EvenCounter`. It would be unsound (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 `hash`, `fresh`, or `cfresh` annotation defaults to the earliest possible if omitted.

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

## 6 Naming: expression names

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

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

```
T name
```

of names associated with it. Values of these types (like type names) can be generated freshly at runtime, freshly at compile-time, or deterministically by hashing, with expression forms `fresh`, `cfresh`, `hash(M.x)`, and `hash(T, e, e)`. We detail the different 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. 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 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  $T$ .<sup>2</sup>

**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

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

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

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

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

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

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

and used as below.

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

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

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

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

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

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

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

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

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

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

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

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

```
module hash N : sig      val f : int -> unit      end
      = struct let f = fun x->IO.print_int x+1      end
```

```
module 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  $T = \text{int} \rightarrow \text{unit}$  is the type of `N.f`; the surrounding hash coercion `hash(int,"",-)` constructs an `int` name from this.<sup>3</sup> 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  $\forall \vec{t}. T_1 \text{ name} \rightarrow T_2 \text{ name}$ .

**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.

<sup>2</sup>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`.

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

```

let c = hash(int,"foo")
DChan.send %[int] (c,v)
—
let c = hash(int,"foo")
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 sets of versions of components interoperating. Typechecking can provide some basic sanity guarantees, but cannot capture these semantic differences.

One alternative is to permit rebinding only to identical copies of modules that the code was initially linked to. Usually, though, more flexibility will be required — to permit rebinding to modules with “small” or “backwards-compatible” changes to their semantics, and to pick up intentionally location-dependent modules. It is impractical to specify the semantics that one depends upon in interfaces (in general, theorem proving would be required at link time, though there are intermediate behavioural type systems). We therefore 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 *resolvespecs*, there is a tension between allowing arbitrary computation in defining the relations and supporting compile-time analysis. Ultimately, it seems desirable to make the basic module primitives parametric on abstract types of versions and constraints — in a

particular distributed code environment, one may want a particular local choice for the languages. For Acute once again we choose not the most general alternative, but instead one which should be expressive enough for many examples, and which exposes some key design points.

**Scenario 1** It is common to use version numbers which are supplied by the programmer, with no checked relationship to the code. As an arbitrary starting point, we take version numbers to be nonempty lists of natural numbers, and version constraints to be similar lists possibly ending in a wildcard *\** or an interval; satisfaction is what one would expect, with a *\** matching any (possibly empty) suffix. 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, per-

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

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

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

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

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

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

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

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

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

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

In the case above this hash would be roughly

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

and where one imports a module with an abstract type field

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

the hash `h =`

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

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

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

At first sight this is quite unpleasant, requiring the importers of a module to ‘know’ representation types which

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

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

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

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

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

On the other hand, we choose not to include *resolvespecs* 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 representations of 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. 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), v))
—
module N : sig val f:int->unit          end
  = struct let f=fun x-> IO.print_int x+1 end
  mark "MARK-N"
  let (g,v)=unmarshal (IO.receive ()) in g () v
```

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.

## 9 Concurrency, mobility, and `thunkify`

Distributed programming 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, 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, which also need typing support.

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

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

There are many possible forms of thunkification: asynchronous or synchronous and with different atomicity and interactions with naming, blocking system calls, and module initialisation. The choices and our rationale are discussed in [Acu]; 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

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

`Tcp_connection_management` maintains TCP connections to TCP addresses (IP address/port pairs), creating them

on demand. `Tcp_string_messaging` uses that to provide asynchronous messaging of strings to TCP addresses. These are both hash modules, with abstract types of handles; they spawn daemons to deal with incoming communications.

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

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

Clients of this libraries can use any of the various ways of creating shared typed names discussed in §6 and §8.3. 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`.

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

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

## 11 Related work

Acute builds on our earlier work: compile-time fresh generation of abstract type names and channel names [Sew01]; hash-generation of effect-free abstract type names [LPSW03]; and dynamic rebinding [BHS<sup>+</sup>03]. There is ex-

tensive related work on module systems, dynamic binding, dynamic type tests, and distributed process calculi. For most of this we refer the reader to the discussion in those papers, confining our attention here to some of the most relevant distributed programming language developments.

Early work on adding local concurrency to ML resulted in Concurrent ML [Rep99] and the initial Facile, both based on the SML/NJ implementation. Facile was later extended with rich support for distributed execution, including a notion of *location* and computation mobility [TLK96]. 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  $\pi$ -calculus core instead of primitive functions or objects. The Distributed Join Calculus [FGL<sup>+</sup>96] and subsequent JoCaml implementation [JoC] modified the  $\pi$  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<sup>#</sup> [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  $D\pi$  [HRY04] and with modal types [MCHP04]. Even where this is possible, however, programmers must still deal with low-level network failure.

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

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

<sup>4</sup>Several, including JoCaml and Nomadic Pict, have ad-hoc 'traders' for establishing initial connections between programs.

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

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

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

## 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.

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, when it is then dereferenced. Since field references inside a thunk remain as pointers, they could easily be rebound with only modest changes to the run-time. Of course compile-time inlining optimisations between parts of code separated by a mark would no longer be possible.

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

More fundamentally: we must study more refined low-level linking, for negotiation and for access control (escaping the linear mark/module structure); 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 support from a Royal Society University Research Fellowship (Sewell), a St Catharine's College Heller Research Fellowship (Wansbrough), EPSRC grant GRN24872, EC FET-GC project IST-2001-33234 PEPITO, and APPSEM 2. We thank Vilhelm Sjöberg and Christian Steinrücken for their implementation work on predecessors of Acute and Gilles Peskine for discussions on references and coloured brackets. Andrew Appel, Matthew Fairbairn, Jean-Jacques Lévy, Luc Maranget, Gilles Peskine, and Alisdair Wren provided comments.

## References

- [Acu] The Acute team. Acute. <http://www.cl.cam.ac.uk/users/pes20/acute>.
- [Ali03] The Alice project, 2003. <http://www.ps.uni-sb.de/alice/>.
- [AVWW96] J. Armstrong, R. Virding, C. Wikstrom, and M. Williams. *Concurrent Programming in Erlang*. Prentice Hall, 1996. 2nd ed.
- [BCF02] N. Benton, L. Cardelli, and C. Fournet. Modern concurrency abstractions for  $C^\sharp$ . In *Proc. ECOOP, LNCS 2374*, 2002.
- [BHS<sup>+</sup>03] G. Bierman, M. Hicks, P. Sewell, G. Stoye, and K. Wansbrough. Dynamic rebinding for marshalling and update, with destruct-time  $\lambda$ . In *Proc. ICFP*, 2003.
- [Car95] L. Cardelli. A language with distributed scope. In *Proc. 22nd POPL*, pages 286–297, 1995.
- [DEW99] S. Drossopoulou, S. Eisenbach, and D. Wragg. A fragment calculus towards a model of separate compilation, linking and binary compatibility. In *Proc. LICS*, pages 147–156, 1999.
- [dot03] Packacking and deploying .net framework applications (.net framework tutorials), 2003. <http://msdn/microsoft.com/library/library/default.asp?url=/library/en-us/dnanchor/html/netdevanchor.asp>.
- [FGL<sup>+</sup>96] C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, and D. Rémy. A calculus of mobile agents. In *Proc. 7th CONCUR, LNCS 1119*, 1996.
- [GMZ00] D. Grossman, G. Morrisett, and S. Zdancewic. Synthetic type abstraction. *ACM TOPLAS*, 22(6):1037–1080, 2000.
- [HL94] R. Harper and M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Proc. 21st POPL*, 1994.
- [HP] R. Harper and B. C. Pierce. Design issues in advanced module systems. Chapter in *Advanced Topics in Types and Programming Languages*, B. C. Pierce, editor. To appear.
- [HRY04] M. Hennessy, J. Rathke, and N. Yoshida. Safedpi: A language for controlling mobile code. In *Proc. FOS-SACS, 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.
- [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] Bent Thomsen, Lone Leth, and Tsung-Min Kuo. A Facile tutorial. In *CONCUR'96, LNCS 1119*, 1996.
- [US01] A. Unyapoth and P. Sewell. Nomadic Pict: Correct communication infrastructure for mobile computation. In *Proc. POPL*, pages 116–127, January 2001.
- [ves] Vesta. <http://www.vestasys.org/>.

\$Id: paper2.mng,v 1.100 2004/07/16 16:41:37 pes20 Exp \$\br/>Time-stamp: <2004-07-16 17:39:59 pes20>

## Appendix

This appendix gives most of the Acute syntax for reference. This is the fully type-annotated source language, including sugared forms, together with other non-source constructs that are needed to express the semantics. The implementation can infer many of the type annotations, and the *mode*, *withspec*, *likespec*, *vce*, *vne*, and *resolvespec* annotations on **module** and **import** default to reasonable values if omitted. The internal parts  $M$ ,  $t$  and  $x$  of identifiers  $M_M$ ,  $t_t$  and  $x_x$  are inferred by scope resolution.

Novel source features are **highlighted in green** and novel non-source constructs are **highlighted in yellow**.

**Abstract names**  $n$     **Store locations**  $l$     **Standard library constants** (with arity)  $x^n$

### Kinds

$K ::= \text{TYPE} \mid \text{EQ}(T)$

### Types

$T ::= \text{int} \mid \text{bool} \mid \text{string} \mid \text{unit} \mid \text{char} \mid \text{void} \mid T_1 * \dots * T_n \mid T_1 + \dots + T_n \mid T \rightarrow T' \mid T \text{ list} \mid T \text{ option} \mid T \text{ ref} \mid \text{exn} \mid M_M.t \mid t \mid \forall t. T \mid \exists t. T \mid T \text{ name} \mid T \text{ tie} \mid \text{thread} \mid \text{mutex} \mid \text{cvar} \mid \text{thunkifymode} \mid \text{thunkkey} \mid \text{thunklet} \mid h.t \mid n$

**Constructors**  $C_0 ::= \dots$      $C_1 ::= \dots$

### Operators

$op ::= \text{ref}_T \mid (=_T) \mid (<) \mid (\leq) \mid (>) \mid (\geq) \mid \text{mod} \mid \text{land} \mid \text{lor} \mid \text{lxor} \mid \text{lsl} \mid \text{lsl} \mid \text{asr} \mid (+) \mid (-) \mid (*) \mid (/) \mid - \mid (@_T) \mid (\wedge) \mid \text{create\_thread}_T \mid \text{self} \mid \text{kill} \mid \text{create\_mutex} \mid \text{lock} \mid \text{try\_lock} \mid \text{unlock} \mid \text{create\_cvar} \mid \text{wait} \mid \text{signal} \mid \text{broadcast} \mid \text{exit}_T \mid \text{compare\_name}_T \mid \text{thunkify} \mid \text{unthunkify}$

### Expressions

$e ::= C_0 \mid C_1 e \mid e_1 :: e_2 \mid (e_1, \dots, e_n) \mid \text{function } mtch \mid \text{fun } mtch \mid l \mid op^n e_1 \dots e_n \mid x^n e_1 \dots e_n \mid x \mid M_M.x \mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \mid \text{while } e_1 \text{ do } e_2 \text{ done} \mid e_1 \&\& e_2 \mid e_1 \mid e_2 \mid e_1 ; e_2 \mid e_1 e_2 \mid !_T e \mid e_1 :=_T e_2 \mid \text{match } e \text{ with } mtch \mid \text{let } p = e' \text{ in } e'' \mid \text{let } x : T \ p_1..p_n = e' \text{ in } e'' \mid \text{let rec } x : T = \text{function } mtch \text{ in } e \mid \text{let rec } x : T \ p_1..p_n = e' \text{ in } e'' \mid \text{raise } e \mid \text{try } e \text{ with } mtch \mid \Lambda t \rightarrow e \mid e \ T \mid \{T, e\} \text{ as } T' \mid \text{let } \{t, x\} = e_1 \text{ in } e_2 \mid \text{marshal } e_1 e_2 : T \mid \text{unmarshal } e \text{ as } T \mid \text{fresh}_T \mid \text{cfresh}_T \mid \text{hash}(X.X)_T \mid \text{hash}(T, e_2)_{T'} \mid \text{hash}(T, e_2, e_1)_{T'} \mid \text{swap } e_1 \text{ and } e_2 \text{ in } e_3 \mid e_1 \text{ freshfor } e_2 \mid \text{support}_T e \mid M_M@X \mid \text{name\_of\_tie } e \mid \text{val\_of\_tie } e \mid \text{namecase } e_1 \text{ with } \{t, (x_1, x_2)\} \text{ when } x_1 = e \rightarrow e_2 \text{ otherwise } \rightarrow e_3 \mid \text{name\_value}(n) \mid h.x \mid e_1 :='_T e_2 \mid \text{marshalz } e_1 e_2 : T \mid \text{RET}_T \mid \text{SLOWRET}_T \mid \text{TERM} \mid \text{op}(op^n)^n e_1 \dots e_n \mid \text{op}(x^n)^n e_1 \dots e_n \mid [e]_{eqs}^T \mid \text{resolve}(M_M.x, M'_{M'}, \text{resolvespec}) \mid \text{resolve\_blocked}(M_M.x, M'_{M'}, \text{resolvespec})$

### Matches and Patterns

$mtch ::= p \rightarrow e \mid (p \rightarrow e \mid mtch)$   
 $p ::= (- : T) \mid (x : T) \mid C_0 \mid C_1 p \mid p_1 :: p_2 \mid (p_1, \dots, p_n) \mid (p : T)$

### Signature and Structure Bodies

$sig ::= \text{empty} \mid \text{val } x_x : T \ sig \mid \text{type } t_t : K \ sig$   
 $str ::= \text{empty} \mid \text{let } x_x : T \ p_1..p_n = e \ str \mid \text{type } t_t = T \ str$

### Version and version constraint expressions

$avne ::= \underline{n} \mid \underline{N} \mid h \mid \text{myname}$   
 $vne ::= avne \mid avne.vne$   
 $vce ::= \dots$

## Source definitions and Compilation Units

```
sourcedefinition ::= module mode  $M_M : Sig$  version vne = Str withspec  
                   import mode  $M_M : Sig$  version vce likespec by resolvespec = Mo  
                   mark MK  
                   module  $M_M : Sig = M'_M$   
  
   mode ::= hash | cfresh | fresh  
   withspec ::= empty | with ! eqs  
   likespec ::= empty | like  $M_M$  | like Str  
   resolvespec ::= empty | STATIC_LINK, resolvespec | HERE_ALREADY, resolvespec | URI, resolvespec  
   Mo ::=  $M_M$  | UNLINKED  
  
compilationunit ::= empty | e | sourcedefinition ;; compilationunit |  
                   includesource sourcefilename ;; compilationunit |  
                   includecompiled compiledfilename ;; compilationunit
```

## Compiled Definitions and Compiled Units

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

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

```
mv ::= marshalled(definitions,  $E_s$ , s, e, T)
```

## Module names (hashes and abstract names)

```
h ::= hash(hmoduleeqs  $M : Sig_0$  version vne = Str) | hash(himport  $M : Sig_0$  version vc like Str) | n  
X ::=  $M_M$  | h
```

## Expression name values

```
n ::=  $n_T$  | hash(h.x)T | hash(T',  $\underline{x}$ )T | hash(T',  $\underline{x}$ , n)T
```

(In the implementation all *h* and **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 ::=  $\emptyset$  | eqs,  $X.t \approx T$ 
```

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

```
E ::= empty |  $E, x : T$  |  $E, l : T \text{ ref}$  |  $E, M_M : Sig$  |  $E, t : K$ 
```

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

```
 $E_n$  ::= empty |  $E_n, n : \text{nmodule}_{eqs} M : Sig_0$  version vne = Str |  $E_n, n : \text{nimport} M : Sig_0$  version vc like Str |  
        $E_n, n : \text{TYPE}$  |  $E_n, n : T \text{ name}$ 
```

## Processes

```
P ::= 0 | (P1 | P2) | n : definitions e | n : MX( $\underline{b}$ ) | n : CV
```

## Single-Machine Configurations

```
config ::=  $E_n$  ;  $\langle E_s, s, \text{definitions}, P \rangle$ 
```