University of Auckland

415.360

Functional Programming

Christian Collberg

March 22, 1996

Function Types

Declaring Infix Functions I

- Sometimes it is more natural to use an infix notation for a function application, rather than the normal prefix one. Example:
  - `5 + 6` (infix)
  - `(+) 5 6` (prefix)
- Gofer 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. Example: `*` has higher precedence than `+`, and hence:

  `3 + 5*4` $\equiv$ `3 + (5*4)`

  `3 + 5*4` $\not\equiv$ `(3 + 5) * 4`

Slide 6–1

Declaring Infix Functions II

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

  `5-3+9` $\equiv$ `(5-3)+9 = 11`

  `OR`

  `5-3+9` $\equiv$ `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` $\equiv$ `5^(3^2)`
- Some operators have **free** (or **no**) associativity. Combining operators with free associativity is an error:

  `5 == 4 < 3` $\Rightarrow$ $\boxed{\texttt{ERROR}}$

Slide 6–2

Declaring Infix Functions III

- The syntax for declaring operators:

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

From the standard prelude:

```
infixr 8 ^
infixl 7 *
infix 7  /, `div`, `rem`, `mod`
infixl 6 +, -
infix 4  ==, /=, <, <=, >=, >
```

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

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

Slide 6–3

Multi-Argument Functions

- Gofer 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:

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

Currying:

```
add ::  Int -> Int -> Int
add a b = a + b
```

Slide 6–4

Currying I

- 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 Gofer's plus operator `(+)`. It has the type `(+) ::  Int -> (Int -> Int)`.
- If we give two arguments to `(+)` it will return an `Int`:

  `(+) 5 3` ⇒ `8`

Slide 6–5

Currying II

- 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, Gofer 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.

Slide 6–6

Currying III

- To summarize, Gofer 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` ⇔ `(f 5) + 1`
  `f 5 6` ⇔ `(f 5) 6`

Slide 6–7

Currying Example I (a)

- 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`

- $\boxed{\texttt{(f 3)}}$ 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
```

Slide 6–8

Currying Example I (b)

- $\boxed{\texttt{(f' 4)}}$ (≡ `(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`

Slide 6–9

Currying Example II (a)

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 Gofer:

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

?  comb 5 3
   10
```

Slide 6–10

Currying Example II (b)

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

comb 5 3 =
   (comb 5) 3 ⇒
   comb^5 3 ⇒
   120 / (fact 3 * (fact 5-3)) ⇒
   120 / (6 * (fact 5-3)) ⇒
   120 / (6 * fact 2) ⇒
   120 / (6 * 2) ⇒
   120 / 12 ⇒
   10

comb^5 r = 120 / (fact r * fact(5-r))

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

Slide 6–11

Associativity

- Function application is $\boxed{\text{left}}$-associative:

  `f a b = (f a) b` | `f a b` $\neq$ `f (a b)`

- The function space symbol '`->`' is $\boxed{\text{right}}$-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 $\boxed{\texttt{Int -> Int}}$. `g` takes a function of type $\boxed{\texttt{Int -> Int}}$ as argument and returns an `Int`:

  `f :: Int -> Int -> Int`

  $\Updownarrow$

  `f' :: Int -> (Int -> Int)`

  $\not\Updownarrow$

  `g :: (Int -> Int) -> Int`

Slide 6–12

What's the Type, Mr. Wolf?

- If the type of a function `f` is

  $\texttt{t}_1$ `->` $\texttt{t}_2$ `->` $\cdots$ `->` $\texttt{t}_n$ `->` `t`

- and `f` is applied to arguments

  $\texttt{e}_1\texttt{::t}_1$, $\texttt{e}_2\texttt{::t}_2$, $\cdots$, $\texttt{e}_k\texttt{::t}_k$,

- and $\texttt{k} \leq \texttt{n}$

- then the result type is given by cancelling the types $\texttt{t}_1 \cdots \texttt{t}_k$:

  $\cancel{\texttt{t}}_1$ `->` $\cancel{\texttt{t}}_2$ `->` $\cdots$ `->` $\cancel{\texttt{t}}_k$ `->`
  $\texttt{t}_{k+1}$ `->` $\cdots$ `->` $\texttt{t}_n$ `->` `t`

- Hence, $\boxed{\texttt{f e}_1\ \texttt{e}_2 \cdots \texttt{e}_k}$ returns an object of type $\boxed{\texttt{t}_{k+1}\ \texttt{->} \cdots \texttt{->}\ \texttt{t}_n\ \texttt{->}\ \texttt{t}}$.

- This is called the **Rule of Cancellation**.

Slide 6–13

Polymorphic Functions I

- 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 Gofer (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 = ...
```

Slide 6–14

Polymorphic Functions II

- `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]` $\Rightarrow$ `3 (list of Int)`
  `length ["Hi ", "there", "!"]` $\Rightarrow$ `3 (list of String)`
  `length "Hi!"` $\Rightarrow$ `3 (list of Char)`

Slide 6–15

Polymorphic Functions III

- 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]
  ```

Slide 6–16

Polymorphic Functions IV

- 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 Gofer's strong typing that we can only create lists of the same type of element. If we tried to do

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

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

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

Slide 6–17

Summary I

- B&W reading list:
  1. Currying: 12–13
  2. Polymorphism: 10
  3. Operator declarations: 38–39
- 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.

Slide 6–18

Summary II

- 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: `(+)`.

Slide 6–19

Homework I

- Define a polymorphic function $\boxed{\texttt{dup x}}$ 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))
```

Slide 6–20

Homework II

- Define a polymorphic function $\boxed{\texttt{copy n x}}$ 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)]
```

Slide 6–21

Homework III

- 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
```

Slide 6–22

Homework IV

- 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
```

Slide 6–23