Functional Programming: Functions and Types

Armando Solar Lezama
Computer Science and Artificial Intelligence Laboratory
M.I.T.

Adapted from Arvind 2010

September 8, 2011
Function Execution by Substitution

\[
\text{plus } x \ y = x + y
\]

1. \(\text{plus } 2 \ 3 \rightarrow 2 + 3 \rightarrow 5\)

2. \(\text{plus } (2*3) \ (\text{plus } 4 \ 5)\)
   \[
   \rightarrow \text{plus } 6 \ (4+5) \rightarrow (2*3) + (\text{plus } 4 \ 5) \rightarrow 6 + (4+5) \rightarrow 6 + 9 \rightarrow 15
   \]
   \[
   \rightarrow \text{plus } 6 \ 9 \rightarrow 6 + 9 \rightarrow 15
   \]

The final answer did not depend upon the order in which reductions were performed.
Confluence

Informally - The order in which reductions are performed in a Functional program does not affect the final outcome.

This is true for all functional programs regardless whether they are right or wrong.

A formal definition will be given later.
Blocks

$let$
\[x = a \times a\]
\[y = b \times b\]
$in$
\[(x - y)/(x + y)\]

- A variable can have at most one definition in a block.
- Ordering of bindings does not matter.
Layout Convention in Haskell

This convention allows us to omit many delimiters

```
let
  x = a * a
  y = b * b
in
  (x - y) / (x + y)
```

is the same as

```
let
  { x = a * a ;
     y = b * b ;}
in
  (x - y) / (x + y)
```
Lexical Scoping

```
let
  y = 2 * 2
  x = 3 + 4
  z = let
    x = 5 * 5
    w = x + y * x
  in
  w
in
  x + y + z
```

Lexically closest definition of a variable prevails.
Renaming Bound Identifiers
(α-renaming)

\[
\begin{align*}
\text{let} & \\
\quad & y = 2 * 2 \\
\quad & x = 3 + 4 \\
\quad & z = \text{let} x = 5 * 5 \\
\quad & \quad w = x + y \cdot x \\
\quad & \quad \quad \text{in} w \\
\quad & \quad \quad \text{in} x + y + z
\end{align*}
\]

\[
\begin{align*}
\equiv & \\
\quad & \text{let} x' = 5 * 5 \\
\quad & \quad w = x' + y \cdot x' \\
\quad & \quad \quad \text{in} w \\
\quad & \quad \quad \text{in} x + y + z
\end{align*}
\]
Lexical Scoping and \( \alpha \)-renaming

\[
\text{\texttt{plus}} \quad x \; y = x + y
\]

\[
\text{\texttt{plus'}} \quad a \; b = a + b
\]

\texttt{plus} and \texttt{plus'} are the same because \texttt{plus'} can be obtained by systematic renaming of bounded identifiers of \texttt{plus}
Capture of Free Variables

\[
\begin{align*}
  f \ x &= \ldots \\
  g \ x &= \ldots \\
  \text{foo} \ f \ x &= f \ (g \ x)
\end{align*}
\]

Suppose we rename the bound identifier \( f \) to \( g \) in the definition of \( \text{foo} \)

\[
\text{foo}' \ g \ x = g \ (g \ x)
\]

\[
\text{foo} \equiv \text{foo}' \quad ? \quad \text{No}
\]

While renaming, entirely new names should be introduced!
Curried functions

\[
\text{plus } x \ y = x + y
\]

\[
\text{let} \quad f = \text{plus } 1
\]

\[
in \quad f \ 3
\]

\[
\rightarrow (\text{plus } 1) \ 3 \rightarrow 1 + 3 \rightarrow 4
\]

syntactic conventions:
\[
e_1 \ e_2 \ e_3 \equiv ((e_1 \ e_2) \ e_3)
\]
\[
x + y \equiv (+) \ x \ y
\]
Local Function Definitions

\[
\text{integrate } dx \ a \ b \ f = \\
\text{let} \\
\quad \text{sum } x \ tot = \\
\quad \quad \text{if } x > b \ \text{then tot} \\
\quad \quad \text{else sum } (x+dx) \ (tot+(f \ x)) \\
\text{in} \\
\quad (\text{sum } (a+dx/2) \ 0) \ * \ dx
\]
Local Function Definitions

integrate \( dx \ a \ b \ f \) =

let

\[
\text{sum } x \ xot = \\
\quad \text{if } x > b \ \text{then } xot \\
\quad \text{else sum } (x+dx) \ (xot+((f x)))
\]

\[
\text{in}
\]

\[
\text{(sum } (a+dx/2) \ 0) \ * \ dx
\]

integrate \( dx \ a \ b \ f \) =

\[
\quad \text{sum } dx \ b \ f \ (a+dx/2) \ 0) \ * \ dx
\]

sum \( dx \ b \ f \ x \ xot =

\[
\quad \text{if } x > b \ \text{then } xot \\
\quad \text{else sum } dx \ b \ f \ (x+dx) \ (xot+(f x))
\]

Any function definition can be “closed” and “lifted”
All expressions in Haskell have a type

\[ 23 :: \text{Int} \]

"23 belongs to the set of integers"
"The type of 23 is Int"

\[ \text{true :: Bool} \]
"hello" :: String
Type of an expression

(sq 529) :: Int
sq :: Int -> Int

"sq is a function, which when applied to an integer produces an integer"

"Int -> Int is the set of functions, each of which when applied to an integer produces an integer"

"The type of sq is Int -> Int"
Type of a Curried Function

\[ \text{plus } x \ y = x + y \]

\[(\text{plus } 1) \ 3 :: \text{Int} \]

\[(\text{plus } 1) :: \text{Int} \rightarrow \text{Int} \]

\[\text{plus} :: \text{Int} \rightarrow (\text{Int} \rightarrow \text{Int}) \]
\( \lambda \)-Abstraction

Lambda notation makes it explicit that a value can be a function. Thus,

\[
\text{(plus 1)} \quad \text{can be written as} \quad \lambda y \rightarrow (1 + y)
\]

(In Haskell \(\lambda x\) is a syntactic approximation of \(\lambda x\))

\[
\text{plus } x \quad y = \quad x + y
\]

can be written as

\[
\text{plus } = \quad \lambda x \rightarrow \lambda y \rightarrow (x + y)
\]

or as

\[
\text{plus } = \quad \lambda x \quad y \rightarrow (x + y)
\]
Parentheses Convention

\[ f \, e_1 \, e_2 \equiv ((f \, e_1) \, e_2) \]
\[ f \, e_1 \, e_2 \, e_3 \equiv (((f \, e_1) \, e_2) \, e_3) \]

application is \textit{left associative}

\[
\text{Int} \to (\text{Int} \to \text{Int}) \equiv \text{Int} \to \text{Int} \to \text{Int}
\]

type constructor \textit{“\to”} is \textit{right associative}
Type of a Block

\[
\text{(let}
\quad x_1 = e_1
\quad \ldots
\quad x_n = e_n
\quad \text{in}
\quad e) :: t
\]

provided

\[
e :: t
\]
Type of a Conditional

\((\text{if } e \text{ then } e_1 \text{ else } e_2 ) :: t\)

provided

\[ e :: \text{Bool} \]
\[ e_1 :: t \]
\[ e_2 :: t \]

The type of expressions in both branches of conditional must be the same.
Polymorphism

\[ \text{twice } f \ x = f \ (f \ x) \]

1. \( \text{twice } (\text{plus } 3) \ 4 \)
   \[ \rightarrow (\text{Plus } 3) \ ((\text{plus } 3) \ 4) \]
   \[ \rightarrow ((\text{plus } 3) \ 7) \]
   \[ ightarrow 10 \]

   \( \text{twice } :: (\text{Int } \rightarrow \text{Int}) \rightarrow \text{Int } \rightarrow \text{Int} \)

2. \( \text{twice } (\text{append } "Zha") \ "Gabor" \)
   \[ \rightarrow "ZhaZhaGabor" \]

   \( \text{twice } :: (\text{Str } \rightarrow \text{Str}) \rightarrow \text{Str } \rightarrow \text{Str} \)
Deducing Types

1. Assign types to every subexpression
   \[ x :: t_0 \quad f :: t_1 \]
   \[ f \, x :: t_2 \quad f \,(f \, x) :: t_3 \]
   \[ \Rightarrow \quad \text{twice} :: t_1 \to t_0 \to t_3 \]

2. Set up the constraints
   \[ t_1 = t_0 \to t_2 \quad \text{because of } (f \, x) \]
   \[ t_1 = t_2 \to t_3 \quad \text{because of } f \,(f \, x) \]

3. Resolve the constraints
   \[ t_0 \to t_2 = t_2 \to t_3 \]
   \[ \Rightarrow \quad t_0 = t_2 \quad t_2 = t_3 \Rightarrow t_0 = t_2 = t_3 \]
   \[ \Rightarrow \quad \text{twice} :: (t_0 \to t_0) \to t_0 \to t_0 \]
Another Example: *Compose*

```
compose f g x = f (g x)
```

What is the type of `compose`?

1. Assign types to every subexpression
   ```
x :: t0   f :: t1   g :: t2
   g x :: t3   f (g x) :: t4
   ⇒ compose :: t1 -> t2 -> t0 -> t4
   ```

2. Set up the constraints
   ```
t1 = t3 -> t4  because of f (g x)
t2 = t0 -> t3  because of (g x)
   ```

3. Resolve the constraints
   ```
   ⇒ compose ::
       (t3 -> t4) -> (t0 -> t3) -> t0 -> t4
   ```
Now for some fun

\[
\text{twice } f \ x = f \ (f \ x)
\]

\[
a = \text{twice}_1 \ (\text{twice}_2 \ \text{succ} \ 4)
\]

\[
b = \text{twice}_3 \ \text{twice}_4 \ \text{succ} \ 4
\]

1. Is \(a=b\) ?
   - yes

2. Are the types of all the twice instances the same?
   - no

\[
\text{twice}_1 :: (I \to I) \to I \to I
\]

\[
\text{twice}_2 :: (I \to I) \to I \to I
\]

\[
\text{twice}_3 :: ((I \to I) \to I \to I) \to (I \to I) \to I \to I
\]

\[
\text{twice}_4 :: (I \to I) \to I \to I
\]

The first person with the right types gets a prize!
Hindley-Milner Type System

Haskell and most modern functional languages follow the Hindley-Milner type system.

The main source of polymorphism in this system is the *Let block*.

The type of a variable can be instantiated differently within its lexical scope.

*much more on this later …*