Global abstraction-safe marshalling
via hash types

James J. Leifer    Gilles Peskine    Peter Sewell    Keith Wansbrough

INRIA Rocquencourt    University of Cambridge
Problem

Consider inter-machine communication (or persistent storage):

(A)

\[
\ldots \\
\text{send (marshal (v : bool))}
\]

(B)

\[
\ldots \\
\text{let } y = \\
\text{unmarshal (receive () : int list)}
\]

A dynamictypecheck of \( t = t_0 \) canensurethesafetyof unmarshal.

Butwhatif \( t \) and \( t_0 \) areML-likeabstracttypes, e.g. 

\[
t = \text{UnbalancedBinaryTree.ty} \\
t_0 = \text{BalancedBinaryTree.ty}
\]

Couldjustconsidertheirconcreterepresentations	obtain type safety, but wewantabstractionsafety too.

Leifer, Peskine, Sewell, Wansbrough. “Global abstraction-safe marshalling via hash types”
Problem

Consider inter-machine communication (or persistent storage):

\[ \text{(A)} \]

\[
\ldots \\
\text{send (marshal (v : t ))} \\
\]

\[ \xrightarrow{v : t} \]

\[ \text{(B)} \]

\[
\ldots \\
\text{let y =} \\
\text{unmarshal (receive () : t')} \\
\]

A \textit{dynamic type check} of \( t = t' \) can ensure the safety of \text{unmarshal}.
Problem

Consider inter-machine communication (or persistent storage):

(A) ...
send (marshal (v : t ))

(B) ...
let y = unmarshal (receive () : t ' )

A dynamic type check of $t = t'$ can ensure the safety of `unmarshal`.

But what if $t$ and $t'$ are ML-like abstract types, e.g.

$$
t = \text{UnbalancedBinaryTree.ty}
$$
$$
t' = \text{BalancedBinaryTree.ty}
$$

Could just consider their concrete representation types to get type safety, but we want abstraction safety too.

Leifer, Peskine, Sewell, Wansbrough. “Global abstraction-safe marshalling via hash types”
Overview

- Examples: communication with abstract types

- Solution: hash types, compilation, and typing

- Theorems

- Conclusions and future work
An even counter: manifest signature

module EvenC = (  
  struct  
    type t = int  
    (* the representation type *)  
    let start = 0  
    let up x = x + 2  
    let get x = x  
  end : EvenCSig)

EvenCSig =  
  sig  
    type t = int  
    (* t is manifestly equal to int *)  
    val start : t  
    val up : t -> t  
    val get : t -> int  
  end
An even counter: abstract signature

module EvenC = (  
  struct
    type t    = int        (* the representation type *)
    let start = 0
    let up x  = x + 2
    let get x = x
  end : EvenCSig)

EvenCSig =  
sig
  type t                     (* t is abstract *)
  val start : t
  val up   : t -> t
  val get  : t -> int
end
Example: identical abstract types

(A)  
module EvenC = (struct  
    type t = int  
    let start = 0  
    let up x = x + 2  
    let get x = x  
end : EvenCSig)  
let x = EvenC.start in  
    send (marshal (x : EvenC.t))

(B)  
module EvenC = (struct  
    type t = int  
    let start = 0  
    let up x = x + 2  
    let get x = x  
end : EvenCSig)  
let y =  
    unmarshal (receive () : EvenC.t)

✓ succeed

Within a single program, two abstract types with the same definition would be different (ML generativity). Between programs, that’s not what we want.
Example: concrete to abstract

(A) 

let x = 3 in 

send (marshal (x : int))

(B) 

module EvenC = (struct
    type t = int
    let start = 0
    let up x = x + 2
    let get x = x
end : EvenCSig)

let y = 

unmarshal (receive () : EvenC.t)

× fail

Allowing unmarshal to succeed would break (B)’s invariants.
**Example:** same external behaviour but different internal invariants

(A)

```ml
module EvenC = (struct
    type t = int
    let start = 0
    let up x = x + 1
    let get x = 2 * x
end : EvenCSig)

let x = EvenC.start in
send (marshal (x : EvenC.t))
```

(B)

```ml
module EvenC = (struct
    type t = int
    let start = 0
    let up x = x + 2
    let get x = x
end : EvenCSig)

let y =
unmarshal (receive () : EvenC.t)
```

× fail

Again, success would not respect (B)’s invariants.
Example: same internal invariants

(A)  
module EvenC = (struct  
    type t = int  
    let start = 0  
    let up x = 2 + x  
    let get x = x  
end : EvenCSig)  
let x = EvenC.start in  
    send (marshal (x : EvenC.t))  

(B)  
module EvenC = (struct  
    type t = int  
    let start = 0  
    let up x = x + 2  
    let get x = x  
end : EvenCSig)  
let y =  
    unmarshal (receive () : EvenC.t)

? maybe

Success would require a theorem prover to perform the verification (unrealistic) or a user-supplied coercion.

Leifer, Peskine, Sewell, Wansbrough. “Global abstraction-safe marshalling via hash types”
## Summary of the main cases

<table>
<thead>
<tr>
<th>Interface</th>
<th>Implementation</th>
<th>Desired behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>same</td>
<td>same code</td>
<td>✓ succeed</td>
</tr>
<tr>
<td>same</td>
<td>same internal invariants</td>
<td>? maybe</td>
</tr>
<tr>
<td>same</td>
<td>same external behaviour but different internal invariants</td>
<td>× fail</td>
</tr>
<tr>
<td>same</td>
<td>different external behaviour</td>
<td>× fail</td>
</tr>
<tr>
<td>different</td>
<td>...</td>
<td>× fail</td>
</tr>
</tbody>
</table>
| ...                | different representation types          | × fail           

Leifer, Peskine, Sewell, Wansbrough. “Global abstraction-safe marshalling via hash types”
How do we get the desired behaviour?

- For communication between programs with identical sources, it’s easy to compare abstract types by their source-code names, e.g. `EvenC.t` would mean the same thing in all copies.

- However, for programs that share only some modules, that would be unsound.

How do we obtain **globally meaningful type names**?

Solution: we construct them from module *hashes*.

![Diagram](A) \[\ldots\] ![Diagram](B) \[\ldots\]

\[v : \text{hash(struct } \ldots \text{end : sig } \ldots \text{end)}.t\]
Solution: hash types

- We can implement them with a cryptographic hash, e.g. md5 (compact fingerprint yet injective in practice).

- We freely look inside their structure in our typing rules, but never need to do this in the implementation.


\[
\text{v: hash(struct ... end: sig ... end).t}
\]
1. Compile-time reduction: hash generation

\[
\text{module EvenC = } \\
\left( \\
\text{struct type t = int let start = 0 ... end} \\
\right) \\
\left( \\
\text{: sig type t val start : t ... end} \\
\right) \\
\text{send (marshal (EvenC.start : EvenC.t))}
\]

\[\rightarrow_c \text{ inlining EvenC} \]

\[
\text{send (marshal (0 : h.t))}
\]

where \( h = \text{hash}\left( \text{struct type t = int let start = 0 ... end} \right) \\
\left( \text{: sig type t val start : t ... end} \right) \]
2. Compile-time reduction: module dependency (1/3)

module EvenC =
  (struct type t = int
    let start = 0 ... end
  : sig type t
    val start : t ... end

module CleanC =
  (struct type s = EvenC.t * bool
    let create = (EvenC.start, true) ... end
  : sig type s
    val create : s ... end

send (marshal (CleanC.create : CleanC.s))

Leifer, Peskine, Sewell, Wansbrough. “Global abstraction-safe marshalling via hash types”
2. Compile-time reduction: module dependency (2/3)

Inlining `EvenC`

```ocaml
c
module CleanC = {
  struct type s = \texttt{h}.t * bool
  let create = (0, true) ... end
  : sig type s
  val create : s ... end
send (marshal (CleanC.create : CleanC.s))
}
```

Where

```ocaml
\texttt{h} = \texttt{hash} {
  struct type t = int
  let start = 0 ... end
  : sig type t
  val start : t ... end
}
```

Leifer, Peskine, Sewell, Wansbrough. “Global abstraction-safe marshalling via hash types”
2. Compile-time reduction: module dependency (2/3)

\[ \longrightarrow_{c} \]

Inlining EvenC

\[
\text{module CleanC} =
\begin{aligned}
\text{struct} &\quad \text{type } s = h \cdot t \ast \text{bool} \\
&\quad \text{let } \text{create} = (0, \text{true}) \ldots \text{end} \\
&\quad : \text{sig} \quad \text{type } s \\
&\quad \text{val } \text{create} : s \ldots \text{end}
\end{aligned}
\]

\[
\text{send (marshal (CleanC.create : CleanC.s))}
\]

Where

\[
h = \text{hash}
\begin{aligned}
\text{struct} &\quad \text{type } t = \text{int} \\
&\quad \text{let } \text{start} = 0 \ldots \text{end} \\
&\quad : \text{sig} \quad \text{type } t \\
&\quad \text{val } \text{start} : t \ldots \text{end}
\end{aligned}
\]
2. Compile-time reduction: module dependency (3/3)

\[ \xrightarrow{c} \text{inlining CleanC} \]

\[
\text{send (marshal } ((0, \text{true}): \mathcal{h}'.s)) \]

where

\[
\begin{align*}
\mathcal{h} &= \text{hash} \\
\begin{pmatrix}
\text{struct} & \text{type } t = \text{int} \\
\text{let} & \text{start} = 0 \quad \ldots \quad \text{end} \\
\text{let} & \text{create} = (0, \text{true}) \quad \ldots \quad \text{end}
\end{pmatrix}
\end{align*}
\]

\[
\begin{pmatrix}
\text{struct} & \text{type } t
\end{pmatrix}
\]

\[
\begin{pmatrix}
\text{val} & \text{start} : t \quad \ldots \quad \text{end}
\end{pmatrix}
\]

\[
\mathcal{h}' &= \text{hash} \\
\begin{pmatrix}
\text{struct} & \text{type } s = \mathcal{h}.t \times \text{bool}
\end{pmatrix}
\]

\[
\begin{pmatrix}
\text{let} & \text{create} = (0, \text{true}) \quad \ldots \quad \text{end}
\end{pmatrix}
\]

\[
\begin{pmatrix}
\text{val} & \text{create} : s \quad \ldots \quad \text{end}
\end{pmatrix}
\]

Leifer, Peskine, Sewell, Wansbrough. “Global abstraction-safe marshalling via hash types”
3. Compile-time reduction: coloured brackets

module EvenC =

(struct type t = int let start = 0 ... end
 : sig type t val start : t ... end)

send (marshal (EvenC.start : EvenC.t))

\[ \rightarrow_c \]

inlining EvenC

send (marshal (0 : h.t))

where

\[ h = \text{hash} (\text{struct type t = int let start = 0 ... end} \\
 : \text{sig type t val start : t ... end}) \]

Coloured brackets are adapted from [Zdancewic, Grossman, & Morrisett]
3. Compile-time reduction: coloured brackets

module EvenC =
  (struct type t = int let start = 0 ... end
   : sig type t val start : t ... end
  )

send (marshal (EvenC.start : EvenC.t))

\[ \rightarrow_c \]

inlining EvenC

send (marshal ( [0]_h^h \cdot t : h.t ))

where

\[ h = \text{hash}\left( \begin{array}{c}
\text{struct type t = int let start = 0 ... end} \\
: \text{sig type t val start : t ... end}
\end{array} \right) \]

Coloured brackets are adapted from [Zdancewic, Grossman, & Morrisett]
The calculus

- call-by-value lambda-calculus;
- second-class, first-order modules;
- communication and parallel composition;
- marshal and unmarshal;
- hashes in the type grammar:

  \[ T ::= \ldots \]

  \| \ h \cdot t \quad \text{(not in user source code)}

- coloured brackets in the expression grammar:

  \[ e ::= \ldots \]

  \| \ [e]^T_h \quad \text{(not in user source code)}

Leifer, Peskine, Sewell, Wansbrough. “Global abstraction-safe marshalling via hash types”
Type equality \( (E \vdash_h T_0 \equiv T_1) \)

- singleton kind equations for module typing [Harper & Lillibridge];
- plus hash transparency when inside coloured brackets:

\[
\frac{E \vdash_h \text{ok}}{
E \vdash_h h \cdot t \equiv T}
\quad \text{if } h = \text{hash}(\text{struct type } t = T \ldots \text{end} : \text{sig type } t \ldots \text{end})
\]

Coloured brackets

- determine where hash transparency occurs:

\[
\frac{E \vdash_h e : T}{
E \vdash_{h'} [e]_h^T : T}
\]
Theorems

- **Type preservation, progress:** for compile-time and run-time reduction. Thanks to brackets, this includes (informally) abstraction preservation.

- **Type coincidence:** ML type equivalence coincides with unmarshal-time syntactic comparison of hash types.

- **Erasure:** after compilation, erasure of all coloured brackets (except in hashes) yields identical run-time behaviour.

Subtleties: handling dependent signatures and tracking colours. (We optimise proofs with a rigorous meta-notion of “similar case”.)
Conclusions and future work

Hashing modules provides a meaningful way of comparing abstract types that are defined in independently compiled distinct programs: as a result, the behaviour we sought “just works”.

What’s next?

- **ML**: multiple type and term fields, polymorphism, functors, nested modules;
- **Beyond**: subtyping, coercions and versioning, dynamic binding for local resources;
- **Implementation**: Jocaml and Ocaml, applications to safe name servers, channels, persistent stores.
Theorems in detail...
Theorem: erasure

After compilation, erasure of all coloured brackets (except in hashes) yields identical run-time behaviour.

Let \textit{erase} be the erasure function.

Let $\overset{\text{uncol}}{\rightarrow}$ be the corresponding run-time reduction relation.

- If $\text{nil} \vdash_{\text{ho}} e : T$ and $e \overset{\text{ho}}{\rightarrow} e'$ then $\text{erase}(e) \overset{\leq 1}{\rightarrow}_{\text{uncol}} \text{erase}(e')$.
- If $\text{nil} \vdash_{\text{ho}} e : T$ and $\text{erase}(e) \overset{\text{uncol}}{\rightarrow} e_0$

  then there exists $e'$ such that $\text{erase}(e') = e_0$ and $e \overset{\geq 1}{\rightarrow}_{\text{ho}} e'$.

As we said, $\overset{\text{uncol}}{\rightarrow}$ does not preserve typing.
Theorem: type coincidence

ML type equivalence coincides with unmarshal-time syntactic comparison of hash types.

Consider the following sequence of module definitions:

\[ D.\_ = \text{module } U_1 = M_1:S_1 \text{ in ...module } U_n = M_n:S_n \text{ in } \_ \]

Let \( \sigma_D \) be the substitution induced by compilation of the modules.

Suppose that no two modules have the same hash.

Then:

\[ U_1:S_1, \ldots, U_n:S_n \vdash \_ \; T_0 =: T_1 \quad \iff \quad \sigma_{D_T 0} = \sigma_{D_T 1} \]

static typing \iff \check{\text{check performed by unmarshal}}