Function Execution by Substitution

plus x y = x + y

1. plus 2 3 → 2 + 3 → 5
2. plus (2*3) (plus 4 5)
   → plus 6 (4+5)
   → plus 6 9
   → 6 + 9
   → 15
   → (2*3) + (plus 4 5)
   → 6 + (4+5)
   → 6 + 9
   → 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

```plaintext
let
  x = a * a
  y = b * 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

\[
\text{let} \quad \begin{align*}
    x &= a \times a \\
    y &= b \times b \\
\end{align*} \quad \text{in} \quad (x - y)/(x + y)
\]

is the same as

\[
\text{let} \quad \begin{align*}
    \{ x &= a \times a ; \\
    y &= b \times b ; \} \\
\end{align*} \quad \text{in} \quad (x - y)/(x + y)
\]

Lexical Scoping

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

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

Lexical Scoping and \(\alpha\)-renaming

\[
\begin{align*}
\text{plus} \ x \ y &= x + y \\
\text{plus}' \ a \ b &= a + b
\end{align*}
\]

\text{plus} and \text{plus}' are the same because \text{plus}' can be obtained by systematic renaming of bounded identifiers of \text{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 \( \text{foo} \)

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

\[
\begin{align*}
  \text{foo} &\equiv \text{foo}’
\end{align*}
\]

Curried functions

\[
\begin{align*}
  \text{plus } x \ y &= x + y \\
  \text{let } f &= \text{plus } 1 \\
  \text{in } f 3
\end{align*}
\]

syntactic conventions:

\[
\begin{align*}
  e_1 \ e_2 \ e_3 &\equiv ((e_1 \ e_2) \ e_3) \\
  x + y &\equiv (+) \ x \ y
\end{align*}
\]
Local Function Definitions

```haskell
integrate dx a b f =
    let
        sum x tot =
            if x > b then tot
            else sum (x+dx) (tot+(f x))
        in
            (sum (a+dx/2) 0) * dx
```

Free variables of `sum`?

```haskell
integrate dx a b f =
    (sum dx b f (a+dx/2) 0) * dx

sum dx b f x tot =
    if x > b then tot
    else sum dx b f (x+dx) (tot+(f x))
```

Any function definition can be "closed" and "lifted"

Types

_All expressions in Haskell have a type_

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

```
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

plus x y = x + y

(plus 1) 3:: Int
(plus 1) :: Int -> Int

plus :: ?
\textit{\textlambda \text{-Abstraction}}

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

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

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

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

can be written as

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

or as

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

\textit{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}

\[\text{Int} \rightarrow (\text{Int} \rightarrow \text{Int}) = \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}\]

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

\[(\text{let} \quad 
\begin{align*}
  x_1 &= e_1 \\
  \quad &\vdots \\
  x_n &= e_n \\
\text{in} \quad 
  e \quad) \quad :: 	
\) \quad 

provided 
\(e :: t\)

Type of a Conditional

\[(\text{if} \quad e \quad \text{then} \quad e_1 \quad \text{else} \quad e_2 \quad) \quad :: 	
\]

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 (plus 3) 4}
   \rightarrow (\text{Plus 3}) \ ((\text{plus 3}) \ 4)
   \rightarrow ((\text{plus 3}) \ 7)
   \rightarrow 10
   \text{twice :: ?}

2. \text{twice (append "Zha") "Gabor"}

   \text{twice :: ?}

Deducing Types

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

What is the most "general type" for twice?

1. Assign types to every subexpression
   \[ x :: t0 \quad f :: t1 \quad f \ x :: t2 \quad f \ (f \ x) :: t3 \]
   \[ \Rightarrow \text{twice :: ?} \]

2. Set up the constraints
   \[ t1 = \quad \text{because of } (f \ x) \]
   \[ t1 = \quad \text{because of } f \ (f \ x) \]

3. Resolve the constraints
Another Example: *Compose*

\[
\text{compose } f \ g \ x = f \ (g \ x)
\]

What is the type of \text{compose}?

1. Assign types to every subexpression
   \[
   x :: t_0 \quad f :: t_1 \quad g :: t_2
   \]
   \[
   g \ x :: t_3 \quad f \ (g \ x) :: t_4
   \]
   \[\Rightarrow \text{compose} ::\]

2. Set up the constraints
   \[
   t_1 = \text{because of } f \ (g \ x)
   \]
   \[
   t_2 = \text{because of } (g \ x)
   \]

3. Resolve the constraints
   \[\Rightarrow \text{compose} ::\]

Now for some fun

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

\[
\begin{align*}
a &= \text{twice}_1 \ (\text{twice}_2 \ \text{succ}) \ 4 \\
b &= \text{twice}_3 \ \text{twice}_4 \ \text{succ} \ 4 \\
\end{align*}
\]

1. Is \(a=b\)?

2. Are the types of all the twice instances the same?

*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 ...*