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)

```
...

send (marshal (v : bool))
```

v ⟶

(B)

```
...
let y =
 unmarshal (receive () : int list)
```

Problem

Consider inter-machine communication (or persistent storage):

(A)

```
...

send (marshal (v : t    ))
```

$\xrightarrow{\texttt{v : t}}$

(B)

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

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

Problem

Consider inter-machine communication (or persistent storage):

(A)

```
...

send (marshal (v : t    ))
```

$\xrightarrow{\texttt{v : t}}$

(B)

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

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

But what if `t` and $\texttt{t}'$ are ML-like abstract types, e.g.

$$\begin{aligned}\texttt{t} &= \texttt{UnbalancedBinaryTree.ty} \\ \texttt{t}' &= \texttt{BalancedBinaryTree.ty} \quad ?\end{aligned}$$

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

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)
```

$\surd$ 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)

```
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)

```
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.

Summary of the main cases

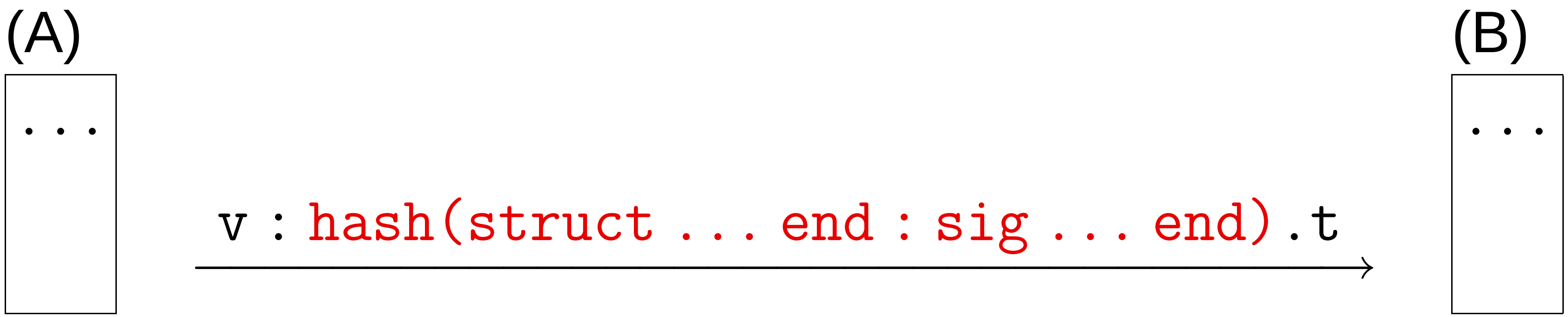
Interface	Implementation	Desired behavior
same	same code	√ succeed
same	same internal invariants	? maybe
same	same external behaviour but different internal invariants	× fail
same	different external behaviour	× fail
different	...	× fail
...	different representation types	× fail

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.

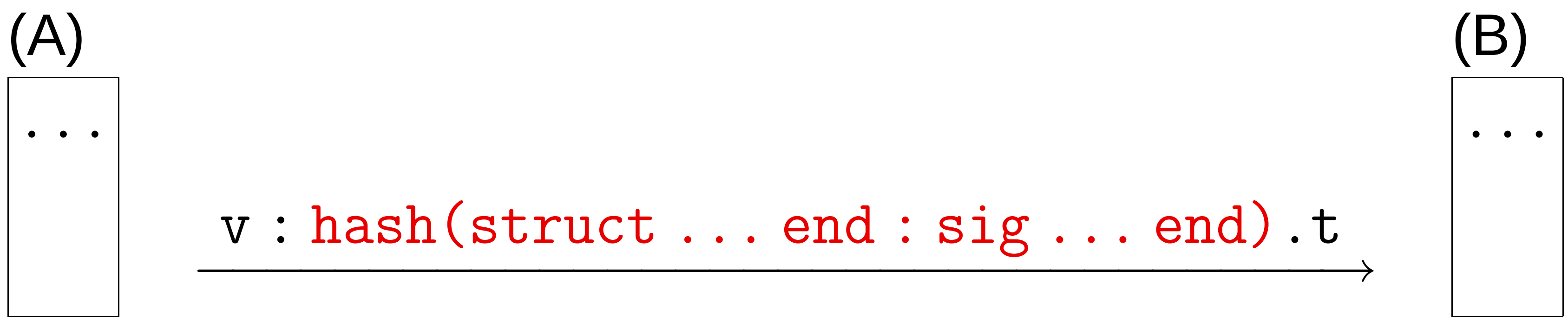
How do we obtain globally meaningful type names?

Solution: we construct them from module ***hashes***.



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.
- What exactly do we hash? A good candidate: abstract syntax trees of module definitions. But module dependencies require care.



1. Compile-time reduction: hash generation

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

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

$\longrightarrow_c$ inlining EvenC

```
send (marshal (0 : h.t))
```

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

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))
```

2. Compile-time reduction: module dependency (2/3)

$\longrightarrow_c$ inlining EvenC

$$\texttt{module CleanC} = \left(\begin{array}{lll} \texttt{struct} & \texttt{type s} = h\texttt{.t} * \texttt{bool} & \\ & \texttt{let create} = (0, \texttt{true}) \quad \ldots & \texttt{end} \\ : \texttt{sig} & \texttt{type s} & \\ & \texttt{val create} : \texttt{s} \quad \ldots & \texttt{end} \end{array}\right)$$

$$\texttt{send (marshal (CleanC.create : CleanC.s))}$$

where

$$h = \texttt{hash}\left(\begin{array}{lll} \texttt{struct} & \texttt{type t} = \texttt{int} & \\ & \texttt{let start} = 0 \quad \ldots & \texttt{end} \\ : \texttt{sig} & \texttt{type t} & \\ & \texttt{val start} : \texttt{t} \quad \ldots & \texttt{end} \end{array}\right)$$

2. Compile-time reduction: module dependency (2/3)

$\longrightarrow_c$ inlining EvenC

$$\texttt{module CleanC} = \left(\begin{array}{lll} \texttt{struct} & \texttt{type s} = h\texttt{.t} * \texttt{bool} & \\ & \texttt{let create} = (\texttt{0}, \texttt{true}) \quad \ldots & \texttt{end} \\ : \texttt{sig} & \texttt{type s} & \\ & \texttt{val create} : \texttt{s} \quad \ldots & \texttt{end} \end{array}\right)$$

$$\texttt{send (marshal (CleanC.create : CleanC.s))}$$

where

$$h = \texttt{hash}\left(\begin{array}{lll} \texttt{struct} & \texttt{type t} = \texttt{int} & \\ & \texttt{let start} = \texttt{0} \quad \ldots & \texttt{end} \\ : \texttt{sig} & \texttt{type t} & \\ & \texttt{val start} : \texttt{t} \quad \ldots & \texttt{end} \end{array}\right)$$

2. Compile-time reduction: module dependency (3/3)

$\longrightarrow_{c}$ inlining `CleanC`

$$\texttt{send (marshal (}\,(0, \texttt{true})\,:\,h'.\texttt{s}\,\texttt{))}$$

where

$$h = \texttt{hash}\begin{pmatrix} \texttt{struct} & \texttt{type t} = \texttt{int} & & \\ & \texttt{let start} = 0 & \ldots & \texttt{end} \\ : \texttt{sig} & \texttt{type t} & & \\ & \texttt{val start} : \texttt{t} & \ldots & \texttt{end} \end{pmatrix}$$

$$h' = \texttt{hash}\begin{pmatrix} \texttt{struct} & \texttt{type s} = h.\texttt{t} * \texttt{bool} & & \\ & \texttt{let create} = (0, \texttt{true}) & \ldots & \texttt{end} \\ : \texttt{sig} & \texttt{type s} & & \\ & \texttt{val create} : \texttt{s} & \ldots & \texttt{end} \end{pmatrix}$$

3. Compile-time reduction: coloured brackets

$$\texttt{module EvenC} =$$
$$\left(\begin{array}{llll} \texttt{struct} & \texttt{type t = int} & \texttt{let start = 0} & \texttt{...} \quad \texttt{end} \\ \texttt{: sig} & \texttt{type t} & \texttt{val start : t} & \texttt{...} \quad \texttt{end} \end{array}\right)$$

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

$\longrightarrow_c$ inlining EvenC

$$\texttt{send (marshal (}\ 0 \ \texttt{:}\ h\texttt{.t))}$$

where

$$h = \texttt{hash}\left(\begin{array}{llll} \texttt{struct} & \texttt{type t = int} & \texttt{let start = 0} & \texttt{...} \quad \texttt{end} \\ \texttt{: sig} & \texttt{type t} & \texttt{val start : t} & \texttt{...} \quad \texttt{end} \end{array}\right)$$

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

3. Compile-time reduction: coloured brackets

$$\texttt{module EvenC =}$$
$$\left(\begin{array}{lll} \texttt{struct} & \texttt{type t = int} & \texttt{let start = 0} \quad \ldots \quad \texttt{end} \\ \texttt{: sig} & \texttt{type t} & \texttt{val start : t} \quad \ldots \quad \texttt{end} \end{array}\right)$$

$$\texttt{send (marshal (EvenC.start : EvenC.t))}$$

$\longrightarrow_c$ inlining EvenC

$$\texttt{send (marshal (}[0]^{h.\texttt{t}}_{h} : h\texttt{.t))}$$

where

$$h = \texttt{hash}\left(\begin{array}{lll} \texttt{struct} & \texttt{type t = int} & \texttt{let start = 0} \quad \ldots \quad \texttt{end} \\ \texttt{: sig} & \texttt{type t} & \texttt{val start : t} \quad \ldots \quad \texttt{end} \end{array}\right)$$

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:

$$\begin{array}{lll} \mathtt{T} ::= & \ldots & \\ & \mid \; h.\mathtt{t} & \text{(not in user source code)} \end{array}$$

- coloured brackets in the expression grammar:

$$\begin{array}{lll} \mathtt{e} ::= & \ldots & \\ & \mid \; [\mathtt{e}]^{\mathtt{T}}_{h} & \text{(not in user source code)} \end{array}$$

Type equality $(\mathtt{E} \vdash_h \mathtt{T}_0 == \mathtt{T}_1)$

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

$$\frac{\mathtt{E} \vdash_h \mathtt{ok}}{\mathtt{E} \vdash_h h.\mathtt{t} == \mathtt{T}} \quad \text{if } h = \mathrm{hash}\begin{pmatrix} \mathtt{struct\ type\ t = T\ \dots\ end} \\ \mathtt{:\ sig\ type\ t\ \dots\ end} \end{pmatrix}$$

Coloured brackets

- determine where hash transparency occurs:

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

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 $\mathtt{erase}$ be the erasure function.

Let $\xrightarrow[\text{uncol}]{}$ be the corresponding run-time reduction relation.

- If $\mathtt{nil} \vdash_{\mathtt{ho}} \mathrm{e}{:}\mathrm{T}$ and $\mathrm{e} \longrightarrow_{\mathtt{ho}} \mathrm{e}'$ then $\mathtt{erase}(\mathrm{e}) \xrightarrow[\text{uncol}]{}^{\leqslant 1} \mathtt{erase}(\mathrm{e}')$.
- If $\mathtt{nil} \vdash_{\mathtt{ho}} \mathrm{e}{:}\mathrm{T}$ and $\mathtt{erase}(\mathrm{e}) \xrightarrow[\text{uncol}]{} \mathrm{e}_0$ then there exists e' such that $\mathtt{erase}(\mathrm{e}') = \mathrm{e}_0$ and $\mathrm{e} \longrightarrow_{\mathtt{ho}}^{\geqslant 1} \mathrm{e}'$.

As we said, $\xrightarrow[\text{uncol}]{}$ 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:

$$\mathtt{D}.\_ = \mathtt{module}\ \mathtt{U}_1 = \mathtt{M}_1{:}\mathtt{S}_1\ \mathtt{in}\ \ldots \mathtt{module}\ \mathtt{U}_n = \mathtt{M}_n{:}\mathtt{S}_n\ \mathtt{in}\ \_$$

Let $\sigma_{\mathtt{D}}$ be the substitution induced by compilation of the modules.

Suppose that no two modules have the same hash.

Then:

$$\mathtt{U}_1{:}\mathtt{S}_1, \ldots, \mathtt{U}_n{:}\mathtt{S}_n \vdash_\bullet \mathtt{T}_0 \mathtt{==} \mathtt{T}_1 \iff \sigma_{\mathtt{D}}\mathtt{T}_0 = \sigma_{\mathtt{D}}\mathtt{T}_1$$

static typing $\iff$ check performed by `unmarshal`