Specialisation of Higher-Order Functions for Debugging

Bernard Pope

Lee Naish

The University of Melbourne,

Australia

Motivation

Motivation (cont'd)

```
map id [1,2,3]
    => [1,2,3], correct?

map (+) [1,2,3]
    => [(+) 1, (+) 2, (+) 3], correct?
```

Mini FP

- $x \in Variables$
- $p \in \mathsf{Primitives}$
- $c \in \mathsf{Data}$ Constructors
- $f \in \mathsf{Type} \; \mathsf{Constructors}$
- $v \in \mathsf{Type} \; \mathsf{Variables}$

Mini FP (cont'd)

```
s \in \mathsf{Type} \; \mathsf{Schemes} dots = \forall \; \overline{v} \; . \; t d \in \mathsf{Declarations} dots = x = e  \mid f \; v_1 \ldots v_k = \{c_i \; t_{i1} \ldots t_{in_i}\}_{i=1}^m t \in \mathsf{Types} dots = v  \mid t_1 \to t_2   \mid f \; t_1 \ldots t_k
```

Mini FP (cont'd)

```
e \in \mathsf{Expressions}
::= \lambda x.e
\mid e_1 \ e_2 \mid
\mid x
\mid c
\mid let \ d_1 \dots \ d_n \ in \ e
\mid case \ e \ of \ a_1 \ \dots \ a_n
\mid p \ e_1 \ \dots \ e_n
a \in \mathsf{Alternatives}
::= c \ e_1 \ \dots \ e_n \mapsto e
\mid x \mapsto e
```

Example

```
main => Int
compose => (Int->Int->Int)
              ->(Int->Int)
                ->Int->Int->Int
       => (Int->Int)
             ->(Int->Int)->Int->Int
twice => (Int->Int)->Int->Int
plus => Int->Int->Int
id => Int->Int
(+) => Int->Int
```

```
main = compose1 plus
                    (plus 1)
                    (twice1 id 5)
compose1 = \f g x . f (g x)
compose2 = \f g x . f (g x)
twice1 = \xspace x compose2 x x
```

```
data F1 a b = F1 (a->b) String

data F2 a b c = F2 (a->b->c) String

apply1 = \x . case x of (F1 f s) -> f

apply2 = \x . case x of (F2 f s) -> f
```

```
compose1 = \f g x . apply2 f (apply1 g x)
compose2 = \f g x . apply1 f (apply1 g x)
twice1 = \x . compose2 x x
```

compare

```
compose :: (b -> c) -> (a -> b) -> a -> c
with:
```

compose2 :: (F1 b c) -> (F1 a b) -> a -> c

and also:

Cloning

Function clone (Q_1, S_1, P_1)

begin

if
$$Q_1 == \emptyset$$
 then P_1

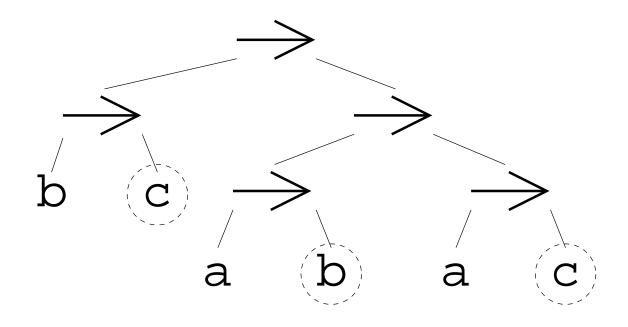
else

$$\begin{array}{l} (\mathsf{x} = \mathsf{e},\,\mathsf{t}) \in \mathsf{Q}_1 \\ \mathsf{Q}_2 \longleftarrow \mathsf{Q}_1 - \{(\mathsf{x} = \mathsf{e},\,\mathsf{t})\} \\ \mathsf{S}_2 \longleftarrow \mathsf{S}_1 \cup \{(\mathsf{x},\,\widehat{\mathsf{t}}\,)\} \\ \mathsf{C} \longleftarrow \mathsf{calls}\;(\mathsf{e},\,\mathsf{t}) \\ (\mathsf{Q}_3,\,\mathsf{P}_2) \longleftarrow \mathsf{newCalls}\;(\mathsf{C},\,\mathsf{S}_2,\,\mathsf{P}_1) \\ \texttt{return}\;\mathsf{clone}\;(\mathsf{Q}_2 \cup \mathsf{Q}_3,\,\mathsf{S}_2,\,\mathsf{P}_2) \end{array}$$

end

Cloning (cont'd)

 $compose :: \forall a, b, c : (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c$



call to compose2:

$$(Int \rightarrow \underline{Int}) \rightarrow (Int \rightarrow \underline{Int}) \rightarrow Int \rightarrow \underline{Int}$$

call to compose1:

$$(Int \rightarrow \underline{Int} \rightarrow \underline{Int}) \rightarrow (Int \rightarrow \underline{Int}) \rightarrow \underline{Int} \rightarrow \underline{Int} \rightarrow \underline{Int}$$

Cloning (cont'd)

The distinguishing type variables (DTVs) of a type scheme:

$$dtv(\forall \overline{v} \cdot t) = dtv(t)$$

$$dtv(f \ t_1 \dots t_k) = \bigcup_{i=1}^k dtv(t_i)$$

$$dtv(v) = \emptyset$$

$$dtv(t_1 \rightarrow v) = dtv(t_1) \cup \{v\}$$

$$dtv(t_1 \rightarrow t_2) = dtv(t_1) \cup dtv(t_2)$$

such that t_2 is not a type variable

Cloning (cont'd)

To compute the abstracted type given a type scheme s and a call type t_c :

$$abstype(s, t_c) = \{(v, order(t_v))\}$$

such that:

$$v \in dtv(s)$$

and:

$$(v, t_v) \in mgu(s, t_c)$$

and:

$$order(v) = 0$$

 $order(f \ t_1 \dots t_k) = 0$
 $order(t_1 \rightarrow t_2) = 1 + order(t_2)$

Encoding

Applications:

 e_1 e_2

such that: $e_2 :: t_1 \rightarrow t_2 \rightarrow \ldots \rightarrow t_{n+1}$

Are transformed using the rule: $encode\{e_1 \ e_2\} \Rightarrow e_1 \ encode\{e_2\}$

where the resulting expression denoted by $encode\{e_2\}$ has type: $F_n\ t_1\ t_2\ \dots\ t_{n+1}$

Encoding (cont'd)

Tricky case:

$$encode\{e_1 \ (x \ e_2)\}$$

such that:
$$(x e_2) :: t_1 \rightarrow t_2 \rightarrow \ldots \rightarrow t_k$$

and x is lambda bound, $e_2 :: t_e$

In the transformed program x will have type: $F_k \ t_e \ t_1 \ t_2 \ \dots \ t_k$

We want the type of the encoded form of $(x e_2)$ to be: $F_{k-1} t_1 t_2 \ldots t_k$

Encoding (cont'd)

Encoding (cont'd)

From the example earlier

```
let list = [1,2,3]
   in . . . map id list . . .
       . . . map (+) list . . .
we get:
   let list = [1,2,3]
   in . . . map1 (F1 id "id") list . . .
       . . . map2 (F2 (+) "(+)") list . . .
map2 :: (F2 a b c) -> [a] -> [F1 b c]
map2 = \floar{1}.
         case 1 of
          [] -> []
          (:) x xs
              -> (:) (toF1 f x) (map2 f xs)
                                        21
```

Discussion

What to do about:

- 1. calls across module boundaries?
- 2. type classes, and other overloaded identifiers?
- 3. polymorphic recursion?

Discussion (cont'd)

Polymorphic recursion:

```
f :: (a -> b) -> c
f = \g . f (\x . g)
.
.
.
. f id . . .
```

For each recursive call the abstracted type is different. This implies an infinite number of clones.

End notes

- 1. Implementation status?
- 2. Other uses of the algorithm?
- Relation to de-functionalisation or firstification
- 4. Data constructors, especially those with explicit functional arguments:

data Foo a
$$b = C (a \rightarrow b)$$