Untyped general polymorphic functions

Martin Pettai

February 5, 2010

Introduction

- We would like to have a functional language where it is possible to define general polymorphic functions
  - Return type of a function is uniquely determined by the argument type
  - All polymorphic functions where the implied function on types belongs to a certain large class of total functions, should be definable
  - Higher-order polymorphic functions
  - Static type checking
- We will see how polymorphic functions can be defined in
  - dynamically typed languages with `typecase`
  - extensional polymorphism, which uses `typecase` in a statically typed language
  - our language, which uses `typecase` in untyped functions in an otherwise statically typed language

Dynamically typed languages with `typecase`

- Run-time values are tagged with types, e.g. 3 is internally `(Int, 3)`
- We can also include pure types (without a value) as ordinary run-time objects
  - `typeof` operator to get the type of a value, e.g. `typeof 3 ==> Int`
- In such a language types can be computed with (e.g. branching, recursion) as easily as values

Dynamically typed languages with `typecase`

- We can easily define polymorphic functions:

```
let f = \ x .
           typecase (typeof x) of
              Int    -> x + 3;
              String -> x ++ "s";
              _      -> "ERROR";
           end
in
   (f 3, f "symbol", f True)
=> (3, "symbols", "ERROR")
```

Dynamically typed languages with **typecase**

- We can use `typecase` inside any expression:

```
let f = \ x .
            100 * typecase (typeof x) of
                     Int    -> x;
                     String -> length x;
                     _      -> 13;
                  end
in
   [f 3, f "symbol", f True]
=> [300, 600, 1300]
```

Dynamically typed languages with `typecase`

- We can have recursion over types:

```
let rec f = \ x .
    typecase (typeof x) of
        Int    -> x;
        List _ ->
            let y = map f x
            in typecase (typeof y) of List a ->
                    typecase a of
                        Int       -> Just (sum y);
                        Maybe Int ->
                            case y of
                                Just z :: zs -> z;
                                _            -> 0;
                            end; end; end; end
```

Dynamically typed languages with `typecase`

- Suppose we also have typed functions, e.g.
  `\ (x : Int) : Int . x + 2` has type `Int -> Int`

Dynamically typed languages with typecase

- We can also have higher-order functions:

```
let reverseargs f =
   let rec revtypes t cont =
      typecase t of
         Unit        -> cont t;
         List _      -> cont t;
         (t1 -> t2) -> revtypes t2 (\ u . t1 -> cont u)
   in let rec proc ts cont =
      typecase ts of
         Unit        -> cont f;
         List _      -> cont f;
         (t -> ts') -> \ (x : t) : ts' .
                                 proc ts' (\ g . cont (g x))
   in
      proc (revtypes (typeof f) (\ t . t)) (\ g . g)
```

Statically typed functional languages

- Polymorphic functions are more difficult to define
- There are only typed functions, no untyped functions
- Usually the argument type and result type must be specified (or inferred by the compiler) and the function type is constructed from these
- These types may contain universally quantified type variables (this gives us parametric polymorphism), e.g. `forall a. List (a,a) -> Maybe a`
- *Ad-hoc* polymorphism is more difficult to achieve

Extensional polymorphism

- Introduced by Dubois, Rouaix, and Weis in 1995
- Example:

```
let rec generic flat =
  case d1 list -> d2 list of
    t1 list list -> t2 list => (function l ->
                                   flat (flatten l))
  | t list -> t list         => (function l -> l)
```

- Branching only on the type of a polymorphic value
- A type inference algorithm is used to annotate subexpressions (including polymorphic variables) with types
- Another algorithm is used to check that polymorphic values are only used at the types for which they are defined
    - For this, branching and recursion on the inferred type is performed

Extensional polymorphism: problems

- Type system is complicated
  - Must include polymorphic types
  - Polymorphic values have several types: the general type scheme, the type scheme for each branch, the inferred types for the used instances
- Type inference is complicated
  - The type of a variable is not constant
  - The return type of a function might not be uniquely determined by the argument type
- Higher-order (impredicative) polymorphism difficult to achieve
  - Higher-order polymorphic types make type inference undecidable

Our approach: drop the polymorphic types

- Because polymorphic types create many problems, we leave polymorphic functions untyped, i.e. they do not have a type in the type system (although they have an implicit type outside the type system)
- Our untyped polymorphic functions can use higher-order polymorphism, typecase, and pure types
- Expressions will be reduced in two phases: static (compile-time) and dynamic (run-time) phase
  - Typecases and other type-level constructs will be reduced in the static phase
  - If (and only if) the program is not type-correct, type errors will occur during static-phase reductions
  - For dynamic-phase reductions, type information is not needed and type errors cannot occur
  - The type system only defines types for the expressions that cannot be reduced further in the static phase (we call those expressions box expressions)
  - Return type of an untyped polymorphic function is uniquely determined by the argument type

Our language: syntax

```
SIMPLETYPE ::= Unit | List SIMPLETYPE
             | SIMPLETYPE -> SIMPLETYPE
EXPR ::= VAR | VAR : SIMPLETYPE | unit | nil SIMPLETYPE
       | cons EXPR EXPR | typeof EXPR | EXPR EXPR
       | tlam VAR ( VAR :< TYPE ) . EXPR
       | vlam VAR ( VAR : EXPR ) : EXPR . EXPR
       | iffun EXPR then EXPR else EXPR
       | iftype EXPR then EXPR else EXPR
       | tcase EXPR of Unit -> EXPR; List VAR -> EXPR;
                      (VAR -> VAR) -> EXPR
       | vcase EXPR of nil -> EXPR; cons VAR VAR -> EXPR
       | SIMPLETYPE
TYPE ::= SIMPLETYPE | Type SIMPLETYPE | Fun NAT
NAT ::= 0 | 1 | 2 | ...
```

Our language: box expressions

```
BOX ::= VAR : SIMPLETYPE | unit | nil SIMPLETYPE
      | cons BOX_v BOX_v | BOX_v BOX_v
      | tlam VAR ( VAR :< TYPE ) . EXPR
      | vlam VAR ( VAR : SIMPLETYPE ) : SIMPLETYPE . EXPR
      | vcase BOX_v of nil -> BOX_v; cons VAR VAR -> BOX_v
      | SIMPLETYPE
BOX_v ::= VAR : SIMPLETYPE | unit | nil SIMPLETYPE
      | cons BOX_v BOX_v | BOX_v BOX_v
      | vlam VAR ( VAR : SIMPLETYPE ) : SIMPLETYPE . BOX_v
      | vcase BOX_v of nil -> BOX_v; cons VAR VAR -> BOX_v
```

Our language: final expressions

```
FINAL ::= VAR : SIMPLETYPE | unit | nil SIMPLETYPE
       | cons FINAL FINAL
       | tlam VAR ( VAR :< TYPE ) . EXPR
       | vlam VAR ( VAR : SIMPLETYPE ) : SIMPLETYPE . EXPR
       | SIMPLETYPE
```

Our language: type rules

$$\frac{}{(x\ \texttt{:}\ t) : t} \qquad \frac{}{\texttt{unit} : \texttt{Unit}} \qquad \frac{}{\texttt{nil}\ \ t : \texttt{List}\ \ t}$$

$$\frac{b_1 : t \qquad b_2 : \texttt{List}\ \ t}{\texttt{cons}\ \ b_1\ \ b_2 : \texttt{List}\ \ t} \qquad \frac{}{\texttt{tlam}\ \ x_1\ \ \texttt{(}\ \ x_2\ \ \texttt{:<}\ \ \tau\ \ \texttt{)}\ \ \texttt{.}\ \ e : \max(\tau, \texttt{Fun}\ \ \texttt{0})}$$

$$\frac{}{(\texttt{vlam}\ \ x_1\ \ \texttt{(}\ \ x_2\ \ \texttt{:}\ \ t_1\ \ \texttt{)}\ \ \texttt{:}\ \ t_2\ \ \texttt{.}\ \ e) : (t_1\ \ \texttt{->}\ \ t_2)} \qquad \frac{}{t : \texttt{Type}\ \ t}$$

$$\frac{b_1 : \texttt{List}\ \ t_1 \qquad b_2 : t_2 \qquad b_3 : t_2}{(\texttt{vcase}\ \ b_1\ \ \texttt{of}\ \ \texttt{nil}\ \ \texttt{->}\ \ b_2\texttt{;}\ \ \texttt{cons}\ \ x_1\ \ x_2\ \ \texttt{->}\ \ b_3) : t_2}$$

$$\frac{b_1 : (t_1\ \ \texttt{->}\ \ t_2) \qquad b_2 : t_1}{b_1\ \ b_2 : t_2}$$

Our language: type ordering

- To be able to verify the termination of type-level recursion, we define on the set `TYPE` a partial order that is well-founded and computable:

$$\frac{t_1 < t_2 \qquad t_2 < t_3}{t_1 < t_3} \qquad \frac{t_1 < t_2}{\texttt{Type}\ \ t_1 < \texttt{Type}\ \ t_2} \qquad \frac{n_1 <_{\texttt{NAT}} n_2}{\texttt{Fun}\ \ n_1 < \texttt{Fun}\ \ n_2}$$

$$\frac{}{t < \texttt{List}\ \ t} \qquad \frac{}{t_1 < (t_1\ \texttt{->}\ t_2)} \qquad \frac{}{t_2 < (t_1\ \texttt{->}\ t_2)}$$

$$\frac{}{t_1 < \texttt{Type}\ \ t_2} \qquad \frac{}{t < \texttt{Fun}\ \ n} \qquad \frac{}{\texttt{Type}\ \ t < \texttt{Fun}\ \ n}$$

Our language: example

```
tlet reverseargs{0} f =
   tlet revtypes{1} t cont{0} =
      tcase t of
         Unit          -> cont t;
         List _        -> cont t;
         (t1 -> t2)    -> revtypes t2 (tlam u . t1 -> cont u)
   in tlet proc{1} ts cont{0} =
      tcase ts of
         Unit          -> cont f;
         List _        -> cont f;
         (t -> ts')    -> vlam (x : t) : ts' .
                                proc ts' (tlam g . cont (g x))
   in
      proc (revtypes (typeof f) (tlam t . t)) (tlam g . g)
```

Our language: example

- If we apply `reverseargs` to `f : Unit -> (Unit -> Unit) -> List Unit -> Unit`, it will reduce in the type level to the expression

```
vlam (x1 : List Unit) : (Unit -> Unit) -> Unit -> Unit .
   vlam (x2 : Unit -> Unit) : Unit -> Unit .
       vlam (x3 : Unit) : Unit .
          (f : Unit -> (Unit -> Unit) -> List Unit -> Unit
              ) x3 x2 x1
```

- which has type `List Unit -> (Unit -> Unit) -> Unit -> Unit`.

Our language vs dynamically typed languages

- If we do not require decidability of type checking
  - We can drop the kind annotations and use the same syntax as the dynamically typed language
  - Type checking only uses type-level information
  - If the type checking terminates, the program is guaranteed not to produce type errors at run time
    - Thus we have statically type-checked the dynamically typed program

Conclusion

- We have a language that allows defining very general higher-order polymorphic functions
  - which can be defined almost as easily as in dynamically typed languages with `typecase`
  - the type system is very simple (no need for polymorphic types)
- But we have not been able to prove the decidability of type checking
  - Maybe it is necessary to change the kind system

The End