CSc 520

Principles of Programming Languages

15: Haskell — Curried Functions

Christian Collberg

collberg@cs.arizona.edu

Department of Computer Science

University of Arizona

Declaring Infix Functions

- Sometimes it is more natural to use an infix notation for a function application, rather than the normal prefix one:
  - `5 + 6` (infix)
  - `(+) 5 6` (prefix)
- Haskell predeclares some infix operators in the standard prelude, such as those for arithmetic.
- For each operator we need to specify its precedence and associativity. The higher precedence of an operator, the stronger it binds (attracts) its arguments: hence:

```
3 + 5*4  ≡ 3 + (5*4)
3 + 5*4  ≢ (3 + 5) * 4
```


Declaring Infix Functions. . .

- The associativity of an operator describes how it binds when combined with operators of equal precedence. So, is

```
5-3+9     ≡ (5-3)+9 = 11
            OR
5-3+9     ≡ 5-(3+9) = -7
```

  The answer is that + and - associate to the left, i.e. parentheses are inserted from the left.
- Some operators are right associative: `5^3^2` ≡ `5^(3^2)`
- Some operators have free (or no) associativity. Combining operators with free associativity is an error:

```
5 == 4 < 3               ⇒ ERROR
```


Declaring Infix Functions. . .

- The syntax for declaring operators:

```
infixr prec oper   -- right assoc.
infixl prec oper   -- left assoc.
infix prec oper    -- free assoc.
```

From the standard prelude:

```
infixl 7 *
infix 7  /, `div`, `rem`, `mod`
infix 4  ==, /=, <, <=, >=, >
```

- An infix function can be used in a prefix function application, by including it in parenthesis. Example:

```
?  (+) 5 ((*) 6 4)
    29
```

Multi-Argument Functions

- Haskell only supports one-argument functions.
- An n-argument function $f(a_1, \cdots, a_n)$ is constructed in either of two ways:
  1. By making the one input argument to f a tuple holding the n arguments.
  2. By letting f "consume" one argument at a time. This is called currying.

Tuple	Currying
`add :: (Int,Int)->Int`	`add :: Int->Int->Int`
`add (a, b) = a + b`	`add a b = a + b`

Currying

- Currying is the preferred way of constructing multi-argument functions.
- The main advantage of currying is that it allows us to define specialized versions of an existing function.
- A function is specialized by supplying values for one or more (but not all) of its arguments.
- Let's look at Haskell's plus operator `(+)`. It has the type

  `(+) :: Int -> (Int -> Int).`
- If we give two arguments to `(+)` it will return an `Int`:

  `(+) 5 3` $\Rightarrow$ `8`

Currying. . .

- If we just give one argument (5) to `(+)` it will instead return a function which "adds 5 to things". The type of this specialized version of `(+)` is `Int -> Int`.
- Internally, Haskell constructs an intermediate – specialized – function:

```
add5 :: Int -> Int
add5 a = 5 + a
```

- Hence, `(+) 5 3` is evaluated in two steps. First `(+) 5` is evaluated. It returns a function which adds 5 to its argument. We apply the second argument 3 to this new function, and the result 8 is returned.

Currying. . .

- To summarize, Haskell only supports one-argument functions. Multi-argument functions are constructed by successive application of arguments, one at a time.
- Currying is named after logician Haskell B. Curry (1900-1982) who popularized it. It was invented by Schönfinkel in 1924. Schönfinkeling doesn't sound too good...
- Note: Function application (`f x`) has higher precedence (10) than any other operator. Example:

  `f 5 + 1` $\Leftrightarrow$ `(f 5) + 1`

  `f 5 6` $\Leftrightarrow$ `(f 5) 6`

Currying Example

- Let's see what happens when we evaluate `f 3 4 5`, where `f` is a 3-argument function that returns the sum of its arguments.

```
f :: Int -> (Int -> (Int -> Int))
f x y z = x + y + z

f 3 4 5 ≡ ((f 3) 4) 5
```


Currying Example...

- `(f 3)` returns a function `f' y z` (`f'` is a specialization of `f`) that adds 3 to its next two arguments.

```
f 3 4 5 ≡ ((f 3) 4) 5 ⇒ (f' 4) 5

f' :: Int -> (Int -> Int)
f' y z = 3 + y + z
```


Currying Example...

- `(f' 4)` (≡ `(f 3) 4`) returns a function `f''z` (`f''` is a specialization of `f'`) that adds (3+4) to its argument.

```
f 3 4 5 ≡ ((f 3) 4) 5 ⇒ (f' 4) 5
         ⇒ f'' 5

f'' :: Int -> Int
f'' z = 3 + 4 + z
```

- Finally, we can apply `f''` to the last argument (5) and get the result:

```
f 3 4 5 ≡ ((f 3) 4) 5 ⇒ (f' 4) 5
         ⇒ f'' 5 ⇒ 3+4+5 ⇒ 12
```


Currying Example

The Combinatorial Function:

- The combinatorial function $\binom{n}{r}$ "n choose r", computes the number of ways to pick r objects from n.

$$\binom{n}{r} = \frac{n!}{r! * (n-r)!}$$

In Haskell:

```
comb :: Int -> Int -> Int
comb n r = fact n/(fact r*fact(n-r))

? comb 5 3
  10
```

Currying Example…

`comb :: Int -> Int -> Int`
`comb n r = fact n/(fact r*fact(n-r))`
`comb 5 3` $\Rightarrow$ `(comb 5) 3` $\Rightarrow$
`comb`^{5} `3` $\Rightarrow$
`120 / (fact 3 * (fact 5-3))` $\Rightarrow$
`120 / (6 * (fact 5-3))` $\Rightarrow$
`120 / (6 * fact 2)` $\Rightarrow$
`120 / (6 * 2)` $\Rightarrow$
`120 / 12` $\Rightarrow$
`10`
`comb`^{5} `r = 120 / (fact r * fact(5-r))`

- `comb`^{5} is the result of partially applying `comb` to its first argument.

Associativity

- Function application is left-associative:
  `f a b = (f a) b` | `f a b` $\neq$ `f (a b)`
- The function space symbol `'->'` is right-associative:
  `a -> b -> c = a -> (b -> c)`
  `a -> b -> c` $\neq$ `(a -> b) -> c`
- `f` takes an `Int` as argument and returns a function of type `Int -> Int`. `g` takes a function of type `Int -> Int` as argument and returns an `Int`:
  `f' :: Int -> (Int -> Int)`
  $\Updownarrow$
  `f :: Int -> Int -> Int`
  $\not\Updownarrow$
  `g :: (Int -> Int) -> Int`

What's the Type, Mr. Wolf?

If the type of a function `f` is

t_1 -> t_2 -> $\cdots$ -> t_n -> t

and `f` is applied to arguments

e_1::t_1, e_2::t_2, $\cdots$, e_k::t_k,

and k $\leq$ n

then the result type is given by cancelling the types $t_1 \cdots t_k$:

$\cancel{t_1}$ -> $\cancel{t_2}$ -> $\cdots$ -> $\cancel{t_k}$ -> t_{k+1} -> $\cdots$ -> t_n -> t

Hence, f e_1 e_2 $\cdots$ e_k returns an object of type

t_{k+1} -> $\cdots$ -> t_n -> t.

This is called the Rule of Cancellation.

Polymorphic Functions

- In Pascal 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. . .

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

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


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`.

Example:

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

Homework

- Define an operator `$$` so that `x $$ xs` returns `True` if `x` is an element in `xs`, and `False` otherwise.

Example:

```
?  4 $$ [1,2,5,6,4,7]
   True

?  4 $$ [1,2,3,5]
   False

?  4 $$ []
   False
```