CSc 372

Comparative Programming Languages

10 : Haskell — Polymorphic Functions

Christian Collberg

collberg+372@gmail.com

Department of Computer Science

University of Arizona

Polymorphic Functions

- In many languages we can't write a generic sort routine, i.e. one that can sort arrays of integers as well as arrays of reals:

```
procedure Sort (
        var A : array of <type>;
        n :  integer);
```

- In Haskell (and many other FP languages) we can write polymorphic ("many shapes") functions.
- Functions of polymorphic type are defined by using type variables in the signature:

```
length ::  [a] -> Int
length s = ...
```

Polymorphic Functions...

- `length` is a function from lists of elements of some (unspecified) type `a`, to integer. I.e. it doesn't matter if we're taking the length of a list of integers or a list of reals or strings, the algorithm is the same.

```
length [1,2,3]                ⇒ 3 (list of Int)
length ["Hi ", "there", "!"] ⇒ 3 (list of String)
length "Hi!"                  ⇒ 3 (list of Char)
```

Polymorphic Functions. . .

- We have already used a number of polymorphic functions that are defined in the standard prelude.
- `head` is a function from “lists-of-things” to “things”:

```
head ::  [a] -> a
```

- `tail` is a function from lists of elements of some type , to a list of elements of the same type:

```
tail ::  [a] -> [a]
```

- `cons "(:)"` takes two arguments: an element of some type `a` and a list of elements of the same type. It returns a list of elements of type `a`:

```
(:)  ::  a -> [a] -> [a]
```

Polymorphic Functions. . .

- Note that `head` and `tail` always take a list as their argument. `tail` always returns a list, but `head` can return any type of object, including a list.
- Note that it is because of Haskell's strong typing that we can only create lists of the same type of element. If we tried to do

  ```
  ? 5 : [True]
  ```

  the Haskell type checker would complain that we were consing an Int onto a list of Bools, while the type of ":" is

  ```
  (:) :: a -> [a] -> [a]
  ```

Context Predicates

The `remdups` Function

- Remember the `remdups` function:

```
remdups [1]          ⇒ [1]
remdups [1,2,1]      ⇒ [1,2,1]
remdups [1,2,1,1,2]  ⇒ [1,2,2]
remdups [1,1,1,2]    ⇒ [1,2,1]
```

- Algorithm in Haskell:

```
remdups ::  [Int] -> [Int]
remdups x:y:xs =
   if x == y then
      remdups y:xs        ⇐ case 1
   else
      x :  remdups y:xs   ⇐ case 2
remdups xs = xs           ⇐ case 3
```

Context Predicates

- Obviously `remdups` should work for any list, not just lists of `Ints`. Removing duplicates from a list of strings is no different from removing duplicates from a list of integers.
- However, there's a complication. In order to remove duplicates from a list, we must be able to compare list elements for equality.
- The polymorphic type

  ```
  [a] -> [a]
  ```

  is therefore a bit too general, since it would allow any type, even one for which equality is not defined.

Context Predicates. . .

- Haskell uses **context predicates** to restrict polymorphic types:

  ```
  remdups ::  Eq [a] => [a] -> [a]
  ```

  Now, `remdups` may only be applied to list of elements where the element type has == and \= defined.

- `Eq` is called a **type class.** `Ord` is another useful type class. It is used to restrict the polymorphic type of a function to types for which the relational operators (`<, <=, >, >=`) have been defined.

Multiple Context Predicates

- Consider the `signum` Function:

```
signum ::   (Num a, Ord a) => a -> Int
signum n |   n == 0     = 0
         |   n > 0      = 1
         |   n < 0      = -1
```

- `signum` can be applied to any type that is a number (hence the `Num a` predicate), and for which the relational operators are defined (`Ord a`).
- Without these restrictions, the polymorphic `signum` function could have been applied to lists, for example, which would not have made sense.

Conclusion

Summary...

- We want to define functions that are as reusable as possible.
  1. Polymophic functions are reusable because they can be applied to arguments of different types.
  2. Curried functions are reusable because they can be specialized; i.e. from a curried function `f` we can create a new function `f'` simply by "plugging in" values for some of the arguments, and leaving others undefined.

Summary

- A polymorphic function is defined using type variables in the signature. A type variable can represent an arbitrary type.
- All occurences of a particular type variable appearing in a type signature must represent the same type.
- An identifier will be treated as an operator symbol if it is enclosed in backquotes: `"`"`.
- An operator symbol can be treaded as an identifier by enclosing it in parenthesis: `(+)`.

Homework

- Define a polymorphic function `dup x` which returns a tuple with the argument duplicated.

Example:

```
?   dup 1
    (1,1)

?   dup "Hello, me again!"
    ("Hello, me again!",
    "Hello, me again!")

?   dup (dup 3.14)
    ((3.14,3.14), (3.14,3.14))
```

Homework

- Define a polymorphic function `copy n x` which returns a list of `n` copies of `x`.

Example:

```
?   copy 5 "five"
    ["five","five","five",
       "five","five"]

?   copy 5 5
    [5,5,5,5,5]

?   copy 5 (dup 5)
    [(5,5),(5,5),(5,5),(5,5),(5,5)]
```

Homework

- Let `f` be a function from `Int` to `Int`, i.e. `f :: Int -> Int`. Define a function `total f x` so that `total f` is the function which at value `n` gives the total `f 0 + f 1 + ··· + f n`.

<u>Example:</u>

```
double x = 2*x
pow2 x = x^2
totDub = total double
totPow = total pow2
?  totDub 5
   30
?  totPow 5
   55
```