

# HashCaml: type-safe marshalling for O'Caml

Mark Shinwell  
(with John Billings, Peter Sewell and Rok Strnisa)

University of Cambridge Computer Laboratory

Semantics Lunch, 5<sup>th</sup> December 2005

# O'Caml marshalling is unsound

Current versions of O'Caml have no checks in place to ensure that marshalled values are unpacked at the correct type.

For example:

```
c90:~$ /usr/local/ocaml-3.09.0/bin/ocaml
      Objective Caml version 3.09.0

# let s = Marshal.to_string 42 [];;
val s : string =
  "\132\149??\000\000\000\001\000 ... \000\000j"
# let t = (Marshal.from_string s 0)();;
Warning X: this argument will not be used by the function.
Bus error
```

# What HashCaml does

HashCaml fixes the unsoundness by providing *type-safe* marshalling operations. They can be applied in polymorphic settings:

```
let f x = Marshal.to_string x [];;  
let s = [f 42; f "foo"];;  
let forty_two = Marshal.from_string (List.hd s);;
```

We make this work by extending the O'Caml system with

- support for discovering the types of values at runtime (but no “full RTTI” or runtime type inference);
- a means of preserving abstraction safety (so one abstract  $M.t$  does not get confused with another).

The type-safe marshalling works even between non-identical programs and/or runtimes.

# What HashCaml doesn't do

Whilst HashCaml is intended in some way to be a port of the Acute language to the O'Caml system, it does not provide all the functionality of Acute. For example HashCaml lacks:

- Dynamic rebinding (the 'mark' scheme) – although some experiments have been done.
- Control of versioning.
- Thunkification (almost like capturing a continuation).
- Marshalling of code between non-identical runtimes.

Additionally it would be nice to do:

- Validation of the contents of marshalled packages to ensure they haven't been tampered with.

## What is provided: quick overview

The following new keywords have been added:

```
fresh
dyntype(exp)
rep(type)
fieldname(value-path)
namecoercion(tycon-path, tycon-path)
ifname exp = exp then exp else exp
```

together with a family of types `'a name` (à la Fresh O'Caml).

Functions for marshalling and unmarshalling are provided in the standard library.

# Marshalling and unmarshalling

Constructs for marshalling and unmarshalling can be built from the dynamic typing keywords and the existing O'Caml marshaller:

```
let to_string v flags = make_package v flags (dyntype(v))
let from_string str : 'a = extract_package str 0 (rep('a))
```

Here, `make_package` and `extract_package` are runtime functions written in C.

- `dyntype(v)` tells the marshaller what dynamic type the value has.
- `rep('a)` tells the unmarshaller what result type the `from_string` function is being used at.

**Q.** How do we implement `dyntype` and `rep`?

**A.** With great difficulty.

Three possible ways of obtaining dynamic type information at runtime:

- Full runtime type information in the heap
  - abstraction boundaries have been lost
  - expensive.
- Stack-based type reconstruction
  - work backwards up the stack looking at callers' argument types
  - fraught with difficulties.
- Addition of explicit type abstractions and applications
  - à la System F, but we pass *type representations*.

# Example

In our case the third scheme amounts to

- turning occurrences of **value identifiers** with polymorphic type schemes into typerrep applications;
- inserting typerrep abstractions at **generalization points**.

For example the following code

```
let f x = dyntype(x)
let foo = [f "foo"; f (24, 7)]
```

translates to

```
let f typerrep_a x = typerrep_a
let foo = [f String "foo"; f (Pair(Int, Int)) (24, 7)]
```

where `String` is the typerrep of the type `string` and `Pair(Int, Int)` is the typerrep of `int × int`.

## More complex patterns

Care must be taken when dealing with the more complicated forms of `let`-binding permitted by O'Caml:

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

should translate to

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

To deal with these cases, we perform the type-passing translation after the pattern matches have been compiled.

- a tricky business...

# Signature constraints

We must also be careful where structures are constrained by signatures. Coercions at these points can lead to implicit type applications which we must insert explicitly.

For example the function `List.iter` defined thus:

```
let rec iter f = function [] -> ()
                    | a::l -> f a; iter f l
```

has type scheme  $\forall\alpha\beta. (\alpha \rightarrow \beta) \rightarrow \alpha \text{ list} \rightarrow \text{unit}$ . We therefore add two `typerep` lambdas (corresponding to the two type parameters).

However in the signature for the `List` the function has type scheme  $\forall\alpha. (\alpha \rightarrow \text{unit}) \rightarrow \alpha \text{ list} \rightarrow \text{unit}$ . Therefore when the function is exposed through the signature (with only one `typerep` lambda) we must add a *coercion wrapper* to instantiate  $\beta$  to `unit`.

# Construction of type representations

Type representations are emitted as 256-bit *parameterized hashes*.

- A parameterized hash consists of:
  - a constant part (whose hash is calculated at compile-time);
  - an indexed set of parameters (corresponding to hash values that may only be calculated at runtime).
- The indices of the parameters may be referenced in the constant part to express that it depends on those values.
- Given a parameterized hash we can generate code to calculate the final hash at runtime.

In the case of typereps the parts of hashes which cannot be calculated at compile-time are those corresponding to:

- type variables (modulo an optimization);
- abstract types in other modules.

# Abstract types in other modules

Given an abstract type  $M.t$  from some other module, what should its type representation be?

- It is the hash of " $t$ " together with the *hash name* of  $M$ , denoted by  $M.myname$ .
- Each module has this `myname` field added by magic.

How is `myname` calculated? At the user's wish,

- either by **hashing the source code of the module**, or
- by **generating a fresh name** at module initialization time.

This protects invariants by preventing confusion between different module implementations. It provides stronger guarantees than 'interface hashing' schemes such as that provided in the Java system.

There are plenty of awkward issues to do with substructures and functors...

# Marshalling at fully polymorphic types

Theoretically we could generalize the type of a marshalled value before packaging it, so this:

```
let s = marshal (fun x -> x);;
```

would yield a package which could be opened and used at many different types.

- This would require a distinguished keyword so the special typing rule could be implemented.
- Also it would require a subsumption check at unmarshal-time:
  - hashes would not do, we would have to send tree structures.

At the moment we want to prevent this and other counter-intuitive behaviour by completely disallowing type variables in the dynamic type of an expression being marshalled.

- Don't yet know how to do this when typereps are just hashes.

Some of the difficulties encountered so far that are specific to O'Caml's implementation are:

- Recovery of typing information after type inference can be hard since imperative algorithms are used.
  - O'Caml does not have any concept of *type scheme*: instead, generalized type variables are marked as such (we write them  $\alpha^*$ ).
  - When copying a type, generic variables are duplicated but non-generic ones are not.
- Internal representations are complicated (cyclic type structures).
- We have to rewrite on different varieties of trees to solve the pattern-matching problem.
- There is special treatment of free type variables in the toplevel.
- Many fancy things to deal with (objects, commuting arguments, etc).

## Example: the perils of imperative type inference

In the following code, the monomorphic function  $g$  has type  $\alpha \rightarrow \text{bool}$ , where  $x$  has type  $\alpha$ .

```
# let f x = let g y = (x = y) in g x
val f : 'a -> bool = <fun>
```

The type-passing version should be:

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

Unfortunately because the  $\alpha$  in the type of  $f$  is generalized, it appears after type inference that  $g$  has type  $\alpha^* \rightarrow \text{bool}$ .

- We save information during type inference to enable the original state to be deduced even after typing.

Correct interoperation with C is important not least because some of the runtime system and standard libraries are implemented in C.

- No type representations are passed to C functions.
- However C functions that take polymorphic functional arguments may need to construct type representations.
- To do this they must be wrapped in an ML function that uses `rep`.
- The unmarshalling function is an example of this as we saw earlier:

```
let from_string str : 'a =  
    extract_package str 0 (rep('a))
```

Our support for dynamic recovery of type information permits code like:

```
# let f x y = (typeof(x) = typeof(y))  
val f : 'a -> 'b -> bool = <fun>
```

- Currently we can only test type representations for equality since they are just hashes.
- Using tree structures for type representations would enable full *intensional type analysis* (“typecase”).

Whilst some of the additions we have made to O'Caml can impair performance, there are many optimizations that could be made.

For example:

- Link-time calculation of `myname` fields.
- Memoization of `typerrep` hashes.
- Analyses to ensure that `typereps` are not passed to functions that will never use them.
- Link-time monomorphism analyses to calculate `typerrep` hashes.

This work is still at a relatively early stage, although progress is relatively rapid.

- Initial indications are that the new facilities integrate well from the programmer's perspective.
- Current priorities are to bootstrap the compiler and improve the test environment.
- We hope to be testing more substantive examples within the next couple of weeks.
- It is hoped eventually that we can optimize enough to leave only a minimal runtime penalty.