Global Abstraction-Safe Marshalling with Hash Types

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

[†]INRIA Rocquencourt
{First.Last}@inria.fr

[‡]University of Cambridge
{First.Last}@cl.cam.ac.uk

Abstract

Type abstraction is a key feature of ML-like languages for writing large programs. Marshalling is necessary for writing distributed programs, exchanging values via network byte-streams or persistent stores. In this paper we combine the two, developing compile-time and run-time semantics for marshalling, that guarantee abstraction-safety between separately-built programs.

We obtain a namespace for abstract types that is global, i.e. meaningful between programs, by hashing module declarations. We examine the scenarios in which values of abstract types are communicated from one program to another, and ensure, by constructing hashes appropriately, that the dynamic and static notions of type equality mirror each other. We use singleton kinds to express abstraction in the static semantics; abstraction is tracked in the dynamic semantics by coloured brackets. These allow us to prove preservation, erasure, and coincidence results. We argue that our proposal is a good basis for extensions to existing ML-like languages, pragmatically straightforward for language users and for implementors.

Categories and Subject Descriptors

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

General Terms Languages, Theory, Verification

Keywords programming languages, ML, type theory, abstract types, marshalling, serialisation, modules, singleton kinds, hashing, distributed programming, lambda calculus

1. Introduction

Problem Type abstraction is a basic tool for modular programming, allowing the programmer to separate the interface and the implementation of an abstract data type, and to limit the scope in which the implementation details are visible. Work on ML-style module systems, including [19, 21, 11, 17], has led to expressive language constructs for controlling abstraction, with modules (structures) that can export abstract types, and also parameterised modules (functors); they have rich notions of type equality to deal with generativity and sharing. This work has largely been in the non-distributed context, concerned only with isolated executions of single programs. There, build-time type checking suffices to guarantee both type-safety and *abstraction-safety* — the property that values of an abstract type can only be inspected or constructed by the code of its definition, and hence that any invariants of this code hold of all values. At run-time, type information can be erased.

In the distributed setting, abstraction-safety is more subtle. One may need to exchange values between multiple executions of the same build, between executions of different builds of the same sources, and between executions of builds of different sources (sharing some modules, perhaps, but not all). This interaction might be by network communication or via a persistent store; in either case, some run-time check is clearly needed to guarantee safety. For abstraction-safety, it does not suffice to check only the underlying representation type; intuitively, we need also that the sender and receiver have compatible invariants. This can be enforced by requiring that they have the same code, but in general, where there is only partial sharing, we shall see that the design of an appropriate check is delicate.

We focus in particular on language support for *marshalling* a value to a byte string and *unmarshalling* such strings back to values. With these, one can implement a variety of useful mechanisms above the standard (byte-string) primitives for network communication and persistence. For example: (1) In the existing distributed languages JoCaml [14] and Nomadic Pict [27] a single program can dynamically distribute computations, which can then interact via typed channels, but unsafe "name servers" are required to bootstrap connections *between* programs. Type- and abstraction-safe marshalling would enable such name servers to be expressed in a safe way. (2) More generally, safe marshalling would enable one to code up a variety of communication abstractions, such as typed channels with differing behaviour (asynchronous, unicast, multicast,...), within a high-level language; they would then be automatically guaranteed to be safe.

Contribution We present a type system and semantics (both compile-time and run-time) for marshalling and unmarshalling values between separate programs. Our solution:

- covers modules that declare abstract types, and ancillary type-sharing constraints;
- involves a dynamic type-check at unmarshal time that guarantees both type-safety and abstraction-safety;
- ensures the resulting dynamic notion of type equality coincides with the usual static notion, so that distributed programming is a smooth extension of local programming;
- "just works" in standard cases for interaction between programs that share some modules, without requiring any shared data beyond the source code for these modules;
- supports controlled abstraction-breaking, where required; and
- is efficiently implementable.

It is therefore a good basis for extensions to existing ML-like languages, pragmatically straightforward for language users and for implementors.

Approach The basic idea of our solution is to construct a global namespace for abstract types, meaningful across all programs, by hashing module declarations. *Hash types* do not

ICFP'03, August 25–29, 2003, Uppsala, Sweden.

appear in source programs, but are constructed at compile-time. For example, consider a module called `N` with the body `struct type t=Trep let x=... end` and the published interface `sig type t val x:... end`. The hash h =

```
hash(module N=struct type t=Trep let x= ... end
               : sig type t      val x: ... end, t)
```

would be constructed to give a run-time analogue of the compile-time abstract type name `N.t`. By constructing hashes carefully, we ensure that a simple run-time syntactic type equality check, at unmarshal-time, corresponds to the compile-time notion of type equivalence used in type-checking.

The standard operational semantics for existentials forgets abstraction. In contrast, we give a run-time semantics that records which subterms can see through which abstractions using *coloured brackets*, adapting a device of [10]. For example, within the code of `N.x`, even after it has been substituted into its usage sites, the type equality h`==Trep` can be used. This enables us to prove type- and abstraction-preservation, progress, and the coincidence result mentioned above. We prove also that an implementation may safely erase all coloured brackets outside hashes at run-time.

Non-goals Our focus in this paper is on what mechanisms are required to guarantee abstraction-safety. We do not address the full ML language; instead, we focus on a core language based on simply-typed λ-calculus with abstract and manifest modules, although we argue that our formal system may be cleanly extended. Dynamic *rebinding* of identifiers within marshalled values is considered in [4]. Moreover, we are not here concerned with low-level representations of marshalled values; we assume some fixed scheme for marshalling simply-typed values. Finally, we protect against confusion, not malice.

Outline We begin in Sec. 2 by examining scenarios in which values of abstract types are communicated between programs, identifying the desired constructs and behaviour from the programmer's point of view. Sec. 3 outlines our solution informally, shows why it provides the desired behaviour, and shows it can be implemented efficiently. In Sec. 4 we present a formal calculus, λ_{hash}, that covers the novel aspects of our solution. It describes networks of interacting separately-built modular programs. In Sec. 5 and Sec. 6 we discuss related and future work and conclude.

2. Abstraction and interaction: the desired behaviour

In this section we discuss the desired behaviour of marshalling in a distributed setting, with a series of informal examples in an ML-like language. Our solution, in the following section, shows how this ideal can be achieved.

We consider an ML-like language in which a program consists of two parts: first a sequence of module declarations, each of which can introduce abstract types; then an expression (the main body of the program). We are concerned with interaction between whole programs, usually built separately. This interaction is via network communication, though it could equally be via a persistent store; in either case, the underlying mechanism simply transmits byte strings. For concreteness, most of our examples involve networks consisting of two machines, `pauillac` and `glia`, running programs, say *Pa* and *Pb*. These network configurations are written `pauillac`[*Pa*] | `glia`[*Pb*]. It then suffices to consider a single communication channel (such as a TCP connection between fixed ports); the language has communication primitives

```
send : string->unit     receive : unit->string
```

We can now explore the desired behaviour of `marshal`$(e:T)$ and `unmarshal`$(e:T)$, which marshal to and from `string`.

2.1 Communication

The simplest example is that of sending a value of a non-abstract type between separately-built programs. Consider the two programs

```
P1a =  send (marshal (5 : int))
P1b =  print_int (unmarshal (receive ():int))
```

If these are built and then executed on the two machines the communication and `unmarshal` should succeed:

`pauillac`[*P1a*] | `glia`[*P1b*] $\surd$

2.2 Respecting types

On the other hand, if one machine sends a `string` that the other attempts to unmarshal as an `int` there should obviously be a run-time failure.

```
P2a  =  send (marshal ("five":string))
P2b  =  print_int (unmarshal (receive ():int))
```

`pauillac`[*P2a*] | `glia`[*P2b*] $\times$

To ease debugging, it is desirable for that failure to occur as early as possible (at unmarshal-time rather than when the `string` is used later) and to be trapped cleanly, raising an exception rather than giving unpredictable behaviour. The implementation must therefore send some form of type representation. The following examples explore the constraints on what this must be.

2.3 Respecting abstractions

Now consider an example with an abstract type. Here the `EvenCounter` module declares a type `EvenCounter.t` which has a representation type of `int` but externally is abstract, as declared in its signature. The operations of `EvenCounter` enforce the invariant that values of `EvenCounter.t` are always represented by even integers. If we allowed an arbitrary integer to be unmarshalled as an `EvenCounter.t` then the abstraction, and this invariant, would be broken; the `unmarshal` should therefore fail.

```
P3a = send (marshal(5:int))

P3b = module EvenCounter =
        struct                   sig
          type t=int               type t
          let start=0       :      val start:t
          let get x = x            val get:t->int
          let up x = x+2           val up:t->t
        end                      end
      print_int (EvenCounter.get
          (unmarshal (receive ()):EvenCounter.t))
```

`pauillac`[*P3a*] | `glia`[*P3b*] $\times$

Marshalling from a different abstract type — say a `TripleCounter.t` — to `EvenCounter.t` should fail similarly.

2.4 Communication between completely-shared sources

For communication between two instances of the same build, which therefore have identical source code, the problem is relatively simple. Below, *P4* declares an abstract type `IntSet.t` of sets of integers, representing them as binary search trees. It makes a run-time determination of which machine it is on and then sends or receives an `IntSet.t`; the unmarshal should succeed. We will develop this example later — suppose this first implementation orders subtrees by <, and has a union operation that does not remove duplicate entries.

```
P4 =
module IntSet =
  struct
    type t = int tree
    let singleton = singleton-code    IntSetStruct
    let mem = mem-code
    ...
  end : sig
    type t
    val singleton : int -> t
    val mem : int -> t -> bool        IntSetSig
    val empty : t
    val add : int -> t -> t
    val union : t -> t -> t
  end
if ...on-machine-pauillac... then
  send (marshal (IntSet.singleton 17 : IntSet.t))
else
  if IntSet.mem 17 (unmarshal(receive():IntSet.t))
  then print "y" else print "n"
```

pauillac[*P4*] | glia[*P4*] $\surd$

By default this should still succeed even if the two machines execute different builds of the same source.

2.5 Communication between partially-shared sources

More generally, one may need communication between programs which share only some modules (perhaps ubiquitous standard libraries, or application-specific libraries). Here *P5a* and *P5b* share the IntSet module from before, but otherwise have different module declarations and main body expressions; their communication of an IntSet.t should succeed.

```
P5a =
module IntSet = IntSetStruct:IntSetSig
send (marshal (IntSet.singleton 17 : IntSet.t))
```

```
P5b =
module IntSet = IntSetStruct:IntSetSig
module M =
  struct let haszero x = IntSet.mem 0 x  end
  : sig  val haszero : IntSet.t -> bool  end
if M.haszero (unmarshal (receive () : IntSet.t))
then print "y" else print "n"
```

pauillac[*P5a*] | glia[*P5b*] $\surd$

2.6 Guaranteeing compatible invariants

In the previous example the two programs had syntactically identical IntSet implementations. Since IntSet does not depend on any other modules, this is a sufficient condition to guarantee that the two abstract types have compatible invariants, i.e. that any value of either will be correctly acted upon by the operations of the other. Moreover, it can be automatically checked, whereas compatibility of invariants cannot even be stated without specifing the behaviour of the two modules, and would then require general theorem-proving to verify. Note that it would not be sufficient to require that the two implementations use the same representation type, or even to require that the implementations are (in the absence of marshalling) observationally equivalent.

For example, suppose that *IntSetStructGt* is similar to *IntSetStruct* but orders subtrees with > rather than <.

```
P6a =
module IntSet = IntSetStructGt:IntSetSig
send (marshal (IntSet.add 0 (IntSet.add 1
          (IntSet.add 2 IntSet.empty)) : IntSet.t))
```

When communicating with *P5b*, which contains the original *IntSetStruct*, the unmarshal should fail, as otherwise an erroneous result could be produced.

pauillac[*P6a*] | glia[*P5b*] $\times$

Later we shall see that a mechanism for intentionally circumventing this restriction, in a controlled way, is sometimes desirable.

2.7 Respecting names (when necessary)

In some cases one has modules with identical implementations that nonetheless provide conceptually different abstract types, for example in the Euro and Pound modules below. Unmarshalling should respect this difference, so the example should fail (just as, within a single ML program, types Euro.t and Pound.t would be incompatible).

```
P7a =
module Euro =
 struct type t=int  let of_int x = x      ... end
 : sig  type t      val of_int : int -> t ... end
send (marshal (Euro.of_int 17 : Euro.t))
```

```
P7b =
module Pound =
 struct type t=int  let of_int x = x      ... end
 : sig  type t      val of_int : int -> t ... end
unmarshal (receive (): Pound.t)
```

pauillac[*P7a*] | glia[*P7b*] $\times$

This restriction is not always useful (e.g. whether an integer set module is called IntSet or Set_Int is likely irrelevant), so the language should support some syntactic way of indicating whether a module name is significant or not.

2.8 Module dependencies

Consider now modules that depend on abstract types declared by other modules. In *P8a* below there is a module IntSet, providing an abstract type IntSet.t, followed by a module SummedIntSet, providing an abstract type of sets of integers augmented with a running sum. The expression part constructs, marshals and sends a value of the SummedIntSet.t abstract type. This SummedIntSet depends on IntSet in three ways: (1) IntSet.t occurs in its representation type IntSet.t $*$ int; (2) IntSet.t occurs in the type of an operation in its signature; and (3) operations from IntSet occur in the definitions of its operations. Any such dependency means that substantive changes to the definition of IntSet must propagate through to give distinct SummedIntSet.t types. On the other hand, any module declarations that are not (transitively) depended upon should have no effect on SummedIntSet.t.

For example, consider also *P8b* below. It has exactly the same text as SummedIntSet but a different implementation of IntSet — suppose *IntSetStruct'* has a different representation type from *IntSetStruct*, or the same representation but incompatible invariants, or different externally-observable behaviour. The *P8a* and *P8b* SummedIntSet.t types should be incompatible, so the unmarshal should fail.

```
P8a =
module IntSet = IntSetStruct:IntSetSig
module SummedIntSet =
  struct
    type t = IntSet.t * int
    let empty = (IntSet.empty,0)
    let sum (x,y) = y
    ...
  end : sig
    type t
    val empty : t
    val singleton : int -> t
    val sum : t -> int
    val to_intset : t -> IntSet.t
    ...
  end
send(marshal((SummedIntSet.singleton 2)
                    : SummedIntSet.t ))
P8b =
module IntSet = IntSetStruct':IntSetSig
module SummedIntSet = ...same text as above...
SummedIntSet.sum
  (unmarshal (receive () : SummedIntSet.t))
```

pauillac[*P8a*] | glia[*P8b*] $\times$

2.9 Mirroring local type sharing: manifest types, functors

The examples in this and subsequent subsections are not covered by the formal calculus of Sec. 4. Nonetheless we argue informally in Sec. 6.2 how they can be treated by straightforward extensions of our main techniques and earlier work.

ML module systems include parametric modules, known as *functors*, for large-scale software structuring and code reuse. In the single-program world there are a number of subtle type-equality issues, related to how generative functors are, and how one can express type sharing constraints [21, 17, 11, 28]. Our marshalling primitives should correctly reflect these subtleties in inter-program communication.

For example, the module `SummedIntSet` above, which explicitly references `IntSet`, might be re-expressed in terms of a functor `F` which takes any argument structure `U` with interface *IntSetSig* and builds a `SummedIntSet`:

```
module IntSet =  IntSetStruct:IntSetSig
module F = functor (U:IntSetSig) ->
  struct type t=U.t*int ... end
  : sig  type t          ... end
module SummedIntSet = F(IntSet)
```

The functor `F` generates an abstract type, so we must consider when that type should be compatible with others. If two separate programs contain this preamble, they should be able to exchange values of their respective `SummedIntSet.t` types. This mirrors the behaviour of OCaml's *applicative functors* [18], in which another instance of the application `F(IntSet)` within the same program would have a type compatible with `SummedIntSet.t`.

Should the functorised and non-functorised (*P8a*) `SummedIntSet.t` be compatible? Again following existing module systems, we should make them incompatible, as otherwise static type equality would depend on module substitution.

Type *sharing* allows functors to express type equalities between their argument and result; unmarshalling should respect these static type equalities. The example `F'` below constructs a type `t` but, in contrast to `F`, does not make that type abstract; instead it makes it manifestly equal to the product of its argument type `U.t` and `int`.

```
module F' = functor (U:IntSetSig) ->
  struct type t=U.t*int ... end
  : sig  type t=U.t*int ... end
```

The application of `F'` to a module `IntSet` creates a static type equality `F'(IntSet).t==IntSet.t*int`, which should also be admitted at run-time.

2.10 Breaking abstractions (simple bidirectional case)

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 Sec. 2.6 restriction.

For example, consider the improved *IntSetStructDeDup* implementation below, in which the operations are similar to *IntSetStruct*, the only difference being that `union` removes duplicates. The compiler cannot verify that *IntSetStructDeDup* has all the semantic properties that the programmer requires of *IntSetStruct*. Hence we provide a way of explicitly declaring that these modules provide compatible types. In *P10a* below, `IntSet'.t` is made equal to `IntSet.t` by the *strong coercion* `...with t =! IntSet.t`. The compiler checks only that the old and new types have compatible representations (here `int tree`), but should respect further abstractions *within* those representation types. This is based on our earlier work of [26].

```
P10a =
module IntSet = IntSetStruct:IntSetSig
module IntSet' =
  struct                         }
    type t = int tree            } IntSetStructDeDup
    ...improved operations...    }
  end                            }
: IntSetSig with t =! IntSet.t
send (marshal (IntSet'.singleton 17 : IntSet'.t))

P10b =
module IntSet = IntSetStruct:IntSetSig
if IntSet.mem 0  (unmarshal(receive():IntSet.t))
then print "y" else print "n"
```

pauillac[*P10a*] | glia[*P10b*] $\surd$

2.11 Breaking abstractions (directed case)

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.

For example, suppose we have a program that uses stored values of *IntSetStruct* and we wish to upgrade both the implementation and the stored values, changing the representation type from binary search trees to red-black trees. The new implementation would have a module declaration:

```
module IntSet2  =
  struct
    type t = int rbtree      IntSetStructRBT
    ...
  end
  : IntSetSig
```

A program to upgrade the stored values can be expressed as below, with an `Upgrade` module that has both types, coerced respectively to be equal to the old and new abstract types. (We are not proposing machinery to *automatically* apply the `upgrade` function.)

```
module IntSet  = IntSetStruct:IntSetSig
module IntSet2  = IntSetStructRBT : IntSetSig
module Upgrade =
  struct
     type t1 = int tree
     type t2 = int rbtree
     let upgrade = ...
  end : sig
    type t1
    type t2
    val upgrade : t1 -> t2
  end
    with  t1 =! Intset.t and t2 =! Intset2.t
...map Upgrade.upgrade over the stored values...
```

Note that the coercion does not require the signature of `Upgrade` to coincide with those of `IntSet` and `IntSet2`. The compiler only checks that `IntSet.t` is represented by `int tree` and `IntSet2.t` by `int rbtree`.

2.12 Forcing generativity

Dually, sometimes it is desirable to *force* a type change between builds even when the code remains identical, to prevent confusion between old and new communicated values. For example, one may have several distributed deployments of the same application which should be kept logically isolated.

2.13 Effectful module initialisation

In our previous examples the components of modules are all values. Generalising this to arbitrary expressions (as ML does), an abstract type definition can be dependent on some computation with side effects.

For example, consider an `NCounter` module that reads its step value from standard input when initialised; the invariant of any instance is then that any value of its `NCounter.t` is a multiple of this step. Two instances of the module can obviously have different invariants, and so marshalling from one to another should fail. Thus each run of a program containing `NCounter` should have an incompatible type `NCounter.t`.

2.14 Marshalling functions and rebinding

In this paper we deal only with marshalling of *closed* values; the semantics ensures that all module and expression declarations are substituted in before a `marshal` operation takes place. Marshalling of functions is therefore semantically straightforward.

A full language should, however, provide some form of *dynamic rebinding* of identifiers when they are unmarshalled, both to achieve the desired semantics where local resources have different behaviour in different contexts, and for performance reasons where much code is shared (and so should not be communicated). The paper [4] addresses dynamic rebinding, in the absence of type abstraction.

3. Solution: hash types as global names

This section introduces our solution informally, from both implementation and semantic viewpoints.

As we have seen, type-safe and abstraction-safe unmarshalling requires some run-time type representation in marshalled values, to permit a dynamic type comparison.

Our solution is based on the observation that hashing module definitions provides a global namespace for abstract types: if an identical module is hashed during builds of two different programs at different sites, the same hash will be obtained. Thus the programs share names for any abstract type provided they share the source code of the module that declares the type (and of its dependencies); no communication (e.g. of GUIDs) is needed at build time.

We regard hashes literally as types — hashes appear as a clause in the type grammar. They do not appear in source programs, but are inserted during compilation; as we shall see in more detail, the compiler replaces occurrences of an abstract type such as `IntSet.t` by the hash of the definition of `IntSet` that is in scope. Semantically, we work with ideal hashing, with a formal syntactic construction `hash(...)`. Implementations would realise this with an actual hash function; we discuss the low-level properties of hashes in Sec. 3.5.

At run-time, after this compile-time type substitution, the types in `marshal(`*e*`:`*T*`)` and `unmarshal(`*e*`:`*T'*`)` are closed, without free module identifiers or type variables. They can therefore be easily represented as byte strings, communicated across the network or stored in a persistent store, and can be compared with simple string equality.

We ensure that this dynamic equality precisely mirrors the static notion of provable type equality by carefully tuning the way in which hashes are generated and used; we show below that our system achieves this. Unmarshalling is therefore not only type-safe, but also abstraction-safe.

The standard operational semantics for abstract types forgets about abstraction as computation proceeds, substituting in representation types and operations. Here, in contrast, we need a run-time semantics that maintains abstraction throughout, both (1) so that our type preservation theorems tell us that abstractions are not broken; and (2) to support the proof that static and dynamic type equality coincide. After a module is reduced away, module code (which may see through the abstract type of that module) is intermixed with body code (which must treat the type as abstract). We therefore use a syntactic construct, *coloured brackets*, adapted from the work of [10], to delimit the regions in which different type equivalences hold. This is not purely a proof technique, however: in some subtle cases the coloured brackets within hashes are needed in compile-time hash generation to correctly distinguish abstract types that would otherwise be aliased. We show that implementations can erase coloured brackets outside hashes after compilation.

3.1 Simple examples

We illustrate the use of hashes in a simple case by referring back to the example of Sec. 2.4, in which a single program, *P4*, was built and run on the two machines. The build process is modelled in our semantics by type-checking, as usual, followed by reductions that substitute out module definitions, inserting hashes as required. Hash generation is deterministic, and hence the result of building *P4* on the two machines is identical. The program has a single module definition. It has a compile-time reduction as below, to an 'executable' *P4'*. (Note that for clarity of exposition, we omit coloured brackets from all reductions until Sec. 3.3; the example reductions as stated are not all type-preserving without them.)

```
P4 =
module IntSet = IntSetStruct:IntSetSig
if ...on-machine-pauillac... then
  send (marshal (IntSet.singleton 17 : IntSet.t))
else
  if IntSet.mem 17 (unmarshal(receive():IntSet.t))
  then print "y" else print "n"
```
$\longrightarrow_c$ (compilation)
```
if ...on-machine-pauillac... then
  send (marshal (singleton-code 17 : h ))
else
  if mem-code 17 (unmarshal (receive () : h))
  then print "y" else print "n"
= P4'
```
where

`h = hash(module IntSet=IntSetStruct:IntSetSig,t).`

Here the definitions of `IntSet.singleton` and `IntSet.mem` have been substituted for their occurrences, and the global name *h* has been substituted for type `IntSet.t`. Notice that *h* is constructed from the entire definition of `IntSet`, including the textual name `IntSet`, the implementation structure *IntSetStruct*, the interface *IntSetSig*, and the type field name `t`. In this simple example `IntSet` has no dependencies, so one can think of hashing its source text; we will discuss later the more interesting case of modules with dependencies, and also the question of exactly what form the hash function takes. Our liberal use of substitution is, of course, a semantic device — in practice compilation would use other representations.

At run-time, the two machines `pauillac` and `glia` execute their independently-compiled copies of *P4'*. Their shared knowledge of the hash *h* acts as a certificate that they may safely share values of their respective abstract types `IntSet.t` and `IntSet.t`.

```
   pauillac[P4'] | glia[P4']
```
$\longrightarrow^*$ (local computation on `pauillac` and `glia`)
```
   pauillac[send(marshal(singleton-code 17:h))]
 | glia[if mem-code 17 (unmarshal(receive():h))
        then print "y" else print "n"]
```
$\longrightarrow^*$ (local computation on `pauillac`, to get v)
```
   pauillac[send(marshalled( v:h ))]
 | glia[if mem-code 17 (unmarshal(receive():h))
        then print "y" else print "n"]
```
$\longrightarrow$ (communication)
```
   pauillac[ () ]
 | glia[if mem-code 17 (unmarshal(marshalled(v:h):h))
        then print "y" else print "n"]
```
$\longrightarrow$ (on `glia`: dynamic type check h=h, succeeds)
```
   pauillac[ () ]
 | glia[if mem-code 17 v
        then print "y" else print "n"]
```
$\longrightarrow^*$ (on `glia`: computation, prints `"y"`)
```
  pauillac[ () ] | glia[ () ]
```

Ultimately, only strings may be communicated across a network. The notation `marshalled` $(v:T)$ denotes a string literal containing representations of value v and its type T. This is only meaningful, and only used, where v and T are both closed.

Notice that the dynamic check is simple: just that the type *h* sent from `pauillac` is identical to the type *h* written into the `unmarshal` on `glia` at compile time. Yet, by virtue of the construction of these hashes, this is sufficient to guarantee both type-safety and abstraction-safety.

Consider now the programs of Sec. 2.1–2.7. How do hashes of modules provide the desired behaviour? In the case of concrete types, the comparison is obvious. For *P1*, `int`=`int`; for *P2*, `string`$\neq$`int`; for *P3*, for no hash *h* do we have `int`=*h*. As we have already seen, in the *P4* case the two programs share an identical hash *h*. For *P5*, in *P5a* and *P5b* the computed hash *h* for `IntSet.t` is identical (in fact the same *h* as above). Thus the *h* substituted for `IntSet.t` in *P5a*'s call to `marshal` will be identical to that in *P5b*'s call to `unmarshal`, and the communication will again succeed, exactly as we desire.

Although *P6a* (Sec. 2.6) contains a type `IntSet.t`, it is clear that the hash *h'* of the modified module `IntSet` differs from the hash *h* of the original module, correctly reflecting the difference in the modules' behaviour. The two programs will, correctly, be unable to communicate; an exception will be raised at the point of the `unmarshal`.

The example of Sec. 2.7 shows why one might wish the textual name (in general, the path) of a module to be included in its hash, along with the module body. The two modules `Euro` and `Pound` are identical in all but name, and so a hash that did not include the name would treat them as interchangeable, clearly leading to dangerous economic confusion, and furthermore differing from the usual semantics of ML-like languages. On the other hand, the programmer should also be able to specify that a name is *not* to be considered part of the module's identity. This can be done simply by having an additional form of module declaration, `module* N = ...` , for which hashing uses a canonical name `*`, not admissible in source programs, instead of the actual name `N`. In this simple scheme both sender and receiver must use the `*`'d form, of course.

3.2 Module dependencies

The example of Sec. 2.8 shows that the same module text defines a different abstract type if its dependencies change, which means that the hash of a module must depend on the hashes of its dependencies. In our substitutive reduction semantics, type dependencies are handled automatically: we have substituted hashes for any types of earlier modules before constructing the hash of a module that depends on them. We shall see how term dependencies are also automatically taken into account. Consider the following (a simplification of *P8a*):

```
module A=struct type t=bool    let x=true   end
         : sig type t          val x:t      end
module B=struct type t=A.t*int let x=(A.x,3) end
         : sig type t          val x:t       end
send (marshal (B.x : B.t))
```
$\longrightarrow_c$ (compilation)
```
module B=struct type t=h*int let x=(true,3) end
         : sig type t       val x:t         end
send (marshal (B.x : B.t))
```
$\longrightarrow_c$ (compilation)
```
send (marshal ((true,3) : h'))
```
where
```
h = hash (
module A=struct type t=bool   let x=true   end
         : sig type t         val x:t      end,t)
h' = hash (
module B=struct type t=h*int  let x=(true,3) end
         : sig type t         val x:t        end,t)
```

Here the hash *h'* for `B` is constructed after the hash *h* for `A` has been substituted for `A.t`, and after the term part `true` has been substituted for `A.x`. It is clear that if `A` changed, h would change, and so h' would change. This would still be true in the (unlikely) case that `B` mentions `A.t` but not `A.x`.

We must also ensure h' depends on h in the (common) case that `B` mentioned `A.x` but not `A.t`, i.e. where `A` is used in `B` only to implement an internal computation. The coloured brackets of the following section will conveniently suffice for this.

3.3 Abstraction-preserving reduction

Some reductions in Sec. 3.1, 3.2 require non-standard type equalities to make them type-preserving. For example, to type the intermediate state in Sec. 3.2 we must have `(true,3)` of type h`*int`, hence we need a type equality identifying h with its representation type `bool`.

We could allow this type equality to be used anywhere, but instead prefer to delimit more precisely which subterms can see through any particular abstraction. We introduce *coloured brackets*, adapted from the work of [10], during module reduction. In the previous example, the first reduction will actually replace $(\mathtt{A.x}, 3)$ by $([\mathtt{true}]_h^h, 3)$ instead of just `(true,3)`. The brackets serve two purposes. First, the lower annotation (the *colour*) is a hash h, indicating that the additional type equivalence h `== bool` is available when typing the *inside* of the bracketed expression. This equivalence is drawn from the structure of h, viz. h`=hash(module A= struct type t=bool...end:...,t)`. Thus, inside the brackets we have `true:`h. Second, the upper h annotation is the type of the bracketed expression as seen from the *outside*, thus reduction is type preserving. (One would often have a more complex type in the upper annotation, not just a hash, e.g. $[\mathtt{(true,3)}]_h^{h\mathtt{*int}}$.)

The reduction semantics of our formal system moves brackets around as required to ensure that abstraction is preserved throughout reduction, and so our type preservation result (Thm. 4.1) covers abstraction. If we did not use brackets but allowed hash type equalities to be used freely, abstraction would become invisible after reduction. The use of brackets also simplifies the statement of our result relating static and dynamic type equality (Thm. 4.7). Moreover, when compiling a module that refers to a term field of a previous one, the presence of brackets ensures that the hash of the later module does indeed depend on the hash of the earlier module.

3.4 Modest implementation demands

Few changes are required in an ML-like language to support the strategy outlined above.

Thm. 4.5 shows that type checking is decidable and that hashes play no role in compile-time type-checking of source code. In particular, we can use traditional type checking and inference algorithms essentially unchanged. Compile-time reduction only builds hashes, without ever looking inside one. Run-time reduction only ever compares hashes by string equality.

Thm. 4.6 shows that almost all coloured brackets and type information can be erased before run-time, with the exception of course of marshal and unmarshal type annotations, and brackets within hashes.

ML-like languages usually support separate compilation of modules. Typically, a compilation phase takes a module and the signatures of the modules it imports and generates code parameterised by these dependencies. For λ_{hash}, the compilation phase would also generate a hash parameterised by the hashes of the imported modules, in other words a hash-to-hash function. An appropriate compositional implementation of hashing must be used to make these efficiently representable. Typically, linking instantiates the parameterised code with jumps to the code of previous modules. For λ_{hash}, the linking phase would do two further things. First, it would patch the type annotations for `marshal` and `unmarshal` in the code by replacing references to module types by their hashes. Second, it would calculate the hash of the module by applying the hash-to-hash function (generated by compilation) to the hashes of previous modules.

3.5 Low-level details of hashes

In our semantics, we work with *ideal* hashing, taking a free constructor `hash(...)` which can be applied to elements of the abstract syntax. We can think of `hash` as a function whose injectivity guarantees abstraction-safety. To avoid communicating large quantities of source code, an implementation would reify `hash` with a fixed-length hash function, giving a safety guarantee that is only as strong as the probability of the absense of collisions.

This must be chosen so that (1) collisions are rare, and (2) hashes are not too costly to compute.

Both MD5 (RFC1321, 128-bit) and SHA-1 (RFC3174, 160-bit) are sufficiently cheap, and may be considered random functions for this application [24]. Let us consider the likelihood of collisions. For n abstract types and N possible hash values, the probability of a collision is approximately $n^2/2N$. Pessimistically assuming 10^{10} programmers in the world, writing 300 lines of code per day with one abstract type per 100 loc, the probability of a collision in a century of abstract types (using MD5) would then be $(10^{15})^2/2^{129} \approx 10^{-9}$. This is much less than the probability of a cosmic-ray-induced processor error in this period.

It may be desirable to have an absolute guarantee of type-safety, while accepting probabilistic abstraction-safety. To achieve this, one could pair hashes with the corresponding underlying representation types. At the other extreme, one could accept a probabilistic guarantee even at simple types, by sending only a hash of the marshalled type. These choices must depend on a risk assessment.

Note that our proposal is aiming to protect only against accidental errors during programming and software deployment, not against malicious attack, and so we are not concerned with deliberate searches for collisions. Protecting against spoofed messages requires largely orthogonal techniques, e.g. message signatures and/or encryption, that are not in the scope of this paper. Moreover, we do not address the problem of communication between untrusting peers, where one must check not just that the type advertised by the peer is compatible with the local type, but also the validity of the byte string's claim to represent a value of the advertised type (see, e.g., [23]).

We hash elements of the abstract syntax, not concrete syntax, for two reasons. Firstly, it ensures hashes are not dependent on, e.g., the choice of newline or newline/CR, or on comments. Secondly, it fits well with the rest of the semantics — recall we must calculate hashes of modules that are the results of module substitutions. In practice optimised calculations would be possible, without requiring the explicit construction of canonical representatives of abstract syntax elements.

Hashing abstract syntax, which we take up to alpha-equivalence, has the (benign) consequence that abstract type equality is not dependent on the names of function parameters. We have both internal (alpha-convertible) and external *module* names in the semantics; external names must be meaningful between programs.

4. Formal system

Our calculus describes networks of machines. Each machine executes a program; a program consists of a sequence of module declarations followed by an expression. The expression language consists of a simply-typed call-by-value λ-calculus with module field references, marshalling, and communication of strings.

Consider a program containing a module declaring an abstract type. There is an abstraction boundary between the module's body and the rest of the program. Inside the boundary, the type's representation is visible; thus the type is said to be *transparent*. Outside, the type's representation is not visible, thus the type is *opaque*. Our calculus tracks this abstraction boundary as reduction proceeds. Compilation replaces the abstract type by a *hash* h and wraps the code e that comes from inside the module definition with *coloured brackets* decorated by h, as in $[e]_h^T$. The distinction between opaque and transparent views is therefore witnessed by the brackets: inside the brackets, we view h as transparent; when outside h is opaque.

In order to express this distinction in our inference rules, we decorate each judgement with a *colour* hm, as in $E \vdash_{hm} e{:}T$. The colour has one of two forms: it can be a hash h, in which case h is transparent and all other hashes are opaque; or it can be the empty colour $\bullet$, in which case all hashes are opaque.

The reader is referred to the companion technical report [16] for the full semantics of the calculus and proofs of results.

4.1 Relation to the informal discussion

For brevity, we take a module language in which structures are type/term pairs, rather than general dependent records from the earlier informal development. The following table summarises the correspondence between the informal and formal module syntax.

`struct type t =` T_0 `let y =` $v^\bullet$ `end`	$\leftrightarrow \quad [T_0, v^\bullet]$
`sig    type t        val y :` T `end`	$\leftrightarrow \quad [X{:}\mathbf{Type}, T]$
`sig    type t =` T_1 `val y :` T `end`	$\leftrightarrow \quad [X{:}\mathbf{Eq}(T_1), T]$

We split module names into two parts, an external name N and an alpha-convertible name U. We write module declarations as $\mathbf{module}\, N_U \ = M{:}S \ \mathbf{in}\ m$, where U binds in m and N neither binds nor is subject to binding. The user would write only one identifier, which would be used for both. External names play no role in the static type system; they are used in hash construction and hence in dynamic type checks.

The formal system omits `=!` coercions and run-time generativity, which should be straightforward extensions. Functors are also omitted, though we include most of the technical machinery they require, expressing abstract and manifest types in signatures using singleton kinds. In Sec. 6.2 we propose extensions for treating these omissions.

4.2 Syntax

We let x, X and U range over expression, type and module variables.

Networks:

$n ::= 0 \ \mid \ m \ \mid \ n|n$

Machines (whole programs):

$m ::= e \ \mid \ \mathbf{module}\, N_U \ = M{:}S\ \mathbf{in}\ m \quad (U \text{ binds in } m)$

Modules:

$M ::= [T, v^\bullet]$ structure ($v^\bullet$ is a value)

$S ::= [X{:}K, T]$ signature (X binds in T)

Types:

$T ::= \mathrm{UNIT} \mid \mathrm{INT} \mid \mathrm{STRING}$	base types
$\mid X \mid T \to T \mid T * ... * T$	variable, function, product
$\mid U.\mathrm{TYPE}$	type part of a module
....	
$\mid h$	hash

Hashes:

$h ::= \mathbf{hash}(N, M{:}[X{:}\mathbf{Type}, T])$	hash
$hm ::= h \mid \bullet$	colour ("hash maybe")

Kinds:

$K ::= \mathbf{Type}$	kind of all types
$\mid \mathbf{Eq}(T)$	kind of types statically equal to T

Expressions:

$e ::= () \mid \underline{n}$	unit, integers
$\mid (e, ..., e) \mid \mathbf{proj}_i\, e$	tuple, projection
$\mid x \mid \lambda x{:}T.e \mid e\ e$	lambda calculus (x binds in e)
$\mid U.\mathrm{term}$	value part of a module
$\mid \mathbf{mar}\,(e{:}T)$	marshalling primitive
$\mid \mathbf{unmar}\ e{:}T$	unmarshalling primitive
$\mid !\,e \mid ?$	send and receive
....	
$\mid \mathbf{marshalled}\,(e{:}T)$	result of marshalling
$\mid \mathbf{UnmarFailure}$	exception caused by $\mathbf{unmar}$
$\mid [e]_{hm}^T$	coloured bracket

User source programs are closed terms of the m grammar which do not contain any of the constructs below the dotted lines.

Values v^{hm} are indexed by a colour. They are defined formally below; they include usual λ-calculus values, $\mathbf{marshalled}\,(v^\bullet{:}T)$ and "necessary" brackets around values. For closed $v^\bullet$, the value $\mathbf{marshalled}\,(v^\bullet{:}T)$ is a string, the sequence of bits that represents the value $v^\bullet$ and the type T.

We work up to alpha-conversion. We write substitutions as follows: $\{x{\leftarrow}e\}A$ replaces x by e in A; we also define substitutions on module components, as in $\{U.\mathrm{TYPE}{\leftarrow}T, U.\mathrm{term}{\leftarrow}e\}A$.

4.3 Static and dynamic semantics

The static type system for programs has judgements for subkinding, type equality, and subsignaturing relations. Module structures M and names U have signatures S, expressions and machines have types T, and types have kinds K. The system also defines correctness of colours hm, environments E, kinds K, and signatures S.

$E \vdash_{hm} K <: K'$	$E \vdash_{hm} T == T'$	$E \vdash_{hm} S <: S'$
$E \vdash_{hm} M{:}S$	$E \vdash_{hm} U{:}S$	$E \vdash_{hm} e{:}T$
$E \vdash_{\bullet} m{:}T$	$E \vdash_{hm} T{:}K$	$\vdash hm\,\mathrm{ok}$
$E \vdash_{hm} \mathrm{ok}$	$E \vdash_{hm} K\,\mathrm{ok}$	$E \vdash_{hm} S\,\mathrm{ok}$

The typing rules are largely standard; the novel rules will be explained below. Recall that judgements are annotated by a colour hm, i.e. an optional hash — the idea being that derivations of judgements annotated by a hash h can make use of the equality between the abstract type h and its implementation.

Type environments may contain bindings for module, type and expression variables. Earlier variables bind in later types, kinds and signatures.

$E ::= \ \mathbf{nil} \mid E, x{:}T \mid E, X{:}K \mid E, U{:}S$

Static typing of networks, $\vdash n\,\mathrm{ok}$, simply means that all machines are well-formed.

We define compile-time reductions $m \longrightarrow_{\mathsf{c}} m'$ of machines (performed after type checking), and run-time reductions $e \longrightarrow_{hm} e'$ and $n \longrightarrow n'$ for expressions and networks.

4.3.1 Singleton kinds

Following [17, 11, 28], we use singleton kinds to handle abstract and concrete signatures in a uniform way. We have two families of kinds: $\mathbf{Type}$ is the kind of all types; and, for any type T, $\mathbf{Eq}(T)$ is the *singleton kind* of all types that are provably equal to T.

A module consists of a structure $[T_0, v^\bullet]$ and a signature $[X{:}K, T]$. The structure has a representation type T_0 and a value $v^\bullet$ — think of a tuple of operations. This $v^\bullet$ must have the type $\{X{\leftarrow}T_0\}T$, and the implementation type T_0 must have the kind K. This is made precise by the following rule:

$$\frac{\begin{array}{ll} E \vdash_{hm} T_0{:}K & E, X{:}K \vdash_{hm} T{:}\mathbf{Type} \\ E \vdash_{hm} v^\bullet{:}T' & E, X{:}\mathbf{Eq}(T_0) \vdash_{hm} T' == T \end{array}}{E \vdash_{hm} [T_0, v^\bullet]{:}[X{:}K, T]} \ (\mathsf{MS.struct})$$

For an abstract module we have $K = \mathbf{Type}$, revealing no information about the representation type, whereas for a concrete module, commonly $K = \mathbf{Eq}(T_0)$, revealing it. This is captured with the type equality relation: in the context of a module declaration $\mathbf{module}\, N_U = M{:}[X{:}K, T]$ $\mathbf{in}$ _, one can use the path $U.\textsc{type}$ to refer to the type part of the module. If it is concrete, with $K = \mathbf{Eq}(T_0)$, one can further use the type equality $U.\textsc{type} == T_0$, whereas if it is abstract $U.\textsc{type}$ is typically not equal to any other type.

The subkinding relation $K <: K'$ places $\mathbf{Type}$ above all singleton kinds. This is used to define subsignaturing and hence, using subsumption, allows a concrete module can be used as if it had an abstract signature.

4.3.2 Hash formation, type equality of hashes

At run-time, we need globally meaningful type names for abstract types, corresponding to the $U.\textsc{type}$ paths used in compile-time type checking. We construct these global names by hashing (well-typed) closed abstract modules, together with the associated external name.

$$\frac{\vdash_\bullet [T_0, v^\bullet]{:}[X{:}\mathbf{Type}, T]}{\vdash \mathbf{hash}(N, [T_0, v^\bullet]{:}[X{:}\mathbf{Type}, T])\ \mathrm{ok}} \ (\mathsf{hmok.hash})$$

As explained informally earlier, judgements annotated by a hash permit an additional type equality: under the colour $h = \mathbf{hash}(N, [T_0, v^\bullet]{:}[X{:}\mathbf{Type}, T])$, h is equal to its implementation T_0:

$$\frac{E \vdash_h \mathrm{ok}}{E \vdash_h h == T_0} \ (\mathsf{Teq.hash})$$

These two rules examine the internal structure of hashes, which might be thought to be computationally problematic. However, while they are semantically necessary, they play no role in user program type-checking (Thm. 4.5) or in execution.

4.3.3 Compile-time reduction and coloured brackets

Module reduction constructs the type representations that will be used at run-time in marshalling and unmarshalling. Reducing a concrete module is simple: we replace references to its type component by its manifest type, and references to its term component by the value inside the module.

$$\begin{array}{l} \mathbf{module}\, N_U = [T_0, v^\bullet]{:}[X{:}\mathbf{Eq}(T_1), T]\ \mathbf{in}\ m \\ \qquad\qquad \longrightarrow_{\mathsf{c}} \quad \{U.\textsc{type}\leftarrow T_1, U.\mathrm{term}\leftarrow v^\bullet\}m \end{array}$$

When it comes to abstract types, things are more interesting. Given an abstract module declaration $\mathbf{module}\, N_U = [T_0, v^\bullet]{:}[X{:}\mathbf{Type}, T]$, we normally have no way of referring to its type other than by name, i.e. $U.\textsc{type}$. However U is not meaningful on other machines, which motivates the introduction of the *hash* of the module, i.e. $h = \mathbf{hash}(N, [T_0, v^\bullet]{:}[X{:}\mathbf{Type}, T])$. Then module reduction replaces references to the type component by h. References to the term component are replaced by the value suitably protected by *h-coloured brackets*, which embody the abstraction boundary around the module's body as discussed above.

$$\begin{array}{l} \mathbf{module}\, N_U = [T_0, v^\bullet]{:}[X{:}\mathbf{Type}, T]\ \mathbf{in}\ m \\ \qquad\qquad \longrightarrow_{\mathsf{c}} \quad \{U.\textsc{type}\leftarrow h, U.\mathrm{term}\leftarrow [v^\bullet]_h^{\{X\leftarrow h\}T}\}m \end{array}$$

In general, in a bracket expression $[e]_{hm}^T$, the lower annotation hm is a colour that indicates what type equalities may be used to type e. If $hm = \mathbf{hash}(N, [T_0, v^\bullet]{:}[X{:}\mathbf{Type}, T])$, then the equality $hm == T_0$ is available when typing e, through (Teq.hash) (Sec. 4.3.2). If $hm = \bullet$, e is typable without any extra equalities. The upper annotation T is the externally visible type of e. The following rule (the only typing rule that mentions brackets) shows this colour change formally.

$$\frac{E \vdash_{hm'} T{:}\mathbf{Type} \qquad E \vdash_{hm} e{:}T}{E \vdash_{hm'} [e]_{hm}^T{:}T} \ (\mathsf{eT.col})$$

4.3.4 Expression reduction

Expression reduction is based on a standard call-by-value λ-calculus semantics. In this subsection, we give the function application rule and bracket-pushing rules. In later subsections, we show the rules for marshalling and communication.

As we show in Thm. 4.6, brackets can be erased before run-time reduction. However, the brackets' presence is necessary for type preservation (Thm. 4.1). Given their presence, we need reduction rules to "push" them inwards so that the brackets do not interfere with computationally significant reductions (Thm. 4.3). To describe the bracket pushing rules, and to achieve type preservation, it is necessary to index the reduction relation, class of values, and reduction contexts by colours.

We write v^{hm} for a value of colour hm. Brackets may appear in a value when used to build a value of an abstract type out of a value of the corresponding implementation type, for example $[3]_h^h$, where the implementation type of h is $\textsc{int}$.

$$\begin{array}{rl} v^{hm} ::= & \underline{n} \mid () \mid (v^{hm}, ..., v^{hm}) \mid \lambda x{:}T.e \\ \mid & \mathbf{marshalled}\,(v^\bullet{:}T) \mid [v^{h_1}]_{h_1}^{h_1} \text{ where } h_1 \neq hm \end{array}$$

The following bookkeeping rules push brackets with manifestly decomposable types inside expressions, and remove them where not necessary.

$$\begin{array}{rcl} [\underline{n}]_{hm'}^{\textsc{int}} & \longrightarrow_{hm} & \underline{n} \\ [()]_{hm'}^{\textsc{unit}} & \longrightarrow_{hm} & () \\ [(v_1^{hm}, ..., v_j^{hm})]_{hm'}^{T_1 * ... * T_j} & \longrightarrow_{hm} & ([v_1^{hm}]_{hm'}^{T_1}, ..., [v_j^{hm}]_{hm'}^{T_j}) \\ [\lambda x{:}T.e]_{hm'}^{T' \to T''} & \longrightarrow_{hm} & \lambda x{:}T'.[\{x\leftarrow [x]_{hm}^{T'}\}e]_{hm'}^{T''} \\ [\mathbf{marshalled}\,(v^\bullet{:}T)]_{hm'}^{\textsc{string}} & \longrightarrow_{hm} & \mathbf{marshalled}\,(v^\bullet{:}T) \end{array}$$

$$[[v^{h_1}]_{h_1}^{h_1}]_{h_2}^{h_1} \longrightarrow_{hm} [v^{h_1}]_{h_1}^{h_1} \quad \text{if } h_1 \neq h_2 \wedge h_2 \neq hm$$

$$[v^{hm_1}]_{hm_1}^{h_2} \longrightarrow_{hm} v^{hm_1} \quad \text{if } hm_1 = hm \vee hm_1 = \bullet$$

Function application introduces brackets to protect the argument, since the formal parameter may itself be used under a bracket in the body of the function. This is a variant of [10], where the formal parameter has to be used at the colour of the function itself.

$$(\lambda x{:}T.e)\, v^{hm} \longrightarrow_{hm} \{x\leftarrow [v^{hm}]_{hm}^T\}e$$

4.3.5 Marshalling

As in [1], $\mathbf{mar}\,(e{:}T)$ "tags" the value of e with a type annotation T, producing a result of type $\textsc{string}$. The dual construct $\mathbf{unmar}\ e{:}T$ produces a value of type T, which the type tag in e must (dynamically) match.

$$\frac{E \vdash_{hm} e{:}T}{E \vdash_{hm} \mathbf{mar}\,(e{:}T){:}\textsc{string}} \ (\mathsf{eT.mar})$$

$$\frac{E \vdash_{hm} T{:}\mathbf{Type} \qquad E \vdash_{hm} e{:}\textsc{string}}{E \vdash_{hm} (\mathbf{unmar}\ e{:}T){:}T} \ (\mathsf{eT.unmar})$$

There is a subtlety here: in the conclusion of (eT.mar), the fact that e has the type T may require the extra type equality provided by hm. Hence we introduce $\mathbf{marshalled}\,(e'{:}T)$, which requires the argument to be not only closed but typable in $\bullet$, i.e. everywhere. Reduction transforms $\mathbf{mar}\,(v^{hm}\!:T)$ into $\mathbf{marshalled}\,([v^{hm}]^T_{hm}\!:T)$, where the brackets serve to ensure that any type equality provided by hm is always available to type v^{hm} (even after sending the marshalled value to another machine). Note that before reducing $\mathbf{mar}\,(v^{hm}\!:T)$, both v^{hm} and T will have been closed by substitution.

$$\mathbf{mar}\,(v^{hm}\!:T) \longrightarrow_{hm} \mathbf{marshalled}\,([v^{hm}]^T_{hm}\!:T)$$

$$\frac{E \vdash_{hm} \text{ok} \qquad \vdash_\bullet e\!:T}{E \vdash_{hm} \mathbf{marshalled}\,(e\!:T)\!:\text{STRING}} \text{ (eT.marshalled)}$$

The unmarshalling of a string first extracts the type tag T from the string and compares it with the tag for the expected type T'. Since T is a valid type for $v^\bullet$ in $\bullet$, it is also one in hm. The type tags T and T' are compared by *syntactic equality*: if the types match, the original value is extracted from the string; otherwise an exception is raised. This dynamic type equivalence is closely related to static equivalence (Thm. 4.7).

$$\begin{array}{lll} \mathbf{unmar}\,(\mathbf{marshalled}\,(v^\bullet\!:T)\!:T') & & \\ & \longrightarrow_{hm} v^\bullet & \text{if } T = T' \\ & \longrightarrow_{hm} \mathbf{UnmarFailure} & \text{otherwise} \end{array}$$

4.3.6 Programs and networks

A machine consists of a series of module declarations followed by an expression. Each module declaration may refer to the previous ones.

$$\frac{E \vdash_\bullet T\!:\mathbf{Type} \qquad E \vdash_\bullet M\!:S \qquad E, U\!:S \vdash_\bullet m\!:T}{E \vdash_\bullet (\mathbf{module}\, N_U \,=M\!:S\ \mathbf{in}\ m)\!:T} \text{ (mT.let)}$$

A network is a parallel juxtaposition of machines. Note that each machine has its own environment: there is no explicit scope that encompasses more than one machine.

$$\frac{\vdash n_1\ \text{ok} \qquad \vdash n_2\ \text{ok}}{\vdash n_1 \mid n_2\ \text{ok}} \text{ (nok.par)} \qquad \frac{\vdash_\bullet m\!:\text{UNIT}}{\vdash m\ \text{ok}} \text{ (nok.mach)}$$

We assume that there is a single channel, which carries values of type STRING. The expression $!\,e$ sends the value of e over that channel, and ? reads a value from that channel. Communication is straightforward as all the work required to make values and types intercomprehensible is done by the marshalling apparatus; for suitable evaluation contexts $CC^\bullet_{hm_1}$ and $CC^\bullet_{hm_2}$ we have just the rule below, writing context application with a dot.

$$CC^\bullet_{hm_1}.!\,v^{hm_1} \mid CC^\bullet_{hm_2}.? \longrightarrow CC^\bullet_{hm_1}.() \mid CC^\bullet_{hm_2}.v^{hm_1}$$

4.4 Results

First, our calculus enjoys type preservation and progress properties.

Theorem 4.1 (type preservation for compile-time, expression, and network reduction)

- if $m \longrightarrow_{\mathsf{c}} m'$ and $\vdash_\bullet m\!:T$ then $\vdash_\bullet m'\!:T$;
- if $e \longrightarrow_{hm} e'$ and $\vdash_{hm} e\!:T$ then $\vdash_{hm} e'\!:T$; and
- if $n \longrightarrow n'$ and $\vdash n$ ok then $\vdash n'$ ok.

Theorem 4.2 (progress for compile-time reduction) If $\vdash_\bullet m\!:\text{UNIT}$ then either

- m is an expression; or
- m reduces, i.e. there exists m' such that $m \longrightarrow_{\mathsf{c}} m'$.

Moreover, compile-time reduction is terminating.

Theorem 4.3 (progress for expressions) If $\vdash_{hm} e\!:T$ then one of the following holds:

- e is a value, i.e. there exists v^{hm} such that $e = v^{hm}$;
- e reduces, i.e. there exists e' such that $e \longrightarrow_{hm} e'$;
- e is blocked waiting for I/O, i.e. there exists $CC^{hm}_{hm_2}$ and e' such that $e = CC^{hm}_{hm_2}.!\,e'$ or $e = CC^{hm}_{hm_2}.?$; or
- e has thrown an exception, i.e. there exists $CC^{hm}_{hm_2}$ such that $e = CC^{hm}_{hm_2}.\mathbf{UnmarFailure}$.

In addition, we have proved a normalisation result for expressions, showing that the rules for coloured brackets do not introduce any divergencies.

Both compile-time machine reduction and run-time expression reduction are deterministic (network reduction is not, of course):

Theorem 4.4 (determinacy for compile-time and expression reduction)

- If $m \longrightarrow_{\mathsf{c}} m'$ and $m \longrightarrow_{\mathsf{c}} m''$ then $m' = m''$; and
- if $e \longrightarrow_{hm} e'$ and $e \longrightarrow_{hm} e''$ then $e' = e''$.

For static type checking:

Theorem 4.5 (decidability of type checking) Type checking is decidable. Furthermore, user source programs can be typed by derivations involving no hashes or coloured brackets.

At run-time, all type annotations except those on $\mathbf{mar}$, $\mathbf{marshalled}$, and $\mathbf{unmar}$ can be erased. Moreover, all coloured brackets can be erased except for those that occur within a hash within one of those remaining annotations. More precisely, we define $\text{erase}(e)$ to be e with all type annotations and brackets erased except that the type annotations on $\mathbf{mar}$, $\mathbf{marshalled}$, and $\mathbf{unmar}$ are left unchanged. We define $\underset{\text{erase}}{\longrightarrow}$ to be like $\longrightarrow_{hm}$ by taking the erase-image of the left- and right-hand sides of each rule (and removing rules that would become $e \underset{\text{erase}}{\longrightarrow} e$).

Theorem 4.6 (erasure preserves reduction outcomes) Assume $\vdash_\bullet e\!:T$. We have that $e \longrightarrow_\bullet e'$ implies $\text{erase}(e) \underset{\text{erase}}{\longrightarrow}^{\leqslant 1} \text{erase}(e')$. Conversely, $\text{erase}(e) \underset{\text{erase}}{\longrightarrow} e_0$ implies that there exists e' such that $\text{erase}(e') = e_0$ and $e \longrightarrow^{\geqslant 1}_\bullet e'$.

Note that brackets are needed in module reduction, to keep track of a module's ancestors as we build its hash.

Finally we show that, under reasonable conditions, static and dynamic type equality coincide. Let D be a module declaration context:

$$\mathbf{module}\, N_{0\,U_0} \,= M_0\!:S_0\ \mathbf{in}\ ...\mathbf{module}\, N_{j\,U_j} \,= M_j\!:S_j\ \mathbf{in}\ \_$$

in the user source language (with no brackets or hashes). Consider a machine $D.C.e$ for some expression context C and an expression $e = (\mathbf{unmar}\,(\mathbf{mar}\,(e_0\!:T_0)\!:T_1))$. One would like this dynamic type check to succeed if and only if T_0 and T_1 are statically provably equal, i.e. iff $U_0\!:S_0, ..., U_j\!:S_j \vdash_\bullet T_0 == T_1$.

Write σ_D for the accumulated substitution defined by the module reduction rules for D (we omit an explicit definition for lack of space). The dynamic check is then $\sigma_D\, T_0 = \sigma_D\, T_1$. We have:

Theorem 4.7 (coincidence between dynamic and static type checking) Suppose that $D.C.e$ is well formed (i.e. $\vdash_\bullet D.C.e\!:\text{UNIT}$), that it contains no hashes, and that its external names $N_0, ..., N_j$ are distinct. Let $E = U_0\!:S_0, ..., U_j\!:S_j$ be the associated environment. Assume that T_0 and T_1 contain no hashes and $E \vdash_\bullet T_i\!:\mathbf{Type}$ for $i = 0, 1$. Then $E \vdash_\bullet T_0 == T_1$ iff $\sigma_D\ T_0 = \sigma_D\ T_1$.

The requirement that the external names $N_0, ..., N_j$ be distinct rules out the rather pathological programs in which there are two module definitions with the same name, one shadowing the other, which have identical structures, signatures, and dependencies. The exclusion of hashes is automatic for user source programs.

One can imagine stronger theorems, relating type equality between two programs that share a common (DAG-)prefix of module definitions, but their statements become rather elaborate.

5. Related work

Modules and generativity There is an extensive literature on ML-style modules, including [19, 21, 11, 17, 28, 7], much of it discussing subtle questions of generativity versus applicativity. To our knowledge, however, none deals with the inter-program case. In [26], fresh type names are generated during call-by-value module reduction, with ν-binders that can extrude across distributed scope. This allows inter-program sharing, and also a `with!` coercion, but at the pragmatically-awkward cost of requiring particular object files to be shared.

Type dynamic Our marshal and unmarshal operations are essentially constructors and destructors for values of dynamic type; **mar** is just `dynamic`, and **unmar** is a restricted form of `typecase`. Our dynamic values have type STRING, emphasising that they may be communicated readily. Type `Dynamic` was first formalised by Abadi et al. [1, 2], who also gives a historical survey. Intensional polymorphism [12, 31] permits run-time type analysis of *all* values.

Marshalling abstract types The problem of marshalling values of abstract (existential) type has not been satisfactorily addressed theoretically before. In several systems, abstract types are run- or build-time generative, so that two executions or builds of the same source will yield distinct types. While communication within such a program can be abstraction-safe, successful communication between builds can only be at the representation type, and hence abstraction-unsafe. This is true, for instance, of [2], TMAL [8], Modula-3 [6, 5], Alice [3], and the typed-channel languages listed below.

Weirich [32] exposes an existential's representation type to type analysis, permitting a type-safe polytypic marshalling function to be written. As future work we hope to expose our global type names at term level (cf. [13]), permitting an *abstraction-safe* polytypic marshalling function to be written. Furuse and Weis [9] argue for ignoring abstraction, checking representation types only.

A number of programming languages feature some form of built-in marshalling (pickling, serialisation, etc.): for example Modula-3, Alice, Java, .NET, and OCaml. Most of these languages serialise the type along with the value in order to permit a check at unmarshal time, and represent the type by a hash. Languages differ, however, in exactly what is hashed — i.e., in what is considered when deciding type equality.

In Modula-3, abstract types are made opaque by *branding*, which may be either by a literal string (analogous to an external name) or a compiler-generated unique identifier. The latter are unique within a program but not necessarily related between programs, so explicit brands must be used for inter-program communication; however, they do not guarantee abstraction-safety for that case. *Revelation* can be used to make an abstraction transparent.

In Alice, abstract type creation is run-time generative, meaning that abstract types from different executions are always distinct. This vacuous abstraction-safety forces the use of representation types for pickling between different programs.

In Java serialisation [29], class equivalence is on fully-qualified class name, the representation type of all fields, and the types of all non-private methods; the implementation is not considered in type equality. A strong coercion (Sec. 2.10) is provided (although compatibility of representation types is not checked until unmarshal time).

In .NET serialisation [20], class equivalence is on the textual name along with the implementation of the entire assembly in which it is defined (a single DLL or EXE, which may comprise many source files). This guarantees data structure invariants are maintained, as in our approach; however, we work on the much finer scale of individual modules, and furthermore we require only source code to be shared, not object files.

OCaml [22] does no typechecking for marshalling at all, and hence is not even type-safe. When unmarshalling a function, it verifies (by a hash) that the communicating builds are identical, thus allowing the code pointers of all closures to be communicated literally.

Coloured brackets Coloured brackets were introduced in [33, 10]; we differ in that we permit a variable to occur in a colour other than the one where it is defined. Our proofs are harder, our β-rule has to introduce extra brackets, but our brackets carry only a single optional hash, rather than a list of hashes. Rossberg [25], like us, is concerned to preserve the opacity of abstract types under reduction due to the presence of typecase. His coercions serve the same purpose as our brackets, but his use of the closed-scope `open` construct instead of dot notation prevents any possibility of sharing values of abstract type between instances.

Typed channels Several languages, e.g. JoCaml [14], Nomadic Pict [27], Facile [30, 15], implement *typed channels*. These permit type- and abstraction-safe communication once the channel is established. Establishing a channel at an abstract type, however, requires the endpoints somehow to share the type already; in the case that the endpoints reside in different programs or instances, this requires an unsafe cast, usually performed (outside the language) by a name server.

6. Conclusions and future work

6.1 Summary

We have proposed a novel and expressive design for guaranteeing type- and abstraction-safe marshalling of data sent between distributed ML programs, that can uniformly treat manifest, abstract, and generative types. The key technical idea is to use hashes of module declarations as globally-meaningful type names, which are inserted at compile-time and then compared dynamically when unmarshalling. We add coloured brackets to delimit the "abstraction boundary" within which hashes are transparent, tracking these brackets through the reductions so as to achieve type and abstraction preservation. Our proposal is a smooth extension of existing ML-like languages: type checking is unchanged, most type information can be erased before run-time, and the dynamic type check closely mirrors static ML type equivalence.

6.2 Future work

In the future, we aim to broaden our solution to be applicable to full-scale languages.

The following extensions will be required to cope with the examples in Sec. 2.9–2.13. The *strong coercion* (Sec. 2.10 and 2.11) used for forcing an abstract type to have the same hash as an earlier module, has a simple compile-time implementation: check the representation types of the two are provably equal, then simply reuse the hash of the earlier module as the type name for the new. This requires the compiler to keep a mapping from hashes to representation types, which is straightforward. *Programmer-requested generativity* (Sec. 2.12) can be dealt with in an implementation by generating a fresh global name (say a random bit string of the same length as hashes) at compile time; its semantics can be modelled by ν-binding. Both this and the strong coercion are very similar to the constructs in our earlier work [26]. *Side-effect-induced generativity* (Sec. 2.13) requires a way to identify simple pure computations in structure bodies that the programmer can easily understand; abstract types of structures with pure computations should be hashes, whereas those of structures with effectful computations should have freshly-generated names. *Functors* (Sec. 2.9) are a more substantial extension, but, at least for a restricted but useful class, should be straightforward. Consider first-order applicative functors [18] and module expressions that are either (i) an explicit structure, possibly multiply-abstracted, or (ii) *pure*, i.e. constructed from module identifiers, abstraction and application. These give rise to *functions* from hashes to hashes; applying these functions gives run-time representations of the compile-time path-based type names.

Other substantial extensions also need to be considered. *Dependent record structures*, i.e. module structures with multiple fields also appear in this paper's informal examples; they should be conceptually straightforward. *Parametric and substructuring polymorphism* within the dynamic check would allow receivers to accept a more general type than that offered by the sender. This is a more substantial extension; it will be a challenge to minimise the transmitted type information required for these dynamic "subtype" checks. One may want to *rebind* (Sec. 2.14) identifiers within a transmitted value to avoid the overhead of sending code already available at the other end, or to obtain location-specific behaviour; here we aim to integrate hash types with [4]. Marshalling *reference cells* exhibits related problems: should the reference be rebound, made remote, or duplicated? More generally, one must consider values mentioning other machine resources: screens, files…

We wish to integrate our work with existing systems for distributed programming which have statically typed channels for normal operation but no safe way of initiating communication, such as JoCaml [14] and Nomadic Pict [27]. We also wish to test the expressiveness of our marshalling primitives by using them to write libraries for safe distributed communication and persistence.

Acknowledgments We acknowledge support from a Royal Society University Research Fellowship (Sewell), EPSRC grant GRN24872 (Wansbrough), EC FET-GC project IST-2001-33234 PEPITO, and APPSEM 2. The authors thank Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and the anonymous referees for their suggestions.

7. REFERENCES

[1] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically typed language. *ACM TOPLAS*, 13(2):237–268, 1991.

[2] M. Abadi, L. Cardelli, B. Pierce, and D. Rémy. Dynamic typing in polymorphic languages. *J. Functional Programming*, 5(1):111–130, 1995.

[3] T. Alice Project. Alice manual: Pickling. `http://www.ps.uni-sb.de/alice/manual/pickling.html`, 2003.

[4] G. Bierman, M. Hicks, P. Sewell, G. Stoyle, and K. Wansbrough. Dynamic rebinding for marshalling and update, with destruct-time λ. In *Proc. ICFP 2003*, 2003. Full version available as UCAM-CL-TR-568. `http://www.cl.cam.ac.uk/~pes20/`.

[5] A. Birrell, G. Nelson, S. Owicki, and E. Wobber. Network objects. *Software – Practice & Experience*, 25(S4):87–130, 1995. Available in slightly different form as SRC-115 revised.

[6] A. Z. Broder et al. Fingerprint.i3. `http://research.compaq.com/SRC/m3sources/html/fingerprint/src/Fingerprint.i3.html`, 1994.

[7] D. Dreyer, K. Crary, and R. Harper. A type system for higher-order modules. In *Proc. 30th POPL, New Orleans*, pages 236–249, 2003.

[8] D. Duggan. Type-safe linking with recursive DLLs and shared libraries. *ACM TOPLAS*, 24(6):711–804, 2002.

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

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

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

[12] R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In *Proc. 22nd POPL*, pages 130–141, 1995.

[13] M. Hicks, S. Weirich, and K. Crary. Safe and flexible dynamic linking of native code. In *Proc. 3rd Workshop on Types in Compilation*, pages 147–176, 2000.

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

[15] F. Knabe. *Language Support for Mobile Agents*. PhD thesis, Carnegie Mellon University, Dec. 1995.

[16] J. J. Leifer, G. Peskine, P. Sewell, and K. Wansbrough. Global abstraction-safe marshalling with hash types. Technical Report RR-4851, INRIA Rocquencourt, 2003. Available from `http://pauillac.inria.fr/~leifer/research.html`. Also published as UCAM-CL-TR-569.

[17] X. Leroy. Manifest types, modules, and separate compilation. In *Proc. 21st POPL*, pages 109–122, 1994.

[18] X. Leroy. Applicative functors and fully transparent higher-order modules. In *Proc. 22nd POPL*, pages 142–153, 1995.

[19] D. MacQueen. Modules for Standard ML. In *Proc. 1984 ACM Symp. LISP and Func. Prog.*, pages 198–207, 1984.

[20] Microsoft Corporation. .NET Framework developer's guide: Serializing objects. `http://msdn.microsoft.com/library/en-us/cpguide/html/cpovrserializingobjects.asp`, 2001.

[21] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[22] Objective Caml. `http://caml.inria.fr`.

[23] B. Pierce and E. Sumii. Relating cryptography and polymorphism. `http://web.yl.is.s.u-tokyo.ac.jp/~sumii/pub/`, July 16 2000. Substantially revised version to appear in J. Comp. Security.

[24] M. J. B. Robshaw. On recent results for MD2, MD4 and MD5. *RSA Laboratories' Bulletin*, (4), Nov. 12 1996.

[25] A. Rossberg. Dynamic opacity for abstract types. Technical report, Programming Systems Lab, Universität des Saarlandes, 2002. `http://www.ps.uni-sb.de/Papers/abstracts/opaque.html`.

[26] P. Sewell. Modules, abstract types, and distributed versioning. In *Proc. 28th POPL*, pages 236–247, 2001.

[27] 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.

[28] C. A. Stone and R. Harper. Deciding type equivalence in a language with singleton kinds. In *Proc. 27th POPL*, pages 214–227, 2000.

[29] Sun Microsystems. Java object serialization specification 1.4.4. `http://java.sun.com/j2se/1.4.1/docs/guide/serialization/`, 2002.

[30] B. Thomsen, L. Leth, and T.-M. Kuo. A Facile tutorial. In *CONCUR'96, LNCS 1119*, pages 278–298, 1996.

[31] S. Weirich. Type-safe cast: Functional pearl. In *Proc. ICFP, Montreal*, pages 58–67, 2000.

[32] S. Weirich. Higher-order intensional type analysis. In *Proc. 11th ESOP, LNCS 2305, Grenoble, France*, 2002.

[33] S. Zdancewic, D. Grossman, and G. Morrisett. Principals in programming languages: A syntactic proof technique. In *Proc. ICFP, Paris*, pages 197–207, Sep 1999.

$Revision: 1.212 $ $Date: 2003/06/25 14:38:47 $