

Marshalling: Abstraction, Rebinding, and Version Control

James J. Leifer[†] Peter Sewell[‡] Keith Wansbrough[‡]

[†]INRIA Rocquencourt
James.Leifer@inria.fr

[‡]University of Cambridge
{Peter.Sewell,Keith.Wansbrough}@cl.cam.ac.uk

Abstract

We discuss the design of programming languages for distributed computation, focussing on support for type-safe marshalling of arbitrary language values. In particular: (1) unmarshalling can involve *rebinding* to local resources; (2) values of *abstract types* can be communicated, and a globally-coherent notion of type equality ensures that unmarshalling respects abstraction; and (3) interoperability between separately-built programs with different *versions* of shared modules is supported, with fine-grain version control.

This paper discusses the design space and describes an experimental language, Acute, which collects a coherent set of design choices. Acute extends an ML fragment with marshalling and versions, it has a complete semantic definition (of typing, compilation, and runtime), and has been implemented.

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

1. Introduction

Distributed computation is ever more widespread and critical, but most programming language design has focussed on problems of local computation, leaving both the *means* of interoperation between programs and the *contents* of messages to the programmer to arrange. Historically, the means have been via system calls or communication libraries; the contents have been in byte-sequence or XML wire formats. We see several key issues:

Firstly, we must deal not just with distributed execution, but with distributed development and deployment, taking place across many administrative domains on a long time-scale. It is therefore impossible to synchronise software updates: there may be many coexisting versions of many applications, sharing some libraries but not others, that need to interoperate.

Secondly, we must deal with partial failure, attack, and, sometimes, mobility of code, of devices, and of running computations. This leads to a great variety of communication and persistent store abstractions, with varying performance, security, and robustness

properties, used for interoperation in different circumstances. Accordingly, we believe a distributed programming language should indeed not have a built-in commitment to any particular *means* of interaction. Rather, it should be at a level of abstraction that makes distribution and communication explicit, but be expressive enough to enable different abstractions to be provided as library modules.

Thirdly, in contrast to the previous point, providing no language support for the *contents* of exchanged data is unsatisfactory, and discards the benefits of a high-level language. Standardised wire formats have an important place, but it is increasingly useful to be able to exchange arbitrary language values, preserving their structure, without requiring the programmer to translate back and forth. Moreover, distributed programming and debugging is particularly challenging; interaction should be guaranteed to be type-safe, with errors detected as early as possible.

Many existing languages do provide some form of *marshalling* or *serialization*, with a dynamic check at unmarshal-time. However, the constructs are often insufficiently expressive and not fully integrated with the rest of the language, leaving the programmer unable to control what is shipped, what is checked dynamically when unmarshalling, and how the shipped contents are linked in to the receiver. In particular, none deal satisfactorily with *type abstraction*, *rebinding* to local resources, and *version change*.

Type abstraction is an important tool for modular development, preventing accidental dependencies on a module's implementation details. Values of an abstract type can only be operated on by its implementation, which will typically preserve some user invariants. Within a single program this is checked by conventional static typing, but what if values are marshalled and unmarshalled elsewhere? Simply ensuring that marshalling is type-safe is not enough: it should also be *abstraction-safe* by default, respecting abstraction boundaries (and hence preserving user invariants) across a distributed system. When versions change, though, some controlled forms of abstraction-breaking may be needed.

Static binding is desirable for most programming, but when we marshal a value it may have to be dynamically *rebound* to location-dependent resources, and also to local versions of modules – though sometimes we may want to ship the entire code-base it depends on. Thus we require expressive language constructs to specify what to ship and what to rebound. There may be version constraints that the value demands of its new context, and version declarations of the resources the receiver offers.

This argument leads us to consider the design of languages with marshalling of arbitrary language values to byte strings (above which various communication library modules can be expressed), type- and abstraction-safe unmarshalling, and a coherent notion of type equality between separately-developed programs that share some or no libraries, potentially of different versions.

In earlier work we studied two aspects of the problem in isolation. The paper [BHS⁺03] discussed rebinding marshalled values to local resources, in a simply-typed lambda calculus setting. The paper [LPSW03] treated marshalling of values with abstract types, but without rebinding, using module *hashes* to construct globally-meaningful type names. Neither dealt with version constraints beyond type equality.

Contribution and Approach In this paper we focus on the integration of rebinding and abstract types. Versions and version constraints are clearly also needed, and included; surprisingly they turn out to play a key role in the integration. We work towards a realistic programming language rather than on purely theoretical calculi. Our contribution takes two forms. On the one hand, we discuss many aspects of the (large!) design space for such languages. On the other, we have designed and implemented a particular language, Acute, in which to exhibit a coherent set of design choices.

Acute is not intended as a full proposal for a production language, but as a vehicle for experimentation and a starting point for debate. It is sufficiently expressive (and has enough library support) to allow non-trivial examples to be written, but yet is still reasonably simple. We observe that some design choices turn out to be forced. Others are more ad-hoc, but are chosen to support a good range of examples.

The core language of Acute is based on ML. We aim towards primitives that would be pragmatically useful, and would be reasonably easy to integrate with either SML or OCaml. Much of the design discussion is applicable also to lower-level languages, e.g. for dynamic linking, kernel modules etc. in the setting of typed intermediate languages or assembly language.

We are primarily concerned with what the user source language should be, and so work with a direct semantics and implementation, eschewing syntactic sugar and encodings. In detail, we:

- support rebinding to local resources, lifting the *lambda-mark* semantics of [BHSSW03] to a module language and using *redex-time* semantics between modules (§2);
- add a simple language of *resolve methods* to describe how to obtain dynamically-required libraries (§2);
- add languages of module versions and version constraints (§3);
- extend the treatment in [LPSW03], of marshalling in the presence of abstract and manifest types, to modules with multiple type fields (§4.1);
- support controlled abstraction-breaking type coercions, making the *with!* of [LPSW03] precise and integrating it with multiple-field modules (§4.2); and, most significantly,
- integrate rebinding and the treatment of abstract types, using *import hashes* to define a notion of type equality that makes sense globally (§4.3).

We have developed a full semantic definition for Acute, using *coloured brackets* [GMZ00, LPSW03] to preserve abstraction during computation (§5), and have implemented it (§6); the implementation will be made available. We discuss related work and conclude in §7 and §8.

The core language of Acute consists of normal ML types and expressions: booleans, integers, strings, tuples, lists, options, recursive functions, pattern matching, references, exceptions, and invocations of OS primitives in standard libraries. To avoid syntax debate we fix on one in use, that of OCaml.

Our module language supports top-level declarations of structures, containing value term fields and type fields, with both abstract and manifest types in signatures.

The main new constructs are as follows. We give just the syntax here, for concreteness; the rest of the paper is devoted to the design rationale and semantics. Expressions are extended with marshalling and unmarshalling; a program is a sequence of module definitions followed by an expression; module definitions are either of marks, definitions of modules, or imports of module identifiers:

```
e ::= ...
| marshal MK e:T | unmarshal e as T
definition ::= ...
| mark MK
| module M:Sig version vne = Str withspec
| import M:Sig version vce likespec
  by resolvespec = Mo
```

Here *Sig* and *Str* are ML-style module interfaces and bodies, with type and term fields. Marks MK name boundaries in module definition contexts at which dynamic rebinding might occur, and imports constrain and specify how rebinding happens. We return later to the version numbers and constraints *vne* and *vce*, and to the other components.

We make several major simplifying assumptions, which should be the subject of future work. We omit: module initialisation; separate interfaces; most subsignaturing; subtyping; recursive modules; fine-grain control over marshalling store fragments; and dealing with OS library values that should not be naively marshalled, e.g. file descriptors. We also omit some straightforward features, simply to keep the language small: core language polymorphism, except for built-in constructors and operators; user-defined type operators; user-defined exceptions; concurrent threads; and functors (we believe that adding first-order applicative functors would be straightforward; going beyond that would be more interesting).

A full version of this paper, including further discussion and the semantic definition of Acute, is available [LSW03].

2. Module-level rebinding, without type fields

We begin by considering the problem of marshalling and rebinding to local resources, deferring consideration of abstract types and rich version languages to the following sections.

marshal and *unmarshal* convert to and from byte-strings, which can be used with any persistent storage or communication primitives. For brevity our examples use loopback communication primitives *send:string->unit* and *receive:unit->string*, which are implemented above the Acute sockets interface. We write *p1 | p2* for the parallel execution of the two separately-built programs *p1* and *p2*. Modulo some minor syntactic issues the examples are all executable Acute code (for example *send* must really be written *IO.send*, and some interfaces and types are elided).

For example (ignore the *mark* and *MK* for now):

```
mark MK
send( marshal MK 5 : int )
|
print_int ( 3 + unmarshal (receive ()) as int)
```

communicates the value 5 from the first program to the second, which prints 8.

For safety, a type equality check is obviously needed, 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

```
mark MK
send( marshal MK "five" : string )
|
print_int ( 3 + unmarshal (receive ()) as int)
```

should raise an exception. Allowing interaction via an untyped medium inevitably means that some dynamic errors are possible, but they should be restricted to clearly-identifiable program points, and detected as early as possible. This error should be detected at unmarshal-time, rather than when the received value is used; for some later design points there will be trade-offs between early and late error detection.

This dynamic check might be of two strengths: we can either assume one only ever unmarshals strings created by marshalling in well-behaved runtimes, in which case it would suffice to include a type in marshalled values and check equality between that and the T of `unmarshal ... as T`, or we can additionally check that the marshalled value is a well-formed representation of something of that type. For robustness the latter is clearly preferable.

Note that the implementation does not need to keep types for all runtime values, but only the types associated with marshal and unmarshal points.

2.1 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, a function (perhaps a large plug-in software component) might mention identifiers for:

- (1) ubiquitous standard library calls, e.g., `print_int`;
- (2) location-dependent application-specific library calls, e.g., routing functions;
- (3) application code which 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.

There is another possibility: a local reference 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 (most sharply, where the function is marshalled to persistent store, and unmarshalled after the original process has terminated). Following the argument of §1, we choose not to build-in any commitment to a particular communications infrastructure, and so do not support distributed references directly, aiming rather to ensure our language is expressive enough to allow libraries of ‘remotable’ references to be written above our lower-level marshalling primitives.

2.2 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 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 1:1 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 so such *negotiation* protocols are not expressible in Acute.

Instead, we make a simple adaption of the mechanism proposed in [BHSSW03] (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 followed by a running expression; those module definitions, together with implicit module definitions for standard libraries, are the resources. We introduce *marks* to name a sequence of module definitions – the sequence above the occurrence of the mark. 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.

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 `function ()->M.y`. This marshal is with respect to mark `MK`, which lies above the definition of module M , so a copy of the M definition is marshalled up with the value `function ()->M.y`. Hence, when this function is applied to `()` in the receiver the evaluation of `M.y` can use that copy, evaluating to 6.

```
mark MK
module M : sig val y:int end = struct let y=6 end
send( marshal MK function ()->M.y : unit->int )
|
(unmarshal (receive ()) as unit->int) ()
```

(Module M has an interface `sig val y:int end` and a body `struct let y=6 end`; we will often omit explicit interfaces where they can be inferred).

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 `() | ((6,3),4)`.

```
module M1 = struct let y=6 end
mark MK
module M2 = struct let z=3 end
send( marshal MK (function ()-> (M1.y,M2.z))
      : unit->int*int )
|
module M1 = struct let y=6 end
module M2 = struct let z=4 end
((unmarshal (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 (a modularised version of [BHS⁺03, Fig. 3]) shows this. The sender has two modules, $M1$ and $M2$, with $M1$ above the mark `MK`. It marshals a value `function ()-> (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 the sender declares

with an explicit import that the code below MK can be rebound to any version (*) of M1. This overrides the default behaviour, which would add a generated import to the marshalled value with a version constraint allowing rebinding only to identical copies of M1. The example terminates with () | (7,3).

```

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

```

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

A language with rebinding cannot use a standard call-by-value operational semantics, that 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*, in which one instantiates an identifier only when it appears in a context which needs to destruct the outermost structure of the value. The destruct-time semantics permits more rebinding, but is also more complex – here, to make the semantics as intuitive as possible, we use redex-time instantiation of module references.

For example, the first occurrence of M.y 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.

```

module M = struct let y=6 end
import M : sig val y:int end version * = M
mark MK
send( marshal MK (M.y, function ()-> M.y)
      : int * (unit->int) )

```

2.4 The structure of marks and modules

In Acute a running program consists of a linear sequence of definitions – marks, module definitions and imports – followed by the expression being evaluated. 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.

Whether this linear structure is ideal, and if not what should replace it, we leave for future work. There are some obvious possible alternatives: 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. In the linear-structure

case, 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.

2.5 Controlling when rebinding happens

We have to choose whether 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
function ()-> M.y

```

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

```

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

```

is roughly the program below.

```

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

2.6 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 URL. 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 `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 for 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://foo.bar/M" = unlinked
M.y + 3
```

Here the `M.y` is in redex-position, so the runtime examines the *resolvespec* associated with the import of *M*. Here that list has just a single element, the URI `http://foo.bar/M`. 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 faulting mechanism is not an exception – after *M* is loaded, the `M.y` is instantiated in its original evaluation context – + 3. It could be encoded, of course (with exceptions and affine continuations, or by encoding imports as option references) but we are focussing on the user language.

In [BHS⁺03] one could limit the resources that a particular unmarshal could rebind to (by specifying a mark), thus enabling one to securely encapsulate unmarshalled code, allowing rebinding only to ‘safed’ versions of libraries. That is clearly desirable, but to support sufficiently-flexible limits it seems necessary to have more structure than the Acute linear sequence of marks and modules.

2.7 The relationship between modules and the filesystem

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

```
compilationunit ::=
  empty
  e
  definition ;; compilationunit
```

```
include sourcefilename ;; compilationunit
include compiledsourcefilename ;; compilationunit
```

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

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

2.8 Marshalling references

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

```
mark MK
let x = ref 5 in send( marshal MK x : int ref )
|
print_int ( !(unmarshal (receive ()) as int ref ) )
```

Here the best choice for the core language semantics seems to be for the marshalled value to include a copy of the reachable part of the store, to be disjointly added to the store of any unmarshaller. Just as in §2.1 we reject the alternative of building in automatic conversion of local references to distributed references, as no single distributed semantics (which here should include distributed garbage collection) will be satisfactory for all applications. A full language must be rich enough to express distributed store libraries above this, of course, and perhaps also other constructs [SY97].

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

3. 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, we believe this is an error-prone approach, liable to end up with semantically-incoherent sets of versions of components inter-operating. 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. This will be desirable where it is critical to have correct behaviour and so one only wants to execute combinations of modules that have been tested together.

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), so we introduce *versions* as crude approximations to semantic module specifications.

We need a language of versions, which will be attached to modules, a language of version constraints, which will be attached to imports, and a version-satisfies-version-constraint relation, which will be checked at link/unmarshal time. An implication relation between version constraints also turns out to be necessary.

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 analysis. Ultimately, it seems desirable to make the basic module primitives parametric on abstract types of versions and constraints – in particular distributed code environments, 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.

As a starting point, we take version numbers to be nonempty lists of natural numbers (as is fairly common) and version constraints to be lists of naturals possibly ending in a wildcard `*`; satisfaction is what one would expect, with `*` matching any (possibly empty) suffix. Many minor enhancements are possible and straightforward. Arbitrarily, we enhance version constraints with closed, left-open and right-open intervals, e.g. `1.5-7`, `1.8.-7`, and `2.4.7-`. The meanings of these numbers and constraints is dependent on some social process. Within a single distributed development environment one needs some shared understanding that new versions of a module will be given new version numbers, commensurate with their semantic changes.

To support tighter version control than this, we add two further mechanisms. As we shall see in §4, we use hereditary hashes of module implementations to construct globally-meaningful names for abstract types; those hashes can also be used in the version languages, as an implementable check for semantic identity.

First, version numbers can mention `myhash`, and version constraints can mention module names `M` (those in scope, obviously). In each case the compiler writes in the appropriate module hash. 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.myhash`, 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 version languages leave power with the code producer, who can “lie” about which version a module is – instead of writing `myhash` they might write the hash of a previous build. This is desirable if they know there are clients out there with an exact-hash constraint, but also know that their semantic change from that previous build will not break any of the clients.

Secondly, therefore, we let the code consumer write a constraint not on the version field of the module it seeks to use, but on its actual hash. Typically one would have a definition of the desired version to hand in the filesystem (and in Acute bring it into scope by including that file), writing `hash=M`. One could also cut-and-paste a hash in explicitly, writing `hash=0xA564C8F3`. If one needs a guarantee that only mutually-tested collections of modules will be executed together, say where one is writing safety-critical software, this might be the desired idiom everywhere.

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

```

avne ::=      Atomic version number expression
  n      natural number literal
  myhash  hash of this module

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

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

```

```

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

vce ::=     Version constraint
  dvce      dotted version constraint
  hash = avce  exact-hash version constraint

```

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

```

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

```

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

It turns out that one needs exact-hash version constraints not just for user-specified tight version constraints, as in the idiom above, but also during marshalling, when one may have to generate imports for module bindings that cross a mark. Exact-hash constraints seem to be the only reasonable default to use there. Mostly one needs only the satisfaction relation; the implication between constraints is needed only when when deals with chains of imports.

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

4. Manifest and abstract type fields

We now turn to modules with type fields. As in ML, and in the two examples below, type fields can either be *abstract*, with the representation type held private to the module body, or *concrete*, with the type field in the signature manifestly equal to its representation type.

```

module Mabstract
: sig type t      val get:t->int      end
= struct type t=int let get=function x->x end

module Mconcrete
: sig type t=int  val get:t->int      end
= struct type t=int let get=function x->x end

```

4.1 Type equality and hashes, without rebinding

To marshal values of abstract types we need a notion of type equality that makes sense 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 a module that defines an abstract type (and share its dependencies) but differ elsewhere, they should be able to exchange values of that type. The solution put forward in [LPSW03] is to use

module *hashes* as global names for abstract types. For example, the type `Mabstract.t` will be compiled (roughly) to `h.t`, where the abstract-syntax hash `h =`

```
hash(
  module Mabstract
  :   sig type t       val get:t->int       end
  = struct type t=int let get=function x->x end
)
```

If one unmarshals a pair of type `Mabstract.t * Mabstract.t`, say, the unmarshal type check will compare with `h.t*h.t`.

In constructing the hash for module `M` we have to take into account any dependencies it has on other modules `M'`, replacing any type `M'.t` and term `M'.x` references. 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.

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

Dealing with modules that are general dependent products, rather than the dependent type/term pairs of [LPSW03], requires addressing several technical points: (1) The semantics requires both external and internal (subject to alpha equivalence) field names. (2) References to previous manifest type fields must be normalised away before hash generation, to ensure that hashes are calculated up to type equality. (3) After calculating the hash, compilation must *selfify* the module by replacing abstract type `t` by the manifest type `t=h.t`. In contrast to [Sew01], this selfification only affects the signature, not the structure. (4) We need to record the set *ats* of which fields were originally abstract, before selfification. For details of these we refer the reader to the technical report [LSW03].

4.2 User type coercions with `with!`

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

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

In [Sew01, LPSW03] we proposed a *strong coercion* to do this. We have now given it a precise semantics integrated with the Acute multi-field structures. By analogy with ML sharing specifications, we allow a module definition to have a *withspec*, a list of equalities between abstract types of the module and abstract types constructed earlier (typically, of previous builds of the same module).

```
definition ::= ...
module M : Sig version vne = Str withspec
withspec ::= with! withspecbody
```

```
withspecbody ::= empty | t = M.t', withspecbody
```

The compiler checks the old and new types have compatible representations (respecting any internal abstraction boundaries); if they do, the new type is compiled to be manifestly equal to (the internal hash-name of) the old type. The *withspec* is, in effect, a declaration by the programmer that the old and new implementations respect the same invariants.

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

4.3 Rebinding

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=function x->x+2 end
module EvenCounter
: sig          = struct
  type t      type t=int
  val start:t let start = 0
  val get:t->int let get = function x->x
  val up:t->t   let up=function x->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 (the latter assuming `with!` is not used, of course).

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.

```
module M : sig      val f:int->int      end
              = struct let f=function x->x+2 end
import M : sig val f:int->int end version * = M
mark MK
module EvenCounter : ... = ...
send( marshal MK (function () -> EvenCounter.get
  (EvenCounter.up EvenCounter.start)):unit->int)
|
module M : sig      val f:int->int      end
              = struct let f=function x->x+3 end
(unmarshal (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-hash constraint `hash=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-hash 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) then 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 any `M` matching the signature and version constraint. In generating the hash for `EvenCounter`, then, 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* – which names the set of all `M` implementations that match that signature and version constraint.

In the case above this hash would be

```
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 version 2.3.* like Graphics2_0
```

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 `Sig`.

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* 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 interoperability (e.g. to have a local code mirror, to change a web code repository to avoid a viral 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.

4.4 Summary

Summarising, our module constructs are:

```
definition ::=
  mark MK
  | module M:Sig version vne = Str withspec
  | import M:Sig version vce likespec
    by resolvespec = Mo
```

The mark `MK` names the sequence of definitions above the mark. A module definition has a name `M`, an ML-style interface and body `Sig` and `Str`, with type fields and (value) term fields, a version number expression `vne`, and a *withspec* which can be used to coerce abstract types to be compatible with previous builds. An import introduces a module identifier `M` with interface `Sig`, which must satisfy the version constraint expression `vce`. The import is either linked to some previous module or import, in which case `Mo` is the name thereof, or `Mo` is unlinked. If a field `M.x` is required of an unlinked import the *resolvespec* is used to determine how to look for a definition. Finally, the *likespec*, typically the name of an earlier version of the module, is necessary for type soundness of imports with abstract types, recording their representation types.

The interplay between versions, types and interfaces is delicate. Versions do not appear directly in the type language or in the signature language, though they do appear in module (and import) hashes, and hence in the runtime type names for abstract types. Version expressions and version constraint expressions can involve both user-supplied version numbers and module hashes. Version constraints can be in terms both of version numbers and, where tighter control is required, module hashes (as a shorthand for semantic identity). A module *withspec* allows compile-time coercion of abstract types, giving them the same runtime type names as previously-built abstract types. (It does not permit coercion of the module hash, however). When writing a new version of a module, one may (or may not) give it a changed version number expression, and, independently, one may coerce some of its abstract types to be compatible with previous versions. Import *likespecs* are analogous, restricting the modules that can be linked to the import to an equivalence class that share representation types, typically named by giving a previously-built sample version.

5. Semantics

The Acute semantics consists of three parts: a type system, a definition of compilation, and an operational semantics describing the runtime behaviour. For lack of space we omit all details, referring the reader to the technical report [LSW03].

It uses several constructs that do not occur in source programs. Most significantly: (1) hashes h of modules and of imports are constructed by compilation, and used at runtime for unmarshal type equality checks; (2) the semantics preserves abstraction boundaries throughout, using *coloured brackets* $[e]_{eqs}^T$ [GMZ00, LPSW03] to

delimit subexpressions in which type equalities *eqs* between abstract types *h.t* and their representations can be used; (3) marshalled values contain a list of definitions, a store, a core value, and its type; and (4) **resolve** expressions are used to express the state of an executing *resolvespec*.

The type system uses singleton kinds to express manifest and abstract types [HL94, Ler94]. Additionally, most type judgements, and the operational relations, are with respect to sets *eqs* of type equalities $h.t = T$, reflecting which abstractions one is within. The most interesting typing rules are for modules, imports, and hashes – hashes behave very like module identifiers, with rules for selfification and for constructing types *h.t* and terms *h.x* (the latter occurring only within other hashes, not in executable code).

The semantics of compilation is a partial function that takes a compilation unit and a filesystem (mapping filenames to source and compiled files) and constructs a compiled sequence of definitions (and, optionally, an expression). It first preprocesses, resolving included files, and typechecks. It then recurses over the resulting *definitions*, constructing module hashes to use at runtime and dealing with *withspecs* and *likespecs*. Some care is required to ensure hashes are up to type equality and alpha equivalence.

The runtime semantics defines unlabelled reductions, for internal computation, transitions labelled *GetURI(URI)*, *DeliverURI(definitions')*, and *CannotFindURI*, for the IO involved in executing *resolvespecs*, and transitions labelled $f v_1 \dots v_n$ and v for invocations of external standard library functions. The main novelties are the semantics of marshalling and unmarshalling, the semantics for resolving unlinked *M.x* in redex-position, and the rules for manipulating coloured brackets to preserve abstraction boundaries.

6. Implementation

As we have seen, there are many design choices in this area, and practical experience with the various possible options is required in order to evaluate them.

Accordingly, we have implemented Acute. On the one hand, the implementation must support non-trivial examples: it must be reasonably efficient, and have sufficient library support and error reporting. On the other hand, it must be easy to see that it is a faithful implementation of the definition, and must be easy to modify to variant definitions as we explore the choices. We therefore take a middle way between a full compiler and a direct implementation of the operational semantics, with a runtime that manipulates OCaml data structures but uses closures. A tool makes it simple to import (monomorphised fragments of) OCaml libraries – at present it provides modules *Char*, *String*, *Graphics*, *Pervasives*, *Sys*, and *Filename*, and a sockets-interface *Tcp* library.

The language definition is large enough that proving standard properties (type preservation and progress) would be a substantial undertaking, which we have not attempted. Instead, to provide some confidence in both the definition and the implementation, we have a mode in which all intermediate states reached in a computation can be type-checked. For this we keep coloured brackets and the internal structure of hashes; a production implementation would erase the former and use an MD5 or SHA1 hash instead of the latter. For the system of [LPSW03] we proved that –where this is meaningful– the dynamic type equality used at unmarshal time coincides with the usual source-program static type equality. We expect a similar fact to hold here.

As an experiment, the implementation is written in Fresh OCaml [SPG03], which provides direct support for dealing with identifiers up to alpha conversion. The polytypic swap operation has proved useful, though the current fresh patterns have not – we often have

term-variable binders that mention non-binding type variables.

Several thousand lines of code have been written in Acute and its predecessors, including *minesweeper* and *blocks* games that marshal parts of their state. These confirm that the performance and expressiveness suffice for our purposes, though further experimentation is required (for example, we are currently writing a communication library using the *Chord* P2P algorithm, and adding the concurrency primitives it needs). The implementation will be made publicly available.

7. Related work

There has been little prior work on good programming-language support for version change and, to the best of our knowledge, none that deals simultaneously with abstract types and rebinding to local resources.

Abstract types and ML modules have long been investigated, dating back to work on CLU, MacQueen’s original proposal [Mac84] and the SML definition [MTH90]. Their type-theoretic foundations were improved with manifest types/singleton kinds by Harper, Lillibridge and Leroy [HL94, Ler94]. Dynamic semantics for existentials, and for dot-notation [CL90], have mostly discarded abstraction boundaries as computation proceeds. Grossman et al. [GMZ00] introduced coloured brackets to track these boundaries. Sewell [Sew01] introduced compile-time new generation of global abstract type names, and a simple form of the *with!* coercion used here to support some limited versioning scenarios. Global abstract type names based on module hashes were introduced by Leifer et al. [LPSW03] to support abstraction-preserving marshalling; coloured brackets were used to make the semantics abstraction-preserving. Rossberg [Ros03] also addressed marshalling with abstract types, using a calculus with run-time new-generation and coercions which play a similar role to coloured brackets, working towards foundations for Alice [Ali03].

Marshalling values of arbitrary language types is closely related to the *type dynamic* of Abadi et al. [ACPP91, ACPR95], who also give a historical survey. Intensional polymorphism [HM95, Wei02] permits run-time type analysis of *all* values.

A number of distributed languages support type- and abstraction-safe communication between computations that originated from the same program, but resort to an ad-hoc mechanism for establishing communication between programs. These include *Facile* [TLK96], *JoCaml* [JoC], and *Nomadic Pict* [SWP99].

Several languages build-in particular higher-level distributed abstractions to their runtimes, taking an opposing view to the one we espoused in §1. Cardelli’s *Obliq* [Car95] is a good example, with network-transparent remote object references, above *Modula3*’s network objects; *JoCaml* provides location-independent communication between mobile computations, with a distributed algorithm involving collapsible forwarding-pointer chains – *Nomadic Pict* adopted a lower level of abstraction specifically to enable such algorithms to be programmed and reasoned about.

Dynamic binding (in the absence of abstract types) has a widely-scattered literature; we refer the reader to [BHS⁺03] for discussion.

Static linking has been studied by Cardelli for a model language [Car97]; Hicks et al. have examined safe dynamic linking of native code [HWC00], extending Typed Assembly Language with constructs similar to *dOpen*. Linking for Java, and in particular binary compatibility, has been studied by Drossopoulou et al. [DEW99]. The *units* of Flatt and Felleisen provide fine-grain control of how imports and exports are linked [FF98].

Explicit versioning has seldom appeared in programming languages, but is common in package management – 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.

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.

The Microsoft .NET framework supports versioning of *assemblies* [dot03], which may consist of several files; they are stored locally in an assembly cache. 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. For remoting, when a “client-activated” object is called, the server will create the highest version compatible with the one requested by the client, or the exact version if no policy file is present. However, if the “class holder” cannot resolve the assembly, the system deletes the version information and falls back on partial binding. The server cannot control the version created, though the client can. When a “server-activated” object is called, the version information is ignored and the latest version (or that from the service’s policy file) of the object is created to service the client – here the server has control.

8. Conclusions and future work

In summary, we have designed and implemented a programming language, Acute, for distributed computation that supports type-safe marshalling, rebinding to local resources, abstract types, and version change. This has required new semantics for variable instantiation and rebinding (a module-level redex-time development of lambda-mark), resolve methods, languages of versions and version constraints, and careful design of runtime type names, with novel import hashes. Moreover, we have discussed the design space within which Acute lies, identifying many important issues.

Future work In the near future we will gain further practical experience with using the Acute constructs, with some larger examples. Thereafter, we will revisit some of the main design decisions, particularly those of §2. We would like to: (1) support negotiation, with some lower-level linking and marshalling operations; (2) reconsider the linear mark and module structure; (3) support finer-grain control over when linking happens; (4) allow computational resolve methods; and (5) support region-based control over what parts of the store to marshal. From §3 we would like to support signed versions, and hereditary version numbers, to specify a coherent collection of dependencies. We believe the treatment of abstract types in §4 is satisfactory. Large examples may require extending Acute with some omitted standard features.

Further challenging problems are the treatment of module initialisation (needing new-generation of type names in effectful cases), subtyping, and OS libraries (perhaps using regions again, to track local values which should not be marshalled). For some purposes it would be useful to have a standard mapping from language types to XML structures, in addition to the internal marshalled format.

Ultimately, there is a need to integrate programming-language versioning constructs and those of system-administration machinery for package management.

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 PEPIN, and APPSEM 2. We thank Vilhelm Sjöberg and Christian Steinrücken for their implementation work on predecessors to Acute.

9. REFERENCES

- [ACPP91] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically typed language. *ACM TOPLAS*, 13(2):237–268, 1991.
- [ACPR95] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Didier Rémy. Dynamic typing in polymorphic languages. *J. Functional Programming*, 5(1):111–130, 1995.
- [Ali03] The Alice project, 2003. <http://www.ps.uni-sb.de/alice/>.
- [BHS⁺03] Gavin Bierman, Michael Hicks, Peter Sewell, Gareth Stoye, and Keith Wansbrough. Dynamic rebinding for marshalling and update, with destruct-time λ . In *Proc. ICFP 2003*, 2003.
- [Car95] Luca Cardelli. A language with distributed scope. In *Proc. 22nd POPL*, pages 286–297, 1995.
- [Car97] L. Cardelli. Program fragments, linking, and modularization. In *POPL’97*, pages 266–277, January 1997.
- [CL90] Luca Cardelli and Xavier Leroy. Abstract types and the dot notation. Technical Report 56, DEC SRC, March 10 1990.
- [DEW99] S. Drossopoulou, S. Eisenbach, and D. Wragg. A fragment calculus towards a model of separate compilation, linking and binary compatibility. In *Proc. LICS*, pages 147–156, 1999.
- [dot03] Packacking and deploying .net framework applications (.net framework tutorials), 2003. <http://msdn/microsoft.com/library/default.asp?url=/library/en-us/dnanchor/html/netdevanchor.asp>.
- [FF98] Matthew Flatt and Matthias Felleisen. Units: Cool modules for HOT languages. In *PLDI*, 1998.
- [GMZ00] D. Grossman, G. Morrisett, and S. Zdancewic. Syntactic type abstraction. *ACM TOPLAS*, 22(6):1037–1080, 2000.
- [HL94] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Proc. 21st POPL*, 1994.
- [HM95] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Proc. 22nd POPL*, 1995.
- [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.
- [HWC00] Michael Hicks, Stephanie Weirich, and Karl Cray. Safe and flexible dynamic linking of native code. In *Proc. 3rd Workshop on Types in Compilation, LNCS 2071*, pages 147–176, 2000.
- [JoC] JoCaml. <http://pauillac.inria.fr/jocaml/>.
- [Ler94] Xavier Leroy. Manifest types, modules, and separate compilation. In *Proc. 21st POPL*, pages 109–122, 1994.
- [LPSW03] James J. Leifer, Gilles Peskine, Peter Sewell, and Keith Wansbrough. Global abstraction-safe marshalling with hash types. In *Proc. 8th ICFP*, 2003.
- [LSW03] James J. Leifer, Peter Sewell, and Keith Wansbrough. Marshalling: Abstraction, rebinding, and version control, 2003. <http://www.cl.cam.ac.uk/~pes20/>.
- [Mac84] David MacQueen. Modules for Standard ML. In *Proc. 1984 ACM Symp. LISP and Func. Prog.*, pages 198–207, 1984.
- [MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.
- [Ros03] Andreas Rossberg. Generativity and dynamic opacity for abstract types. In *Proc. 5th PPDP*, August 2003.
- [Sew01] Peter Sewell. Modules, abstract types, and distributed versioning. In *Proc. 28th POPL*, pages 236–247, 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] Peter Sewell, Paweł T. Wojciechowski, and Benjamin C. Pierce. Location-independent communication for mobile agents: a two-level architecture. In *Internet Programming Languages, LNCS 1686*, pages 1–31, 1999.
- [SY97] T. Sekiguchi and A. Yonezawa. A calculus with code mobility. In *Proc. 2nd FMOODS*, pages 21–36, 1997.
- [TLK96] Bent Thomsen, Lone Leth, and Tsung-Min Kuo. A Facile tutorial. In *CONCUR’96, LNCS 1119*, pages 278–298, 1996.
- [Wei02] Stephanie Weirich. Higher-order intensional type analysis. In *Proc. 11th ESOP, LNCS 2305, Grenoble, France, 2002*.

\$Id: paper.mng,v 1.106 2003/11/30 21:46:25 pes20 Exp \$