Type-Safe Distributed Programming for OCaml

John Billings Peter Sewell Mark Shinwell Rok Strniša

Computer Laboratory, University of Cambridge

https://www.cl.cam.ac.uk/users/pes20/hashcaml

Abstract

Existing ML-like languages guarantee type-safety, ensuring memory safety and protecting the invariants of abstract types, but only within single executions of single programs. Distributed programming is becoming ever more important, and should benefit even more from such guarantees. In previous work on theoretical calculi and the Acute prototype language we outlined techniques to provide them for simple languages.

In this paper we put these ideas into practice, describing the *HashCaml* extension to the OCaml bytecode compiler, which supports type-safe and abstraction-safe marshalling, together with related naming constructs. Our contribution is threefold: (1) We show how to define globally meaningful runtime type names for key OCaml type constructs that were not covered in our previous work, dealing with the generativity issues involved: user-defined variant and record types, substructures, functors, arbitrary ascription, separate compilation, and external C functions. (2) We support marshalling within polymorphic functions by type-passing, requiring us to build compositional runtime type names and revisit the OCaml relaxed value restriction. We show that with typed marshalling one must fall back to the SML97 value restriction. (3) We show how the above can be implemented with reasonable performance as an unintrusive modification to the existing OCaml language, implementation, and standard libraries. An alpha release of HashCaml, capable of bootstrapping itself, is available, along with an example type-safe distributed communication library written in the language.

Categories and Subject Descriptors D.3.1 [**Programming Languages**]: Formal Definitions and Theory; D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms Languages, Reliability, Security

Keywords HashCaml, Objective Caml, type-safe marshalling

1. Introduction

The development of functional programming languages such as ML and Haskell demonstrates — and rests upon — the use of expressive type systems. These provide strong guarantees, ensuring not just memory safety but also that high-level abstractions introduced by the programmer are respected. Unfortunately such guarantees are only provided within single executions of single programs: when data is transferred between executions, either via the network or via persistent storage, one loses the support of the standard type systems. Ironically, this is just at points where early detection of errors would be most useful, as in the distributed or persistent setting, with multiple versions of programs interacting, there is great potential for confusion. Further, it is, in principle, an unnecessary loss, as for two programs to interact usefully there must be *some* shared knowledge of the structure of any data that will be exchanged. How, then, should we design typed programming languages to make best use of this shared knowledge, to detect errors as early as possible, and thereby to support distributed programming as well as ML and Haskell support local computation?

In previous work we have studied several aspects of this problem, developing theoretical calculi that construct globally meaningful runtime type names freshly [31] and based on code hashes [22], calculi of dynamic rebinding [4], and a prototype language, Acute, to explore the design space further [33, 34].

In the current work we further develop these theoretical ideas and put them firmly into practice. We have designed and implemented an extended version of the OCaml 3.09.1 bytecode compiler, of Leroy et al. [24]. This *HashCaml* extension supports type-safe marshalling, including marshalling values of abstract types, marshalling within polymorphic functions, and various constructs for distributed naming. Our contribution is threefold:

(1) We show how to define globally meaningful runtime type names for key OCaml type constructs that were not covered in our previous work, dealing with the generativity issues involved: user-defined variant and record types, substructures, functors, arbitrary ascription, separate compilation, and external C functions (§4).

(2) We support marshalling within polymorphic functions by type-passing, requiring us to build compositional runtime type names and revisit the OCaml relaxed value restriction. We show that with typed marshalling one must fall back to the SML97 value restriction (§5).

(3) We show how the above can be implemented, with reasonable performance, as an unintrusive modification to the existing OCaml language, implementation, and standard libraries (§6).

We prefix these in §3 with an overview of the main problems and a summary of the HashCaml features, and in §7 we detail an example demonstrating how the new constructs support typed distributed programming: a library written in HashCaml that provides polymorphic, typed, local and distributed, channel-based communication. We discuss related work in §8 and conclude in §9. A preliminary alpha release of HashCaml is publicly available [6] and, except where otherwise stated, the code in this paper is taken from working examples in the distribution.

The addition of type-safe marshalling is in one sense a modest change to OCaml — the language syntax and static type system are largely unchanged, as is the semantics of (almost all) existing programs. The semantics of type-safe marshalling, however, cuts

ML'06 September 16, 2006, Portland, Oregon, USA.

across much of the language: all its types and values. We must consider what the dynamic type equality is across all of the existing type system, and also between different program scopes; and we need reasonable behaviour for marshalling arbitrary values of the language. We are proposing what we consider to be reasonable choices for OCaml, but many points are equally applicable to SML or other languages with type abstraction. For future language designs these questions should be considered from the outset: one would like the static and dynamic type equalities to coincide exactly, where they are compatible, but, (as we will show) the existing OCaml (or SML) static equality is not always suitable for distributed programming.

2. Background

To make this paper self-contained, we briefly recapitulate the main line of argument from our earlier work [34], referring the reader there for a full discussion.

1. Marshalling There are many desirable communication and persistent storage abstractions, with very different properties (synchronisation, performance, reliability, secrecy, authentication, etc.). Rather than build a fixed selection into a programming language, we think it better to provide *marshalling* (or *serialization*) of arbitrary values to byte strings; various communication and storage abstractions can then be provided as libraries above this.

2. Type-Safe Marshalling For early detection of errors, marshalling should be *type-safe*. In particular, unmarshalling should fail if the type of the marshalled value is not what is expected (this is a dynamic failure, but it should be at unmarshal-time rather than during later computation). Accordingly, marshalled values must contain a *runtime type representation*. For example, this should succeed (ignore the `[]` and `0` arguments):

```
let s = Marshal.to_string  false  []
let b = true && (Marshal.from_string s 0)
```

whereas the following code should raise an exception at the `unmarshal`, when an equality test fails between the type representation in the marshalled value (`int`) and that of the expected type (`bool`). (In OCaml the unmarshal succeeds, producing a value of type `bool` which is not equal to `true` or to `false`.)

```
let s = Marshal.to_string  17 []
let b = true && (Marshal.from_string s 0)
```

Realistic examples would typically have the two lines of code in two distinct programs. Note that the types should be inferred where possible, as above.

3. Abstraction-Safe Marshalling and Globally Meaningful Type Names For a language with simple types one could ensure type-safety by choosing some runtime type representation that is isomorphic to the source-language abstract syntax of types. However, idiomatic ML programming makes frequent use of programmer-defined abstract types; the standard type system guarantees that within a single execution any invariants of these types are protected. To maintain that same level of protection across a distributed system our previous work proposed building globally meaningful type names for abstract types in two ways, either *freshly* (pseudo-random long bitstrings) or from *hashes* of the modules that define them. For example, in Acute one could define an abstract type `EvenCounter.t` as below; the runtime type name for this type was essentially h`.t`, where h was the hash of the module definition (in fact of the abstract syntax of the structure and signature, up to alpha equivalence and type equality, and taking module dependencies into account). This ensures that the invariant of `EvenCounter.t`, that its values will always be non-negative even numbers, is preserved even when its values can be marshalled between programs.

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

These runtime type names were built so that common cases of interoperation between different programs, which are not identical but do share the definitions of any types used for interaction, would just work — without any need to exchange runtime type names during development.

4. Parametric Polymorphism and Type Passing Idiomatic ML programming also makes heavy use of parametric polymorphism, and many of the anticipated uses of marshalling are within polymorphic functions — for example, within the implementation of a function `send : fd -> 'a -> unit` that takes a TCP socket file descriptor and a value to send, marshals the value, and sends it over the TCP connection. Each execution of this marshal requires a runtime type representation for the type at which `send` is instantiated. In Acute, parametric polymorphism was via explicit type abstractions and applications, so runtime type representations automatically flowed to the marshal and unmarshal points where they were needed.

3. Putting it into practice: HashCaml

The Acute language was an experimental prototype, not a production system. The language was large enough to write nontrivial examples but omitted many standard features — most importantly, the module system was rudimentary and there were no user-definable datatypes. The implementation runtime was an interpreter over the abstract syntax of the semantics, which made it easy to maintain in tight correspondence with the Acute definition, but was very slow (around three orders of magnitude slower than OCaml bytecode). The library support was very limited.

Problems Our main goal in the current work is to integrate type-safe and abstraction-safe marshalling into a production language, OCaml. There are three main problems that must be addressed, which we introduce below and solve in §4, §5 and §6.

Globally Meaningful Type Names Firstly, we must consider what runtime type names to use for types that involve the OCaml type features not covered in our previous work: user-defined variant and record types, substructures, functors, arbitrary ascription, and separate compilation. We have to balance a tension between:

(1) making dynamic type equality (induced by unmarshal-time comparison of our runtime type names) correspond to the existing OCaml static type equality;

(2) making the dynamic type equality useful for distributed programming; and

(3) doing this at a reasonable implementation cost (both runtime performance and compiler complexity).

The key question here is the *generativity* of type definitions — when should two similar type definitions give rise to compatible types. This is a fundamental and much-discussed issue in the design of static typing for datatypes and modules [20]; as we shall see, the distributed setting introduces new questions and affects the best choices.

Parametric Polymorphism and Type Passing Secondly, in contrast to the explicit polymorphism of Acute, in OCaml (as in ML in general), polymorphism is implicit in the term language. There is little type information available from the existing runtime structures in the implementation. We provide the runtime type information needed at marshal and unmarshal points within polymorphic functions by a type-passing translation, inserting explicit type-representation abstractions and applications during compilation. For example, we compile

```
let f : 'a list -> int
  = function xs -> ...
```

to something like

```
let f : typerep -> 'a list -> int
  = function tvar_a xs -> ...
```

As we shall see, this gives rise to semantic questions related to the compositionality of type representations, the treatment of non-ground types, and the value restriction. For the latter, we show that the addition of type-safe marshalling forces a switch from the OCaml relaxed value restriction of Garrigue [15] to the SML97 value restriction [40].

Implementation Thirdly, we need an implementation design that fits well with the existing OCaml implementation, providing reasonable performance without requiring radical changes to the existing language, compiler or runtime. This is intertwined with the previous problems: in places the user-visible semantics of our new constructs arises from a pragmatic compromise between (1), (2), and (3) above. Accordingly, in §4 and §5 we describe enough of our implementation strategy to motivate these user-visible choices. Internal implementation problems are addressed in §6, especially the problem of designing an internal type representation that is compositional but also high-performance, without (say) recomputing actual type hashes at every use of generalized value identifiers. There are also challenging implementation questions arising from pattern matching, signature coercions, normalisation before hashing, and the OCaml imperative generalization code.

HashCaml: Overview We solve these problems in the context of HashCaml, an extension of the OCaml bytecode compiler. Native code compilation and the interactive top level are not supported, but the standard OCaml tools (`ocamllex`, etc.) work, with the exception of `camlp4`, `ocamldoc`, and `ocamldebug` (to the best of our knowledge there would be no serious problems for these). The compiler can bootstrap itself. The implementation is intended to cover all of the standard part of the language except marshalling of polymorphic variants and objects, for which warnings will be issued.

The changes with respect to OCaml are not very intrusive. For the language itself, existing OCaml programs that do not use marshalling should behave identically except that (1) we impose the standard value restriction on type generalization, which is slightly more restrictive than the restriction implemented in OCaml, and (2) there is some performance cost. We return to both points in detail later in the paper. Programs that use marshalling type-correctly should typically just work. As for the implementation, the patch consists of a set of fairly small changes to the vanilla OCaml source tree, plus additional C and OCaml source files of around 5 000 lines.

HashCaml: Summary of New Language Constructs We briefly summarise the new constructs here, explaining in the following sections how they are used. OCaml provides standard library functions for marshalling arbitrary values to byte strings, but (as the documentation warns) this marshalling is not type-safe: *"Anything can happen at run-time if the object in the file does not belong to the given type."* In HashCaml the behaviour of this OCaml standard library module `Marshal` has been modified to make marshalling type-safe and abstraction-safe: attempts to unmarshal a value at the wrong type will cause a runtime exception. The signature is unchanged, and the old unsafe behaviour is available via a module `Oldmarshal`.

Runtime type representations for abstract types are built variously from hashes of module definitions or from pseudo-random numbers, at runtime or compile time. The former is the default; the latter is available by annotating structure definitions `fresh` or `cfresh`, or for an entire implementation file (corresponding to a module of the same name) writing `typemode cfresh` or `typemode fresh` as the first item in the source file.

There is a new built-in abstract type `typerep` of runtime type representations, with constructs `rep(`*t*`)` to build the representation of a type *t* and `dyntype(`*e*`)` to build the representation of the runtime type of an expression.

There is a new family of types `'a name`, represented as 256-bit values. Expression-level names can be generated in several ways:

(1) freshly: just write `fresh` (of type `'a name`);
(2) from a pair of a type representation and a string, writing `hashname(`*t*`,` *s*`)`, where *t* is a type and *s* is a string, to yield a value of type *t* `name`;
(3) using the module hashes produced for calculating type representations, for example `fieldname M.f` (of type *t* `name` where `M.f`:*t*); and
(4) by applying name coercions `namecoercion(`*path1*`,`*path2*`,`*e*`)` where *path1* and *path2* are type constructor paths (referring to type constructors of the same arity) and *e* has type `[`*ty1*`...`*tyn*`]` *path1* `name`, yielding a value of type `[`*ty1*`...`*tyn*`]` *path2* `name`.

There is also a special conditional for comparing names:

`ifname` *e1*`=`*e2* `then` *e3* `else` *e4*

where *e1*:*t1* `name`, *e2*:*t2* `name`, *e3*:*t*, *e4*:*t*, and *e3* is type-checked in an environment where *t1* and *t2* are unified.

4. Globally Meaningful Type Names

In traditional static type systems for ML-like languages there is a nontrivial notion of static type equality, and the literature contains much discussion of generativity and sharing for abstract types. Turning to their dynamic behaviour, these languages were designed so that implementations could erase type information before execution, and work on ML-like module systems typically has either no dynamic semantic definition or one over a type-erased language (or one in which abstraction boundaries are erased). When we add typed marshalling, however, we have to deal with two type equality relations: the static one of the type system and the dynamic relation checked at unmarshal time. One might seek to ensure simply that the two coincide exactly, but, unfortunately, the situation is rather more complex than this. Firstly, in the static type system one is only concerned with comparison of two types that are simultaneously in scope (perhaps indirectly via value identifiers) within a single program, whereas with marshal/unmarshal, one can pull typed values out of *any* scope, from different executions of different programs, and compare them. Secondly, the existing OCaml static type equality seems to be somewhat finer than is desirable for distributed programming.

In this section we go through the features of the OCaml static type system (except polymorphic variants and objects) that were not covered in our previous work, showing for each how we define runtime type names that give a reasonable dynamic equality. The HashCaml static type system is identical to that of OCaml, except for the (straightforward) static typing of the new constructs and the value restriction. The HashCaml dynamic type equality does not coincide exactly with this, but in many simple cases, where two types can be compared both statically and dynamically, the two equalities will coincide.

Throughout we discuss the semantics by example. The existing OCaml does not have a complete formal definition (type system or operational semantics) and we have not produced one for HashCaml. However, we are partly guided by our earlier formal work, especially the full semantics defined for Acute [32].

Structures and Module Ascription In SML and OCaml abstract types are introduced primarily by signature ascription. In the example below, `M1.t` is abstract and not statically compatible with `M.t` (which *is* statically compatible with `int`), so the definition of `id` does not typecheck.

```
module M
= struct
    type t = int
    let x = 3
  end
module M1 : sig type t  val x:t end = M
let id : M.t -> M1.t = fun z->z
```

In HashCaml the same is also true dynamically — the unmarshal below fails, but unmarshalling `s` at type `M.t` or `int` would succeed.

```
module M
= struct
    type t = int
    let x = 3
  end
module M1 : sig type t  val x:t end = M
let s = Marshal.to_string M.x []
let (z : M1.t) = Marshal.from_string s 0
```

The static type systems of Leroy [23] and Harper and Lillibridge [19] provide this static type behaviour essentially by using the source-code paths of abstract types (e.g. `M1.t`) as the compile-time names of abstract types. In HashCaml, as in our earlier work, we can provide corresponding dynamic type behaviour using a runtime type name h`.t` for `M1.t`, where h is a hash of the module definition (so runtime type name equality checking captures any invariants of the abstract type). This is very similar to the Acute example of §2, but in Acute a module definition and ascription were tied together, whereas in OCaml and HashCaml they are separate constructs.

Abstract types with Effectful Definitions Some abstract types have invariants which depend on IO or store effects, e.g. the n-counter below from [33]. For these, hash generation of runtime type names will not guarantee preservation of invariants; instead the programmer can specify that a runtime type name should be dynamically created, freshly at module initialisation time, with the `fresh` annotation.

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

One can also specify `cfresh` for a compile-time-fresh name. We leave this choice, of hash (the default), fresh, or cfresh naming, entirely to the programmer. One might think that the language should guarantee invariant preservation by performing some valuability analysis, forbidding the hash and cfresh cases if module initialisation is effectful, but the Acute example libraries for Nomadic Pict and Ambient-style communication showed that this is not desirable — one may need type compatibility between different (distributed) runtime instances of abstractions that each have local store.

The `myname` Implementation Strategy In the preceding example the `fresh` annotation is, perhaps surprisingly, associated with a structure rather than with a module declaration or signature ascription form. This is a pragmatic choice: in the OCaml implementation, structures are compiled essentially to records, and a new ascription, e.g. `module M :` *Sig* `= M'`, has essentially no existence at runtime (except perhaps for coercion functions) — it does not build a new record. There is therefore no convenient place to keep runtime type data per-ascription, and so we keep it with the original structure, simply adding a single new '`myname`' field that contains either the hash or fresh/cfresh name associated with the structure. This field is invisible to user code. Wherever we need the runtime type name of an abstract `M.t`, we generate lambda code (an OCaml compiler intermediate representation) that essentially builds the pair `(M.myname, "t")`, and whenever we need to build a module hash that mentions `M` we generate lambda code that refers to `M.myname`.

Ascription, again A visible consequence of this `myname` implementation is that, while in SML and OCaml multiple ascriptions give rise to statically different types, in HashCaml multiple ascriptions give dynamically *compatible* types. For example, the following does not compile in OCaml

```
module M = struct type t = int  let x = 3 end
module M1 : sig type t val x:t end = M
module M2 : sig type t val x:t end = M
let id : M1.t -> M2.t = fun x -> x
```

whereas in HashCaml the unmarshal below succeeds.

```
module M = struct type t = int  let x = 3 end
module M1 : sig type t val x:t end = M
module M2 : sig type t val x:t end = M
let s = Marshal.to_string M1.x []
let (z : M2.t) = Marshal.from_string s 0
```

One could bring the static and dynamic type equality into sync by either (i) changing the HashCaml compilation of ascriptions to really generate new records, potentially with new names, or (ii) change the static type system to be more liberal. As we wish to keep the HashCaml implementation a small diff to that of OCaml, and we suspect that uses of multiple ascription to intentionally generate new abstractions are rare, we do neither, accepting the discrepancy.

Substructures and Subascriptions OCaml supports nested structures, potentially with ascriptions at each level, as in the example below.

```
let s1=ref "" ;;  let s2=ref "" ;;  let s3=ref "" ;;
module M1
: sig
    module M2
     : sig type t  val x:t end
  end
= struct
    module M2  : sig type t  val x:t  val f:t->t end
    = struct
        type t = int
        let  x = 2
        let  f = fun z->z+2
        let  _ = (s3 := Marshal.to_string (3:t) [])
      end
      let _=(s2:=Marshal.to_string(M2.f M2.x:M2.t) [])
  end;;
let _ = (s1:=Marshal.to_string (M1.M2.x : M1.M2.t) [])
```

Here the `struct` defining `M2` lies under two ascriptions. Statically, there really are three different types here: inside the body of `M2`'s `struct` one can do anything with an `int` (so there is no invariant); inside the body of `M1`'s `struct` one can treat `M2.t` only abstractly, but with both `x` and `f` operations (which ensure the 'even' invariant); outside the definition of `M1` one can treat `M1.M2.t` only abstractly and only with operation `x` (so the invariant that all values of `M1.M2.t` are equal to `2` is guaranteed).

To the best of our knowledge this expressivity is not widely used to deliberately protect strengthened invariants, but the general pattern arises often. In particular, an OCaml compilation unit in file `foo.ml` implicitly defines a module `Foo` ascripted to `foo.mli` (if that exists); the body of `foo.ml` itself often contains ascripted module definitions.

Note that, outside the definition of `M1`, conventional module type systems (and the OCaml implementation) do not record the existence of the inner ascription: one just has a type environment assigning `M1` the signature `sig module M2:sig type t val x:t end end`.

Statically, the three types `t`, `M2.t` and `M1.M2.t` exist in distinct scopes and the question of comparing them does not arise. With marshalling, however, one can pull values out of their scope (as shown) and (not shown) pull them back in with unmarshalling and perform dynamic type equality tests, so the runtime types used for the three marshals are critical. In HashCaml these type representations are (hashes of) `int`, *h2*`.t` and *h1*`.M2.t` respectively, where *h2* and *h1* are the values of the `myname` fields of `M2` and `M1`. All three are distinct, so marshalling and unmarshalling across the abstraction boundaries will fail.

This seems to be the most useful semantics, but it could also be desirable to be able to marshal a concrete value (within an abstraction boundary) at an abstract type that it would have outside, e.g. marshalling `3` above within `M2` at type `M2.t` or `M1.M2.t`. How best to specify such behaviour in the source program is unclear, though, as (i) there may be many possible dynamic types to choose from, and (ii) the relevant paths are not in scope at the marshal point.

Substructures and Prefix Hashing For equality testing of runtime type names to guarantee anything about the invariants of abstract types, when building module hashes (that these type names are constructed from) we have to take account of their dependencies. With substructures there are several forms of dependency: (1) references to external modules via their paths; (2) references to identifiers declared in the prefix of a superstructure; and (3) the dependency of a structure on its substructures. Case (1) is dealt with uniformly by referring to the `myname` fields of those modules, as in the previous paragraph. For (3) every module also depends on the `myname`s of its submodules; so in the example below `M.myname` depends on `N.myname` and `O.myname`, in addition to the hash of its normalized structure and the `myname`s of its external dependencies (i.e. `List.myname`).

```
module M
= struct
    let iterate = List.iter
    let x = 7
    module N = struct  let y = x  end
    module O = struct  let w = 4  end
  end
```

For (2) we have a design choice. One could make the hash of a module take account of all of the superstructure prefix, so `N.myname` would involve hashing the definitions of `iterate` and `x`, and `O.myname` would involve hashing the definitions of `iterate`, `x`, and `N`. This would be pleasingly regular but awkward in practice: recall that OCaml `.ml` files essentially define structures, so any definition of an abstract type in a structure at the top level of an `.ml` file, e.g. an application of the standard library `Set.Make` functor, would always depend on all of the preceding definitions in the file. We therefore include prefix definitions in a substructure hash only if there is some dependency. In more detail, if a module refers to its prefix, the module's `myname` depends on the hash of the normalized structure of the prefix, and on the external dependencies of the prefix. For example `N.myname` also depends on `hash(normalize(prefix(N)))`, since it refers to its prefix (`M.x`), and on `List.myname`. On the other hand, `O` is independent of its prefix and makes no external references, so its `myname` would be the same in every context.

Normalization w.r.t. Type Abbreviations The static type system operates up to type abbreviations, so within the scope of `type t=int` the types `t` and `int` can be used interchangeably. Our implementation should do the same, building both runtime type representations and module hashes. We believe this to be straightforward, but at the time of writing our implementation does not.

Functors OCaml functors are applicative. For example, in the scope of the definitions

```
module F (X:sig end) : sig type t end
                     = struct type t=int end
module U = struct end
module M = F(U)
module M'= F(U);;

((fun x->x) : M.t -> F(U).t);;
((fun x->x) : M.t -> M'.t);;
```

we have static type equalities `M.t = M'.t = F(U).t`. In HashCaml we echo this in the construction of hashes and hence in the dynamic type equality: by default the result structure of a functor application has a hash-built `myname` that depends on the `myname` fields of the argument structures. The static type system's path `F(U).t` thus matches the dynamic type equality — `M` and `M'` refer to two different runtime structures, but their `myname` fields have the same value, built out of the hash of `U` and the hash of the body of `F`, and so the runtime types for `M.t` and `M'.t` will be equal. Common cases, e.g. marshalling a value of a set or hashtable abstract type produced by applying an OCaml standard library functor, should therefore just work. The lack of normalisation w.r.t. type abbreviations can be problematic here, however, as variations in the behaviour OCaml type inference can lead to visible differences.

In the current implementation each usage of an application path (such as `F(U).t`) gives rise to another evaluation of the functor body. This has a performance cost, but more seriously means that functor bodies that involve effects, or are annotated `fresh`, will give erroneous behaviour. Further work is needed here.

Abstract Type Operators Abstract type operators are applicative in both static and dynamic semantics, as one would expect. For example, in

```
module M
= struct
    type 'a t = int * 'a
    let f x = (3,x)
  end
module M1 : sig type 'a t  val f:'a -> 'a t end = M
let s = Marshal.to_string (M1.f true) []
let (z : bool M1.t) = Marshal.from_string s 0
```

the runtime type name for (the type operator application) `bool M1.t` is essentially `((M1.myname,"t"),[bool])`.

Variant types In OCaml identical variant type definitions in different modules are not statically compatible, so the code below does not typecheck.

```
module M1
= struct
    type t = C of int
  end
module M2
```

```
= struct
    type t = C of int
  end
let f : M1.t -> M2.t = fun x->x
```

There are many plausible choices for how the dynamic type names of such types could be built. In increasing coarseness of dynamic type equality: (i) freshly at runtime, (ii) from a hash of the type definition and its path, (iii) from a hash just of the type definition, (iv) from a hash of a normalised definition, ignoring the order of clauses, (v) from a hash of the structure of the definition, ignoring the names of constructors, or (vi) any of the above, chosen per-type by the programmer. In any of the hash cases such a hash must properly take any dependencies on other types into account.

For HashCaml we adopt (iii), so the unmarshal below succeeds.

```
module M1
= struct
    type t = C of int
  end
module M2
= struct
    type t = C of int
  end
let s = Marshal.to_string (M1.C 3) []
let x = match (Marshal.from_string s 0) with M2.C(y)->y
```

Option (i), while fine for a single run of a single program, would prevent almost any interesting communication; option (v) would fail to catch many accidental errors; and option (vi) seems unduly complex. The choice amongst the others is debatable. We suspect that (ii) would be annoying in the distributed setting as it would require too much similarity between the module structure of two communicating programs. Here again, then, the desired dynamic type equality seems to be coarser than the OCaml static type equality, and in principle one might slightly change the latter to bring them into sync.

Order-normalisation, option (iv), would mean that the following unmarshal succeeds; we would like to implement this in future releases.

```
module M1
= struct
    type t1 = C1 of int | D1 of t2
    and  t2 = C2 of bool | D2 of t1
  end
module M2
= struct
    type t2 = D2 of t1 | C2 of bool
    and  t1 = D1 of t2 | C1 of int
  end
let s = Marshal.to_string (M1.C1 3) []
let x = match (Marshal.from_string s 0)
        with M2.C1(y) -> y | M2.D1 _ -> 3
```

Records Record types behave analogously: in OCaml they are statically generative, but we build their runtime type names from hashes of their definitions, at present not order-normalised, and not including their path.

Record fields in OCaml can have locally-universally-quantified type variables (as we saw used in §7), and therefore may need type passing. At the time of writing our implementation treats simple cases of this correctly but needs further work.

Separate Compilation The `myname`-based implementation means that separate compilation just works, without any need for extra computation at link time, again keeping the changes to the OCaml implementation small. Various optimisations would be possible with more link-time work. As compilation units are files rather than explicit `structs` we need a different syntax for letting the programmer specify whether the `myname` field should be hash, cfresh or fresh generated: we allow a `typemode cfresh` or `typemode fresh` to appear as the first item in a `.ml` file (the hash case is the default, as for explicit `structs`).

Standard Library The OCaml standard library is available in HashCaml without any special treatment: definitions of abstract types are produced by hashing as described above.

C functions The use of external functions raises two issues. Firstly, in building hashes of modules that refer to C functions we include the function name at usage points and any `extern` declarations. We do not include hashes of the associated C source files. Secondly, type passing: for a C function used at a HashCaml type that is higher-order and polymorphic, one may need to manually pass type representation arguments back and forth. One such had to be adapted in the standard library.

5. Polymorphism and Type Passing

As outlined in §3, in order to support marshalling and unmarshalling within polymorphic functions we must ensure that enough type information is available at runtime, at all marshal and unmarshal points. We do so with an explicit type-passing translation: roughly, at each polymorphic generalisation point we insert an additional lambda for each generalised type variable, and at each use of an identifier with a nontrivial type scheme we insert applications.

This has a performance cost, which we return to in §6 and §9, but also semantic implications, which we explain here.

Marshalled Type Representations Our runtime type representations in marshalled values are unstructured, simply 256-bit numbers. The marshalled type representation of an abstract type `M.t` is based on either a hash of the definition of `M` (and its dependencies) or a fresh name. Marshalled representations of concrete types are essentially hashes of their structure.

The only operation we need at runtime on these type representations is an equality test, at unmarshal time, so no additional structure is required.[1] Using hashing to build the marshalled representations of concrete types is not forced but it does simplify things, keeping the representations of small and constant size.

With marshalled type representations generated freshly and by hashing, the type safety guarantees provided by the language are predicated on the probability of collision being small and the hash function being good. We believe our use of SHA-256 hashes and pseudo-random 256-bit numbers suffices to make the probability of accidental collision negligible, e.g. compared to the probability of cosmic-ray-induced errors.

Compositionality of Runtime Type Representations The internal runtime representation of types used within a program must support different operations to that used in marshalled values: we must be able to (a) compute a marshalled type representation from an internal one, but also (b) construct internal representations of composite types from other internal representations. For example, the `pair_and_marshal` below involves a marshal at type `'a * 'a` where `'a` is instantiated to `int`. The construction of the internal representation for this type must be compositional in the type structure, otherwise the unmarshal, at type `int*int`, would fail. We show how this is achieved in §6.

```
let pair_and_marshal : 'a -> string
  = function x -> Marshal.to_string (x,x) []
let s = pair_and_marshal 17
let n = let (x1,x2)=(Marshal.from_string s 0) in x1+x2
```

[1] If unmarshalling involved checking specialisation of type schemes or subtype relationships, or to support intensional type analysis on the type representations in marshalled values, more structure would be needed.

Non-Ground Runtime Types HashCaml supports marshalling within polymorphic functions, but not marshalling *at* type schemes, because the latter (a) would require structured type representations, but also (b) would be out of step with the rest of OCaml, in which, as in SML, functions cannot be abstracted on parameters with proper type schemes. Hence the following fails in HashCaml, as the inferred type of the argument to `Marshal.to_string` is not ground.

```
let x = []
let s = Marshal.to_string x []
```

On the other hand the following succeeds, as type inference does infer a monotype for `x`

```
let f x =
  let s = Marshal.to_string x [] in
  x + 1 in
f 0
```

and the following, with an explicit type annotation, also succeeds.

```
let x=function y -> y
let s=Marshal.to_string (x:int->int) [Marshal.Closures]
```

In general it is hard to statically detect (un)marshals at non-ground types (an inter-module analysis would be required), so the error in the first case is dynamic, an exception at the marshal point. This is unfortunate, but we think unlikely to be a problem in practice, as such errors should be detected early in testing.

As an alternative, one could use the type system to track types for which an internal runtime representation is present [16]. One would add a type `'a withtyperep`, have the compiler only permit construction of values of that type of compile-time-monomorphic values, e.g. in user code invoking entry points to communication library code, and have library code pass around values of such types. This could be heavy to use but would give more predictable errors.

The Value Restriction SML97 adopted the *value restriction* [40], allowing polymorphic generalisation only for syntactic values. In OCaml 3.09.1 generalization is more liberal, in three ways: (1) whether a conditional expression is generalizable is independent of the condition; (2) certain application expressions are generalizable; and (3) type variables that occur only on the right of an arrow can be generalised, as proposed by Garrigue [15]. All three would lead to problems with our type-passing translation, as inserting the internal type-representation lambdas would change the evaluation order. Hence, HashCaml imposes the value restriction.

We give concrete examples showing the problem for (1) and (3). For (1), the following is typable in OCaml, with `f` of type scheme $\forall$`'a. 'a -> 'a`.

```
let r = ref 0
let s = ref ""
let f = if (r:=1;true) then
          (fun x-> s := Marshal.to_string x []; x)
        else (fun x-> x)
let y = !r
let z = (f true, f 3)
```

In HashCaml the body of `f` would need a runtime type representation for its argument, in order to build the marshalled value, yet if we were to naively add a type passing lambda around the body of `f` then we would observably change the evaluation order, as `y` would end up bound to `0` instead of `1`. (The use of the reference `s` just enables the marshalled values to escape — this might just as well be via network communication.)

For (3), the example below is legal OCaml 3.09.1 code that requires the Garrigue relaxed generalisation condition to typecheck.

```
let r = ref 0
let s = ref ([] : string list)
let f = (r := 1 + !r;
         fun () -> let x = [] in
                   (s:=Marshal.to_string x [] :: !s; x))
let z = (  3 :: (f ())  ,  "foo" :: (f ())  )
```

Here `f` is bound to an expression that is not a syntactic value (or a conditional or non-expansive application), but a generalized type scheme $\forall$`'a. unit -> 'a list` can be inferred, as the `'a` only occurs on the right of the `->`. This generalization is sound in OCaml, intuitively because no values of type `'a` can be returned [15]. With typed marshalling, however, it would be problematic. The two uses of `f` in the last line are at types `unit->int list` and `unit->string list` respectively, so, in the two executions of `f`, the `x` is used at `int list` and `string list`. If the example were allowed in HashCaml then, for marshalling to be type-safe, the marshalled values produced should be `([] : int list)` and `([] : string list)`. Hence any implementation of `f` would actually use its type argument at runtime — but adding a type lambda around the definition of `f` would observably alter the order of evaluation, delaying the assignment to `r` and causing it to happen twice.

Alternatives to type passing Instead of passing type representations, one might aim to recover the type information of values to be marshalled from data in the heap and call stack, perhaps building on work on tag-free garbage collection [17]. In the absence of type abstraction this might work well, but here it seems we would need to propagate abstraction boundaries at runtime, perhaps using some form of the coloured brackets introduced by Grossman et. al [18] and further developed in the Acute semantics [22, 33]. This would be complex and costly.

6. Implementation

In this section we describe our solutions to the main implementation issues that do not impinge directly on the user-visible semantics. Documentation for the internal details of the implementation is available [7].

External and Internal Runtime Type Representations and Type Normalisation As we saw in §5, the external runtime type representations used in marshalled values (against which we do an equality check at unmarshal time) are unstructured 256-bit strings, but the internal runtime type representations (passed in to polymorphic functions, and used to build the external representations) must be composable. The simplest possibility would be to use 256-bit strings also for the internal representations, composing them by applying the hash function at every node of a type abstract syntax tree, e.g. with `rep(`*t1*`*`*t2*`)` = SHA-256(`rep(`*t1*`)`, `*`, `rep(`*t2*`)`), but this would be very expensive. We therefore use a more sophisticated internal representation, ensuring that construction of a new type representation (e.g. when one function, polymorphic in `'a`, calls another at type `'a * 'a`), involves only one allocation and no hashing.

Building these representations occurs in several stages, which we illustrate for the example type `'a * (int M.t)`, where `M.t` is abstract. First we build a normalised tree:

```
NTtuple
  [NTvar 42;
   NTconstructed (
     SHA-256(NTDexternal_abstract "t"),
     [M],
     [NTconstructed (
       SHA-256(NTDbuiltin_abstract "int"),
       [],
       [])])]
```

Here the type variable 'a has been replaced by a canonical index, 42 (in the generated lambda code this will be captured by the appropriate type representation lambda); the paths of external modules on which this type depends have been collected and duplicates removed (the [M]); and the remaining (essentially closed) parts of the structure have been hashed (though in this example there is no variant type or record type definition). The tree is then injectively flattened into a list of strings, type variable references and module references:

```
[FNstring "T(";
 FNtyvar 42;
 FNstring ")(E(...hash value...))";
 FNmyname (...OCaml type environment..., M);
 FNstring "(E(...hash value...))"]
```

Lambda code is then emitted to build a block with corresponding structure: various strings; a real pointer to a block which will contain the type representation for 'a; and a reference to the myname field of M. At marshal and unmarshal points these structures are flattened and a SHA-256 hash of the resulting string computed.

Computation of `myname` values and AST normalization Generating lambda code to compute the myname values of hashed modules involves the following: (1) normalisation w.r.t. alpha equivalence; (2) removal of irrelevant information, e.g. stamps, to ensure that the myname of a submodule is independent of its supermodules if it does not reference them; (3) collection of paths to external modules (with duplicates removed), to generate references to their mynames; and (4) dealing with functors etc. as outlined in §4.

Pattern matching For a polymorphic generalization of a compound pattern, e.g.

```
let (x,y) = (fun x->x),(fun y->y)
```

we generate lambda code in two phases: first adding type representation lambdas for the type variables needed by *all* parts of the right hand side, and later introducing dummy applications for unused ones. This rewriting on lambda code means that we do not have to change the (complex) OCaml pattern matching code.

OCaml generalization The OCaml implementation does not distinguish between types and type schemes; instead, to perform generalization, it imperatively marks the appropriate type variables. This is safe for OCaml, but as we examine the typed syntax tree after type inference it could give the wrong result, e.g. for the example below. Here g has a trivial type scheme, but after type inference its type variables will be marked as general due to the outermost let.

```
let f x = let g y = (x=y) in g x
```

We therefore adapt the type inference algorithm to record, when it meets a value identifier, which type variables in its type are marked as general at that time.

Signature coercions A module's interface may have less general types than the actual implementation underneath. For example, the standard library List.iter code has inferred type

```
val iter : ('a -> 'b) -> 'a list -> unit
```

but is declared in the interface file as

```
val iter : ('a -> unit) -> 'a list -> unit
```

This would cause problems with a naive type-passing translation as the first would expect two type representation arguments but clients of the interface file would provide only one. Now, if an implementation has an interface then the interface must be compiled first. We can therefore detect this situation and for each such identifier, say old, both (a) rename all occurrences of it within the structure, say to new, and (b) add a wrapper function named old that accepts as many type representation parameters as the interface requires and then calls new, supplying it with parameters built from its arguments and the signature coercion.

Runtime support There are three small additions to the OCaml runtime: (1) random256.c is used for generating pseudo-random cfresh and fresh mynames at compile-time and runtime; (2) hash256.c is used for (i) flattening of type representation blocks and (ii) for hashing of strings representing flattened type representation or AST blocks; and (3) polymarshal.c is an interface for the new Marshal module. All of the above were written in C to minimize the runtime overhead of the additional functionality.

Marshalling Values The HashCaml Marshal library uses the underlying OCaml marshaller underneath, so the semantics of marshalling values is unchanged. In particular, a copy of the reachable part of the store is included; and flags allow control of whether sharing is preserved and whether closures are marshalled. Closures can only be unmarshalled in an identical runtime, as marshalled values include only function pointers, not bytecode. (A preliminary experiment by Billings, compiling a small fragment of Acute to a modified OCaml bytecode, suggests that supporting bytecode marshalling between non-identical runtimes would be feasible without a huge performance hit [5].)

Marshalling of exception values in OCaml produces surprising results, as noted by Verlaguet [38]. Pattern matching on exceptions is relative to the pointer that references the exception. When an exception is unmarshaled, it is reallocated with a new pointer that does not correspond to an exception pointer. Because of this, any pattern matching performed on an unmarshaled exception, e.g. as below, will fail.

```
exception E;;
let s=Marshal.to_string E [];;
print_string (match Marshal.from_string s 0 with
    E -> "true"
  | _ -> "false") ;;
(* output : false *)
```

Fixing this would involve changing the OCaml runtime representation of exceptions, for which in principle we should fresh- or hash-generate runtime names.

Other issues There are many other local issues, related to the intricacies of the existing OCaml implementation, that have had to be dealt with. Dealing with De Bruijn indexing has been significant, for expression identifiers, type parameters, record polytype quantifiers, and functor arguments. Typechecking for OCaml lambda code or bytecode would have significantly eased development. On the other hand, using OCaml lambda-code binding for plumbing type variable representations, and myname data, has meant that much of our implementation is orthogonal to internal details of the existing compiler.

Performance data Here we give some brief preliminary performance data, showing that the overhead of type passing can often be negligible and that the system as a whole is usable. For the first, we ran the test suite from the OCaml CVS repository (fib, takc, taku, sieve, quicksort, quicksort.fast, fft, fft.fast, soli, soli.fast, boyer, kb, nucleic, bdd, hamming, sorts, almabench, almabench.fast). Executing those tests is less than 2% slower (302s vs 297s, in the noise). Executing artificial benchmarks that make very heavy use of type passing and/or marshalling can be from 2 to 12 times slower.

On the other hand, compilation is significantly slower than the OCaml ocamlc. On a G5 PowerPC make world is around 3.5 times slower, and make bootstrap around 9 times slower; making the test suite from the OCaml CVS is also around 3.5 times slower. Little attempt has been made so far to optimise the new compiler phases.

Outstanding implementation problems For completeness we summarise the known open problems in the implementation. At the time of writing (in the HashCaml 3.09.1-alpha-785 release): (1) types are not normalised w.r.t. type abbreviations, above; (2) type normalisation for records with universally-quantified type variables is not always correct; (3) type normalisation should be De Bruijn indexed per structure, not per type; and (4) functor applications are re-evaluated for extended module paths, giving incorrect semantics for non-pure functor bodies.

7. Example: Distributed Channel Library

In this section we show by example how the HashCaml marshalling and naming facilities can be used to write safe communication abstractions, discussing exactly what safety properties are guaranteed. The example (loosely based on an earlier Acute library) is a type-safe distributed channel library for asynchronous messaging over typed channels; it is built above the untyped byte-stream network connections provided by the Sockets API for TCP.

From a practical point of view, this example demonstrates the feasibility of creating a simple yet robust typed communication layer as a library in a high-level language; the code is thread-safe and gracefully handles any TCP errors. It is implemented in fewer than 500 lines of code.

7.1 Library Interface

The distributed channel library consists of two modules: `LChan` and `DChan`. `LChan` provides facilities for local channels, whilst `DChan` implements distributed channels using `LChan`.

Both use typed channel names: here *t* `chan` is the type of names of channels that carry values of type *t*. Channel names are implemented simply as HashCaml names (internally represented as 256-bit values).

```
type 'a chan = 'a name
```

The interface to `LChan` is given below. Prior to transferring any data, it is necessary to invoke `init`, which returns an abstract handle to an empty channel-manager state. The client may then asynchronously send data, and register receiver functions, on specified channels. Sent messages and pending receivers are queued on the specified channel; whenever there is one of each, the receiver function is applied to the message, after which both are deleted.

```
module type LChan = sig
  type com_handle
  val init:unit -> com_handle
  val send:com_handle -> 'a chan -> 'a -> unit
  val register_recv:com_handle->'a chan->('a->unit)->unit
end
```

The `DChan` module is an 'internet-aware' version of `LChan`. At initialisation a server is bound to a local internet address to listen for incoming messages. The `send` function transfers messages to remote hosts, while receiver functions are registered with the local server.

```
module type DChan = sig
  type com_handle
  val init:Unix.inet_addr -> port -> com_handle
  val send:Unix.inet_addr -> port -> 'a chan -> 'a ->unit
  val register_recv:com_handle->'a chan->('a->unit)->unit
end
```

For brevity we describe a slightly simplified version of the two modules. In the actual implementation, one can specify timeout and replication options when registering a receiver, and can deregister a receiver.

7.2 Library Implementation

In this section we outline the implementation of the channel library. We focus upon the use of the HashCaml marshalling and naming primitives and the safety guarantees they provide.

Message types The `DChan` module defines a polymorphic type for network messages, containing both a channel and a data field:

```
type 'a message = {
  msg_chan = 'a chan;
  msg_dat  = 'a }
```

Prior to marshalling these messages onto the wire, we existentially package them so that they effectively have type

```
wire_message = ∃'a. 'a message
```

OCaml does not directly support existentials, but does support record fields with universally-quantified type variables; we use those to encode existentials in a module `Msg`.

Sending messages The `DChan.send` function transfers a value to another host over the specified channel. The code performs four main tasks: (i) the data and channel are packed into an existentially-quantified message, as above; (ii) the message is marshalled to a string; (iii) a TCP connection is established with the destination host; and (iv) the marshalled message is sent over the TCP connection.

```
let send : Unix.inet_addr->port->'a chan->'a->unit
= function ip port chan dat ->
    ...
    let msg = Msg.pack {
      msg_chan = chan;
      msg_dat  = dat } in
    ...
    Marshal.to_string msg [Marshal.Closures] ;
    ...
```

Here the HashCaml implementation of `Marshal.to_string` builds a marshalled value by applying the untyped OCaml marshaller to a pair of a runtime type representation of the type of `msg` (i.e. `wire_message`) and the value `msg`. That runtime type representation is essentially a 256-bit hash of the structure of `wire_message`.[1]

Receiving messages The `DChan` implementation contains a `recv` function that is invoked whenever an incoming connection is made to the local server. The code effectively performs the inverse of the visible `DChan.send` function, (i) unmarshalling the message, and (ii) unpacking the data. The data is then forwarded to the local channel module.

```
let recv ... =
  ...
  let msg = Marshal.from_string s 0 in
  let unpack = { f = fun m ->
    LChan.send local_com_handle m.msg_chan m.msg_dat } in
  Msg.use unpack msg
  ...
```

When the message is unmarshalled, HashCaml performs a dynamic equality type check, verifying that the received type representation from the marshalled value is equal to the locally-computed type representation for the expected `wire_message` type. If they differ then an exception is raised at the unmarshal point.

[1] The `Marshal.Closures` flag, as for the underlying OCaml marshaller, enables marshalling of function pointers. This is necessary here, whether or not the user data is functional, as the existential encoding involves a function type — but it has the unfortunate consequence that this library in its current form cannot be used for communication between differing programs, as the underlying marshaller depends on identity of binaries for function-pointer

Different instances of the channel library (in different executions of the code, possibly on different machines) can therefore interact. Further, (apart from the existential encoding issue[1]) different implementations of the channel library could interact so long as the definition of `wire_message` in each is identical. If this is done correctly then the unmarshal will never fail, but if an incompatible implementation is accidentally connected to the IP address and port of an instance of `DChan` then the unmarshal exception would be raised.

HashCaml protects against accidental error, not malicious attack, just as the OCaml static type system does; protecting against forged messages (that claim to have type `wire_message` but actually have a different structure) requires additional mechanisms that we discuss later.

Local channels The `LChan` module uses a hashtable (from the OCaml standard library) to map from channel names to channel data structures, each storing a queue of received data values and a queue of registered receiver functions for a single channel. We maintain the invariant that at least one of these is empty, i.e. we only enqueue data or a receiver if it cannot be immediately consumed.

```
type 'b channeldata = {
  h_chan : 'b name;
  h_dat : 'b list ref;
  h_recvs : ('b -> unit) list ref }
```

To store channel data structures for channels carrying arbitrary types we must existentially pack both them and the hashtable keys; the hashtable is effectively of type

```
(∃'a. 'a chan) -> (∃'b. 'b channeldata)
```

The local channel library `send` and `register_recv` functions take as arguments an `'a chan` and either an `'a` or an `'a->unit`. Their implementations first do a hashtable lookup, using an existentially-packed version of their channel name argument, and returning an existentially-packed channel data structure. They then use the HashCaml `ifname` to compare the supplied and stored names; in the 'equal' branch of the conditional the types of the two names are unified so the channel data structures can be manipulated. We are performing a dynamic, term-level equality check between names to obtain a static, type-level guarantee that the associated types of values are the same.

7.3 Library Usage

A client can use the `send` and `register_recv` functions at any type. In the following example we show a simple scenario in which a fresh channel name is constructed and then this channel is used to send an integer value locally.

```
let local_com_handle = LChan.init ()
let f : int -> unit = fun i -> Printf.printf "%d\n" i
let c : int name = fresh
let _ = LChan.register_recv local_com_handle c f
let _ = LChan.send local_com_handle c 42
```

In more interesting examples, one might want to register a receiver on a channel in program A and also send the channel name to program B (possibly on a remote machine), expecting it to use `DChan` to send back values on that channel. How, though, should the two programs first establish a shared, typed, channel name? There are two possibilities:

(1) If the two programs share a type and a string, then we can use the `hashname` construct to hash this pair to form a channel name. This is the minimum information necessary for constructing a common name.

(2) Providing that the two programs share some code, we can use the `fieldname` construct to create a channel name which depends on this shared code, instead of a simple string (as in (1)). Hence, the channel name is related to the context it is used in, and not simply to the type being transferred, making unintended name equalities less likely.

The following example illustrates the use of the `fieldname` construct. Say A and B share a function `M.f`, and B wants to invoke it at A, i.e. make a remote procedure call.

```
module M = struct
  let f : int -> unit = fun i -> Printf.printf "%d\n" i
end
```

First, A uses the `fieldname` construct to build an `(int->unit) name`, then applies the `namecoercion` construct to build an `int name`. Then the function is registered with this name, which is used as a channel name on which its arguments, of type `int`, can be received.

```
type 'a arrow = 'a -> unit
type 'a id = 'a
let c : (int -> unit) name = fieldname M.f
let c' : int name = namecoercion (arrow, id, c)

let _ = DChan.register_recv dist_com_handle c' M.f
```

B also constructs the same `int name` (first four lines above), and uses it to make a remote invocation.

```
let _ = DChan.send ip port c' 42
```

Several other naming scenarios are possible. For example, as in Acute, one can build structures of compile-time-fresh names, e.g. when behaviour changes at major version-number releases, and then distribute the resulting object files to be linked in with clients.

We have achieved type-safe remote function invocation in just a few lines of code built over `DChan`. Moreover, if one wanted slightly different communication semantics, e.g. to discard messages that arrive for a channel on which no receiver has been registered, or to return an error to the sender, or to store them persistently, it would be a simple matter of recoding the `DChan` library; the HashCaml language would be unchanged.

Note that, when using the `DChan` library to communicate values of abstract types, the invariants of those types will be preserved: it will not be possible to establish a shared channel name on which one program can receive a value from a differing implementation of an abstract type and misinterpret it.

8. Related work

There is little other work on type-safe marshalling for ML-like languages, and almost none that deals with dynamic type equality across programs in the presence of abstract types. A notable exception is the Alice language of Rossberg et. al [30, 29]. Alice provides type-safe marshalling (there, 'pickling'). Abstract types are all dynamically generative, but to establish shared abstract types across a distributed system one can pre-evaluate a component and distribute the result. This is rather less flexible than HashCaml.

Furuse and Weis [14] discuss type-safe marshalling for OCaml but ignore abstraction boundaries. The GCaml language of Furuse [13] provides runtime type analysis for a variant of OCaml but again ignores abstraction — indeed, two variant type definitions that have the same structure but different constructor names will be treated as compatible. Henry, Mauny, and Chailloux [21] address the problem of type-checking an untrusted marshalled value, ensuring that it has the expected low-level structure and so will not crash the local runtime, but again working up to a coarse notion of type isomorphism.

marshalling. We would like to add an explicit existential type to HashCaml, which would be straightforward and solve the problem.

Several languages build in a form of typed distributed communication directly, including Facile [37], JoCaml [12], and Nomadic Pict [35], with typed channels (building on work on the π-calculus [26], CML [28], and Pict [27]). These support type-safe distributed executions that arise from single executions of single programs (with the distributed parts all spawned from a single initiator), but some ad-hoc 'trader' mechanism is needed to establish connections between multiple executions or different programs. Further, building communication into the language runtime entails a commitment to a particular synchronisation and failure semantics. In HashCaml, in contrast, one can simply write multiple type-safe communication libraries for different circumstances.

A type Dynamic was introduced in Amber [8]. Leroy and Mauny added a variant to CAML [25], including an `extern` primitive to marshal values of type `dyn`. This included definitions of relevant types, with the corresponding `intern` checking that the reader program defines types with the same names and similar structure. Abadi et. al study type Dynamic with and without parametric polymorphism [2, 3]. None of these address abstraction boundaries.

With sufficiently expressive intensional type analysis [9, 39] one could define type-safe marshalling within the language itself, but there is a tension here: if the language is this expressive, allowing abstract types to be analysed, then abstraction boundaries are not enforced even within a single program.

Java's serialization facility [1] includes version identifiers (`serialVersionUID`s) with the class definitions of serialized objects. These default to 64-bit hashes of all method and field signatures, but not of method bodies. Abstraction safety is not guaranteed, since changing a method body does not change the version identifier. Additionally, class authors may override the version identifiers for purposes of compatibility.

The .NET platform supports *sharable assemblies*, which include a public key, file hashes, and a version number. To the best of our knowledge, however, type identity is not dependent on code hashes, giving weaker type-safety guarantees.

9. Discussion

We have described the design and implementation of HashCaml, an extension of the OCaml bytecode compiler with support for type-safe and abstraction-safe marshalling and related naming constructs, and have demonstrated with an example communication library how this supports type-safe distributed programming. The implementation covers all the standard part of OCaml except marshalling of polymorphic variants and objects; it includes the standard libraries; and it has performance reasonably close to `ocamlc` (and around three orders of magnitude faster than Acute). An alpha release is available and should be usable for non-trivial software; feedback on its use would be very welcome.

In the rest of this section we discuss some of our choices and possible future work.

Optimization The performance of the compiler and of the generated code is clearly usable but less than ideal, and there is much scope for optimization. As for execution times, one would like marshalling to impose *no* additional cost on code that does not use it, but at present our implementation adds type-passing everywhere. It should be reasonably straightforward to (conservatively) statically analyse where runtime type representation data is actually needed and (for example) generate two copies of every function, one with type-passing and one without. One might also detect polymorphic functions that are in fact only used at one type. Memoizing the SHA-256 computation of marshalled type representations is likely worthwhile. In fact, the number of distinct types that occur in a large program (especially the marshalled types) is likely small enough that it is worth precomputing all their marshalled representations, together with the actions of the implicit type operators in the program (e.g. λ`'a. 'a*'a`) over those.

Type-passing revisited Of course, a simpler way to remove the cost of type-passing is simply to forbid it — to disallow marshalling at types that mention free type variables. It would then be straightforward to compute all marshalled type representations at compile-time. Whether this would be a serious loss of expressivity is not clear. The `DChan` library of §7, for example, does not require type-passing. The library has a polymorphic interface, with `send` taking an `'a` and an `'a name` for any `'a`, but the marshalled values sent over the wire are all of a single closed existential type. On the other hand, a library function `send : fd -> 'a -> unit`, that takes a value, marshals it, encrypts the resulting string, and writes that to a file descriptor, would require type-passing. To use such a function successfully the sender and receiver would have to share implicit knowledge about the expected types of values, e.g. that at this point in a protocol a value of such-and-such a type is expected. More experience in typed distributed programming is needed to understand whether this latter is an important scenario. In any case, it would be straightforward to provide a HashCaml compiler option to disable type-passing (but retain the hash and fresh computation of runtime type representations, which is the larger part of our work).

Threat model HashCaml protects against accidental error, not malice. It does not attempt to ensure that *no* information about the representation of an abstract type can leak out, as that already does not hold in OCaml (`compare`, `Obj.magic`, etc. mean that no parametricity property holds). On the other hand, distinguishing abstract types from their representations, as we do, should catch many programmer errors.

To protect against maliciously-produced marshalled values one may either (1) ensure that marshalled values only come from trusted sources, e.g. by implementing (within HashCaml) cryptographically-protected communication libraries, and/or (2) check the *structure* of marshalled values at unmarshal-time, as in [21] (though that does not check abstract type invariants). One might combine the two approaches, with the HashCaml type equality check and this low-level structure check.

Other OCaml features Adapting the `ocaml` interactive top level to HashCaml is not completely trivial, as (a) later definitions can instantiate free type variables from earlier ones, and (b) the top level allows shadowing of type definitions. Adapting the `ocamlopt` native-code compiler involves no major problems, as far as we know, but signature coercions are implemented slightly differently. Understanding what dynamic type equality one should use for OCaml object and polymorphic variant types is an open problem.

Other Acute features In the Acute design [33, 34] we argued that several additional features are desirable for typed distributed programming: flexible dynamic rebinding to local resources, marshalling of functions between non-identical programs, polytypic name support and swap operations (as in Fresh OCaml [36]), explicit versioning, and thunkification of executing threads into marshallable values. Addition of these would be well worth-while, but (at least for the latter) would involve rather more substantial changes to the existing OCaml language and implementation.

Subtyping, richer module systems, and structured runtime type representations In the large-scale distributed setting, in which differing versions of programs must interoperate, supporting unmarshalling up to a subtype relation would be very useful. Work by Deniélou and Leifer addresses this issue in the presence of type abstraction boundaries [10]. Other work by Peskine is addressing the design of runtime type names for a rich module system, along the lines of Dreyer et. al [11]. The type-safe marshalling machinery of HashCaml also supports a limited form of `typecase`, with type equality testing. To experiment with richer forms of intensional type analysis [9] in OCaml, the HashCaml type-passing could be adapted to work with more structured type representations.

Dynamic type equality Finally, with an eye to future language design, we note that our work here has essentially involved designing a reasonable dynamic type equality (and one that is global, defined across scopes and between programs), for a language with an existing nontrivial static type equality. Ideally the two should be designed together and, where comparison is meaningful, they should coincide. Generativity of type definitions has been extensively discussed in the literature. In the distributed setting, as marshalling makes it possible to compare types across scopes and between programs, there are additional subtle choices that must be made. Further, for marshalling to be useful —i.e. for typed communication between programs to not be unreasonably restricted— there are additional constraints. Within a single ML program all type definitions could often be (statically and dynamically) generative, but for distributed programming that is no longer the case.

Acknowledgements We acknowledge support from a Royal Society University Research Fellowship (Sewell), EPSRC grant GRT11715, and APPSEM 2. We thank Jacques Garrigue for discussions on type passing, Julien Verlaguet for work on a predecessor system, and Adam Biltcliffe, James Leifer, Benjamin Pierce, Tom Ridge, Viktor Vafeiadis, and Alisdair Wren for comments.

References

[1] Java^{TM} object serialization specification 1.5.0. Technical report, Sun Microsystems, Apr. 2004.

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

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

[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.

[5] J. Billings. A bytecode compiler for Acute, 2005. Computer Science Tripos Part II Dissertation, University of Cambridge.

[6] J. Billings, P. Sewell, M. Shinwell, and R. Strniša. HashCaml 3.09.1-alpha-785. `https://www.cl.cam.ac.uk/users/pes20/hashcaml`, Apr. 2006.

[7] J. Billings, P. Sewell, M. Shinwell, and R. Strniša. The implementation of HashCaml, Apr. 2006. `https://www.cl.cam.ac.uk/users/pes20/hashcaml`.

[8] L. Cardelli. Amber. In *Combinators and Functional Programming Languages, LNCS 242*, pages 21–70, 1986.

[9] K. Crary, S. Weirich, and G. Morrisett. Intensional polymorphism in type erasure semantics. In *Proc. ICFP*, pages 301–313, 1998.

[10] P.-M. Deniélou and J. J. Leifer. Abstraction preservation and subtyping in distributed languages, Sep. 2006. In *Proc. ICFP*, 2006.

[11] D. Dreyer, K. Crary, and R. Harper. A type theory for higher-order modules. In *Proc. POPL*, 2003.

[12] C. Fournet, F. L. Fessant, L. Maranget, and A. Schmitt. The JoCaml language beta release documentation and user's manual, Jan. 2001. `http://moscova.inria.fr/jocaml/`.

[13] J. Furuse. *Extensional Polymorphism: Theory and Applications*. PhD thesis, Université Paris 7, 2002.

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

[15] J. Garrigue. Relaxing the value restriction. In *International Symposium on Functional and Logic Programming, Nara, LNCS 2998*, Apr. 2004.

[16] J. Garrigue. Personal communication, Sept. 2005.

[17] B. Goldberg. Tag-free garbage collection for strongly typed programming languages. *Sigplan*, 26(6):165–176, 1991.

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

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

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

[21] G. Henry, M. Mauny, and E. Chailloux. Typer la désérialisation sans sérialiser les types. In *Journées Francophones des Langages Applicatifs*, Jan. 2006.

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

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

[24] X. Leroy et al. Objective Caml 3.09.1. `http://caml.inria.fr`, Jan. 2006.

[25] X. Leroy and M. Mauny. Dynamics in ML. *Journal of Functional Programming*, 3(4):431–463, 1993.

[26] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, Parts I + II. *Information and Computation*, 100(1):1–77, 1992.

[27] B. C. Pierce and D. N. Turner. Pict: A programming language based on the pi-calculus. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*, pages 455–494. MIT Press, 2000.

[28] J. H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.

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

[30] A. Rossberg, D. L. Botlan, G. Tack, T. Brunklaus, and G. Smolka. Alice through the looking glass. In *Trends in Functional Programming, Vol. 5*, Feb. 2006.

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

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

[33] P. Sewell, J. J. Leifer, K. Wansbrough, F. Zappa Nardelli, M. Allen-Williams, P. Habouzit, and V. Vafeiadis. Acute: High-level programming language design for distributed computation. In *Proc. ICFP*, Sept. 2005.

[34] P. Sewell, J. J. Leifer, K. Wansbrough, F. Zappa Nardelli, M. Allen-Williams, P. Habouzit, and V. Vafeiadis. Acute: High-level programming language design for distributed computation. Dec. 2005. Submitted for publication. `https://www.cl.cam.ac.uk/users/pes20/acute/paper3.ps`.

[35] 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.

[36] M. R. Shinwell, A. M. Pitts, and M. J. Gabbay. FreshML: Programming with binders made simple. In *Proc. ICFP*, 2003.

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

[38] J. Verlaguet. Acaml: An extension of OCaml with Acute-like marshalling, Oct. 2005. Masters Dissertation.

[39] D. Vytiniotis, G. Washburn, and S. Weirich. An open and shut typecase. In *Proc. ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI)*, Jan. 2005.

[40] A. K. Wright. Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8(4):343–355, 1995.

$Date: 2006-04-05 20:15:20 +0100 (Wed, 05 Apr 2006) $ $Rev: 830 $: