Efficiently Scrapping Boilerplate Code in OCaml

Dmitry Boulytchev and Sergey Mechtaev

Software Engineering Chair
Saint-Petersburg State University
{dboulytchev, mechtaev}@gmail.com

ACM SIGPLAN Workshop on ML, 2011

18 September 2011

Tokyo, Japan

“Scrap Your Boilerplate” — Motivation

```
type expr = Add of expr * expr | Const of int | Var of string

let rec increment = function
  Add   (x, y) -> Add   (increment x, increment y)
| Const x      -> Const (x + 1)
| Var   v      -> Var v

let rec suffixize = function
  Add   (x, y) -> Add (suffixize x, suffixize y)
| Const x      -> Const x
| Var   v      -> Var (v ^ "_suffix")
```

“Scrap Your Boilerplate” — Motivation

```
type expr = Add of expr * expr | Const of int | Var of string

let rec increment = function
  Add   (x, y) -> Add   (increment x, increment y)
| Const x      -> Const (x + 1)
| Var   v      -> Var v

let rec suffixize = function
  Add   (x, y) -> Add (suffixize x, suffixize y)
| Const x      -> Const x
| Var   v      -> Var (v ^ "_suffix")
```

- “Interesting function” —
  - `(fun x -> x+1)`
  - `(fun s -> s ^ "_suffix")`

“Scrap Your Boilerplate” — Motivation

```
type expr = Add of expr * expr | Const of int | Var of string

let rec increment = function
  Add   (x, y) -> Add   (increment x, increment y)
| Const x      -> Const (x + 1)
| Var   v      -> Var v

let rec suffixize = function
  Add   (x, y) -> Add (suffixize x, suffixize y)
| Const x      -> Const x
| Var   v      -> Var (v ^ "_suffix")
```

- “Interesting function” —
  - `(fun x -> x+1)`
  - `(fun s -> s ^ "_suffix")`
- Generic transformation.

“Scrap Your Boilerplate” — Generic Transformation

$$\mathsf{gmapT}^t : (\forall \alpha . \alpha \to \alpha) \to t \to t$$

“Scrap Your Boilerplate” — Generic Transformation

$$\text{gmapT}^{t} : (\forall\alpha.\alpha \to \alpha) \to t \to t$$

$\text{gmapT}^{\text{expr}} f =$ **function**

```
    Add (x, y) -> Add (f x, f y)
  | Const i -> Const (f i)
  | Var n -> Var (f i)
```

“Scrap Your Boilerplate” — Lifting

$$\text{lift} : (t \to t) \to \forall \alpha . \alpha \to \alpha$$

“Scrap Your Boilerplate” — Lifting

$$\mathsf{lift} : (t \to t) \to \forall \alpha . \alpha \to \alpha$$

$$\mathsf{lift}\,(f : t \to t) = \lambda x : \alpha \,.\, \begin{cases} f\,x & , \quad \alpha = t \\ x & , \quad \textit{otherwise} \end{cases}$$

"Scrap Your Boilerplate" — `everywhere`

$$\mathsf{everywhere} f\,(x : t) = f\,(\mathsf{gmapT}^{t}\,(\mathsf{everywhere} f)\,x)$$

“Scrap Your Boilerplate” — `everywhere`

$$\textsf{everywhere} f\,(x : t) = f\,(\textsf{gmapT}^t\,(\textsf{everywhere} f)\,x)$$

$$\textsf{increment} = \textsf{everywhere}(\textsf{lift}(\lambda x : \textsf{int}\,.\,x + 1))$$

“Scrap Your Boilerplate” — `everywhere`

$$\text{everywhere} f\,(x:t) = f\,(\text{gmapT}^{t}\,(\text{everywhere} f)\,x)$$

$$\text{increment} = \text{everywhere}\big(\text{lift}\big(\lambda x : \text{int}\,.\,x + 1\big)\big)$$

$$\text{suffixize} = \text{everywhere}\big(\text{lift}\big(\lambda x : \text{string}\,.\,x + \text{"_suffix"}\big)\big)$$

Weak Type Equality

- Encoding of type equality relation:

```
type ('a, 'b) eq

val refl    : unit -> ('a, 'a) eq
val symm    : ('a, 'b) eq -> ('b, 'a) eq
val trans   : ('a, 'b) eq -> ('b, 'c) eq -> ('a, 'c) eq
val coerce  : ('a, 'b) eq -> 'a -> 'b
```

Weak Type Equality

- Encoding of type equality relation:

```
type ('a, 'b) eq

val refl    : unit -> ('a, 'a) eq
val symm    : ('a, 'b) eq -> ('b, 'a) eq
val trans   : ('a, 'b) eq -> ('b, 'c) eq -> ('a, 'c) eq
val coerce  : ('a, 'b) eq -> 'a -> 'b
```

- Implementation:

```
type ('a, 'b) eq = ('a -> 'b) * ('b -> 'a)

let refl   ()             = id, id
let symm   (j, l)         = (l, j)
let trans  (f, g) (j, k)  = compose j f, compose g k
let coerce                = fst
```

Type Markers

- Representing types by values:

```
type 'a marker

val make: unit -> 'a marker
val compare : 'a marker -> 'b marker -> ('a, 'b) eq option
```

Type Markers

- Representing types by values:

```
type 'a marker

val make: unit -> 'a marker
val compare : 'a marker -> 'b marker -> ('a, 'b) eq option
```

- Sample implementation:

```
type 'a marker = unit ref

let make () = ref ()
let compare x y =
  if x == y then Some (Obj.magic (refl ())) else None
```

Lifting

- Type of “interesting function”:

```
type lifted = {f : 'a . 'a marker -> 'a -> 'a}
```

Lifting

- Type of “interesting function”:

```
type lifted = {f : 'a . 'a marker -> 'a -> 'a}
```

- Lifting primitive:

```
let lift (m : 'a marker) (f' : 'a -> 'a) =  {
  f = fun n x ->
        match compare m n with
        | None -> x
        | Some e ->
            coerce e (f' (coerce (symm e) x))
}
```

Lifting Example

```
# let int_m : int marker = make ();;
val int_m : int Type_marker.marker = <abstr>
# let inc = lift int_m (fun x -> x+1);;
val inc : Syb.lifted = {f = <fun>}
# inc.f string_m "abc";;
- : string = "abc"
# inc.f int_m 1;;
- : int = 2
# inc.f int_m "abc";;
Characters 12-17:
  inc.f int_m "abc";;
              ^^^^^
Error: This expression has type string but an expression
       was expected of type int
```

Type Information

```
type 'a typeinfo = {
  marker : 'a marker;
  gmapT  : transform -> 'a -> 'a
}
```

Type Information

```
type 'a typeinfo = {
  marker : 'a marker;
  gmapT  : transform -> 'a -> 'a
}

and transform =  {
  transform : 'a . 'a typeinfo -> 'a -> 'a
}
```

Specifying Type Information (I)

- Shallow case:

```
let int = {
  marker = int_m;
  gmapT  = fun _ x -> x
}
```

Specifying Type Information (I)

- Shallow case:

```
let int = {
  marker = int_m;
  gmapT  = fun _ x -> x
}
```

- Non-recursive case:

```
type t = A of a | B of b
let t = {
  marker = t_m;
  gmapT  = fun t ->
    function
    | A x -> A (t.transform a x)
    | B x -> B (t.transform b y)
}
```

Specifying Type Information (II)

- Recursive case:

```
let expr =
  let rec inner () = {
    marker = expr_m;
    gmapT  = fun t ->
      function
      | Add  (x, y) -> Add (t.transform (inner ()) x,
                             t.transform (inner ()) y
                            )
      | Var   s     -> Var (t.transform string s)
      | Const i     -> Const (t.tansform int i)
  }
  in inner ()
```

Implementing `everywhere`

```
let everywhere ti f =
  let rec transform : 'a . 'a typeinfo -> 'a -> 'a =
    fun ti x -> f.f ti.marker (ti.gmapT {transform} x)
  in
  transform ti
```

Implementing `everywhere`

```
let everywhere ti f =
  let rec transform : 'a . 'a typeinfo -> 'a -> 'a =
    fun ti x -> f.f ti.marker (ti.gmapT {transform} x)
  in
  transform ti

let increment = everywhere expr
  (lift int_m (fun i -> i+1))

let suffixize = everywhere expr
  (lift string_m (fun s -> s ^ "_suffix"))
```

Implementing `everywhere`

```
let everywhere ti f =
  let rec transform : 'a . 'a typeinfo -> 'a -> 'a =
    fun ti x -> f.f ti.marker (ti.gmapT {transform} x)
  in
  transform ti

let increment = everywhere expr
  (lift int_m (fun i -> i+1))

let suffixize = everywhere expr
  (lift string_m (fun s -> s ^ "_suffix"))
```

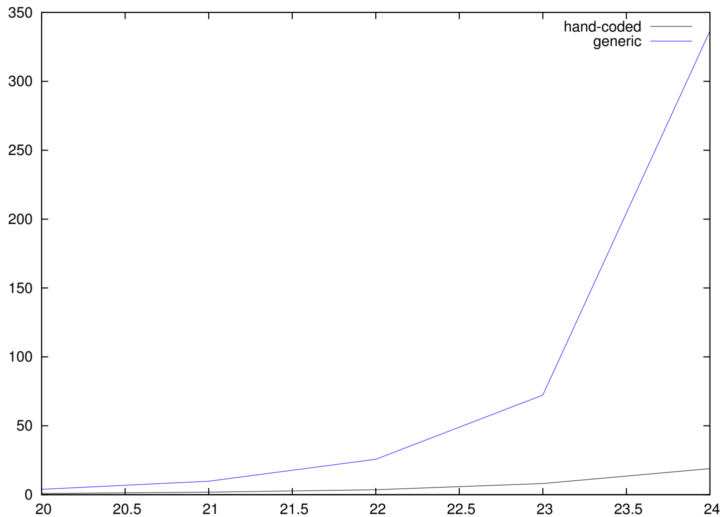
```
# suffixize (Add (Var "a", Const 1));;
- : expr = Add (Var "a_suffix", Const 1)
# increment (Add (Var "a", Const 1));;
- : expr = Add (Var "a", Const 2)
```

Canonical Example

```
datatype company = C of dept list
and dept         = D of name * manager * subunit list
and subunit      = PU of employee | DU of dept
and employee     = E of person * salary
and person       = P of name * address
and salary       = S of float
and manager      = employee
and name         = string
and address      = string

let increase =
  everywhere company (lift salary (function S x -> x *. 1.5))
```

Performance Issue



Specialization on Data Type: Lifting

```
let lift (m : 'a marker) (f' : 'a -> 'a) = {
  f = fun n  x  ->
        match compare m n with
        | None ->  x
        | Some e ->
              coerce e (f' (coerce (symm e) x))
}
```

Specialization on Data Type: Lifting

```
let lift (m : 'a marker) (f' : 'a -> 'a) =  {
  f = fun n ->
        match compare m n with
        | None -> fun x -> x
        | Some e ->
            fun x -> coerce e (f' (coerce (symm e) x))
}
```

Specialization on Data Type: Lifting

```
let lift (m : 'a marker) (f' : 'a -> 'a) =  {
  f = fun n ->
        match compare m n with
        | None -> fun x -> x
        | Some e ->
            let back = coerce e in
            let from = coerce (symm e) in
            fun x -> back (f' (from x))
}
```

Specialization of Data Type: Type Information

```
type expr = Add of expr * expr | Var of string | Const of int
let expr =
  let rec inner () = {
    marker = expr_m;
    gmapT  = fun t ->
      function
      | Add  (x, y) -> Add ( t.transform (inner ()) x,
                             t.transform (inner ()) y
                           )
      | Var   s     -> Var ( t.transform string s)
      | Const i     -> Const ( t.tansform int i)
  }
  in inner ()
```

Specialization of Data Type: Type Information

```
type expr = Add of expr * expr | Var of string | Const of int
let expr =
  let rec inner () = {
    marker = expr_m;
    gmapT  = fun t ->
      let t_expr = t.transform (inner ()) in
      let t_int = t.transform int in
      let t_string = t.transform string in
      function
      | Add  (x, y) -> Add (t_expr x, t_expr y)
      | Var   s     -> Var (t_string s)
      | Const i     -> Const (t_int i)
  }
  in inner ()
```

Specialization of Data Type: `everywhere`

```
let everywhere ti f =
  let rec transform : 'a . 'a typeinfo -> 'a -> 'a =
    fun ti x -> f.f ti.marker (ti.gmapT {transform} x)
  in
  transform ti
```

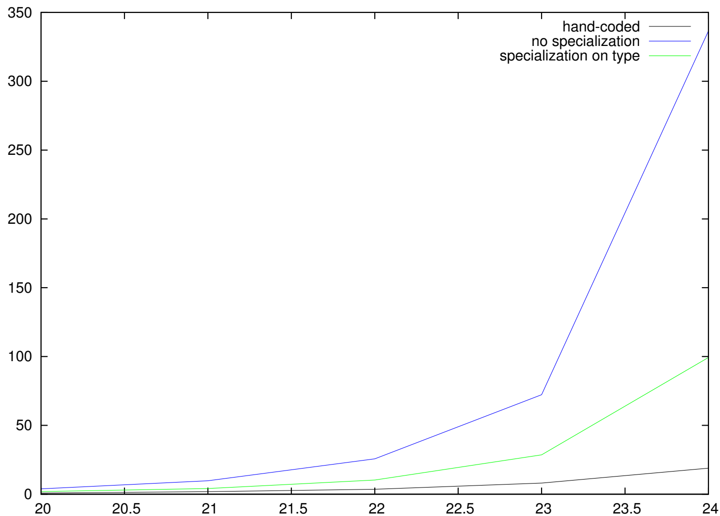
Specialization of Data Type: `everywhere` Loops

```
let everywhere ti f =
  let rec transform : 'a . 'a typeinfo -> 'a -> 'a =
    fun ti -> compose (f.f ti.marker) (ti.gmapT {transform})
  in
  transform ti
```

Specialization of Data Type: `everywhere` Revised

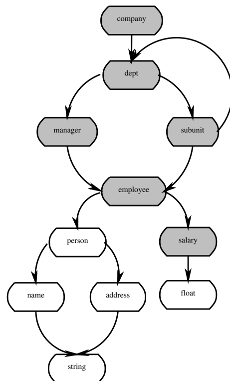
```
let everywhere ti f =
  let context = M.create () in
  let rec transform : 'a . 'a typeinfo -> 'a -> 'a =
    fun ti ->
      let m = ti.marker in
      let f = f.f m in
      try
        let tr = M.find context m in
        compose f (fun x -> !tr x)
      with Not_found ->
        let tr = M.stub () in
        M.add context m tr;
        tr := ti.gmapT {transform};
        M.remove context m;
        compose f !tr
  in
  transform ti
```

Performance Comparison

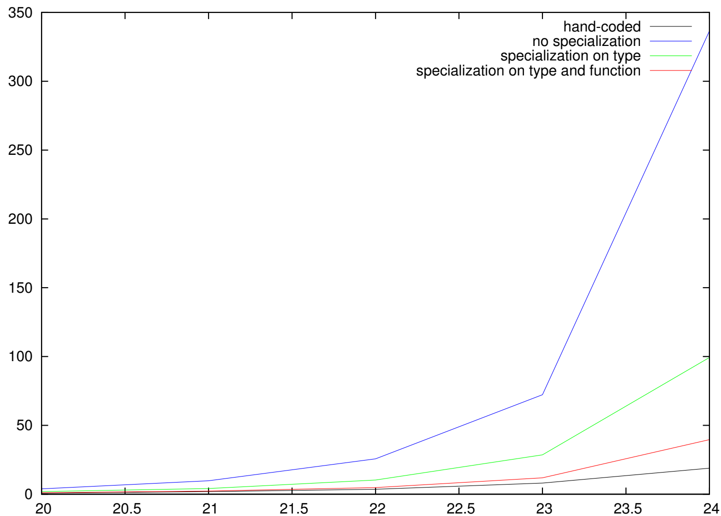


Specialization on Interesting Function

- Idea — prune the traversal of "non-interesting data".
- Trivial analysis over the type graph.
- Cumbersome to implement due to the lack of explicit type representation.
- Can be integrated in the implementation of `everywhere`.



Performance Comparison



Drawbacks and Limitations

- Data structures with cycles and sharing;

Drawbacks and Limitations

- Data structures with cycles and sharing;
- Type marker equality vs. true type equality:

```
module F (X : sig type t end) =
  struct
    datatype t = A of X.t | B
  end
module A = F (struct type t = int end)
module B = F (struct type t = int end)
```

Thank you!

- Source code and supporting materials:
  `http://oops.math.spbu.ru/syb-ocaml`