General polymorphism with type-level primitive recursion and its implementation in Fumontrix

Martin Pettai

October 4, 2008

Introduction

- We will first see how polymorphic functions can be defined in
    - dynamically typed languages with `typecase`
    - statically typed languages (especially Haskell with GHC extensions)
    - statically typed languages with a `dynamic` type
- Then we will give an overview of type-level programmming in Fumontrix
- And finally see how type-level programmming can be used to define very general polymorphic functions in Fumontrix

Dynamically typed languages with `typecase`

- We can easily define polymorphic functions:

```
let f = \ x .
           typecase 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 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:

```
f = \ x .
    typecase x of
        Int    -> x;
        List _ ->
            let y = map f x
            in typecase 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
```

Statically typed functional languages

- Polymorphic functions are more difficult to define
- 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`

Statically typed functional languages: GHC

- Ad hoc polymorphism is not always available. In Haskell it requires type classes (and if the result type depends on the argument type, also functional dependencies, and thus, GHC extensions)

```
class C1 a b | a -> b where
   f1 :: a -> b
instance C1 Bool Integer where
   f1 b = bool2int b
instance C1 Integer Bool where
   f1 x = int2bool x

(f1 3, f1 True) => (True, 1)
```

Statically typed functional languages: GHC

- Now suppose we want to define a polymorphic function whose return type is defined by recursion over the argument type
- This was very easy in dynamically typed languages

Statically typed functional languages: GHC

- In GHC, we have to use a separate type class for each branching

```
class C4a a b | a -> b where
   f4a :: a -> b
instance C4a Bool Integer where
   f4a b = bool2int b
instance C4a Integer Bool where
   f4a x = int2bool x

class C4 a b | a -> b where
   f4 :: a -> b
instance C4 Integer Bool where
   f4 x = True
instance (C4 a b, C4a b c) => C4 [a] c where
   f4 x = f4a (f4 (head x))
```

Statically typed functional languages: GHC

- In GHC, the use of default cases is limited

```
class C8 a where
   f8 :: a -> Integer
instance C8 (Integer,b,c,d) where
   f8 (x,_,_,_) = x
instance C8 (a,Integer,c,d) where
   f8 (_,y,_,_) = 10*y
instance C8 (a,b,Integer,d) where
   f8 (_,_,z,_) = 100*z
instance C8 (a,b,c,Integer) where
   f8 (_,_,_,w) = 1000*w
instance C8 (a,b,c,d) where
   f8 (_,_,_,_) = 0
```

- Here only types with at most one `Integer` component are accepted
- To accept all 16 cases, 16 separate instance declarations must be given

Statically typed functional languages: GHC

- If we want the result type to depend on the argument type (i.e. use functional dependencies), default cases cannot be used at all
- This is not accepted by GHC:

```
class C7 a b | a -> b where
   f7 :: a -> b
instance C7 Bool Int where
   f7 = bool2int
instance C7 a Bool where
   f7 = const True
```

Statically typed functional languages: GHC

- If we want the result type to depend on the argument type (i.e. use functional dependencies), default cases cannot be used at all
- This is not accepted by GHC:

```
class C7 a b | a -> b where
   f7 :: a -> b
instance C7 Bool Int where
   f7 = bool2int
instance C7 a Bool where
   f7 = const True
```

- In a dynamically typed language, this is easily defined:

```
f7 = \ x .
        typecase x of
           Bool -> bool2int x;
           _    -> True;
        end
```

Statically typed functional languages: GHC

- Thus GHC has several limitations and drawbacks in defining polymorphic functions:
    - To define non-parametrically polymorphic functions, it is necessary to use type classes
        - Each branching (typecase) requires a separate type class, each case requires a separate instance
        - Type classes and instances can only occur at the top level of module, thus the structure of function definition is lost
        - Type-level programming with GHC type classes is more similar to logic programming than functional programming
        - Default cases (logical negation) are not allowed, except in some limited situations
        - Type checking can be non-terminating
        - Higher-order functions cannot be used

Statically typed languages with a `dynamic` *type*

- These languages are mostly static but have a special type `dynamic`, which can contain values of any type, and this type is determined at run time
- Values of type `dynamic` are constructed using the keyword `dynamic`, e.g. `dynamic(5:Int)` or `dynamic((\ x:Int . x+2) : Int -> Int)`
- Types can be used for branching at run time using the `typecase` operator
- Functions of type `dynamic -> dynamic` can be as polymorphic as the functions in dynamically typed languages

Statically typed languages with a `dynamic` *type*

- These languages have several drawbacks:
  - Result type of a `dynamic` function application is determined at run time, it cannot be determined statically from the argument type as in non-`dynamic`-type functions

What we need

- We need a language that
  - allows a large class of type-level total functions to be defined
  - maintains decidability of type checking
  - allows type-level programming as easily as data-level programming, e.g. using similar syntax
    - λ-expressions
    - higher-order functions
    - case expressions with overlapping patterns

Strong functional programming

- All (data-level) definable functions are total
  - Primitive recursion
  - Higher-order functions
  - Thus higher-order primitive recursive functionals of finite type can be defined

Ackermann function

- Definition:

$$A(0, n) = n + 1$$
$$A(m + 1, 0) = A(m, 1)$$
$$A(m + 1, n + 1) = A(m, A(m + 1, n))$$

- This is not a primitive recursive function, but it is a primitive recursive functional of finite type

Fumontrix

- The language I implemented for my master thesis
- A lazy statically typed functional language
- Syntax similar to Haskell
- Interpreter also implemented in Haskell
- It was created to remove some shortcomings of Haskell (GHC)

Fumontrix: the type level

- Type level functions are defined using λ-abstractions:
  - `\ x .  Pair x x`
  - can be used as anonymous functions
- They can also be bound to type level variables
  - These declarations (and all other declarations) can occur in any `let`-expression

```
let
   data Pair A B = Pair A B
in
   (exists B : * -> * -> * . Int -> Int -> B Int Int)(
      let
         type f = \ x . x -> x -> Pair x x
      in
         Pair $: Int $: Int : f Int);

main = m1
```

Fumontrix: primitive recursion

- There are also primitive recursive type-level λ-abstractions:

```
data Zero;
data Succ A;
type tfSum = \ a . \ b rec .
   case b of
      Zero    -> a;
      Succ b' -> Succ (rec b');
   end;
```

Fumontrix: Ackermann function

- Here is the Ackermann function in Fumontrix:

```
type acker = \ m rec:f * -> * . \ n rec .
   case m of
      Zero    -> Succ n;
      Succ m' ->
         case n of
            Zero    -> rec:f m' (Succ Zero);
            Succ n' -> rec:f m' (rec n');
         end;
   end;
```

- This uses a higher-order function to have two nested primitive recursions
- acker has kind `* -> * -> *`

Fumontrix: functions from types to values

- We can also have functions that transform types to values

```
type typeNatToValue = \ a rec @ .
   case a of
      Zero   -> value 0;
      Succ b -> value 1 + (type rec b);
   end;
```

- `typeNatToValue` has kind `* -> @`

Fumontrix: functions from values to types

- We can also have functions that transform values to types but here the only necessary function is the operator `typeof`, the other functions of kind `@ -> *` can be expressed as a composition of a function of kind `* -> *` with `typeof`.

Fumontrix: polymorphic functions

- We can also have functions that transform values to values and these can be used to implement polymorphic functions over values
- Here is a simple ad-hoc-polymorphic function:

```
type f = \ x : @ .
    case typeof type x of
        Int ->
            value (type x) != 0;
        Bool ->
            value if (type x) 1 0;
    end;
```

  - `type f (value 2)` ⇒ `True`
  - `type f (value True)` ⇒ `1`

Multi-stage programming

- In Fumontrix there are two evaluation stages:
  - Static stage, where type level (and kind level) expressions are evaluated
  - Dynamic stage, where data level expressions are evaluated
- This is similar to multi-stage programming, which has been implemented in the language MetaML:
  - MetaML multi-stage programming can have more than 2 stages
    - but all these stages are data-level
  - It has operators ˜ and $\langle\cdot\rangle$ instead of `type` and `value`
- Both Fumontrix stages and MetaML allow to define some type-safe macros

Multi-stage programming: macros

- MetaML:

```
fun exp (n,x) = (* : int x <int> -> <int> *)
   if n = 0 then
      <1>
   else
      <~x * ~(exp (n-1, ~x))>
```

- Fumontrix:

```
type exp =
   \ n rec @ -> @ . \ x : @ .
      case n of
         Zero    -> value 1;
         Succ n' -> value (type x) * (type rec n' x);
      end;
```

Multi-stage programming: macros

- MetaML:
- `exp (3, <x>)` ⇒ `<x * x * x * 1>`
  - Internally

    ```
    <let val d =
        let val e = x %* 1
        in x %* e end
     in x %* d end>
    ```
- Fumontrix:
- `type exp (Succ (Succ (Succ Zero))) (value x)` ⇒ `x * x * x * 1`
  - Internally

    ```
    type value (type value x) *
               (type value (type value x) *
                           (type value (type value x) *
                                       (type value 1)))
    ```
  - These `type value` constructs are needed to preserve lexical scoping, each contains a link to the environment where the data-level expression inside the construct is to be evaluated

Fumontrix: general polymorphic functions

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

Fumontrix: general polymorphic functions

```
type f (value 3)
   ==> 3
type f (value 3 :: 4 :: Nil)
   ==> Just 7
type f (value (3 :: 4 :: Nil) ::
              (30 :: 40 :: Nil) :: Nil)
   ==> 7
type f (value ((3 :: 4 :: Nil) ::
               (30 :: 40 :: Nil) :: Nil)
               ::
              ((300 :: 400 :: Nil) ::
               (3000 :: 4000 :: Nil) :: Nil)
               :: Nil)
   ==> Just 707
```

Fumontrix: general polymorphic functions

- Fumontrix allows to define all polymorphic functions for which
    - the result type is a function of the argument type
    - and this function on types is a primitive recursive functional of finite type
- These functions do not have an (explicit) type in the type system (since the equality of even two primitive recursive functions is undecidable) but they can be considered to have an implicit type outside the type system.
    - In other statically typed functional languages functions must have an explicit type (provided by the programmer or by the compiler) in the type system, so that they can be used as arguments of higher-order functions
    - In Fumontrix only kinds need to be explicit, types can be checked using typecase expressions
    - Thus in Fumontrix we can have more general polymorphism than in other statically typed functional languages

Conclusion

- We have implemented a language Fumontrix that
  - has powerful type-level programming
    - where all functions are total
    - which can be used almost as easily as data-level programming
  - allows defining very general polymorphic functions
    - which can be defined almost as easily as in dynamically typed languages with `typecase`

The End