Introduction to Functional Programming in Haskell & the Hindley-Milner Type System

Kung Chen

National Chengchi University, Taiwan

2008 Formosan Summer School of Logic, Language and Computation

Agenda

FLOLAC 2008

- Unit I: FP in Haskell
  - Basic Concepts of FP
  - Haskell Basics
  - Higher-Order Functions
  - Defining New Types
  - Lazy Evaluation
- Unit 2: Intro. to Type Systems for FP
  - The Lambda Calculus
  - Typed Lambda Calculi
  - The Hindley-Milner Type System

Unit I: FP in Haskell

Basic Concepts of

Functional Programming

What is Functional Programming?

FLOLAC 2008

Generally speaking:

- Functional programming is a style of programming in which the *<u>primary method of computation is the application of functions to arguments</u>*

•Define a function square: $\boxed{\underbrace{\text{square}}_{\text{Function name}}\ \underbrace{x}_{\text{Formal parameter}} = \underbrace{x * x}_{\text{Function body: an expression}}}$

What is Functional Programming?

Generally speaking:

- Functional programming is a style of programming in which the *primary method of computation is the application of functions to arguments*

square x = x * x

No parentheses: `square(5)`

Substitute the *argument 5* into the body of the function

Function application:

square 5
  = { applying *square* }
   *5 * 5*
  = { applying * }
   25

Functions and Arguments

- Similarly an argument may itself be a function application:

square (*square* 3)

  = { apply inner *square* }

square (3 * 3)

  = { apply * }

square (9)

  = { apply outer *square* }

9 * 9

  = { apply * }

81

Programming Paradigms

- *FP* is a *programming paradigm …*
- A programming paradigm
  – is a way to think about programs, programming, and problem solving,
  – is supported by one or more programming languages.
- Various Programming Paradigms:
  – Imperative (Procedural)
  – Functional
  – Object-Oriented
  – Logic
  – Hybrid

Imperative vs. Functional

- Imperative languages specify the steps of a program in terms of *assigning values to variables.*

```
int sum (int n, int list[]) {
   int total = 0;
   for (int i = 0; i < n; ++i)
      total += list[i];
   return s;
}
```

Variable assignments

```
sum []       = 0
sum (x:xs) = x + sum xs
```

[]-empty list; “:”-cons a list

Equations

There is no loop!
Recursive, please!

Imperative vs. Functional

FLOLAC 2008

In C, the sequence of actions is	*Applying functions:*

i = 1
total = 1
i = 2
total = 3
i = 3
total = 6
i = 4
total = 10
i = 5
total = 15
	sum [1,2,3,4,5]
	= { apply *sum* }
	1 + *sum* [2,3,4,5]
	= { apply *sum* }
	1 + (2 + *sum* [3,4,5])
	= { apply *sum* }
	1 + (2 + (3 + sum [4,5])
	= { apply *sum* }
	…
	= { apply + }
	15

Functional Programming

- Functional programs work exclusively with values, and expressions and functions which compute values.
- A *value* is a piece of data.
  - `2, 4, 3.14159, ”John”, (0,0), [1,3,5],...`
- An *expression* computes a value.
  - `2+5*pi, length(l)-size(r)`
- Expressions combine values using *functions* and *operators*.

Why FP?
What's so Good about FP?

- To get experience of a different type of programming
- It has a solid mathematical basis
  - Referential Transparency and Equation Reasoning
  - Executable Specification
  - …
- It's fun!

Referential Transparency

Can we replace `f(x) + f(x)` with `2*f(x)`?

Yes, we can!

- **If the function f is *referential transparent.***
  - In particular, a function is referential transparency if *its result depends only on the values of its parameters.*
  - This concept occurs naturally in mathematics, but is broken by imperative programming languages.

Referential Transparency…

- Imperative programs are *not RT due to side effects*.
- Consider the following C/Java function `f`:

```
int y = 10;
int f(int i) {
   return i + y++;
}
```

then `f(5)+f(5)= 15+16 = 31`

but `2*f(5)= 2*15 = 30!`

Referential Transparency…

- In a purely functional language, *variables* are similar to variables in *mathematics*: they hold a value, but they can’t be updated.
- Thus all functions are RT, and therefore *always yield the same result* no matter how often they are called.

Equational Reasoning

FLOLAC 2008

- RT implies that *“equals can be replaced by equals”*
- *Evaluate* an expression by substitution . I.e. we can replace a function application by the function definition itself.

```
double x   = 2 * x
even x     = x mod 2 == 0
```

```
   even (double 5)
⇒ even (2 * 5)
⇒ even 10
⇒ 10 mod 2 == 0
⇒ 0 == 0
⇒ True
```

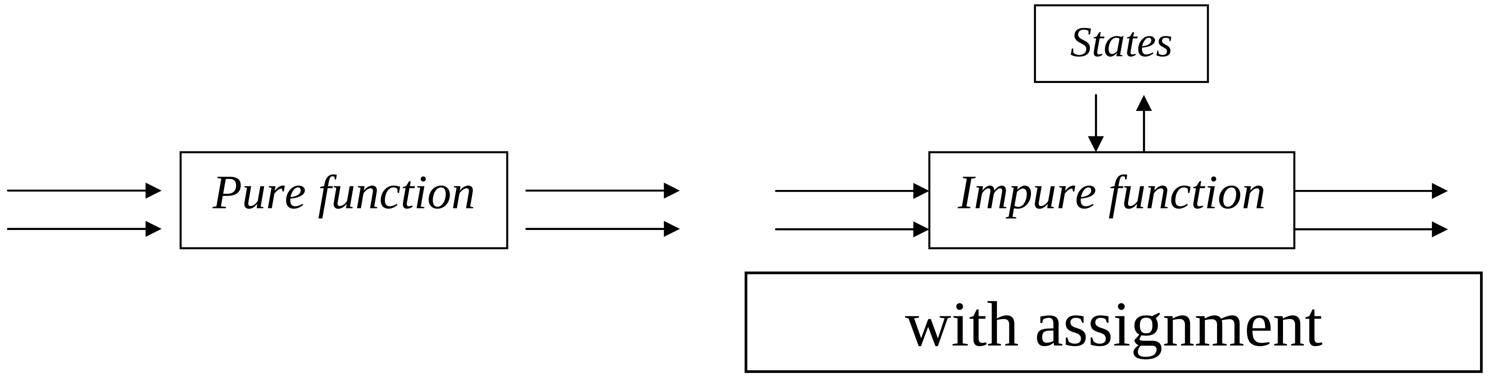
[5/x]: x換成5

even’s definition, [10/x]

Computation in FP

FLOLAC 2008

- Achieved via ***function application***
- Functions are mathematical functions ***without side-effects***.
  - *Output is solely dependent of input.*



Can replace `f(x) + f(x)` with `2*f(x)`

What's so Good about FP?

- Referential Transparency and Equation Reasoning
- **Executable Specification**
- …

Quick Sort in C

FLOLAC 2008

```
qsort( a, lo, hi ) int a[ ], hi, lo;
{   int h, l, p, t;
    if (lo < hi)
    {   l = lo;   h = hi;   p = a[hi];
        do
        {  while ((l < h) && (a[l]  <= p))   l =  l  + 1;
           while ((h > l) && (a[h] >= p))   h = h – 1 ;
           if (l < h) [ t = a[l];  a[l] = a[h];  a[h] = t; }
        } while (l < h);
        t = a[l];   a[l] = a[hi];  a[hi] = t;
        qsort( a, lo, l-1 );   qsort( a, l+1, hi );
    }
}
```

Quick Sort in Haskell

- *Quick sort*: the program is the specification!

```
qsort []     = []
qsort (x:xs) = qsort lt ++ [x] ++ qsort greq
       where lt   = [y | y <- xs, y < x]
             greq = [y | y <- xs, y >= x]
```

```
List operations:
  [] the empty list
  x:xs adds an element x to the head of a list xs
  xs ++ ys concatenates lists xs and ys
  [x,y,z] abbreviation of x:(y:(z:[]))
```

Historical View: Pioneers in FP

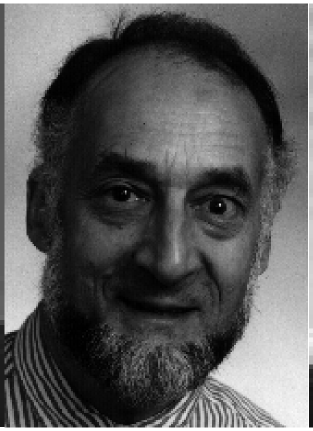
FLOLAC 2008

McCarthy:Lisp

Landin:ISWIM

Steele:Scheme

Milner:ML



Backus:FP

Church: Lambda Calculus

Curry: Combinatory Logic

Background of Haskell

What is Haskell?

FLOLAC 2008

- Haskell is a *purely* functional language created in 1987 by scholars from Europe and US.
- Haskell was the first name of H. Curry, a logician
- Standardized language version: **Haskell 98**
- Several compilers and interpreters available
  - Hugs, Gofer, , GHCi, Helium
  - GHC (Glasgow Haskell Compiler)
- Comprehensive web site: `http://haskell.org/`

Haskell Curry (1900-1982)

Haskell vs. Miranda

1970s - 1980s:

David Turner developed a number of *lazy* functional languages, culminating in the Miranda system.

If Turner had agreed, there will be no Haskell?!

Features of Haskell

- **pure** (referentially transparent) — no side-effects
- **non-strict** (lazy) — arguments are evaluated only when needed
- **statically strongly typed** — all type errors caught at compile-time
- **type classes** — safe overloading
- …

Why Haskell?

FLOLAC 2008

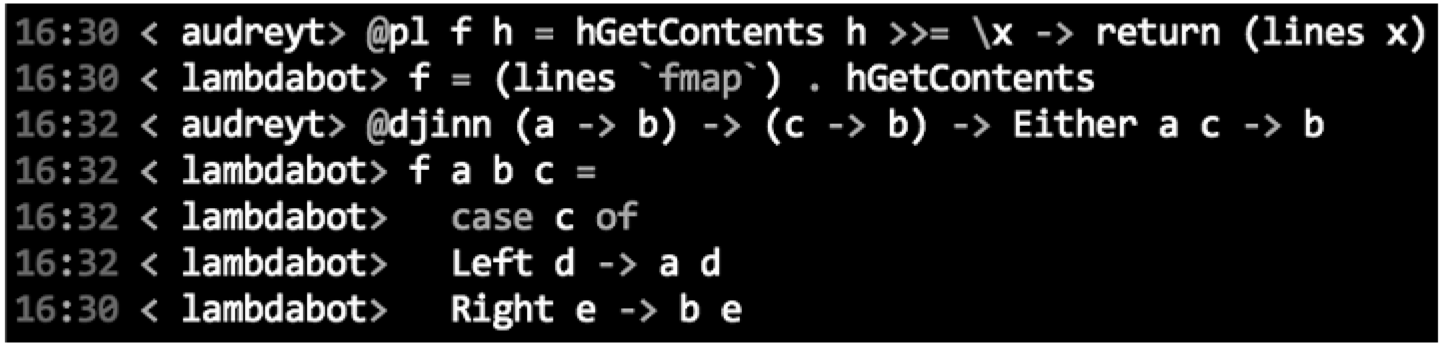
- A language that doesn't affect the way you *think about programming*, is *not worth* knowing.

--Anan Perlis

The recipient of the first ACM Turing Award

Any software written in Haskell?

- Pugs
  - Implementation of Perl 6
- darcs
  - Distributed, interactive, smart RCS
- lambdabot
- GHC



```
16:30 < audreyt> @pl f h = hGetContents h >>= \x -> return (lines x)
16:30 < lambdabot> f = (lines `fmap`) . hGetContents
16:32 < audreyt> @djinn (a -> b) -> (c -> b) -> Either a c -> b
16:32 < lambdabot> f a b c =
16:32 < lambdabot>   case c of
16:32 < lambdabot>   Left d -> a d
16:30 < lambdabot>   Right e -> b e
```

A chat between developers of the Pugs project

From freenode, #perl6, 2005/3/2
http://xrl.us/e98m

```
19:08 < malaire> Does pugs yet have system() or backticks or qx// or any way to use
                 system commands?
19:08 < autrijus> malaire: no, but I can do one for you now. a sec
19:09 < malaire> ok, I'm still reading YAHT, so I won't try to patch pugs just yet...
19:09 < autrijus> you want unary system or list system?
19:09 < autrijus> system("ls -l") vs system("ls", "-l")
19:10 < malaire> perhaps list, but either is ok
19:11 < autrijus> \\n   Bool     pre     system  (Str)\
19:11 < autrijus> \\n   Bool     pre     system  (Str: List)\
19:11 < autrijus> I'll do both :)
19:11 < autrijus> done. testing.
19:14 < autrijus> test passed. r386. enjoy
19:14 < malaire> that's quite fast development :)
19:14 < autrijus> :)
```

Haskell vs. Scheme/ML

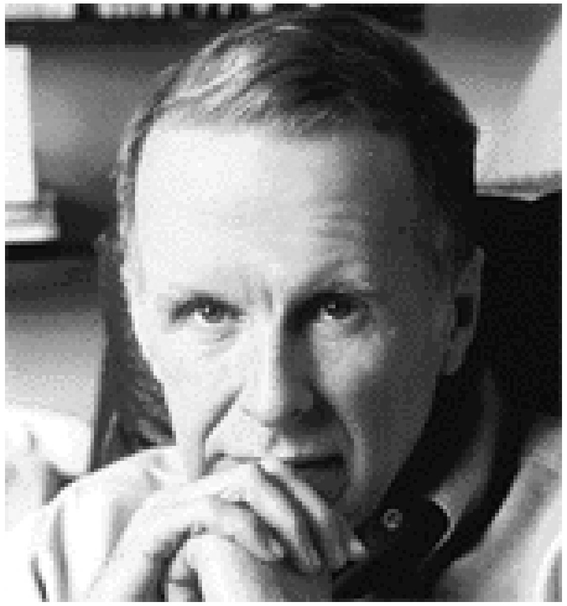
- Haskell, like Lisp/Scheme, ML (Ocaml, Standard ML) and F#, is based on Church's lambda (λ) calculus
- Unlike those languages, Haskell is *pure* (no updatable state)
- Haskell uses "monads" to handle stateful effects
  - cleanly separated from the rest of the language
- Haskell "enforces a *separation* between Church and State"

"FP" is another less-known FPL

FLOLAC 2008

Can Programming Be Liberated from the von Neumann Style?

1977 Turing Award Lecture

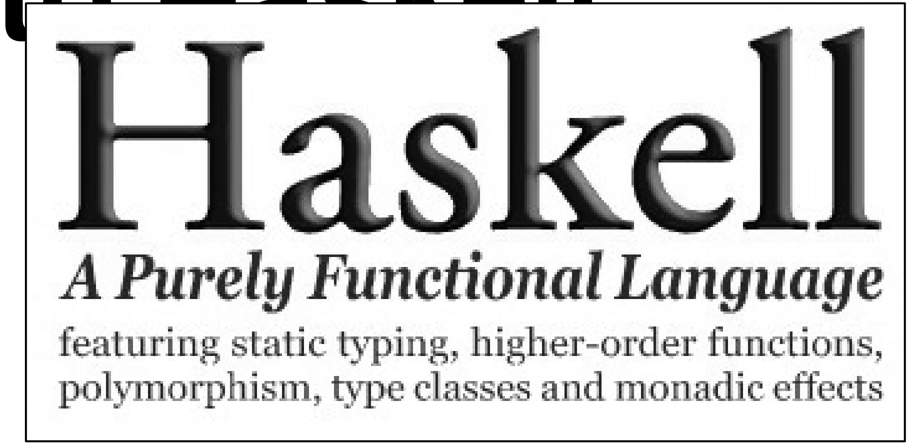

1924-2007

Late 1970s:

John Backus develops FP, a now-called combinator-based FPL.

29

Back to Haskell


Haskell
A Purely Functional Language
featuring static typing, higher-order functions, polymorphism, type classes and monadic effects

The Basics

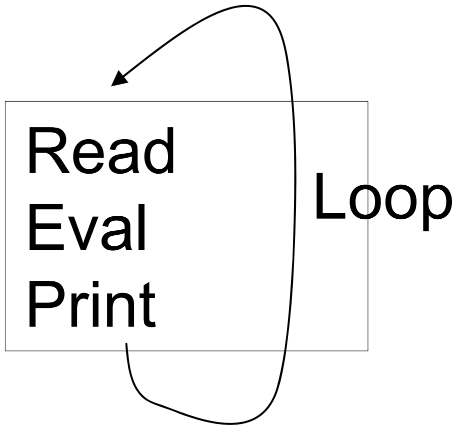
Running Haskell Programs

FLOLAC 2008

- Pick a Haskell Implementation
- We’ll use *Hugs or GHCi*
- Interpreter mode (Hugs):

```
> 5+2*3
11

> (5+2)*3
21

> sqrt (3^2 + 4^2)
5.0
```

ugs > prompt means
he Hugs system is ready
<u>luate an expression.</u>



Hugs: a Haskell Interpreter

`http://www.haskell.org/hugs`

```
__   __ __  __  ____   ___      _________________________________________
||   || ||  || ||  || ||__      Hugs 98: Based on the Haskell 98 standard
||___|| ||__|| ||__||  __||     Copyright (c) 1994-2003
||---||         ___||           World Wide Web: http://haskell.org/hugs
||   ||                         Report bugs to: hugs-bugs@haskell.org
||   || Version: Nov 2003       _________________________________________

Hugs mode: Restart with command line option +98 for Haskell 98 mode


Type :? for help
Prelude>
```

winHugs: a Windows GUI

Hugs

- The Hugs interpreter does two things:
- Evaluate expressions
- Evaluate commands, e.g.
  - **:quit** quit
  - **:load <file>** load a file
  - **:r** redo the last load command
  - **:?** help
  - …

Preparing Haskell Programs

FLOLAC 2008

- Create and Edit a file with a Haskell program
  - File name extension: .hs or .lhs
  - Literate Haskell Programs
    - Description and Comments about the program
    - >Haskell
    - >code
- Load the source program in to Hugs
  - Enter the expression to evaluate
  - Read-Evaluate-Print loop

Running Haskell with GHC

- By Haskell Group at Glasgow University, UK
- Get GHC from http://haskell.org/ghc/
- GHC is a compiler; GHCi is the interpreter version
- $ ghc Main.hs
  - → Main.hi
  - → Main.c
  - → a.out or Main.exe
- $ ghci Main.hs
  Prelude Main> *QuickSort [9, 4, 1, 2, 6]*
  [1,2,4,6,9]

The Standard Prelude

The library file Prelude.hs provides a large number of standard functions. In addition to the familiar numeric functions such as + and *, the library also provides many useful functions on lists.

- Calculating the length of a list:

```
> length [1,2,3,4]
4
```

The Standard Prelude ...

- Appending the elements of two lists:

```
> [1,2,3] ++ [4,5,6]
[1,2,3,4,5,6]
```

- Selecting the first element of a list:

```
> head [1,2,3,4]
1
```

- Removing the first element of a list:

```
> tail [1,2,3,4]
[2,3,4]
```

Function Application

In mathematics, function application is denoted using *parentheses*, and multiplication is often denoted using juxtaposition or space.

```
f(a,b) + c d
```

In Haskell, function application is denoted using *space*, and multiplication is denoted using `*`.

$$\boxed{\underbrace{\texttt{f a b}}_{} \texttt{ + c*d}}$$

Function Application …

FLOLAC 2008

- Function application ("calling a function with a particular argument") has higher priority than any other operator.
- In math (and Java) we use parentheses to include arguments; in Haskell *no parentheses* are needed.

```
f a + b
```

means

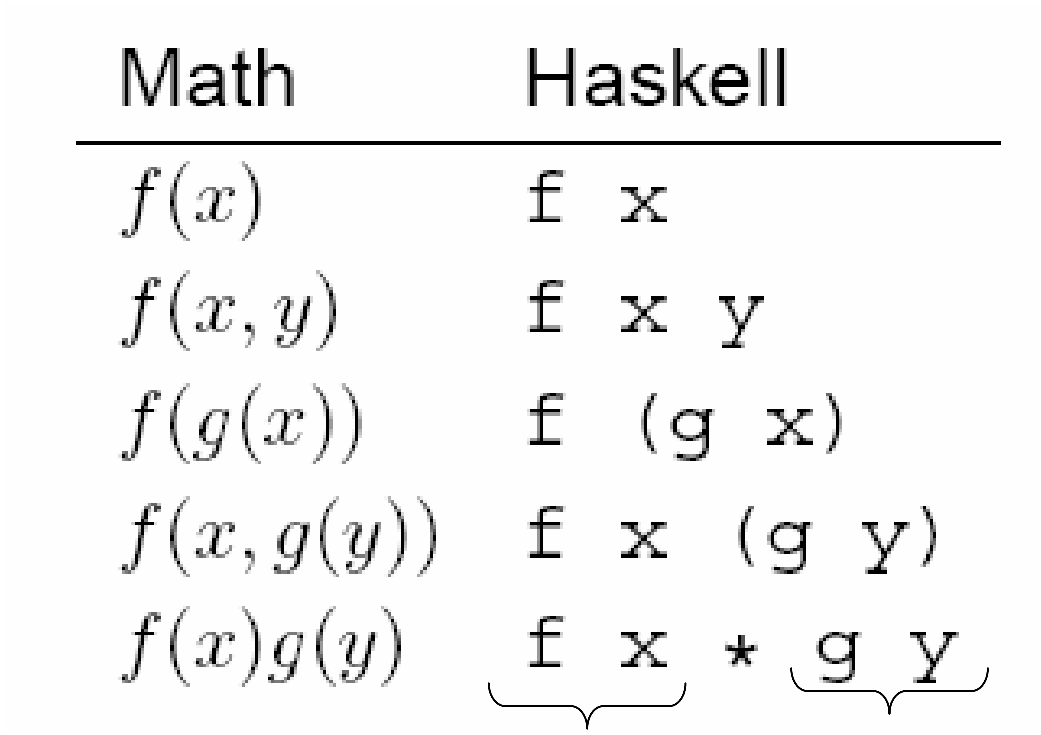
`(f a) + b` not $\boxed{\texttt{f (a+b)}}$

- since function application binds harder than plus.

Summary: Function Application …

FLOLAC 2008

•Here’s a comparison between mathematical notations and Haskell:



Math	Haskell
$f(x)$	`f x`
$f(x, y)$	`f x y`
$f(g(x))$	`f (g x)`
$f(x, g(y))$	`f x (g y)`
$f(x)g(y)$	$\underbrace{\texttt{f x}} * \underbrace{\texttt{g y}}$

Programs as Sets of Definitions

FLOLAC 2008

- A very simple functional program (also known as a *functional script*) in Haskell
  - A set of definitions

Type Signature

```
square     :: Integer -> Integer
square x  =  x * x
```

Definition (i.e. equation)

```
smaller        :: (Integer, Integer) -> Integer
smaller (x,y) =  if x <= y then x else y


main = print (square(smaller(5, 3+4)))
```

Main expr to eval

Definitions

- A Haskell program is a sequence of definitions followed by an expression to evaluate.
- A *definition* gives a name to a value.
- Haskell definitions are of the form:

```
name :: type
name = expression
```

- Examples:

```
size :: Int
size = (12+13)*4
```

Function Definitions

FLOLAC 2008

- A *function definition* specifies how the result is computed from the arguments.

Function types specify the types of the *arguments* and the *result*.

```
average :: Float->Float->Float
```

$$\text{average}\ \underbrace{x\quad y}_{\text{parameters}} = \underbrace{(x+y)/2}_{\text{body}}$$

parameters

The **body** specifies how the result is computed. **No 'return'**

Function Notation

FLOLAC 2008

•*Function arguments need **not** be enclosed in brackets!*

Example:

```
square :: Float -> Float
square x = x*x
```

Calls: square **2.5** $\longrightarrow$ 6.25

Not
`square(2.5)`

`square (1.2+1.3)` $\longrightarrow$ 6.25

Brackets are for grouping only!

Simple Types

FLOLAC 2008

`Integer`	Unbounded integer numbers
`Int`	32-bit integer numbers
`Rational`	Unbounded rational numbers
`Float, Double`	Single- and double-precision floating point numbers
`Bool`	Boolean values: `True` and `False`
`Char`	Characters, e.g., `'a'`

The Booleans

- type `Bool`
- operations

`&&`	and
`||`	or
`not`	not

```
• exOr :: Bool -> Bool -> Bool
exOr x y = (x || y) && not (x && y)
```

The integers

- type `Int`: range `−2147483648`…`2147483647`
- type `Integer`: range unbounded
- operations

`+`	sum
`*`	product
`^`	raise to the power
`-`	difference
`div`	whole number division
`mod`	remainder
`abs`	absolute value
`negate`	change sign

Relational Operators

`>`	greater than
`>=`	greater than or equal to
`==`	equal to
`/=`	not equal to
`<=`	less than or equal to
`<`	less than

`(==)` for integers and Booleans. This means that `(==)` will have the type

```
Int -> Int -> Bool
Bool -> Bool -> Bool
```

Indeed `t -> t -> Bool` if the type t carries an equality.

```
(==) :: Eq a => a -> a -> Bool
```

Operators: Prefix and Infix

FLOLAC 2008

- Operators: infix. Use *parentheses* for prefix.
- Functions: prefix. Use *backquotes* for infix.

```
> 4*12-6
 42
> (<) 2 3
 True

> div 126 3
 42
> 126 ‘div‘ 3
 42
```

Precedence & Associativity

FLOLAC 2008

Op	Precedence	Associativity	Description
`^`	8	right	Exponentiation
`*, /`	7	left	Mul, Div
`` `div` ``	7	free	Division
`` `rem` ``	7	free	Remainder
`` `mod` ``	7	free	Modulus
`+, -`	6	left	Add, Subtract
`==,/=`	4	free	(In-) Equality
`<,<=,>,>=`	4	free	Relational Comparison

The characters

- type `Char`

`'a'`	
`'\t'`	tab
`'\n'`	newline
`'\\'`	backslash
`'\''`	single quote
`'\"'`	double quote
`'\97'`	character with ASCII code 97, i.e., `'a'`

Some operations:

$\texttt{toUpper 'a'} \longrightarrow \texttt{'A'}$

$\texttt{Ord 'a'} \longrightarrow \texttt{97}$

Composite Types: Lists

FLOLAC 2008

A list of values enclosed in square brackets.

A list of integers.

```
[1,2,3], [2] :: [Int]
```

homogeneous

Some operations:

```
[1,2,3] ++[4,5]  ──────→  [1,2,3,4,5]

head [1,2,3] ──────→  1

last [1,2,3] ──────→ 3

tail [1,2,3] ──────→ [2,3]
```

We can have lists of lists:

```
[ [1,3], [0, 5, 6], [4] ] :: [ [Int] ]
```

Quiz

FLOLAC 2008

How would you add *4* to the end of the list [1,2,3]?

Quiz

How would you add 4 to the end of the list [1,2,3]?

```
[1,2,3] ++ [4] ⟶ [1,2,3,4]
```

[4] not 4!
++ combines two *lists*,
and 4 is not a list.

Types: Strings

FLOLAC 2008

Any characters enclosed in double quotes.

```
"Hello!" :: String
```

List of Chars [Char]

Some operations:

$$\texttt{"Hello " ++ "World"} \longrightarrow \texttt{"Hello World"}$$

$$\texttt{First "Hello"} \longrightarrow \texttt{'H'}$$

Composite Types: Tuples

FLOLAC 2008

- A *tuple* is a sequence of components that may be of *different types*

```
(1, 4)                    :: (Int, Int)
(False, ‘b’, 4.294 )      :: (Bool, Char, Float)
(“Fish”, [True, True] )   :: (String, [Bool])
```

Tuples may contain basic types or list types

Tuple types

- The number of components in a tuple is called its *arity*.
- *Arity* cannot be 1.
- The tuple of *arity* zero () is called the *empty tuple*, while a tuple of *arity* 2 is called a *pair*, one of arity 3 a *triple*, and so on

Note that tuples are enclosed in parentheses, not square brackets

Tuples and Lists

FLOLAC 2008

You can have lists of tuples and tuples of lists

```
[(1, True),(4, False)]             :: [(Int, Bool)]

(1.4, [3, 5, 64, 7, 12], True) :: (Float, [Int], Bool)
```

The definition of the tuple provides its arity – in cases above the tuples have arity of 2 and 3 respectively

Function Types

- A function is a mapping of arguments of one type to results of another type
- T1 -> T2 maps arguments of type *T1* to results of type *T2*

```
~        :: Bool -> Bool
isDigit :: Char -> Bool
```

A Note on Function Types

- Function types associate to right.

```
maxOf3 :: Int -> Int -> Int -> Int
```

means

```
maxOf3 :: Int -> (Int -> (Int -> Int))
```

•Functions are values, and partial application is OK.

```
let m = maxOf3 5
 in let mm = m 8
    in mm 12
```

$\Longrightarrow$ 12

Multi-Parameter Functions

- A simple way (but usually not the right way) of defining a *multi-parameter* function is to use tuples:

```
add :: (Int,Int) -> Int
add (x,y) = x+y
```

- Evaluate

```
add (40,2)
```

- We get 42
- Later, we'll learn about *Curried Functions*.

Comments

- *Line comments* start with `--` and go to the end of the line:

```
--This is a line comment.
```

- *Nested comments* start with `{-` and end with `-}`:

```
{-
    This is a comment.
    {-
      And here’s another one....
    -}
 -}
```

Function Definition by Cases and Recursion

The abs function

- The absoulte value (*abs*) function:
  - abs x = |x|
- The definition is *by cases (multiple equations):*
  - $\text{abs } x = \begin{cases} x, & \text{if } x >= 0 \\ -x, & \text{if } x < 0 \end{cases}$
- How to define in Haskell?

```
abs x | x >= 0  = x
abs x | x < 0   = -x
```

A *guard*. An equation is used if its guard is True.

Evaluating abs

```
abs x | x >= 0  = x
abs x | x < 0   = -x
```

Prelude> abs (-2)

- First equation, x = -2
- What is -2 >= 0? → False
- Second equation, x = -2 again
- What is -2 < 0? → True
- Result is –x, that is –(–2)

2

Try the equations *in order*, use the first with a True guard

Other Forms

- Fully explicit

```
abs x | x >= 0 = x
abs x | x < 0  = -x
```

- Abbreviated left hand side

```
abs x | x >= 0 = x
      | x < 0  = -x
```

- Abbreviated last guard

```
abs x | x >= 0 = x
      | otherwise  = -x
```

- *"if" expression*

```
abs x =
    if x >= 0 then x else -x
```

Function Definition by Cases

```
fun v1 v2 … vn
  | g1            = e1
  | g2            = e2
  …
  | otherwise = e_r
```

$\}$ Guarded equations

```
max3 :: Int -> Int -> Int -> Int
max3 i j k | (i >= j) && (i >= k) = i
           | (j >= k) = j
           | otherwise = k
```

Function Definition by Cases

```
fun v1 v2 … vn
  | g1              = e1
  | g2              = e2
  …
  | otherwise = er
```

$\Leftrightarrow$

```
fun v1 v2 … vn =
   if g1 then e1
   else if g2 then e2
   else if · · ·
   else er
```

```
max3 :: Int -> Int -> Int -> Int
max3 i j k =
     if (i >= j) && (i >= k) then i
     else if (j >= k) then j
     else k
```

Recursive Functions

fac n = 1 * 2 * … * n

```
fac :: Int -> Int
fac n
 | n==0 = 1
 | otherwise = fac (n-1) * n
```

```
fac 0 = 1
fac n | n > 0 = fac (n-1) * n
```

or

```
fac :: Int -> Int
fac n = if n == 0 then 1
        else fac (n-1) * n
```

Evaluating Factorials

```
fac :: Int -> Int

fac 0 = 1

fac n | n > 0 = fac (n-1) * n
```

$$\begin{tikzcd}
\text{fac 4} \arrow[dd] & \text{?? 4 == 0} \arrow[r] & \text{False} & \text{fac 2 * 3 * 4} \arrow[d] \\
& \text{?? 4 > 0} \arrow[r] & \text{True} & \text{fac 1 * 2 * 3 * 4} \arrow[d] \\
\text{fac (4-1) * 4} \arrow[d] & & & \text{fac 0 * 1 * 2 * 3 * 4} \arrow[d] \\
\text{fac 3 * 4} \arrow[uuurrr] & & & \text{1 * 1 * 2 * 3 * 4} \arrow[d] \\
& & & \text{24}
\end{tikzcd}$$

Expensive to calculate...

Stack (space)

Time

```
fac 5
5 * (fac 4)
5 * 4 * (fac 3)
5 * 4 * 3 * (fac 2)
5 * 4 * 3 * 2 * (fac 1)
5 * 4 * 3 * 2 * 1 * (fac 0)
5 * 4 * 3 * 2 * 1 * 1
.
.
.
120
```

```
fac 0  = 1
fac n | n > 0 = n * fac (n-1)
```

Tail Recursion

FLOLAC 2008

- Tail recursion: recursive call occurs last
- The technique of *accumulating parameters*

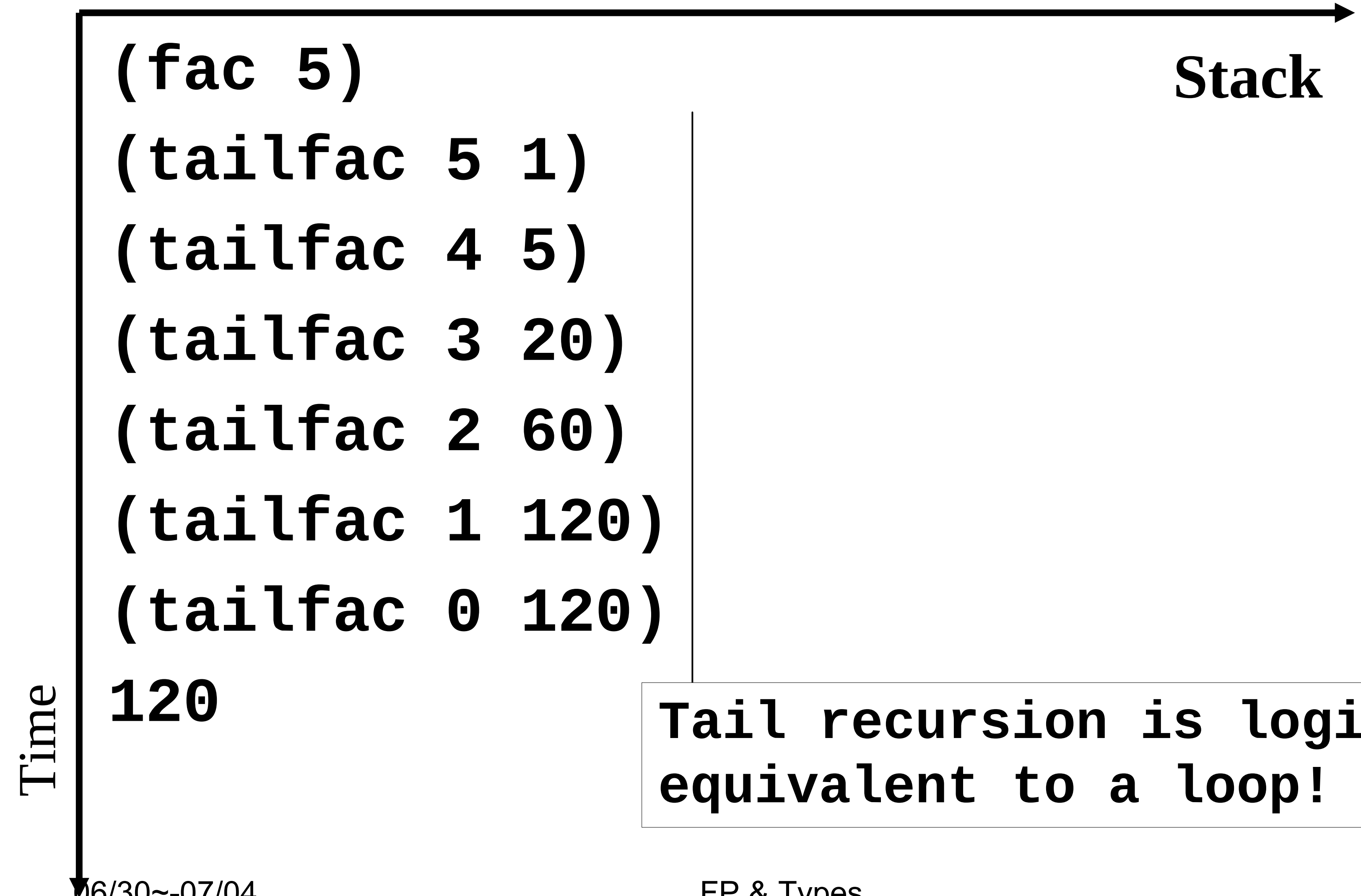
```
fac n = tailfac n 1
    where tailfac n acc
               | n==0 = acc
               | n>0  = tailfac (n-1) n*acc
```

- Local definitions

```
fac 5 → tailfac 5 1
      → tailfac 4 5*1
      → tailfac 3 4*5*1
      ...
```

A Better Process: Tail Recursion

FLOLAC 2008



```
(fac 5)
(tailfac 5 1)
(tailfac 4 5)
(tailfac 3 20)
(tailfac 2 60)
(tailfac 1 120)
(tailfac 0 120)
120
```

Tail recursion is logically equivalent to a loop!

Local Definitions: the `where` clause

FLOLAC 2008

- The `where-clause` follows after a function body:

```
fun args = <fun body>
     where
           decl_1
           decl_2
           . . .
           decl_n
```

```
maxOf3 :: Int -> Int -> Int -> Int

maxOf3 x y z = maxOf2 u z
               where
                 u = maxOf2 x y
```

Local Definitions: the `let` clause

```
let
  <local definitions>
in
  <expression>
```

```
fac n = let tailfac n acc
              | n==0 = acc
              | n>0  = tailfac (n-1) n*acc
        in
            tailfac n 1
```

The `let` Clause

```
f :: [Int] -> [Int]
f [ ] = [ ]
f xs =
     let
        square a = a * a
        one = 1
     in
        (square (head xs) + one) : f (tail xs)
```

```
f [3,2]
→ (square 3 + one) : f [2]  → … → [10,5]
```

The Layout Rule

Indentation determines where a definition ends:

```
circumference r =
      2 * pie * r

area r
 = pie * r * r

bad x = area x
+ circumference x     -- Error: offside!
```

Example

FLOLAC 2008

縮排而且對齊

```
let
     y = x + 2
     x = 5
in
     x / y
```

- same as:

```
let y = {x + 2; x = 5} in x / y
```

Example

The layout rule avoids the need for explicit syntax to indicate the grouping of definitions.

```
a = b + c
    where
      b = 1
      c = 2
d = a * 2
```

means

```
{a = b + c
     where
       {b = 1;
        c = 2};
d = a * 2}
```

implicit grouping

explicit grouping

The error Function

- *error* string can be used to generate an error message and terminate a computation.
- This is similar to Java's exception mechanism, but a lot less advanced.

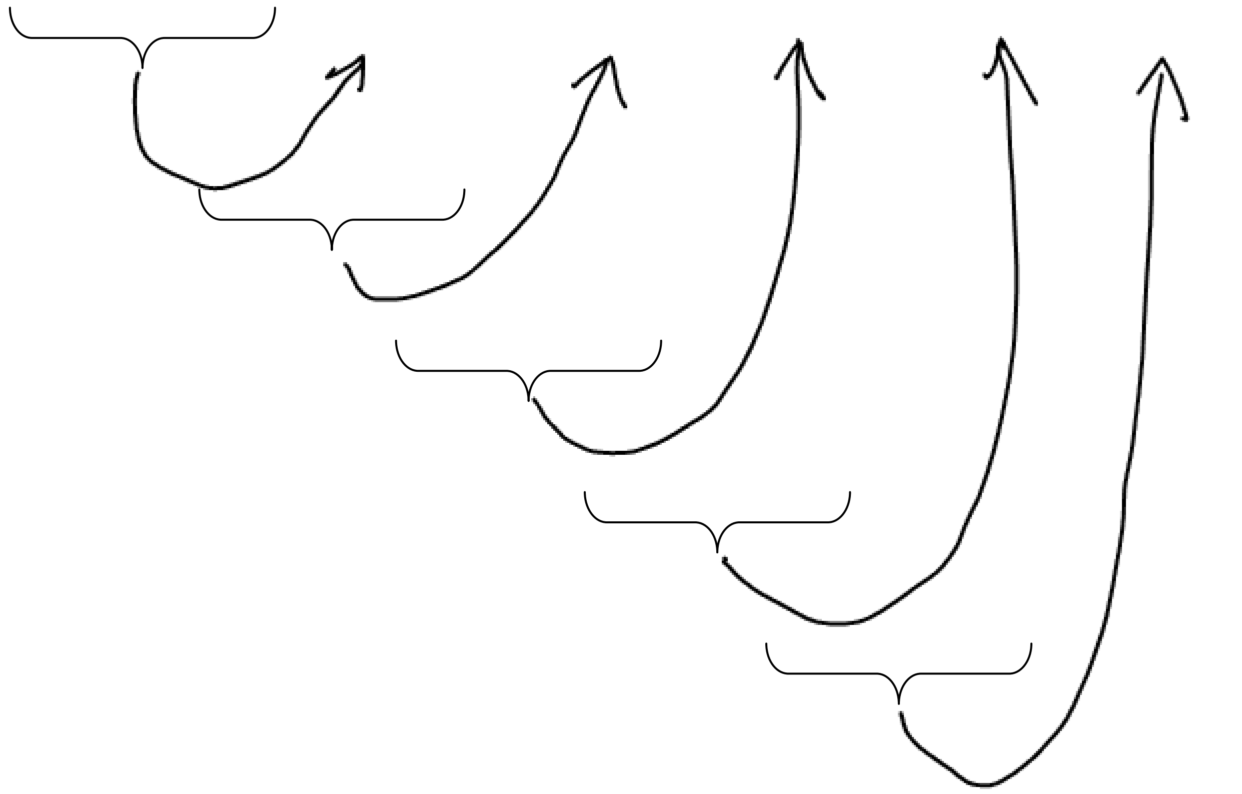
```
fac   :: Int -> Int
fac n = if n<0 then
          error "illegal argument"
        else if n <= 1 then 1
             else n * fac (n-1)
```

- `> f (-1)`

  Program error: illegal argument

Example: Fibonacci Numbers

1 1 2 3 5 8 13 21...



Computing Fibonacci Numbers

```
fib n | n > 1 = fib (n-1) + fib (n-2)
fib 0 = 1
fib 1 = 1
```

- Here there are *two* base cases
  - Neither can be reduced to a smaller problem by the recursive case.
- This definition is not very efficient – why not?

Tree Recursion

Inefficient!

$$\begin{tikzcd}
 & & & \texttt{(fib 5)} \arrow[dll] \arrow[drr] & & & & \\
 & \texttt{(fib 3)} \arrow[dl] \arrow[dr] & & & & \texttt{(fib 4)} \arrow[dl] \arrow[dr] & & \\
\texttt{(fib 1)} & & \texttt{(fib 2)} \arrow[dl] \arrow[d] & & \texttt{(fib 2)} \arrow[dl] \arrow[d] & & \texttt{(fib 3)} \arrow[dl] \arrow[dr] & \\
 & \texttt{(fib 0)} & \texttt{(fib 1)} & \texttt{(fib0)} & \texttt{(fib 1)} & \texttt{(fib 1)} & & \texttt{(fib 2)} \arrow[dl] \arrow[d] \\
 & & & & & & \texttt{(fib 0)} & \texttt{(fib 1)}
\end{tikzcd}$$

Repetitive computation

Rewrite it as tail recursive!

Pattern Matching

Pattern Matching

FLOLAC 2008

- Pattern matching is a simple and intuitive way of defining a function.
- The library function ~ returns the negation of a logical value:

```
~          ::    Bool -> Bool
~ False    =     True
~ True     =     False
```

Constant pattern; order matters

Pattern Matching

FLOLAC 2008

- We can also use pattern matching for functions that take more than one argument
- The library function (*&&)* returns the negation of a logical value

```
(&&)            ::      Bool -> Bool -> Bool
True && True     =      True
True  && False   =      False
False && True    =      False
False && False   =      False
```

Pattern Matching

- We can simplify the definition of (&&) by using the *wildcard* character _

```
(&&)              ::   Bool -> Bool -> Bool
True && True      =    True
 _   &&  _        =    False
```

- This is also good because if the first argument is *False* then it doesn’t need to evaluate the second argument

Pattern Matching

FLOLAC 2008

- Haskell has a naming convention that means that we *cannot use the same variable name* for more than one argument in an equation, so

```
b && b        = b
_ && _        = False
```
✗

would not be allowed, and needs to be rewritten as

```
b && c        | b==c        = b
              | otherwise = False
```
✓

Tuple Patterns

- A tuple of patterns is itself a pattern which matches any tuple of the same arity whose components match the corresponding patterns *in order*
- Constant patterns
  - `()`
  - `(1, 5)`
  - `('a', 5.5, "abcd")`
  - `("nested", (100, 'A'), (1,5,9))`
- Patterns with variables
  - `(1, x)`
  - `(s, i)`
  - `("nested", t1, t2)`

Tuple Patterns

- The library functions *fst* and *snd* select the first and second components of a pair

```
fst          :: (a,b) -> a
fst (x,_)    = x

snd          :: (a,b) -> a
snd (_,y)    = y
```

```
>fst (5, 'a') → 5      --(x binds to 5)
>snd (5, 'a') → 'a'    --(y binds to 'a')
```

More Selector Functions

- For pairs, we have

```
fst (x,y) = x        snd (x,y) = y
```

- For triples, we define

```
fst3 (x,y,z) = x
snd3 (x,y,z) = y
trd3 (x,y,z) = z
```

What would the type of the result be?

- No general selectors such as:

```
select 3 (x,y,z) = z
```

Selection using Pattern Matching

FLOLAC 2008

•Other than using special functions to select elements from a large tuple,
we can use pattern matching. Example:

```
(x1, x2, x3) = a_triple_value
```

Example:

```
(x1, x2, x3) = (100, ‘A’, “Math”)
```

Then `x1=100, x2=‘A’, x3=“Math”`.

List Patterns

- A list of patterns is also a pattern
- It matches any list *of the same length* whose elements all match the corresponding patterns in order. Example:

•Suppose we have a function *test* that checks if a list contains <u>precisely three characters and the first of these is the letter ‘a’</u>

```
test              :: [Char] -> Bool
test  [’a’,_,_]   = True
test  _           = False
```

List Patterns

FLOLAC 2008

- Lists are constructed one element at a time from the empty list
- The *cons* (construct) operator : produces a new list by adding a new element to the front of an existing list:

```
[3,5,7]
        = { apply cons }
3 : [5,7]
        = { apply cons }
3 : (5 : [7])
        = { apply cons }
3 : (5 : (7 : []))
```

•***cons* associates to the right:**

```
3:5:7:[]
```

Defining Functions with List Patterns

FLOLAC 2008

- We can use the *cons* function *(:)* to extend the *test* function to check the first element of a list of any length, not just three

```
test            :: [Char] -> Bool
test ('a':_)    = True
test _          = False
```

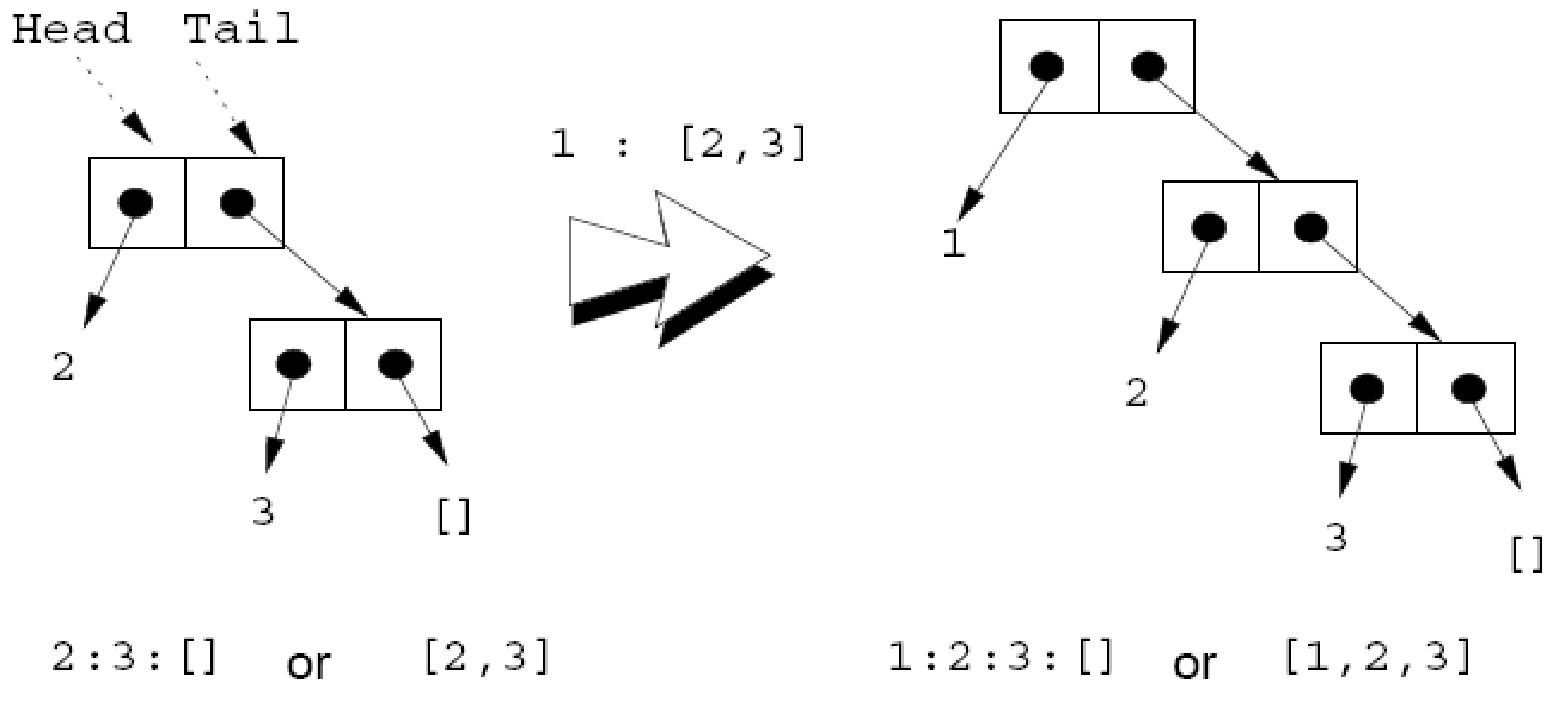
Defining Functions with List Patterns

FLOLAC 2008

- *Null*, *head*, and *tail* work in a similar manner

```
null           :: [a] -> Bool
null []        = True
null (_:_)     = False
head           :: [a] -> a
head (x:_)     = False
tail           :: [a] -> [a]
tail (_:xs)    = False
```

Internal Representation of Lists



Lists are Homogenous

- Lists of lists:

```
[1]:[[2],[3]]  ⇒     [[1],[2],[3]]
```

- Note that the elements of a list must be of *the same type*!

```
[1, [1], 1]              ⇒     Illegal!
[[1], [2], [[3]]]        ⇒     Illegal!
[1, True]                ⇒     Illegal!
```

Integer Patterns

- Haskell also allows integer patterns of the form *n*+*k* where *n* is an integer variable and *k*>0 and an integer constant
- *Pred* maps 0 to itself and any other number to the number preceding it

```
pred           :: Int -> Int
pred 0         = 0
pred (n+1)  = n
```

Recursion over Lists

FLOLAC 2008

•Compute the length of a list.

```
length ::[Int] -> Int
length xs = if xs ==[]    then 0
            else 1 + length (tail xs)
```

•This is called recursion on the tail .

•Using pattern matching:

```
length []     =  0
length (x:xs) = 1 + length xs
```

Evaluating Recursive Functions

```
length [] = 0
length (x : xs) = 1 + length xs
```

```
  length (1 : 2 : 4 : [])
⇒   [ x ← 1 , xs ← 2 : 4 : [] ]
  1 + length (2 : 4 : [])
```

Evaluating Recursive Functions

```
length [] = 0
length (x : xs) = 1 + (length xs)
```

```
  length (1 : 2 : 4 : [])
⇒    [ x ← 1 , xs ← 2 : 4 : [] ]
  1 + length (2 : 4 : [])
⇒    [ x ← 2 , xs ← 4 : [] ]
  1 + 1 + length (4 : [])
⇒    [ x ← 4 , xs ← [] ]
  1 + 1 + 1 + length []
⇒    []
  1 + 1 + 1 + 0
```

Polymorphic Functions & Types

- The `length` function does not care about the *element type* of its list parameter.

```
length [1,2,3]          ⇒    3
length [True, False]    ⇒    2
length [‘a’,‘b’,’c’,’d’] ⇒    4
```

- Indeed, `length` is a polymorphic function, and its type is:

```
length ::[a] -> Int
```

Here ***a*** is a *type variable* that can be instantiated to any types.

Sum and Product of a List

```
sum           :: [Int] -> Int
sum []        =  0
sum (x:xs)  = x + sum xs
```

```
product           :: [Int] -> Int
product []        =  0
product (x:xs) = x * product xs
```

Type Declarations and Checking

FLOLAC 2008

- In Java and most other languages the programmer has to declare what type variables, functions, etc have. We can do this too, in Haskell:

```
> 6*7 :: Int
  42
```

- `::Int` asserts that the expression `6*7` has the type `Int`.
- Haskell will check for us that we get our types right:

```
> 6*7 :: Bool
 ERROR
```

Type Inference

- We can let the Haskell interpreter *infer the type of expressions*, called type inference.
- The command `:t(ype) expression` asks Haskell to
- print the type of an expression:

```
> :type "hello"
  "hello" :: String
```

-
  ```
  > :type True && False
    True && False :: Bool
  ```

-
  ```
  > :t True && False :: Bool
    True && False :: Bool
  ```

Exercise

FLOLAC 2008

- Define a function `upto` such that for `m,n:Int` and `m <= n`

```
upto m n = [m, m+1, ..., n]
```

Variable Naming Convention

- When we write functions over lists it's convenient to use a consistent variable naming convention. We let

x, y, z, · · ·	denote *list elements*.
xs, ys, zs, · · ·	denote *lists of elements*.
xss, yss, zss, · · ·	denote *lists of lists of elements.*

List Concatenation

- `xs ++ ys` `--`also known as `append xs ys`

```
(++) :: [a] -> [a] -> [a]
[] ++ ys           = ys
(x : xs) ++ ys     = x : (xs ++ ys)
```

```
[1,2,3] ++ [4,5,6]
        = { apply ++ }
1: ([2,3] ++ [4,5,6])
        = { apply ++ }
1: (2: ([3] ++ [4,5,6]))
…
1: (2: (3: [4,5,6])))
        = { list notation }
[1,2,3,4,5,6]
```

List Concatenation

•Concatenate multiple lists in a list:

```
concat             :: [[a]] -> [a]
concat []           = []
concat (xs:xss) =  xs ++ concat xss
```

Examples:

```
concat []              =  []
concat [[]]            =  []
concat [[1], [3,5]] = [1,3,5]
```

More Polymorphic Recursive List Functions: reverse

FLOLAC 2008

- Reverse: reverse the order of the elements in a list

```
reverse  :: [a] -> [a]
reverse []       = []
reverse (x : xs) = reverse xs ++ [x]
```

Example

```
reverse [1,2,3,4]     ⇒   [4,3,2,1]
```

But, its Time complexity: **O(n^2)**

•Let’s define a *tail recursive* version of the reverse.

O(n)

Tail Recursive “reverse”

```
reverse   :: [a] -> [a]
reverse xs = rev2 xs []

rev2                :: [a] -> [a] -> [a]
rev2 []      ys   = ys
rev2 (x:xs) ys   = (rev2 xs) (x:ys)
```

“A LISP (FP) programmer knows
the *value* of everything
and the *cost* of nothing.”
--Alan Perlis

Zipping/Unzipping two lists

FLOLAC 2008

```
zip                :: [a] -> [b] -> [(a, b)]

zip []      ys   =  []
zip xs      []   =  []
zip (x:xs)  (y:ys)  =   (x,y) : zip xs ys
```

Ex: `zip [1,2] ['a','b'] = [(1,'a'),(2,'b')]`

```
Unzip           :: [(a,b)] -> ([a], [b])
unzip []                 =  []
unzip ((x,y) : ps)  =  (x:xs, y:ys)
                      where
                         (xs,ys) = unzip ps
```

Yet more list functions in the Prelude

FLOLAC 2008

- Many more list functions in the Prelude:
  - Take, drop, (!!), …

- `Prelude>` **`take 3 "catflap"`**
  `"cat"`
- `Prelude>` **`drop 2 [‘d',‘r',‘o',‘p']`**
  `“op"`
- `Prelude>` **`“abcde” !! 3`**
  `d`

Exercises:

- Defining the *drop* function:
  - drop 2 [1,2,3,4,5] = [3,4,5]

```
drop                 :: Int -> [a] -> [a]
```

- Defining the *init* function:
  - init [1,2,3,4,5] = [1,2,3,4]   --remove the last element

```
init                 :: [a] -> [a]
```

Mutual Recursion

FLOLAC 2008

- Functions that reference to each other
- Example: given a list, selecting *even* or *odd* positions from it.

```
evens::[a]->[a]
odds ::[a]->[a]
```

evens “abcde”
 = { apply *evens* }
’a’ : *odds* “bcde”
 = { apply *odds* }
’a’ : *evens* “cde”
 = { apply *evens* }
’a’ : ’c’ : *odds* “de”
 = { apply *odds* }
’a’ : ’c’ : evens “e”
…

Mutual Recursion

•Given a list, selecting *even* or *odd* positions from it.

```
evens              ::    [a] -> [a]
evens []           =     []
evens (x : xs)     =     x : odds xs

odds               ::    [a] -> [a]
odds []            =     []
odds (_ : xs)      =     evens xs
```

Arithmetic Sequences

- Haskell provides a convenient notation for lists of numbers where the difference between consecutive numbers is constant.

```
[1..3]   ⇒ [1,2,3]
 [5..1]   ⇒   []
```

- A similar notation is used when the difference between consecutive elements is = 1: Examples:

```
[1,3..9]   ⇒ [1,3,5,7,9]
[9,8..5]   ⇒ [9,8,7,6,5]
[9,8..11] ⇒ []
```

Or, in general:

```
  [m,k..n] ⇒ [m,m+(k-m)*1,m+(k-m)*2,· · · ,n]
```

List Comprehension

List comprehensions allow many functions on lists to be performed in a clear and precise manner

List Comprehension

FLOLAC 2008

- Mathematical form

  $\{ x^2 \mid x \in \{1..5\} \}$

  produces the set {1,4,9,16,25}

- Haskell

```
> [ x^2 | x<-[1..5] ]
[1,4,9,16,25]
```

where

|	means “such that”
<-	means “is drawn from”; “for each element in”

Generators

FLOLAC 2008

- The expression x<-[1..5] is called a *generator*
- Generators can also use *patterns* when drawing elements from a list.

Suppose *ps* is a list of pairs:

```
[(1,True), (2,False), (5,False), (9,True)]
```

If we want to extract all pairs of the form (*x*, *True*) then we can do this using the generator

```
> [ x | (x,True)<-ps ]
  [1,9]
```

Generators

FLOLAC 2008

- We can also use wildcards in generators
- If we take the same list of pair *ps*

  [(1,*True*), (2,*False*), (5,*False*), (9,*True*)]

then

```
> [ x | (x,_)<-ps ]
  [1,2,5,9]
```

extracts the list of the first components of the pairs

Generators

- The library function *length* is also defined using a wildcard within a generator

```
length    :: [a] -> Int
length xs =  sum [1 | _<-xs]
```

- The length is calculated by creating a list that contains the value 1 for each element in *xs*, then summing this new list

Multiple Generators

- List comprehensions can have multiple generators separated by *commas*
- We can generate a list of all possible pairings of the elements in two lists using

```
>[(x,y)| x<-[1,2], y<-[8,9] ]
[(1,8),(1,9),(2,8),(2,9)]
```

- The second generator cycles faster than the first generator.
- Swap the order:

```
>[(x,y)| y<-[1,2], x<-[8,9] ]
```

Generators

- A later generator can also depend on the value of an earlier generator
- The following list comprehension produces a list of all possible ordered pairings of the elements of [1..3] in order:

```
➢ [(x,y)| x<-[1..3], y<-[x..3] ]

[(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]
```

Generators

- Similarly we could define the library function *concat*, which concatenates lists, by using one generator to select each list then a second generator to select each element within the list

```
concat        :: [[a]] -> [a]
concat xss  =  [x | xs<-xss, x<-xs]
```

Guards

- As well as using generators to create sets, we can also use *guards* to filter the values produced by generators
- If a guard is *True* then the value is retained, otherwise it is discarded

```
> [x | x<-[1..10], even x]
[2,4,6,8,10]
```

- The function *even x* is the guard function

Guards

- Similarly we can produce a function that maps a positive integer to its list of positive factors

```
factors    ::   Int -> Int
factors n =     [x | x <-[1..n],
                     n ‘mod‘ x==0 ]
```

- So

```
> factors 15
[1,3,5,15]
```

Guards

- We can extend this to find primes, as a prime is a number whose only factors are 1 and the number itself

```
prime       ::    Int -> Bool
prime n     =     length (factors n == 2)
```

So

```
> prime 15        > prime 7
False             True
```

Guards

- We can use guards to implement a look-up table where a list of pairs of keys and values represents the data
- If the keys are of an equality type then we can create a function *find* that returns a list of all values associated with a given key

String Comprehensions

FLOLAC 2008

- List comprehensions can be used to define functions on strings
- The function *digits* returns the list of integer digits from a string

```
digits      :: String -> [Int]
digits xs = [ord x - ord ’0’ | x <- xs,
                                isDigit x ]
```

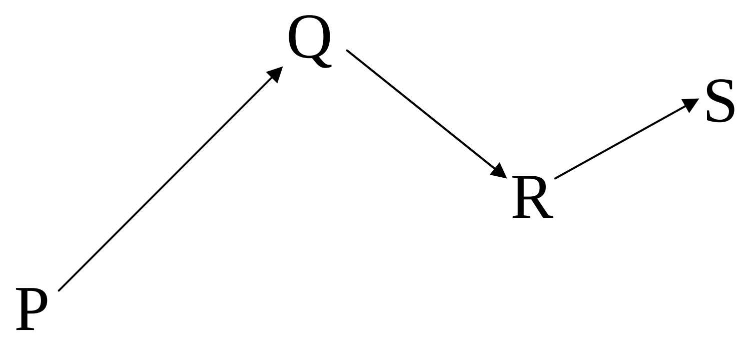
So

```
> digits “1*5+3”
 [1,5,3]
```

An Longer Example

An Example:

Computing *path distance*



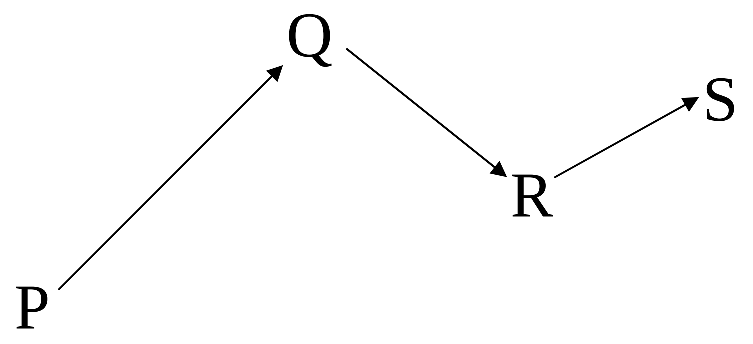
Representing a Point

```
type Point = (Float, Float)
```

x- and y-coordinates.

```
distance :: Point -> Point -> Float
distance (x, y) (x’, y’) =
            sqrt ((x-x’)^2 + (y-y’)^2)
```

Representing a Path



```
type Path = [Point]

examplePath = [p, q, r, s]

path_length = distance p q + distance q r
                  + distance r s
```

Two Useful Functions

FLOLAC 2008

- `init xs` -- all but the last element of xs,
- `tail xs` -- all but the first element of xs.

```
init [p, q, r, s]  ⇒ [p, q, r]
tail [p, q, r, s]  ⇒ [q, r, s]


zip …                [(p,q), (q,r), (r,s)]
sum [1,2,3] ⇒ 6
```

The pathLength Function

```
pathLength :: Path -> Float
pathLength xs = sum [ distance p q
      | (p,q) <- zip (init xs) (tail xs)]
```

Example:

```
pathLength [p, q, r, s] ⇒
 distance p q + distance q r + distance r s
```

Higher-Order Functions

- Functions take functions as *arguments*
- Functional values and Lambda Expressions
- Functions return functions as *results*.

A Motivating Example

FLOLAC 2008

Write a Haskell function **incAll** that adds **1** to each element in a list of numbers.

E.g., **`incAll [1, 3, 5, 9] = [2, 4, 6, 10]`**

```
incAll :: [Int] -> [Int]

incAll [] = []
incAll (n : ns) = n+1 : incAll ns
```

A Motivating Example, cont’d

- Write a Haskell function **lengths** that computes the lengths of each list in a *list of lists.*

E.g.,

```
lengths [[1,3], [], [5, 9]] = [2, 0, 2]
lengths ["but", "and, "if"]] = [3, 3, 2]
```

```
lengths :: [[a]] -> [num]

lengths [] = []
lengths (l : ls)
          = (length l) : lengths ls
```

Similarity and Abstraction

```
incAll [] = []
incAll (n : ns) = (+) n 1       : incAll ns
```

```
lengths (l : ls)  = (length l) : lengths ls
lengths [] = []
```

$\Downarrow$

Let f be (+) or length: `f (hd l)  : recCall (tail l)`

$\Downarrow$

$l = [l_1, l_2, \ldots l_n]$:

$[f\ l_1, f\ l_2, \ldots f\ l_n]$

List map function

- Given a function and a list (of appropriate types), applies the function to each element of the list.

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

map f [] = []
map f (x : xs) = (f x) : map f xs
```

$$[l_1,\ l_2,\ \ldots,\ l_n] \xrightarrow{\texttt{map f}} [f\ l_1,\ f\ l_2,\ \ldots\ f\ l_n]$$

Using `map`

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

```
incAll = map (plus 1)
         where plus m n = m + n

lengths = map (length)
```

Note that `plus :: Int -> Int -> Int`,

so

`(plus 1) :: Int -> Int`.

Functions of this kind are sometimes referred to as **partially evaluated (applied)**.

Partial Applications

Any function may be called with fewer arguments than it was defined with.

The result is a *function* of the remaining arguments.

If
```
f ::Int -> Bool -> Int -> Bool
```
then
```
f 3 :: Bool -> Int -> Bool

f 3 True :: Int -> Bool

f 3 True 4 :: Bool
```

Bracketing Function Calls and Types

FLOLAC 2008

We say function application "brackets to the left"

function types "bracket to the right"

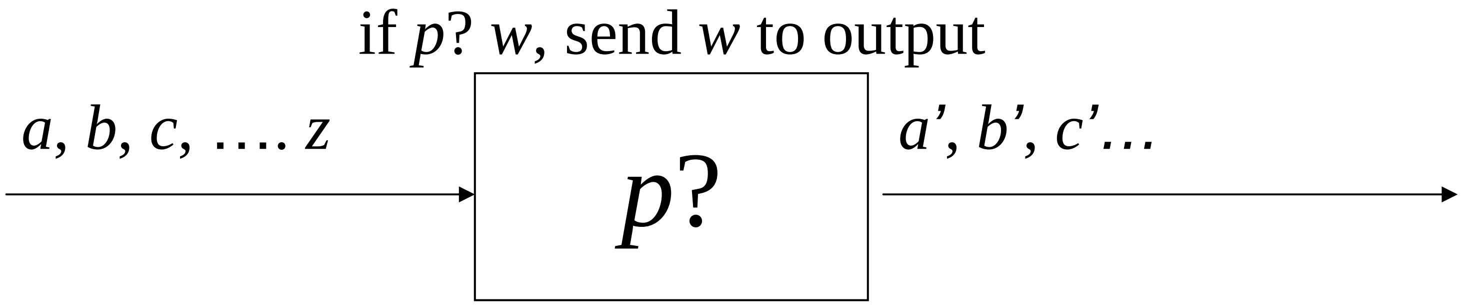
If
```
f ::Int -> (Bool -> (Int -> Bool))
```
then
```
f 3 :: Bool -> (Int -> Bool)
(f 3) True :: Int -> Bool
((f 3) True) 4 :: Bool
```

Functions really take only *one* argument, and return a function expecting more as a result.

Another HoF: List filtering

FLOLAC 2008

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



filter ***even*** **[1,2,3,4,6] = [2,4,6]**

```
even x = x ‘mod’ 2 == 0
```

Lambda Expressions

- Functions can also be defined using *lambda expressions*
- These are nameless functions made up of
  - A pattern for each of the arguments
  - A body that shows how the result can be calculate from the arguments
- These are shown in Haskell using \ or mathematically using λ

```
Example:  \x  -> (x, x, x)
```

```
\ parameter -> body
```

Lambda Expressions

FLOLAC 2008

- The *square* function could also be implemented as a lambda expression

```
\x -> x * x
```

- Lambda expressions can be used in the same way as other functions

```
> (\x->x*x) 2
 4
```

```
map square
[1,2,4]
 ≡
map (\x->x*x)
[1,2,4]
```

```
filter (\x -> x `mod` 2 ==0) [2,3,5,6,7]
```

-> has lowest precedence, extends to the right

Lambda Expressions

- Lambda expressions can also be used to show the meaning of curried expressions

```
add x y = x + y
```

can be understood as

```
add = \x -> (\y -> x + y)
```

which shows that the function takes a number *x* which returns a function which in turn takes another number *y* and returns the sum of the two numbers

More About Functional Values

- Functions returning functions
- Partial Application
- Curried Functions

Sections

Haskell distinguishes **operators** and **functions**: *operators* have **infix** notation (e.g. **1 + 2**), while *functions* use **prefix** notation (e.g. **plus 1 2**).

Operators can be converted to functions by putting them in brackets: **(+) m n = m + n**.

Sections are *partially evaluated operators*. E.g.:

- **(+ m) n = m + n**
- **(0 <)   x = 0 < x**
- **(0 :) l   = 0 : l**

Using `map` More

FLOLAC 2008

```
squareAll = map (^2)
squareAll [1,2,3,4] = [2,4,9,16]
```

- What do the following functions do?

```
1. addNewlines = map (++ "\n")
     addNewlines :: [[Char]] -> [[Char]]

2. stringify = map (: [])
      stringify :: [Char] -> [String]
```

Functions Returning Functions

FLOLAC 2008

- Another view of *partial application: functions returning functions.* Example:
  - ***makeAdder n*:** creates a function add *n* to its argument

```
makeAdder :: Int->(Int->Int)
makeAdder n = \x -> x+n
    or
makeAdder = \n -> \x -> x+n
```

```
  incAll: [Int]->[Int]
  incAll = map (makeAdder 1)
```

Currying

There is a one-to-one correspondence between the types **(A,B) -> C** and **A -> (B -> C)**.

Given a function **f :: (A,B) -> C** ,
its *curried* equivalent is the function

```
curriedF :: A -> B -> C

curriedF a b  =  f (a,b)
```

Currying in Haskell

•Haskell functions are implicitly curried; multiple arguments can be applied one at a time.

```
plus x y = x + y

plus1 =  plus 1

plus1 5 = 6
```

•But `add (x, y) = x + y` requires a pair of arguments: `add(1, 5)`

fold (reduce) functions

Motivating Examples

1. **`product`**: multiplies all the elements in a list of numbers together.

```
product [2,5,26,14] = 2*5*26*14 = 3640
```

```
product :: [Int] -> Int
product [] = 1
product (n : ns) = n * product ns
```

2. **`concat`**: Concatenate multiple lists

```
concat [[2,5], [], [26,14]]= [2,5,26,14]
```

```
concat  :: [[a]] -> [a]
concat []          = []
concat (xs:xss) =  xs ++ xss
```

Folding

A general pattern for the functions **product** and **concat** is *replacing constructors with operators*. For example, **product** replaces **:** (cons) with ***** and **[]** with **1**:

```
1 : (2 : (3 : (4 : []))))
1 * (2 * (3 * (4 *  1)))
```

- **concat** replaces **:** (cons) with **++** and **[]** with **[]**:

```
[2,5] : ([] : ([3,4] : []))
[2,5] ++([] ++([3,4] ++[]))
```

Folding Right

Haskell has a built-in function, **foldr**, that does this replacement:

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

foldr f e [] = e
foldr f e (x : xs) = f x (foldr f e xs)
```

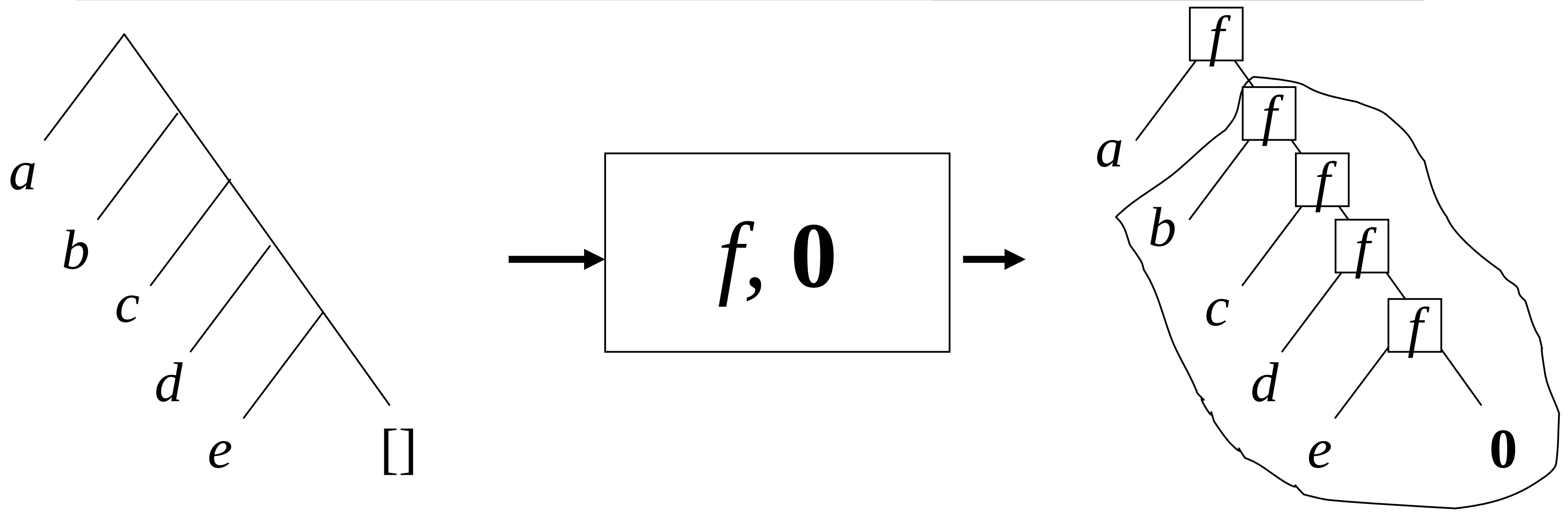
$$\texttt{(*) 1 } \texttt{(}\underbrace{\texttt{2 * (3 * (4 * 1))}}_{\text{recusive call}}\texttt{)}$$

```
product = foldr (*)  1

concat  = foldr (++) []
```

Visualizing *foldr*

FLOLAC 2008

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f e [] = e
foldr f e (x : xs) = f x (foldr f e xs)
```



```
foldr (-) 0 [1,2,3,4,5] = (1-(2-(3-(4-(5-0)))))
                        = 3
```

Folding Left

FLOLAC 2008

Another direction to fold: **foldl**:

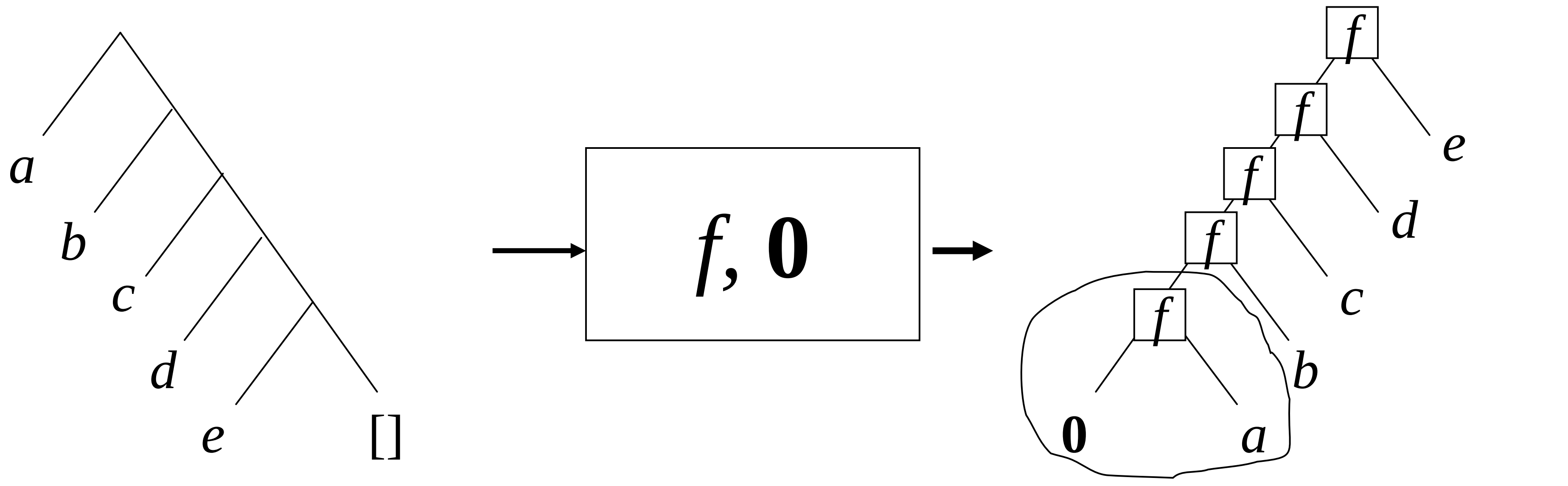
```
foldl :: (b -> a -> b) -> b -> [a] -> b

foldl f e [] = e
foldl f e (x : xs) = foldl f (f e x) xs
```

- `product  = foldl (*) 1`
- `concat = foldl (++) []`
  - `foldl max 0 [1,2,3] = 3`

    where `max a b = if a > b then a else b`

Folding Left (reduce)

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f e [] = e
foldl f e (x : xs) = foldl f (f e x) xs
```



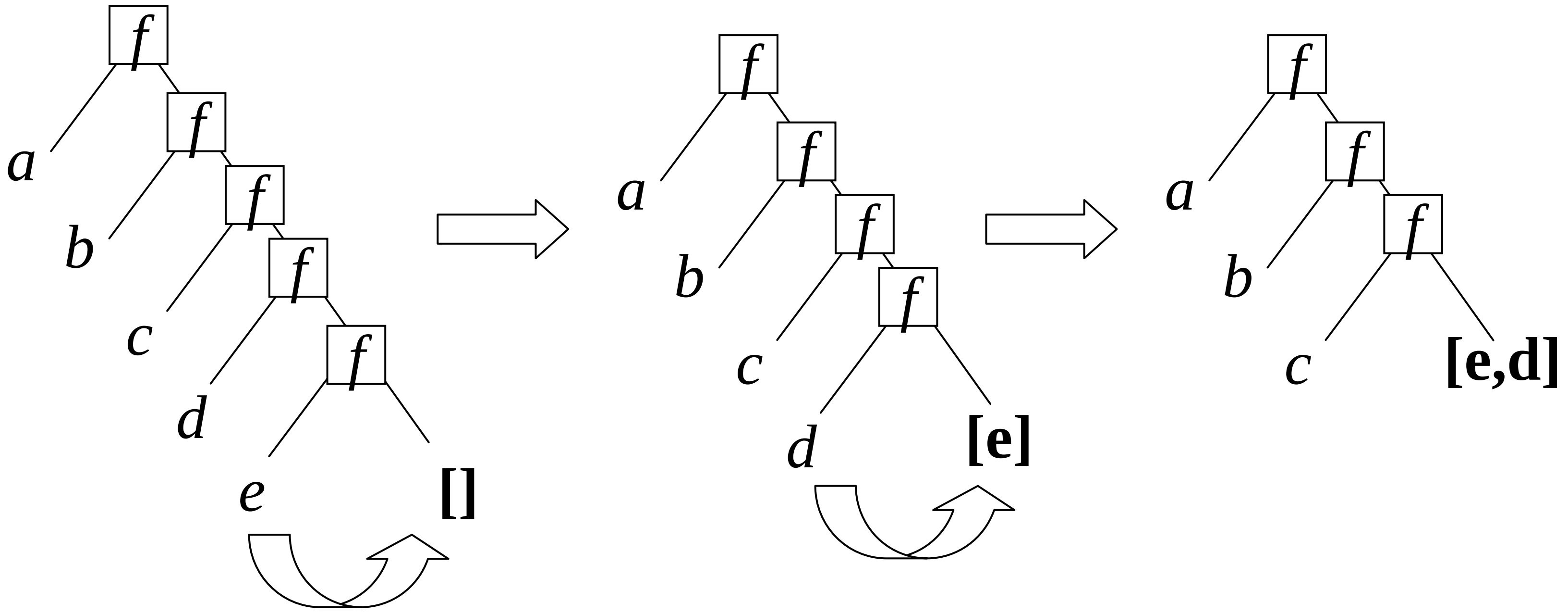
```
foldl (-) 0  [1,2,3,4,5] = (((((0-1)-2)-3)-4)-5)
                           = -15
```

Reversing a list using *foldr*

```
reverser   :: [a] -> [a]
reverser = foldr snoc []
      where snoc x xs = xs ++ [x]
```

--O(N^2)

--Add 'x' to the end of xs

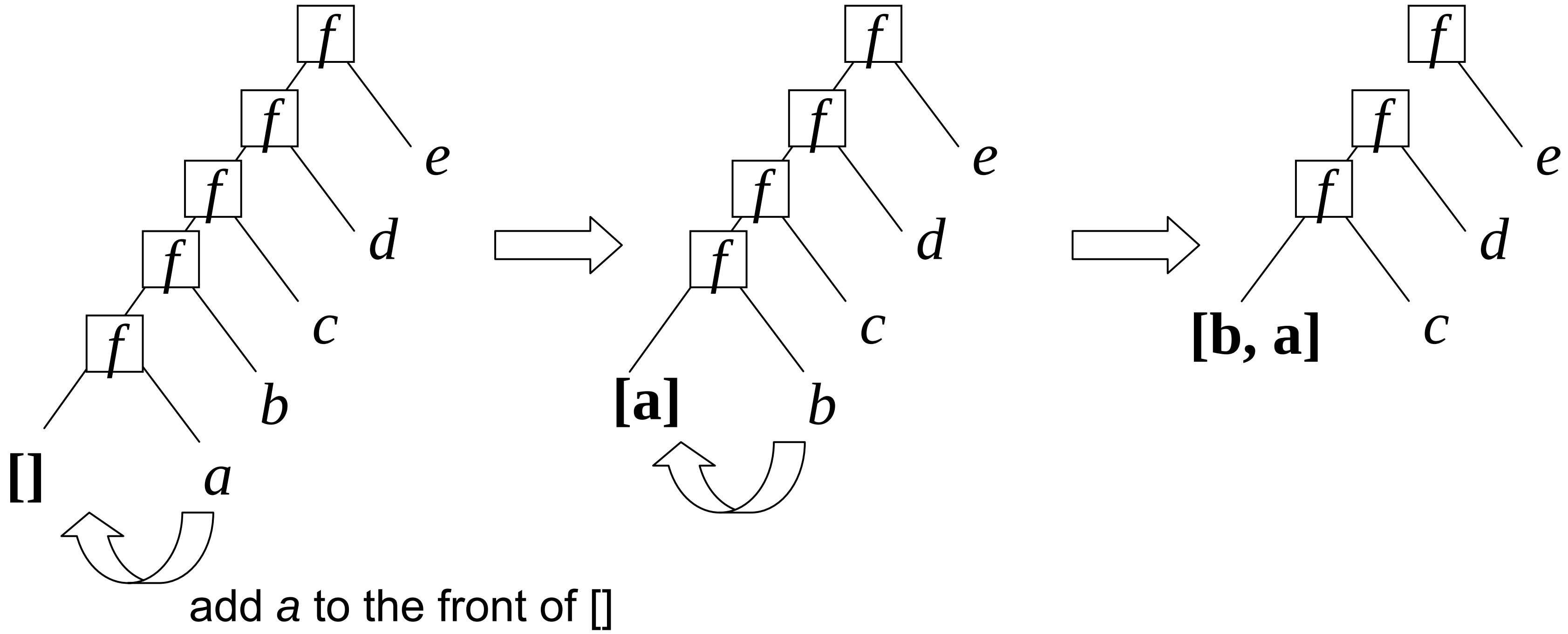


Add 'e' to the end of []

Reversing a list using *foldl*

FLOLAC 2008

```
reversel :: [a] -> [a]
reversel  = foldl cons []                    --O(N)
                 where cons xs x = x : xs
```

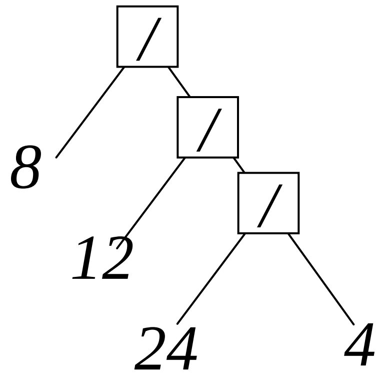


Specialized fold

FLOLAC 2008

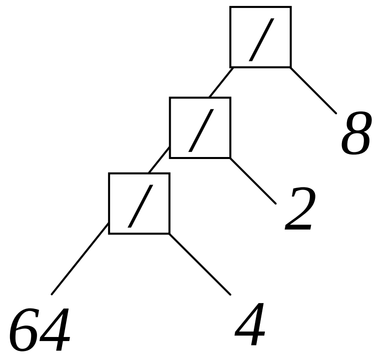
```
foldr1 :: (a -> a -> a) -> [a] -> a
```

```
foldr1 (/) [8,12,24,4] = 4.0
```



```
foldl1 :: (a -> a -> a) -> [a] -> a
```

```
foldl1 (/) [64,4,2,8] = 1.0
```



Combing Map and Reduce

Consider the three sums

FLOLAC 2008

- $1 + 2 + \ldots + 100 = (100 * 101)/2$
- $1 + 4 + 9 + \ldots + 100^2 = (100 * 101 * 102)/6$
- $1 + 1/3^2 + 1/5^2 + \ldots + 1/101^2 = \pi^2/8$

$$\sum_{k=1}^{100} k$$

$$\sum_{k=1}^{100} k^2$$

$$\sum_{k=1,odd}^{101} k^{-2}$$

In mathematics they are all captured by the notion of a **sum**:

$$\sum_{x \in l} f(x)$$

Can we express this abstraction directly?

Look at the three functions

$$\sum_{k=1}^{100} k$$ = sum-integers 1 100

```
sumIntegers k  n  =
     if   k > n  then  0 else
          k  +  (sum-integers  (k+1)  n)
```

$$\sum_{k=1}^{100} k^2$$ = sum-squares 1 100

```
sumSquares  k  n  =
   if  k > n  then  0 else
     (square k) + (sum-squares  (k+1)  n)
```

$$\sum_{k=1,odd}^{101} k^{-2}$$ = pi-sum 1 101

```
piSum  k  n  =
   if  k > n  then  0 else
    (1/(square  k)) +  (pi-sum   (k+2)  n)
```

Abstraction from the three functions

$$\sum_{x \in l} f(x)$$

```
sum  f  next  k n  =
  if  k > n then  0 else
     (f  k) +
        sum  f  next  (next k)  n
```

```
sumIntegers k  n  =
    if   k > n  then  0 else
         k  +  (sum-integers  (k+1)  n)
```

```
sumSquares  k  n  =
   if  k > n  then  0 else
     (square k) + (sum-squares  (k+1)  n)
```

```
piSum  k  n  =
   if  k > n  then  0 else
    (1/(square  k)) +  (pi-sum   (k+2)  n)
```

- sumIntegers = sum (\x->x) (+1)
- sumSquares = sum (\x->x^2) (+1)
- piSum = sum si (+2)
  where si x = 1/(x*x)

```
Id x = x
```

Using `map` and `reduce`

To implement summation: $\sum_{x \in l} f(x)$

```
sum f l =  foldl (+) 0 (map f l)
```

E.g.,

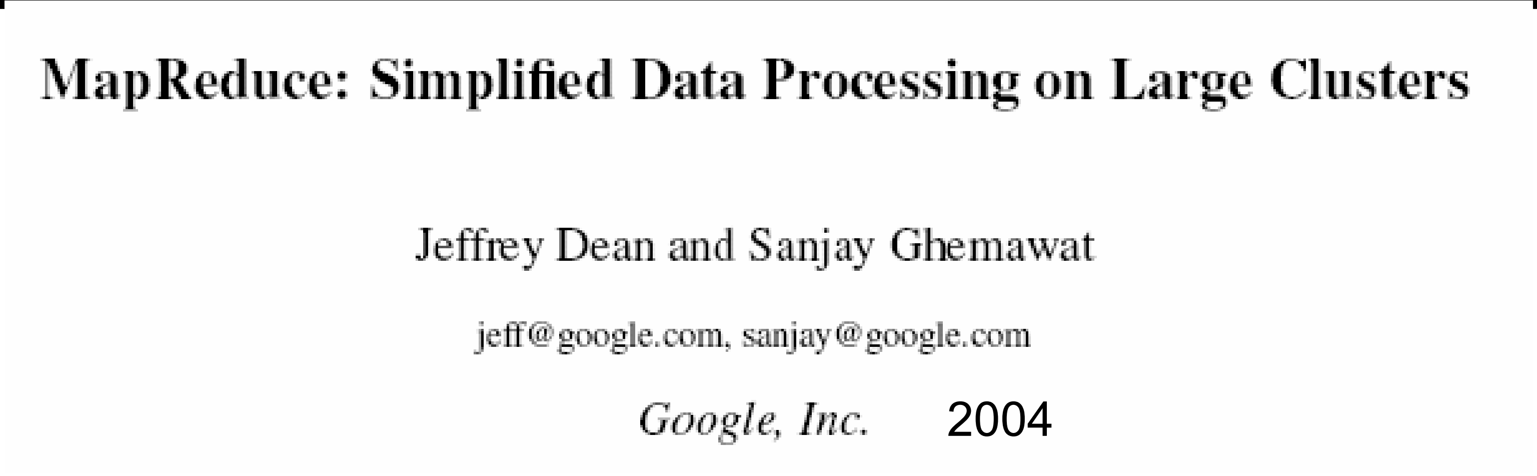
$\sum(x)$: > `sum  (\x->x) [1, 2, 3]`

value: 6

$\sum(x^2)$: > `sum  (\x->x*x) [1, 2, 3]`

value: 14

Google is using FPL, too



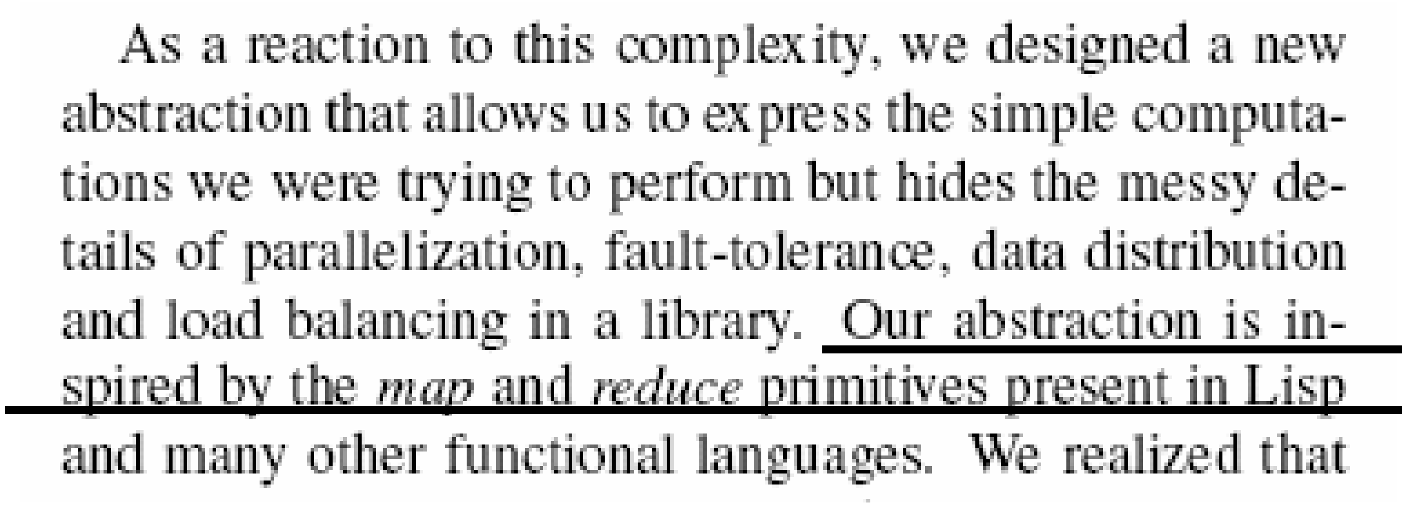
MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

2004



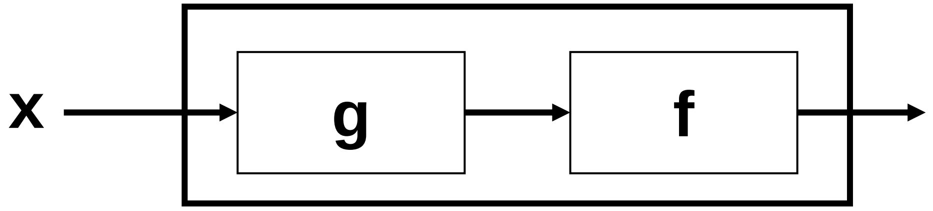
As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. Our abstraction is inspired by the *map* and *reduce* primitives present in Lisp and many other functional languages. We realized that

Function Composition

FLOLAC 2008

Function composition is a higher-order function.

```
compose ::
  (b -> c) -> (a -> b) -> a -> c
compose f g x = f (g x)
```



There is a Haskell operator **.** that implements **compose**:

```
infixr . 9
(f . g) x = f (g x)
```

Composition Example

FLOLAC 2008

Define a function count which counts *the number of lists of length n* in a list L:

```
    count 2 [[1],[],[2,3],[4,5],[]] = 2
```

Using recursion:

```
count :: Int -> [[a]] -> Int
count [] = 0
count n (x:xs)
     | length x == n = 1 + count n xs
     | otherwise     = count n xs
```

Using functional composition:

```
count' n = length . filter (==n) . map length
```

Composition Example

- Double the numbers in a list

```
double :: [Int] -> [Int]
double xs = map (* 2) xs
```

- Remove negative numbers from a list

```
positive :: [Int] -> [Int]
positive xs = filter (0<) xs
```

- Double the positive numbers in a list

```
doublePos :: [Int] -> [Int]
doublePos xs = map (* 2) (filter (0<) xs)
```

or

```
doublePos = map (* 2) . filter (0<)
```

Defining New Data Types

- Enumerated types
- Parameterized types
- Recursive types

Type Declarations

•A new *name* for an existing type can be defined using a type declaration.

```
type String = [Char]
```

--`String` is a *synonym* for the type `[Char]`.

•Type declarations can be used to make other types easier to read. For example, given

```
type Pos = (Int,Int)
```

•We can define

```
left   :: Pos → Pos
left (x,y) = (x-1,y)
```

Type Declarations

- Like function definitions, type declarations can also have *parameters*. For example, given

```
type Pair a = (a,a)
```

we can define:

```
bits  :: Pair Int
bits   = (0,1)

copy  :: a → Pair a
copy x = (x,x)
```

- Type declarations can be *nested*:

```
type Pos   = (Int,Int)
type Trans = Pos → Pos
```
✓

- However, they *cannot be recursive*:

```
type Tree = (Int,[Tree])
```
✗

Defining New Types

FLOLAC 2008

- Enumerated

```
data Bool = False | True
```

- Parameterized (polymorphic)

```
data Maybe a = Nothing | Just a
```

- Recursive

```
Data List a = Nil | Cons a (List a)
```

Enumerated

Example:

```
data Bool = False | True
```

Bool is a new type, with two *new values* False and True.

- **data** is a keyword - defines a new (*algebraic)* data type.
- `Bool` is the *type name.*
- `True`, `False` are *constructors.*
- `True:: Bool, False ::Bool`
- The type name and constructors must begin with an *upper case letter.*

Enumerated

Values of new types can be used in the same ways as those of built in types. For example, given

```
data Answer = Yes | No | Unknown
```

we can define:

```
answers      :: [Answer]
answers       = [Yes,No,Unknown]

flip         :: Answer → Answer
flip Yes      = No
flip No       = Yes
flip Unknown = Unknown
```

Enumerated

The constructors in a data declaration can also have *parameters*. For example, given

```
data Shape = Circle Float
           | Rect Float Float
```

we can define:

```
square            :: Shape
square             = Rect 1 1

area              :: Shape → Float
area (Circle r) = pi * r^2
area (Rect x y) = x * y
```

Continued:

```
data Shape = Circle Float
           | Rect Float Float
```

- *Shape* has values of the form `Circle r` where `r` is a float, and `Rect x y` where `x` and `y` are floats.

- `Circle` and `Rect` can be viewed as <u>functions</u> that simply construct values of type Shape:

```
Circle :: Float → Shape

Rect   :: Float → Float → Shape
```

Parameterized (Polymorphic)

Not surprisingly, data declarations themselves can also have *parameters*. For example, given

```
data Maybe a = Nothing | Just a
```

we can define:

```
zero :: Maybe Int
zero  = Just 0

app  :: (a → b) → Maybe a → Maybe b
app f Nothing  = Nothing
app f (Just x) = Just (f x)
```

Recursive Types

In Haskell, new types can be defined in terms of themselves. That is, types can be recursive.

```
data Nat = Zero | Succ Nat
```

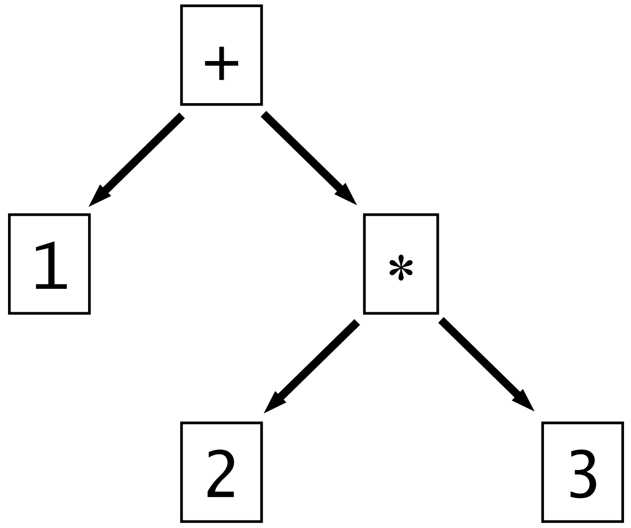
Nat is a new type, with constructors Zero :: Nat and Succ :: Nat → Nat.

Nat contains the following infinite sequence of values:

```
Zero
Succ Zero
Succ (Succ Zero)
•
•
•
```

Modeling Arithmetic Expressions

```
1 + (2 * 3)
```



Arithmetic Expressions

- We can define a suitable new recursive type to represent these expressions

```
data Expr = Val Int
          | Add Expr Expr
          | Mul Expr Expr
```

- So the tree for 1 + 2 * 3 could be represented as

```
Add (Val 1) (Mul (Val 2) (Val 3))
```

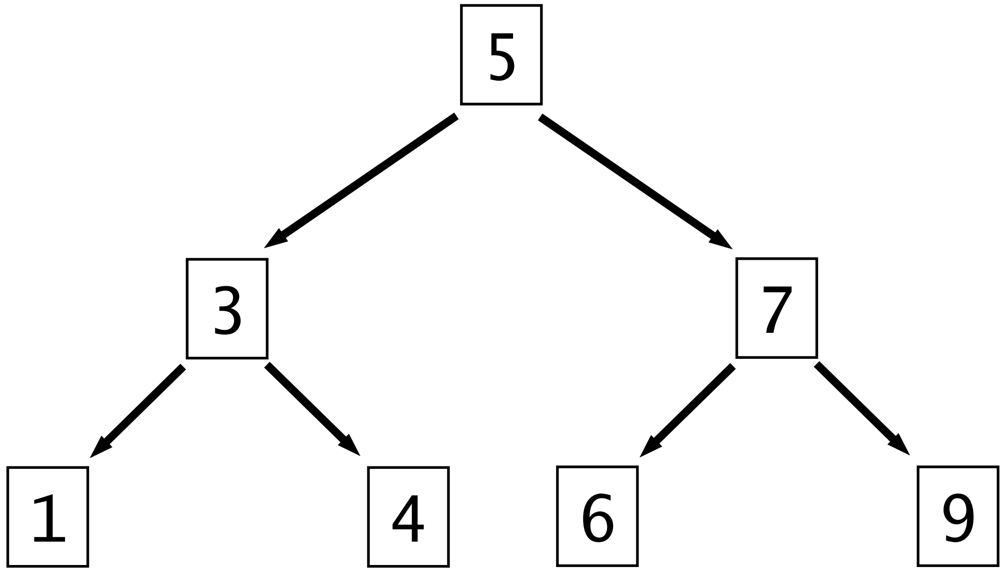
Arithmetic Expressions

- We can define recursive functions to process expressions

```
size             :: Expr -> Int
size (Val n)     = 1
size (Add x y)   = size x + size y

eval             :: Expr -> Int
eval (Val n)     = n
eval (Add x y)   = eval x + eval y
eval (Mul x y)   = eval x * eval y
```

Binary Trees

In computing, it is often useful to store data in a two-way branching structure or binary tree.



Binary Trees

Using recursion, a suitable new type to represent such binary trees can be defined by:

```
data Tree = Leaf Int
          | Node Tree Int Tree
```

For example, the tree on the previous slide would be represented as follows:

```
Node (Node (Leaf 1) 3 (Leaf 4))
     5
     (Node (Leaf 6) 7 (Leaf 9))
```

Binary Trees

- The function *flatten* returns the list of all integers contained in the tree

```
flatten                ::Tree -> [Int]
flatten (Leaf n)       = [n]
flatten (Node l n r)= flatten l
                         ++ [n]
                         ++ flatten r
```

- If the tree flattens to an ordered list then the tree is a *search tree*
- Our example flattens to [1,3,4,5,6,9]

Searching a Binary Tree

We can define a function find that decides if a given integer occurs in a binary tree:

```
find                  :: Int → Tree → Bool
find x (Leaf n)        = x==n
find x (Node l n r)    = x==n
                         || find x l
                         || find x r
```

However, this function simply traverses *the entire tree*, and hence for our example tree may require up to seven comparisons to produce a result.

Binary Search Trees

Search trees have the important property that when trying to find a value in a tree we can always decide which of the two sub-trees it may occur in:

```
find x (Leaf n)                = x==n
find x (Node l n r) | x==n = True
                    | x<n  = find x l
                    | x>n  = find x r
```

For example, trying to find any value in our search tree only takes at most three comparisons.

Lazy Evaluation

Haskell is Lazy

Haskell only evaluates a sub-expression if it's necessary to produce a result.

This is called **lazy** (or **non-strict**) **evaluation**

```
Main> head []
program error: empty argument list

Main> fst (0, head [])
0
Main>
```

Patterns Force Evaluation

Haskell **will** evaluate a subexpression to test if it matches a pattern. Suppose we define:

```
myFst (x, 0) = x
myFst (x, y) = x
```

Then the second argument is always evaluated:

```
Main> myFst (0, maxList [])
program error: empty argument list
Main>
```

Lazy But Productive

Haskell will produce as much of a result as possible:

```
Main> [1, 2, div 3 0, 4]
[1,2,
program error: [primQrmInteger 3 0]

Main> map (1/) [1, 2, 0, 7]
[1.0,0.5,
 program error: [primDivDouble 1.0 0.0]
```

Lazy Evaluation

Lazy evaluation: a sub-expression is evaluated only if it is necessary to produce a result.

The Haskell interpreter implements **topmost-outermost** evaluation:

> **Rewriting is done as near the "top" of the parse tree as possible**.

For example:

```
reverse (1 : ((f 2) : [])) --[1, f 2]
```

Topmost-Outermost

```
reverse (n : ns) = snoc n (reverse ns)
snoc h tl = tl ++ [h]
```

$$
\begin{array}{l}
\quad\underline{\texttt{reverse (1 : ((f 2) : []))}} \\
\Rightarrow \\
\quad\underline{\texttt{(snoc 1 (reverse ((f 2) : [])))}} \\
\Rightarrow \\
\quad\texttt{(}\underline{\texttt{reverse ((f 2) : [])}}\texttt{) ++ [1]} \\
\Rightarrow \\
\quad\texttt{(}\underline{\texttt{(snoc (f 2) (reverse []))}}\texttt{) ++ [1]} \\
\Rightarrow \\
\quad\texttt{((reverse []) ++ [(f 2)]) ++ [1]}
\end{array}
$$

Topmost-Outermost

$$
\begin{array}{l}
\quad \texttt{((}\underline{\texttt{reverse []}}\texttt{) ++ [(f 2)]) ++ [1]} \\
\Rightarrow \\
\quad \texttt{(}\underline{\texttt{[] ++ [(f 2)]}}\texttt{) ++ [1]} \\
\Rightarrow \\
\quad \underline{\texttt{[(f 2)] ++ [1]}} \\
\Rightarrow \\
\quad \texttt{[(f 2),1]}
\end{array}
$$

`(f 2)` is not evaluated!

Infinite Lists

Haskell has a "dot-dot" notation for lists:

```
Main> [0..7]
[0,1,2,3,4,5,6,7]
```

The upper bound can be omitted:

```
Main> [1..]
[1,2,3,4,5,6,7, ...
...
2918,2919,291<<not enough heap space --
task abandoned>>
```

Using Infinite Lists

Haskell gives up displaying a list when it runs out of memory, but infinite lists like **[1..]** can be used in programs that only use a part of the list:

```
Main> head (tail (tail (tail [1..])))
4
```

This style of programming is often summarized by the phrase "**generators** and **selectors**"

- **[1..]** is a generator
- **head.tail.tail.tail** is a selector

Generators and Selectors

Because Haskell implements lazy evaluation, it only evaluates as much of the generator as is necessary:

```
Main> head (tail (tail (tail [1..])))
5
Main> reverse [1..]
ERROR - Garbage collection fails to
reclaim sufficient space
Main>
```

Another Selector

FLOLAC 2008

The built-in function **takeWhile** returns the longest initial segment that satisfies a property **p**:

```
takeWhile :: (a -> Bool) -> [a] -> [a]
takeWhile p [] = []
takeWhile p (x : xs)
  | p x       = x : takeWhile p xs
  | otherwise = []
```

```
Main> takeWhile (<10) [1, 2, 13, 3]
[1,2]
```

Selectors

Note that evaluation of **`takeWhile`** stops as soon as the given property doesn't hold, whereas evaluation of **`filter`** only stops when the end of the list is reached:

```
Main> takeWhile (<10) [1..]
[1,2,3,4,5,6,7,8,9]

Main> filter (<10) [1..]
[1,2,3,4,5,6,7,8,9
                                 ERROR!
```

Eratosthenes' Sieve

A number is **prime** iff

- it is divisible only by 1 and itself
- it is at least 2

The sieve:

- start with all the numbers from 2 on
  - delete all *multiples* of the *first* number from the remainder of the list
  - repeat

Eratosthenes' Sieve

```
primes :: [Int]
primes = sieve [2..]
 where
  sieve (x:xs) =
    x : sieve [ y | y <- xs, y `mod` x /= 0 ]
```

```
Main> take 5 primes
 [2,3,5,7,11]
```

Never-Ending Recursion

The expression **[n..]** can be implemented generally by a function:

```
natsfrom :: num -> [num]
natsfrom n = n : natsfrom (n+1)
```

This function can be invoked in the usual way:

```
Main> natsfrom 0
[0,1,2,3,....                    ERROR!

Main> take 3 (natsfrom 0)
[0,1,2]
```

Iterate

```
-- iterate f x == [x, f x, f (f x), ...]
```

```
iterate :: (a -> a) -> a -> [a]
iterate f x = x : iterate f (f x)
```

```
Main> iterate (*2) 1
[1,2,4,8,16,32,64,128,256,512,1024,...

Main> iterate (drop 3) ”abcdef”
[”abcdef”, ”def”, ””, ””, ...
```

Problem: Grouping List Elements

```
group :: Int -> [a] -> [[a]]
group = ?

Main> group 3 ”apabepacepa!”
[”apa”,”bep”,”ace”,”pa!”]
```

```
Hint: map (take 3) (iterate (drop 3) ”abcdef”)
      => map (take 3)[”abcdef”, ”def”, ””, ””, ...
      => [”abc”, ”def”, ””, ””, ...
```

```
group :: Int -> [a] -> [[a]]
group n = takeWhile (not . null)
        . map (take n)
        . iterate (drop n)
```

Suggested Reading

- Paul Hudak, *"Conception, Evolution, and Application of Functional Programming Languages,"* ACM Computing Surveys 21/3, 1989.
- Paul Hudak and Joseph H. Fasel, *"A Gentle Introduction to Haskell,"* ACM SIGPLAN Notices, vol. 27, no. 5, May 1992. <Haskell tutoria>l
- Simon Thomson, *The Craft of Functional Programming*, 2nd Ed., Addison-Wesley,1999.
- Graham Hutton, *Programming in Haskell,* Cambridge Univ. Press, 2007

More to learn about Haskell

- Type classes
- Constructor classes
- IO Monads
- State handing in a monadic style
- …
- Various research-oriented extensions in GHC

Acknowledgement

- Some of the materials presented here are taken from the slides prepared by :
- Professor G. M. Hutton, Nottingham Univ., UK
- Professor J. Hughes, Chalmers Univ., Sweden
- Professor N. Whitehead, University of Akureyri in Iceland
- Professor G. Malcolm, Univ. of Liverpool, UK

Unit 2: Type Systems for FP

Part I: the λ Calculus

The foundation of all FP languages.

The λ-Calculus

FLOLAC 2008

The λ-calculus was developed by the logician **Alonzo Church** in 1930’s as a tool to study *functions and computability*.

λ-calculus in Computer Science

- Computability
  - λ-definability, Church 1930's
  - Equivalent to *Turing Machines*, Turing 1937
  - Equivalent to *recursive functions*, Kleene 1936
- Programming languages, 1960's
  - Naming, functions
  - Lisp, Algol 60, ISWIM
- Language theory, 1970's
  - Semantics: operational and denotational
  - Type systems

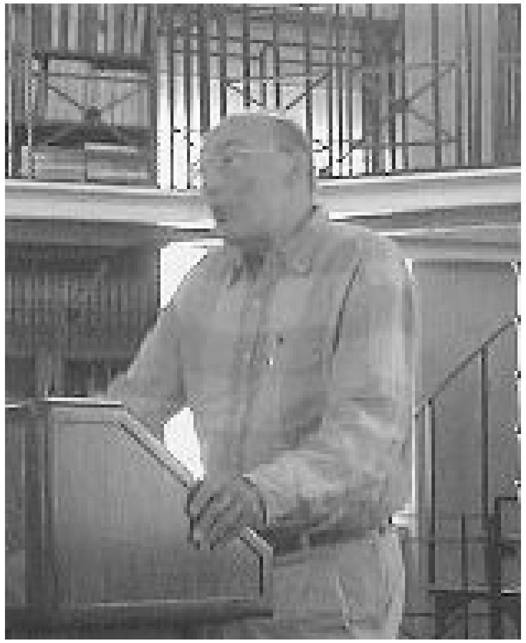
Original Aims of the λ-calculus

- A foundation for logic (1930’s)
  - failed
- A theory of functions (Church 1941)
  - model for computable functions
- Success 30 years later in Computer Science!

The Next 700 PL's

FLOLAC 2008

Peter Landin develops ISWIM, the first *pure* functional language, based strongly on the lambda calculus, with no assignments.



" Whatever the next 700 languages turn out to be, they will surely be variants of lambda calculus."

(Landin 1966)

Lambda calculus with constants

Lambda Calculus: Variants

- The <u>pure lambda calculus</u> (LC) is a *untyped* language composed entirely of functions
- The simply typed lambda calculus (SLC)
- The polymorphic typed lambda calculus (PLC)
- …

Pure Untyped λ-calculus

- Syntax is simple: •M,N are called λ-terms or λ-expressions

$$– \; M,N ::= \underset{\substack{\uparrow\\ \text{variable}}}{x} \;\mid\; \underset{\substack{\uparrow\\ \text{abstraction}}}{\lambda x.M} \;\mid\; \underset{\substack{\uparrow\\ \text{application}}}{M\,N}$$

- No types: e.g., **`(λx.x)y`**; **`(λx.x)(λx.x)`**
- No numbers or operations
    - can be added
    - values are function abstractions
- Functions are nameless
    – No “let *f* = λx.M in N”

Syntax of λ-Terms

FLOLAC 2008

- Examples:
  - λx.x ： the identity function
  - (λy. λx. x) f g： discards the first argument
- Notational conventions:
  - applications associate to the *left* (like in Haskell):
    - "y z x" is "(y z) x"
  - the body of a lambda *extends* as far as possible to the *right*:
    - "$\lambda x.\underbrace{x\ \lambda z.x\ z\ x}$" is "λx.(x λz.(x z x))"
  - "λx. λy. x y" often abbreviates to "λx y. x y"

Terminology

- Bound variables (parameters)
- Free variables
- Example:

- $\boxed{\lambda x.x\ y}$

y is free in the term $\lambda x.x\ y$

x is bound
in the term $\lambda x.x\ y$

Terminology

- λx.M

the scope of x is the term M

- λx.x y

y is free in the term λx.x y

x is bound
in the term λx.x y

```
FV(x) = {x}
FV(λx.M) = FV(M) \
{x}
FV(M N) = FV(M) ∪
FV(N)
```

Open Closed

Open	Closed
– $FV(E) \neq \{\}$	– $FV(E) = \{\}$
– xz	– λx.x
– λx.xz	– λx.λy.xy
– (λx.x)y	– (λx.x)(λy.y)
– (λy.(λx.xz)y)w	– λf.λg.λx.f x (g x)

- Ex. Underline the bound variables

Evaluating λ- Terms

- Function application is straightforward:

```
( λ x.(f x)) y --> f y
```

substitute **y** for **x** in **(f x)**

•Reduce all applications **(λx.L)N**

•Until none can be found

Evaluating λ- Terms

- β -reduction

$(\lambda \underline{x}.\ x\ x)\ \underline{(\lambda y.\ y)}$

$\dashrightarrow_{\beta}$ x x [λy.y / x]

== (λy. y) (λy. y)

$\dashrightarrow_{\beta}$ y [λy.y / y]

== λy. y

M [N/x] is the term in which all free occurrences of *x* in *M* are replaced with *N.*

This replacement operation is called **substitution.** we will define it carefully later in the appendix

Examples of β-reduction

FLOLAC 2008

1. $(\lambda \boldsymbol{x} . \boldsymbol{x})\, \boldsymbol{a} \underset{[a/x]}{\rightarrow_\beta} \boldsymbol{a}$

2. $(\lambda \boldsymbol{x} . \lambda \boldsymbol{y} . \boldsymbol{x})\, \boldsymbol{a}\, \boldsymbol{b} \underset{[a/x]}{\rightarrow_\beta} (\lambda \boldsymbol{y} . \boldsymbol{a})\, \boldsymbol{b} \underset{[b/y]}{\rightarrow_\beta} \boldsymbol{a}$

3. $(\lambda \boldsymbol{x} . \boldsymbol{x}\, \boldsymbol{a})\, (\lambda \boldsymbol{x} . \boldsymbol{x}) \underset{[\lambda x.x/x]}{\rightarrow_\beta} (\lambda \boldsymbol{x} . \boldsymbol{x})\, \boldsymbol{a} \underset{[a/x]}{\rightarrow_\beta} \boldsymbol{a}$

4. $(\lambda \boldsymbol{x} . \lambda \boldsymbol{y} . \boldsymbol{x}\, \boldsymbol{y})\, \boldsymbol{y} \underset{[y/x]}{\rightarrow_\beta} (\lambda \boldsymbol{y} . \boldsymbol{y}\, \boldsymbol{y})$ $\boxed{y \text{ Become bound}}$

Name capturing error!

A Similar Example in C Macro

FLOLAC 2008

- Name capturing problem in macro expansion

```
#define swap(X,Y) [ int tmp=X; X=Y; Y=tmp; ]
```

```
int a, b;
a = 5;
b = 10;
swap(a, b);
```

=> OK

```
[int tmp=b; b=a;
     a=tmp;]
```

```
int a, tmp;
a=5;
tmp = 10;
swap(a, tmp);
```

=> **oops!** tmp got trapped

```
[int tmp=a; a=tmp;
     tmp=tmp; ]
```

Renaming Bound Variables

- Names of *bound variables (parameters)* do not matter.
- Example: $\lambda x.\ x =_\alpha \lambda y.\ y =_\alpha \lambda z.\ z$
  - But NOT:

    $\lambda y.\ \underline{x}\ y =_\alpha \lambda y.\ \underline{z}\ y$

- This is called α **conversion** in lambda calculus

$$\boxed{\lambda x\ .\ E \quad =_\alpha \lambda z\ .\ E[z/x] \qquad \text{(z is not free in E)}}$$

$\lambda y.\ \underline{x}\ y[x/y]$ will make the “free” x captured.

Example Revisited

4. $(\lambda x . \lambda y . x\, y)\, y \rightarrow_\beta \lambda \overset{\curvearrowleft}{y} . y\, y$

$\boxed{y \text{ Become bound}}$

$\Downarrow$ Renaming the bounded y

4. $(\lambda x . \lambda y . x\, y)\, y \rightarrow_\alpha (\lambda x . \lambda z . x\, z)\, y$

$\rightarrow_\beta (\lambda z . y\, z)$

[y/z]

Normal Forms

- Evaluation via β-reduction
- Terms **$(\lambda$x.L)N** are called β-redexes

- β-normal form = no β-redexes

- **$(\lambda$x.xx)y** $\leftarrow$ a β-redex

- $\rightarrow_\beta$ **yy** $\leftarrow$ β-normal form

- Not all λ-terms have β-nf

An example with no NF

$(\lambda x.\ x\ x)\ (\lambda x.\ x\ x)$

$\to_\beta\ x\ x\ [\lambda x.\ x\ x/x]$

$==\ (\lambda x.\ x\ x)\ (\lambda x.\ x\ x)$

$\to$ … looping, no normal form

$\Omega = (\lambda x.\ x\ x)$

$\boxed{\Omega\Omega \text{ has no } \beta\text{-nf}}$

- In other words, it is simple to write non-terminating computations in the lambda calculus

Evaluation Strategy (Order)

- A term may have many *redexes:*

$$\underline{(\underline{\lambda x.(\lambda y.y)z)}\ (\underline{(\lambda z.z)w)}}$$

- Which application first?
- Does it matter?
- Yes:
  - Full Beta Reduction
  - Normal Order
  - Call-By-Name (CBN)
  - Call-By-Value (CBV) (Applicative Order), etc.

Full Beta Reduction

FLOLAC 2008

- Any redex can be chosen, and evaluation proceeds until no more redexes found.
- For example,

$$
\begin{aligned}
& (\lambda x.\underline{(\lambda y.y)z})\ ((\lambda z.z)w) \\
\text{-->}_{\beta}\ & \underline{(\lambda x.z)\ ((\lambda z.z)w)} \\
\text{-->}_{\beta}\ & z
\end{aligned}
$$

Normal Order Reduction

- Deterministic strategy which chooses the *leftmost, outermost redex*, until no more redexes.
- Example:

$$
\begin{aligned}
& \underline{(\lambda x.(\lambda y.y)z)\ ((\lambda z.z)w)} \\
\longrightarrow_\beta\ & (\lambda y.y)z \\
\longrightarrow_\beta\ & z
\end{aligned}
$$

Why Not Normal Order?

FLOLAC 2008

- In most (all?) programming languages, *functions are considered values (fully evaluated)*
- Thus, no reduction is done inside of functions (under the **lambda**)

  $\boxed{\lambda x.\ M \text{ is a value}}$, not reducible
- No popular programming language uses normal order

Call by Name; Call by Value

- Consider the application: (λ **x. E) e$_1$**

- Call by value: evaluate the argument e_1 to a value before β reduction

- Call by name: reduce the application, *without* evaluating e_1
- In both cases: a lambda abstraction: λ x. E is a value.

Call-By-Name/Call-By-Value

FLOLAC
2008

- CBN example
- CBV example

$\underline{\texttt{id (id (}\lambda\texttt{z. id z))}}$

$\to_\beta\ \underline{\texttt{id (}\lambda\texttt{z. id z)}}$

$\to_\beta\ \lambda\texttt{z. id z}$

$\texttt{(id }\underline{\texttt{(id (}\lambda\texttt{z. id z))}}$

$\to\ \underline{\texttt{id (}\lambda\texttt{z. id z)}}$

$\to\ \lambda\texttt{z. id z}$

where $\texttt{id} = \lambda\texttt{x.x}$

Order of Evaluation May Matter Much

- CBV (Inner redex):

$(\lambda y\,.\,\lambda z\,.\,z)\;\underline{((\lambda x\,.\,x\,x)\;(\lambda x\,.\,x\,x))} \rightarrow_\beta$

$(\lambda y\,.\,\lambda z\,.\,z)\;((\lambda x\,.\,x\,x)\;(\lambda x\,.\,x\,x)) \rightarrow_\beta\;\ldots$

- CBN (Outer redex):

$\underline{(\lambda y\,.\,\lambda z\,.\,z)}\;((\lambda x\,.\,x\,x)\;(\lambda x\,.\,x\,x)) \rightarrow_\beta$

$(\lambda z\,.\,z)$

1st sequence is infinite. 2nd has normal form.

Normalization Theorem

If a λ-expression E has a normal form, then the *normal order strategy* will terminate in a normal form. (Curry & Feys, 1958)

Church-Rosser Corollary

The normal form of a λ-expression, if it exists, is unique.

$$\begin{tikzcd}
 & \mathbf{E} \arrow[dl] \arrow[dr] & \\
\mathbf{E1} \arrow[dr] & & \mathbf{E2} \arrow[dl] \\
 & \mathbf{nf} &
\end{tikzcd}$$

Comparison

- The call-by-value strategy is *strict*
- The arguments to functions are always evaluated, whether or not they are used by the body of the function
- *Non-strict* (or *lazy*) strategies evaluate only the arguments that are actually used
  - call-by-name
  - call-by-need

LC and Type Theories

- Russell's paradox:

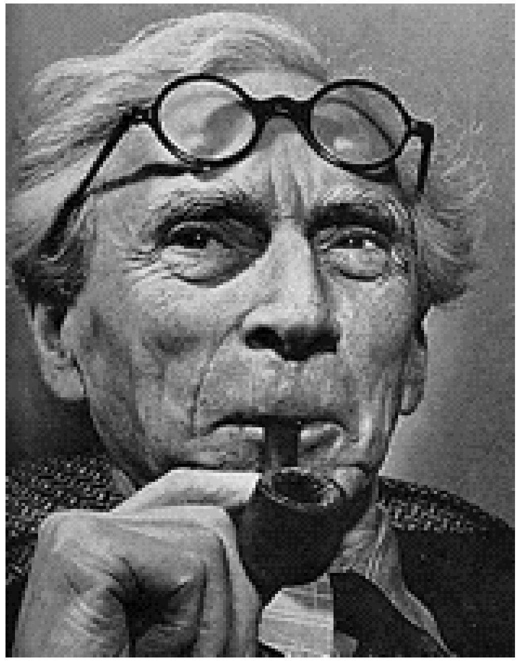
$$\boxed{R = \{\ X \mid X \notin X\ \}, \quad \text{is} \quad R \in R?}$$

- Russell developed type theory, attempting to solve the paradox.

- Church encounters similar issues in pure LC:

$$\boxed{\Omega = (\lambda x.x\ \ x)\ ,\ \Omega\ \Omega \ \text{ has no NF}}$$

- Church proposed the simply typed LC (1941)



Lambda Calculus and Programming Languages

Programming in the Lambda Calculus

We can do everything

- The lambda calculus can be used as an “assembly language”
- We can show how to *compile* useful, high-level operations and language features into the lambda calculus
  - Result = adding high-level operations is convenient for programmers, but not a computational necessity
  - Result = make your compiler intermediate language simpler

Compile the Let Expressions

- Given the let expressions in Haskell

```
let x = e1 in e2
```

- Question: can we implement this construct in the lambda calculus?

$$\begin{tikzcd}
\boxed{\textbf{\textit{source}} \textbf{ = lambda calculus + let}} \arrow[d, "\textbf{translate/compile}"] \\
\boxed{\textbf{\textit{target}} \textbf{ = lambda calculus}}
\end{tikzcd}$$

Compile the Let Expressions

- Given the let expressions in Haskell

```
let x = e1 in e2
```

- Question: can we implement this construct in the lambda calculus?

Example:

```
let f = \x.xz  in \y.f (f y)
```

$\Downarrow$

```
( \f.\y.f (f y) )(\x.xz)
```

Compile the Let Expressions

- Given the let expressions in Haskell

```
let x = e1 in e2
```

- Question: can we implement this construct in the lambda calculus?

Rule:

$$\boxed{\texttt{let f = }\lambda\texttt{x.M in N}}$$

$$\Downarrow$$

$$\boxed{(\lambda\texttt{f.N})(\lambda\texttt{x.M})}$$

•The let-expr is a kind of syntactic sugar

Encoding Booleans in LC

- We will represent "true" and "false" as *functions* named "true" and "false"
  - how do we define these functions?
  - think about how "true" and "false" can be used
  - they can be used by a testing:
    if b then x else y or as a function: if b x y

if *true* x y = x
if *false* x y = y

$\Rightarrow$

if = λtorf . λx. λy . torf x y

$\Downarrow$

true x y = x
false x y = y

Encoding Booleans

- the encoding:

true = λt. λf. t

false = λt. λf. f

if = λx. λthen. λelse. x then else

if *true* (λx.t1) (λx.t2)

= (λx. λthen. λelse. x then else) (λt. λf. t) (λx.t1) (λx.t2)

$\longrightarrow_{\beta}^{*}$ (λt. λf. t) (λx.t1) (λx.t2)

-->* λx.t1

$\longrightarrow_{\beta}^{*}$ Zero or more steps of beta reduction

Encoding Booleans

true = λt. λf. t false = λt. λf. f

and = λb. λc. b c false

and true true

-->* true true false

-->* true

and false true

-->* fals true false

-->* false

β omitted

Encoding Natural Numbers in Lambda Calculus

FLOLAC 2008

- A natural number is a *function* that given an operation *f* and a starting value *s*, applies f a number of times to s:

$$\left.\begin{array}{l} 0 =_{def} \lambda f.\ \lambda s.\ s \\ 1 =_{def} \lambda f.\ \lambda s.\ f\ s \\ 2 =_{def} \lambda f.\ \lambda s.\ f\ (f\ s) \\ \ldots \end{array}\right\}$$

Church numerals

$$\boxed{n\ =_{def}\ \lambda f.\lambda s.\ f^{n}\ s}$$

Computing with Natural Numbers

- The successor function

$$\mathit{succ}\ n =_{\mathrm{def}} \lambda f.\ \lambda s.\ f\ (n\ f\ s)$$

- Addition

$$\mathit{add}\ n_1\ n_2 =_{\mathrm{def}} n_1\ \mathrm{succ}\ n_2$$

- Multiplication

$$\mathit{mult}\ n_1\ n_2 =_{\mathrm{def}} n_1\ (\mathrm{add}\ n_2)\ 0$$

- Testing equality with 0

$$\mathit{iszero}\ n =_{\mathrm{def}} n\ (\lambda b.\ \mathrm{false})\ \mathrm{true}$$

Computing with Natural Numbers. Example

FLOLAC 2008

Given: succ n $=_{def}$ λf. λs. f (n f s)

0 $=_{def}$ λf. λs. s

1 $=_{def}$ λf. λs. f s

succ 0 =

(λn.λf. λs. f (n f s)) 0 =

($\lambda\underline{n}$.λf. λs. f ($\underline{n}$ f s)) (λf. λs. s) $\rightarrow$

(λf. λs. f (($\lambda\underline{f}$. λs. s) f s) $\rightarrow$

(λf. λs. f (($\lambda\underline{s}$. s) s) $\rightarrow$

λf. λs. f s = 1

Computing with Natural Numbers. Example

mult 2 2 $\rightarrow$

2 (add 2) 0 $\rightarrow$

(add 2) ((add 2) 0) $\rightarrow$

2 succ (add 2 0) $\rightarrow$

2 succ (2 succ 0) $\rightarrow$

succ (succ (succ (succ 0))) $\rightarrow$

succ (succ (succ (λf. λs. f (0 f s)))) $\rightarrow$

succ (succ (succ (λf. λs. f s))) $\rightarrow$

succ (succ (λg. λy. g ((λf. λs. f s) g y)))

succ (succ (λg. λy. g (g y))) $\rightarrow^*$ λg. λy. g (g (g (g y))) = 4

Encoding pairs

- would like to encode the operations
  - *mkPair e1 e2*
  - *fst p*
  - *snd p*
- pairs will be *functions*
  - when the function is used in the *fst* or *snd* operation it should reveal its first or second component respectively

Encoding Pairs

- A pair is a function that given a *Boolean* returns the left or the right element

$$\boxed{\begin{array}{ll} \text{mkpair } x\ y & =_{\text{def}} \lambda\, b.\ x\ y \\ \text{fst } p & =_{\text{def}} p \text{ true} \\ \text{snd } p & =_{\text{def}} p \text{ false} \end{array}}$$

- Example:

  fst (mkpair x y) $\rightarrow$ (mkpair x y) true $\rightarrow$ true x y $\rightarrow$ x

and we can go on...

- lists, trees and other datatypes
- recursion, ...
- ...
- the general trick:
  - values will be functions – construct these functions so that they return the appropriate information when called by an operation

•Lambda calculus with *predefined constants*

Recursion in the Lambda Calculus

Recursion in the LC

FLOLAC 2008

- The Y combinator

  `Y ≡ λf.(λx.f(x x)) (λx.f(x x))`

- **`Y`** has the property: for every function F,

  `Y F = F(Y F)`

- In other words, (Y F) is the fixed point of F
- We can use Y to implement recursion in the LC.

Solution

```
Y F
 ≡ (λf.(λx.f(x x)) (λx.f(x x))) F
```
$\rightarrow_\beta$ `(λx.F(x x)) (λx.F(x x))`

$\rightarrow_\beta$ `F ( (λx.F(x x)) (λx.F(x x)) )`

$\leftarrow_\beta$ `F ((λf.(λx.f(x x)) (λx.f(x x))) F)`

`≡ F (Y F)`

So, if we let `X ≡ Y F` then this tells us

`X = F X`

in other words, `X` is a fixed point of `F`.

Recursion

- Factorial in Haskell:

```
fact = \n -> if (n==0) then
                  1
               else
                  (n*(fact (n-1)))
```

  - **Ex. Write fact in λ-calculus by using the Y combinator.**

- Hint: consider the term
- **`F ≡ λf.λn.if (isZero n) 1 (n*f (pred n))`**
- Ex. Evaluate **`fact 0`**, **`fact 1`** and **`fact 2`**.

Solution

FLOLAC 2008

```
fact ≡ Y F
     ≡ Y ( λf.λn.if (isZero n) 1 (n*(f (pred
  n))) )
```

```
fact 2
  = Y F 2
  = F (Y F) 2
  = (λf.λn.if (isZero n) 1 (n*(f (pred n)))) (Y F) 2
  = (λn.if (isZero n) 1 (n*((Y F) (pred n)))) 2
  = if (isZero 2) 1 (2*((Y F) (pred 2)))
  = 2*(Y F (pred 2))
  = 2*(Y F 1)
  = 2*(fact 1)      and so on...
```

Appendix: Formal Treatment of Substitutions

Name Capturing

- $(\lambda x.\lambda y.x)y \rightarrow_\beta \lambda y.y$ X

- Replacing doesn't always work
- But if we α-convert first

  - $(\lambda x.\lambda y.x)y \equiv_\alpha (\lambda x.\lambda y'.x)y$
  - $\rightarrow_\beta \lambda y'.y$

- Now define substitution `M[N/x]` to do this

Substitution M[N/x]

- $x[N/x] \equiv$
- $y[N/x] \equiv$ $(y \neq x)$
- $(PQ)[N/x] \equiv$
- $(\lambda x.L)[N/x] \equiv$
- $(\lambda y.L)[N/x] \equiv$ $(y \neq x)$

- Hint: Take care with **$(\lambda$y.L$)$**. Consider the cases
  - $y \notin FV(L)$ **and** $y \notin FV(N)$ **and only rename y when necessary.**

Substitution M[N/x]

- We assume that $y \neq x$ throughout.
- The first three cases are easy.

- $x[N/x] \equiv N$
- $y[N/x] \equiv y$
- $(PQ)[N/x] \equiv P[x:=N]\ Q[x:=N]$

- In the next case the λx guarantees that x does not appear free in the term $(\lambda x.L)$, so there are no free occurences to substitute for.
- $(\lambda x.L)[N/x] \equiv \lambda x.L$

Substitution M[N/x]

FLOLAC 2008

- The final case is the tricky one.

- $(\lambda y.L)[N/x] \equiv \lambda y.L$, if $x \notin FV(L)$
- $\lambda y.L[N/x]$, if $y \notin FV(N)$
- $\lambda y'.L[y'/y'][N/x]$, otherwise
- where $y' \notin FV(L) \cup FV(N)$

- If $x \notin FV(L)$ then there are no x's to replace with
- N's, so the term stays the same. If $y \notin FV(N)$then there will be no y's accidentally captured by the λy so we can keep λy. But otherwise we must find a fresh variable y' and replace λy by $\lambda y'$.

Lambda Calculus with Constants and Types

Example: Extended LC

•Lambda calculus with *Booleans* and *natural numbers*

```
E  ::=  constants: 1, 2, 3, …
         succ, iszero
         true, false,
         &&(and), ||(or), !(not),
       | variable:  x, y, z, …
       | λx.E
       | E1 E2
       | if E1 then E2 else E3
```

Evaluation Rules for the Extended LC

- Based on β-reduction
- Extended to Booleans and numbers
- Reduced to values:
  - 0, 1, 2, …
  - true, false
  - λx.E
- *Values are normal forms*.

Some extended rules:

```
iszero 0          → true
iszero (succ n)   → false

pred 0            → 0
pred (succ n)     → n

if true then e1 else e2
                  → e1
if false then e1 else e2
                  → e2

        e1 → e2
  ---------------------
   succ e1 → succ e2
...
```

Evaluation Rules for the Extended LC …

- *Not all normal forms are values*
  - E.g., (x y)
- So, <u>reduction (evaluation) may **get stuck**</u>
  - Got a normal form, but *not a value.* For example:

```
(λx. succ x) true → succ true →??
```

Reproduce it in LC:

```
succ true = (λn.λf.λs.f (n f s))(λt.f.t)
           → λf.λs.f ((λt.f.t) f s)
           → λf.λs.f f  --Not a number!
```

Introducing Types

- Def: a term is **stuck** if it is in normal form and not a value
- Stuck terms model *runtime errors*
  - “succ true”
- It’s a kind of type error!
- A key goal of types and type systems will be to remove such runtime errors
  - **Int** = [0, 1, 2, …], *succ, pred*, …
  - **Bool** = [true, false], *and, or, not*
  - We cannot mix **Int** with **Bool** values arbitrarily.

Lambda Calculus with Constants and Types

Based on the Simply Typed Lambda Calculus (SLC)

Function Types

We introduce function types: A → B is the type of functions with a parameter of type A and a result of type B.

Types are defined by this grammar:

$$\boxed{\begin{aligned} T ::=\ & \mathrm{Int} \\ \mid\ & \mathrm{Bool} \\ \mid\ & T \to T \end{aligned}}$$

By convention, → associates to the *right*, so that A → B → C means A → (B → C).

Examples:	Int → Int → Int	curried function of two arguments
	(Int → Int) → Int	function which is given a function

Types and Type Errors

FLOLAC 2008

We type the **`succ`** function and Boolean value **`true`** as

```
succ : Int -> Int
true : Bool
```

```
   f : T1 -> T2
   e : T1
------------------
 f e : T2
```

Then

```
“succ  true”
```

is not acceptable!

We’ll introduce <u>typing rules</u> to filter out (type checking) such expressions.

Lambda Calculus with Types

FLOLAC 2008

To make it easier to define the typing rules, we will modify the syntax so that a λ-abstraction *explicitly specifies the type of its parameter.*

- **And more operators, such as '+', '==', '&&'**

values v ::= *integer literal*
| true | false
| λx**:T**.e

Type declaration for parameters

expressions e ::= v
| x
| e + e | e == e | e && e | if e then e else e
| e e

types Int
| Bool
| T → T

Examples of Expressions

```
2, true, x

x+20-y*5

(x>y) || (y>10 && z==1)

if x==2 then 10 else 20

succ (if x==2 then 10 else 20)

(if (x==0) then f else g) (y+5)
```

Examples of Functions

FLOLAC 2008

```
λx:Int.x+2

λb:Bool.λx:Int.if b then x else -x

λf:Int->Int.λx:Int.f (f x)

(λf:Int->Int.λx:Int.f (f x)) succ

λx:Int.λf:Int->Int.λg:Int->Int.
               if (x==0) then f else g
```

Type Checking for Function Application

- In function application, the type of the argument must be the same with that of the parameter.

```
    e1 : T1 -> T2         (premises, or
    e2 : T1                assumptions)
--------------------
 e1 e2 : T2               (conclusion)
```

```
(λf:Int->Int.λx:Int.f (f x)): (Int->Int)->Int
                        succ: Int->Int
----------------------------------------------
 (λf:Int->Int.λx:Int.f (f x)) succ : Int
```

Determining the Type of an Expression

Type Checking: Does *e* has a type τ?

- τ is a meta-variable representing a type

```
τ ::= Int
    | Bool
    | τ₁ → τ₂
```

Type Judgments

- A *type judgment* has the form

$\boxed{\Gamma \vdash \text{exp} : \tau}$ "exp has type τ under TE Γ"

- Γ is a typing environment
  - Supplies the types of variables and functions
  - Γ is a list of the form [x : τ, . . .]
- exp is a program expression
- τ is a *type* to be assigned to exp
- |- pronounced "turnstyle", or "entails" (or "satisfies")

Example Valid Type Judgments

FLOLAC 2008

- [ ] |- true or false : Bool
- [x : Int] |- x + 3 : Int
- [p : Int -> String] |- (p 5) : String

•Type judgments are derived via *typing rules.*

Format of Typing Rules

FLOLAC 2008

Assumptions:

$$\frac{\Gamma_1 \mid\!- \text{exp}_1 : \tau_1 \quad . \ . \ . \quad \Gamma_n \mid\!- \text{exp}_n : \tau_n}{\Gamma \mid\!- \text{exp} : \tau}$$

Conclusion:

- Idea: Type of expression determined by type of its *syntactic components*
- Rule without assumptions is called an *axiom*
- Γ may be omitted when not needed

Axioms - Constants

$$\frac{}{\vdash n : \mathrm{Int}} \quad \text{(assuming } n \text{ is an integer constant)}$$

$$\frac{}{\vdash \mathrm{true} : \mathrm{Bool}} \qquad \frac{}{\vdash \mathrm{false} : \mathrm{Bool}}$$

- These rules are true with any typing environment
- *n* is a meta-variable

Typing Environment

- *A typing environment* Γ keeps track of the types of *free identifiers* occurred in expressions

  Γ = **`[…, x:Int, f:Int->Int, …]`**

- We view a TE as a finite fun from *identifiers* to types

  Γ : Ide → Type

  So, given Γ as above, Γ(x) = **`Int`**

- No *multiple* bindings for any id:

  Γ′ = **`[…, `*`x:Int`*`, f:Int->Int, x:Bool, …]`** ✗

Axioms - Variables

- Typing rule for variables: (Var)

$$\frac{}{\Gamma \;\texttt{|-}\; \mathtt{x} : \tau} \quad \text{if } \Gamma(\mathtt{x}) = \tau$$

- We can also include the types for pre-defined identifiers (functions) in Γ. For example:
  - Γ **`= […, succ:Int->Int, …]`**

Simple Rules - Arithmetic

Primitive operators ($\oplus \in$ { +, -, *, …}):

$$\frac{\Gamma \mid\!\!- \; e_1 : \text{Int} \qquad \Gamma \mid\!\!- \; e_2 : \text{Int}}{\Gamma \mid\!\!- \; e_1 \oplus e_2 : \text{Int}}$$

Relations (~ $\in$ { < , > , =, <=, >= }):

$$\frac{\Gamma \mid\!\!- \; e_1 : \text{Int} \qquad \Gamma \mid\!\!- \; e_2 : \text{Int}}{\Gamma \mid\!\!- \; e_1 \sim e_2 : \text{Bool}}$$

Simple Rules - Booleans

Logical Connectives:

$$\frac{\Gamma \mid\!- \; e_1 : \text{Bool} \qquad \Gamma \mid\!- \; e_2 : \text{Bool}}{\Gamma \mid\!- \; e_1 \mathbin{\&\&} e_2 : \text{Bool}}$$

$$\frac{\Gamma \mid\!- \; e_1 : \text{Bool} \qquad \Gamma \mid\!- \; e_2 : \text{Bool}}{\Gamma \mid\!- \; e_1 \mathbin{||} e_2 : \text{Bool}}$$

Simple Example

- Let Γ = [x:Int ; y:Bool]
- Show Γ |- y || (x + 3 > 6) : Bool
- Start building the proof tree from the *bottom up*

$$\frac{?}{\Gamma \text{ |- } y \text{ || } (x + 3 > 6) : \text{Bool}}$$

Simple Example

FLOLAC 2008

- Let Γ = [x:Int ; y:Bool]
- Show Γ |- y || (x + 3 > 6) : Bool
- Which rule has this as a conclusion?

$$\frac{?}{\Gamma \mid\!- \; y \;||\; (x + 3 > 6) : \text{Bool}}$$

Simple Example

- Let Γ = [x:Int ; y:Bool]
- Show Γ |- y || (x + 3 > 6) : Bool
- *Booleans:* ||

$$\frac{\Gamma \text{ |- } y : \text{Bool} \qquad \Gamma \text{ |- } x + 3 > 6 : \text{Bool}}{\Gamma \text{ |- } y \text{ || } (x + 3 > 6) : \text{Bool}}$$

Simple Example

- Let Γ = [x:Int ; y:Bool]
- Show Γ |- y || (x + 3 > 6) : Bool
- Pick an assumption to prove

$$\dfrac{\dfrac{?}{\Gamma \vdash y : \mathrm{Bool}} \qquad \Gamma \vdash x + 3 > 6 : \mathrm{Bool}}{\Gamma \vdash y \mathbin{||} (x + 3 > 6) : \mathrm{Bool}}$$

Simple Example

- Let Γ = [x:Int ; y:Bool]
- Show Γ |- y || (x + 3 > 6) : Bool
- Which rule has this as a conclusion?

$$\dfrac{\dfrac{?}{\Gamma \text{ |- } y : \text{Bool}} \qquad \Gamma \text{ |- } x + 3 > 6 : \text{Bool}}{\Gamma \text{ |- } y \text{ || } (x + 3 > 6) : \text{Bool}}$$

Simple Example

- Let Γ = [x:Int ; y:Bool]
- Show Γ |- y || (x + 3 > 6) : Bool
- *Axiom for variables*

$$\dfrac{\dfrac{}{\Gamma \mathrel{|\!-} y : \mathrm{Bool}} \qquad \Gamma \mathrel{|\!-} x + 3 > 6 : \mathrm{Bool}}{\Gamma \mathrel{|\!-} y \mathbin{||} (x + 3 > 6) : \mathrm{Bool}}$$

Simple Example

- Let Γ = [x:Int ; y:Bool]
- Show Γ |- y || (x + 3 > 6) : Bool
- Pick an assumption to prove

$$\dfrac{\dfrac{}{\Gamma \mid\!- \; y : \text{Bool}} \qquad \dfrac{?}{\Gamma \mid\!- \; x + 3 > 6 : \text{Bool}}}{\Gamma \mid\!- \; y \mathbin{||} (x + 3 > 6) : \text{Bool}}$$

Simple Example

- Let Γ = [x:Int ; y:Bool]
- Show Γ |- y || (x + 3 > 6) : Bool
- Which rule has this as a conclusion?

$$\dfrac{\dfrac{}{\Gamma \;|\text{-}\; y : \text{Bool}} \qquad \dfrac{?}{\Gamma \;|\text{-}\; x + 3 > 6 : \text{Bool}}}{\Gamma \;|\text{-}\; y \;||\; (x + 3 > 6) : \text{Bool}}$$

Simple Example

- Let Γ = [x:Int ; y:Bool]
- Show Γ |- y || (x + 3 > 6) : Bool
- *Arithmetic relations*

$$\dfrac{\dfrac{}{\Gamma \vdash y : \text{Bool}} \qquad \dfrac{\Gamma \vdash x + 3 : \text{Int} \qquad \Gamma \vdash 6 : \text{Int}}{\Gamma \vdash x + 3 > 6 : \text{Bool}}}{\Gamma \vdash y \mid\mid (x + 3 > 6) : \text{Bool}}$$

Simple Example

- Let Γ = [x:Int ; y:Bool]
- Show Γ |- y || (x + 3 > 6) : Bool
- Pick an assumption to prove

$$\dfrac{\dfrac{}{\Gamma \mid\!- \; y : \text{Bool}} \qquad \dfrac{\Gamma \mid\!- \; x + 3 : \text{Int} \qquad \dfrac{?}{\Gamma \mid\!- \; 6 : \text{Int}}}{\Gamma \mid\!- \; x + 3 > 6 : \text{Bool}}}{\Gamma \mid\!- \; y \;||\; (x + 3 > 6) : \text{Bool}}$$

Simple Example

- Let Γ = [x:Int ; y:Bool]
- Show Γ |- y || (x + 3 > 6) : Bool
- Which rule has this as a conclusion?

$$\frac{\dfrac{}{\Gamma \mid\!- \; y : \mathrm{Bool}} \qquad \dfrac{\Gamma \mid\!- \; x + 3 : \mathrm{Int} \qquad \dfrac{?}{\Gamma \mid\!- \; 6 : \mathrm{Int}}}{\Gamma \mid\!- \; x + 3 > 6 : \mathrm{Bool}}}{\Gamma \mid\!- \; y \mid\mid (x + 3 > 6) : \mathrm{Bool}}$$

Simple Example

- Let Γ = [x:Int ; y:Bool]
- Show Γ |- y || (x + 3 > 6) : Bool
- *Axiom for constants*

$$\dfrac{\dfrac{}{\Gamma \mid\!- y : \mathrm{Bool}} \qquad \dfrac{\Gamma \mid\!- x + 3 : \mathrm{Int} \qquad \dfrac{}{\Gamma \mid\!- 6 : \mathrm{Int}}}{\Gamma \mid\!- x + 3 > 6 : \mathrm{Bool}}}{\Gamma \mid\!- y \mid\mid (x + 3 > 6) : \mathrm{Bool}}$$

Simple Example

- Let Γ = [x:Int ; y:Bool]
- Show Γ |- y || (x + 3 > 6) : Bool
- Pick an assumption to prove

$$\dfrac{\dfrac{}{\Gamma \mid\!- \ y : \mathrm{Bool}} \qquad \dfrac{\dfrac{?}{\Gamma \mid\!- \ x + 3 : \mathrm{Int}} \quad \dfrac{}{\Gamma \mid\!- \ 6 : \mathrm{Int}}}{\Gamma \mid\!- \ x + 3 > 6 : \mathrm{Bool}}}{\Gamma \mid\!- \ y \mathbin{||} (x + 3 > 6) : \mathrm{Bool}}$$

Simple Example

- Let Γ = [x:Int ; y:Bool]
- Show Γ |- y || (x + 3 > 6) : Bool
- Which rule has this as a conclusion?

$$\dfrac{\dfrac{}{\Gamma \mid\!\!- \; y : \text{Bool}} \qquad \dfrac{\dfrac{?}{\Gamma \mid\!\!- \; x + 3 : \text{Int}} \quad \dfrac{}{\Gamma \mid\!\!- \; 6 : \text{Int}}}{\Gamma \mid\!\!- \; x + 3 > 6 : \text{Bool}}}{\Gamma \mid\!\!- \; y \mid\mid (x + 3 > 6) : \text{Bool}}$$

Simple Example

- Let Γ = [x:Int ; y:Bool]
- Show Γ |- y || (x + 3 > 6) : Bool
- *Arithmetic operations*

$$\dfrac{\dfrac{}{\Gamma \vdash y : \mathrm{Bool}} \qquad \dfrac{\dfrac{\Gamma \vdash x : \mathrm{Int} \quad \Gamma \vdash 3 : \mathrm{Int}}{\Gamma \vdash x + 3 : \mathrm{Int}} \quad \dfrac{}{\Gamma \vdash 6 : \mathrm{Int}}}{\Gamma \vdash x + 3 > 6 : \mathrm{Bool}}}{\Gamma \vdash y \,||\, (x + 3 > 6) : \mathrm{Bool}}$$

Simple Example

- Let Γ = [x:Int ; y:Bool]
- Show Γ |- y || (x + 3 > 6) : Bool
- Pick an assumption to prove

$$\dfrac{\dfrac{}{\Gamma \text{ |- } y : \text{Bool}} \qquad \dfrac{\dfrac{\Gamma \text{ |- } x : \text{Int} \quad \dfrac{?}{\Gamma \text{ |- } 3 : \text{Int}}}{\Gamma \text{ |- } x + 3 : \text{Int}} \quad \dfrac{}{\Gamma \text{ |- } 6 : \text{Int}}}{\Gamma \text{ |- } x + 3 > 6 : \text{Bool}}}{\Gamma \text{ |- } y \mid\mid (x + 3 > 6) : \text{Bool}}$$

Simple Example

- Let Γ = [x:Int ; y:Bool]
- Show Γ |- y || (x + 3 > 6) : Bool
- Which rule has this as a conclusion?

$$\dfrac{\dfrac{}{\Gamma \mid\!- \; y : \mathrm{Bool}} \qquad \dfrac{\dfrac{\Gamma \mid\!- \; x : \mathrm{Int} \quad \dfrac{?}{\Gamma \mid\!- \; 3 : \mathrm{Int}}}{\Gamma \mid\!- \; x + 3 : \mathrm{Int}} \quad \dfrac{}{\Gamma \mid\!- \; 6 : \mathrm{Int}}}{\Gamma \mid\!- \; x + 3 > 6 : \mathrm{Bool}}}{\Gamma \mid\!- \; y \,\|\, (x + 3 > 6) : \mathrm{Bool}}$$

Simple Example

- Let Γ = [x:Int ; y:Bool]
- Show Γ |- y || (x + 3 > 6) : Bool
- *Axiom for constants*

$$\dfrac{\dfrac{}{\Gamma \mathrel{|\!\!-} y : \mathrm{Bool}} \qquad \dfrac{\dfrac{\Gamma \mathrel{|\!\!-} x : \mathrm{Int} \quad \dfrac{}{\Gamma \mathrel{|\!\!-} 3 : \mathrm{Int}}}{\Gamma \mathrel{|\!\!-} x + 3 : \mathrm{Int}} \quad \dfrac{}{\Gamma \mathrel{|\!\!-} 6 : \mathrm{Int}}}{\Gamma \mathrel{|\!\!-} x + 3 > 6 : \mathrm{Bool}}}{\Gamma \mathrel{|\!\!-} y \mathbin{||} (x + 3 > 6) : \mathrm{Bool}}$$

Simple Example

- Let Γ = [x:Int ; y:Bool]
- Show Γ |- y || (x + 3 > 6) : Bool
- Pick an assumption to prove

$$\dfrac{\dfrac{}{\Gamma \mathrel{|-} y : \mathrm{Bool}} \qquad \dfrac{\dfrac{\dfrac{?}{\Gamma \mathrel{|-} x : \mathrm{Int}} \quad \dfrac{}{\Gamma \mathrel{|-} 3 : \mathrm{Int}}}{\Gamma \mathrel{|-} x + 3 : \mathrm{Int}} \quad \dfrac{}{\Gamma \mathrel{|-} 6 : \mathrm{Int}}}{\Gamma \mathrel{|-} x + 3 > 6 : \mathrm{Bool}}}{\Gamma \mathrel{|-} y \mathbin{||} (x + 3 > 6) : \mathrm{Bool}}$$

Simple Example

- Let Γ = [x:Int ; y:Bool]
- Show Γ |- y || (x + 3 > 6) : Bool
- Which rule has this as a conclusion?

$$\frac{\dfrac{}{\Gamma \vdash y : \mathrm{Bool}} \qquad \dfrac{\dfrac{\dfrac{?}{\Gamma \vdash x : \mathrm{Int}} \quad \dfrac{}{\Gamma \vdash 3 : \mathrm{Int}}}{\Gamma \vdash x + 3 : \mathrm{Int}} \quad \dfrac{}{\Gamma \vdash 6 : \mathrm{Int}}}{\Gamma \vdash x + 3 > 6 : \mathrm{Bool}}}{\Gamma \vdash y \mathbin{||} (x + 3 > 6) : \mathrm{Bool}}$$

Simple Example

- Let Γ = [x:Int ; y:Bool]
- Show Γ |- y || (x + 3 > 6) : Bool
- *Axiom for variables*

$$\frac{\dfrac{}{\Gamma \text{ |- } y : \text{Bool}} \qquad \dfrac{\dfrac{\dfrac{}{\Gamma \text{ |- } x : \text{Int}} \quad \dfrac{}{\Gamma \text{ |- } 3 : \text{int}}}{\Gamma \text{ |- } x + 3 : \text{Int}} \quad \dfrac{}{\Gamma \text{ |- } 6 : \text{Int}}}{\Gamma \text{ |- } x + 3 > 6 : \text{Bool}}}{\Gamma \text{ |- } y \text{ || } (x + 3 > 6) : \text{Bool}}$$

Simple Example

- Let Γ = [x:Int ; y:Bool]
- Show Γ |- y || (x + 3 > 6) : Bool
- No more assumptions! DONE!

$$\dfrac{\dfrac{}{\Gamma \;|\!-\; y : \mathrm{Bool}} \qquad \dfrac{\dfrac{\dfrac{}{\Gamma \;|\!-\; x : \mathrm{Int}} \quad \dfrac{}{\Gamma \;|\!-\; 3 : \mathrm{Int}}}{\Gamma \;|\!-\; x + 3 : \mathrm{Int}} \quad \dfrac{}{\Gamma \;|\!-\; 6 : \mathrm{Int}}}{\Gamma \;|\!-\; x + 3 > 6 : \mathrm{Bool}}}{\Gamma \;|\!-\; y \;||\; (x + 3 > 6) : \mathrm{Bool}}$$

If-Expressions

- If_then_else rule:

$$\boxed{\frac{\Gamma \mid\!\!- e_1 : \mathrm{Bool} \quad \Gamma \mid\!\!- e_2 : \tau \quad \Gamma \mid\!\!- e_3 : \tau}{\Gamma \mid\!\!- (\text{if } e_1 \text{ then } e_2 \text{ else } e_3) : \tau}}$$

- τ is a type variable (meta-variable)
  – it can take any type at all
  – All instances in a rule application *must get same type*

- I.e., the Then branch, Else branch and if_then_else must all have same type

Examples of IF

FLOLAC 2008

```
if x==2 then 10 else 20
```
✓

```
if x==2 then 10 else false
```
✗

Function Application

- Application rule: (App)

$$\boxed{\frac{\Gamma \mid\!\!- \; e_1 : \tau_1 \to \tau_2 \qquad \Gamma \mid\!\!- \; e_2 : \tau_1}{\Gamma \mid\!\!- \; (e_1 \; e_2) : \tau_2}}$$

- If you have a function *expression* e_1 of type $\tau_1 \to \tau_2$ applied to an argument of type τ_1, the resulting expression has type τ_2

Application Examples

FLOLAC 2008

$$\frac{\Gamma \;|\!- (\lambda \mathtt{f:Int\text{->}Int}.\lambda \mathtt{x:Int.f\ (f\ x)}) : (\mathrm{Int\text{->}Int})\text{->}\mathrm{Int}\text{->}\mathrm{Int} \qquad \Gamma \;|\!- \mathit{succ} : \mathrm{Int\text{->}Int}}{\Gamma \;|\!- (\lambda \mathtt{f:Int\text{->}Int}.\lambda \mathtt{x:Int.f\ (f\ x)})\ \mathit{succ} : \mathrm{Int\text{->}Int}}$$

$$\frac{[\mathrm{f:Int\text{->}Int,\ g:Int\text{->}Int,\ b:Bool}] \;|\!- \mathtt{if\ b\ then\ f\ else\ g} : \mathrm{Int\text{->}Int}}{[\mathrm{f:Int\text{->}Int,\ g:Int\text{->}Int,\ b:Bool}] \;|\!- (\mathtt{if\ b\ then\ f\ else\ g})\ \mathit{5} : \mathrm{Int}}$$

Function Rule

- Rules describe types, but also how the environment Γ may change
- λ-fun rule: (Abs)

$$\boxed{\dfrac{[x : \tau_1] \cup \Gamma \mid\!- \; e : \tau_2}{\Gamma \mid\!- \; \lambda x.e : \tau_1 \rightarrow \tau_2}}$$

We often write $\boxed{\Gamma\mathtt{.x:T} = \Gamma \cup \mathtt{[x:T]}}$ --extends Γ

•If **x** $\in$ dom(Γ), then Γ**.x:T** means that the new binding of x will replace the original one.

Function Example

FLOLAC 2008

$$\frac{[y : \text{int}\,] \cup \Gamma \mid\!- \; y + 3 : \text{int}}{\Gamma \mid\!- \; \lambda y.y + 3 : \text{int} \rightarrow \text{int}}$$

```
[succ:Int->Int].x:Int  |- succ: Int->Int
-----------------------------------------
[succ:Int->Int ].x:Int |- x: Int
-----------------------------------------App
[succ:Int->Int ].x:Int |- (succ x) : Int
-----------------------------------------
[succ:Int->Int]        |- λx.(succ x) : In->Int
```

Anther Fun Example

```
Γ |- λf:Int->Int.λx:Int.f (f x)): ?
```

•Move f and x to Γ

```
Γ.f:Int->Int.x:Int |- f:Int->Int (Var)
Γ.f:Int->Int.x:Int |- x:Int (Var)
—————————————————————————————————————————— (App)
Γ.f:Int->Int.x:Int |- f x: Int
     Γ.f:Int->Int.x:Int |- f:Int->Int
—————————————————————————————————————————————————— (App)
     Γ.f:Int->Int.x:Int |- f (f x)): Int
————————————————————————————————————————————————————————— (Abs)
        Γ.f:Int->Int |- λx:Int.f (f x)): Int->Int
—————————————————————————————————————————————————————— (Abs)
       Γ |- λf:Int->Int.λx:Int.f (f x)):
                                (Int->Int)->Int->Int
```

Typing Rules for the LC with Constants & Types

Γ `|- i : Int` if *i* is an integer literal

Γ `|- true : Bool` $\quad$ Γ `|- false : Bool`

$$\frac{x{:}T \in \Gamma}{\Gamma \;|\text{--}\; x : T}$$

Γ `|- x:T` **type judgement**

$$\frac{\Gamma|\text{-}\ E1{:}Int \quad \Gamma|\text{-}\ E2{:}Int}{\Gamma|\text{-}\ E1 + E2 : Int} \qquad \frac{\Gamma|\text{-}\ E1{:}Bool \quad \Gamma|\text{-}\ E2{:}Bool}{\Gamma|\text{-}\ E1 \,\&\&\, E2 : Bool}$$

$$\frac{\Gamma|\text{-}\ E1{:}Int \quad \Gamma|\text{-}\ E2{:}Int}{\Gamma|\text{-}\ E1 == E2 : Bool} \qquad \frac{\Gamma|\text{-}E1{:}Bool \quad \Gamma|\text{-}E2{:}T \quad \Gamma|\text{-}E3{:}T}{\Gamma|\text{-}\ if\ E1\ the\ E2\ else\ E3\ {:}T}$$

$$\frac{\Gamma.x{:}T1\ |\text{-}\ E{:}\ T2}{\Gamma|\text{-}\ \lambda x{:}T1.E : T1\text{->}T2} \qquad \frac{\Gamma|\text{-}\ E1{:}T1\text{->}T2 \quad \Gamma|\text{-}\ E2{:}T1}{\Gamma|\text{-}\ E1\ E2 : T2}$$

Typing Built-in Operators/Fun

- Alternative: treat built-in operators like literal constants, and include their types in Γ

```
Γ |- &&   : Bool->Bool->Bool

Γ |- +    : Int->Int->Int
Γ |- succ : Int->Int
   . . .                     . . .
```

- Then, no need to have special rules for them

Type Safety

- Well-typed programs won’t get stuck!

- Theorem: If *e* is a closed expression of type T (|- e : T), then for all e' such that e ->* e', it is the case that either

  (A) e' is a *value* (say, v') and |- v' : t, or

  (B) exists e'' such that e' -> e''.

If |- e_0: T, **then** e_0 -> e_1 ->e_2 -> … -> v

The Simply Typed Lambda Calculus $\lambda^{\rightarrow}$

FLOLAC 2008

- The extended lambda calculus is based on the simply typed lambda calculus.
- The SLC was originally introduced by Alonzo Church in 1940 as an attempt to avoid paradoxical uses of the untyped lambda calculus.
- In the SLC, β-reduction is Strong normalizing: all terms will be evaluated to a normal form.

Limitations of the SLC

- Types are monomorphic.

```
|-- λx:Int.x+1 : Int->Int
```
is OK

- But what is the type for the *identity* function?

```
|-- λx:?. x : ?
```

```
|-- λx:Int. x : Int->Int?
|-- λx:Bool. x : Bool->Bool?
|-- λx:Int->Int. x : (Int->Int)->(Int->Int)?
...
```

Parametric Polymorphism

- Polymorphism: allow many types for a value (hence also for variable, expression)
- Introducing *type variables* and ∀ quantification to express parametric polymorphism.

•Let α be a type variables representing any types. We can type the *id* function as follows.

$$\boxed{\vdash \lambda x{:}\alpha.x : \forall\alpha.\alpha \rightarrow \alpha}$$

Parametric Polymorphism…

Polymorphic type: $\forall\alpha.\alpha$ `->` α

The α can be instantiated to any types:

```
Int -> Int

Bool -> Bool

(Int->Int)->(Int->Int)

...
```

The Polymorphic Lambda Calculus (PLC)

A.K.A

- Second-Order Lambda Calculus
- System F

Motivating PLC

- Like SLC, use explicit typing for fun parameters
  - λx:T. E
- Extend types with generic *type variables* and *quantification*
  - $\forall\alpha.\alpha$ **->** α
- Enhance *terms* with types
  - Type generalization: $\Lambda\alpha.\lambda$x:α.E , a polymorphic term
  - Type application: ($\Lambda\alpha$. λx:α. E) (Int->Int)
    - Replace α with Int->Int

Types of the PLC

Syntax:

Types

τ ::=	**T**	type constannts, (Int, Bool,…)
|	α	type variables
|	$\tau \rightarrow \tau$	function types
|	$\forall\alpha.\tau$	*polymorphic types*

Examples:

`Int, Int->Bool, Int->Int->Bool, …`

$\alpha \rightarrow \beta$ $\forall\alpha.\alpha\text{->}\alpha$

$\forall\alpha.\alpha \rightarrow \forall\beta.\beta$ $\forall\alpha.\forall\beta.(\alpha \rightarrow \beta) \rightarrow \forall\gamma.\gamma$

Terms of the PLC

Terms

$M ::=$	**c**	constants
|	**x**	**v**ariables
|	λ**x:**τ. M	function
|	M M	function application
|	$\Lambda\alpha$(M)	*type generalization* $\quad \boxed{\cong \Lambda\alpha\texttt{.M}}$
|	*M* τ	*type application*

Examples:

Id = $\Lambda\alpha(\lambda x{:}\alpha.x)$ --type generalization (abstraction)

($\Lambda\alpha$.λx:α.x)(Int->Int) --type application (specialization)

Functions on Types

- In PLC, $\Lambda\alpha$ (M) is an anonymous notation for the function F mapping each type τ to the value of M[τ/ α].
- I.e., computation in PLC involves β-*reduction* for such functions on types.

$$\boxed{(\Lambda\alpha(M))\ \tau \rightarrow \quad M[\tau/\alpha]}$$

e.g., ($\Lambda\alpha(\lambda x{:}\alpha.x)$) (Int->Int) $\rightarrow$ λx:Int->Int.x

as well as the usual form of β-reduction from λ -calculus

$$\boxed{(\lambda x{:}\tau.M1)\ M2 \quad \rightarrow \quad M1[M2/x]}$$

Reduction in the PLC

In summary, we apply *substitution* on terms as well as types explicitly.

$$\boxed{\begin{aligned}(\lambda\, x : \tau\,(M_1))\, M_2 &\to M_1[M_2/x]\\ (\Lambda\,\alpha\,(M))\,\tau &\to M[\tau/\alpha].\end{aligned}}$$

PLC vs. SLC

In this system of PLC:

- Two new kinds of terms (expressions):
  - $\Lambda\alpha$ (M) (typically, α is used in M)
  - Application with *type* operand: M τ (τ a type)
- The first kind of expression is also a value
- To the type language we add:
  - Type variables – α
  - Universal types of the form $\forall$

Polymorphism in PLC, 1

Example: the identity function

$$\boxed{\text{Id} = \Lambda\alpha\,(\lambda x{:}\alpha.x) \quad \text{has type} \quad \forall\alpha.\alpha\text{->}\alpha}$$

We can apply Id to many kinds of arguments:

- Id Int 5 = $\Lambda\alpha\,(\lambda x{:}\alpha.x)$ Int 5 → $(\lambda x{:}\text{Int}.x)$ 5→ 5
- Id Bool true = $\Lambda\alpha\,(\lambda x{:}\alpha.x)$ Bool true →* true

Polymorphism in PLC, 2

FLOLAC 2008

Example: applying a function twice

$$\boxed{\textit{twice} = \Lambda\alpha\ (\lambda f{:}\alpha\to\alpha.\ \lambda x{:}\alpha.\ f\ (f\ x)))}$$

has type $\forall\alpha.\ (\alpha\to\alpha)\to\alpha\to\alpha$

and can be applied to arguments of different types:

a) twice Int (λx:Int.x+2) 5 --[Int/α]

→ (λf:Int->Int.λx:Int.f (f x)) (λx:Int.x+2) 5

→ ((λx:int. x+2) ((λx:int. x+2) 5))

→* 9

b) twice Bool (λx:Bool. x) false →* false

Polymorphism in PLC, 3

FLOLAC 2008

- Polymorphic function parameters
- Consider the following function application in LC:

$(\lambda f. (f\ 5, f\ \mathit{True}))\ \ (\lambda x.x)$ --(,) is a pair

Here the function parameter *f* is applied to two types of arguments: *Int* and *Bool*

In PLC, $(\lambda x.x)$ is $\Lambda\alpha.\lambda x{:}\alpha.x$ with type $\forall\alpha.\alpha\text{->}\alpha$
so we let f has the polymorphic type: $\lambda f{:}\forall\alpha.\alpha\text{->}\alpha$
And rewrite the above example as:

$(\lambda f{:}\forall\alpha.\alpha\text{->}\alpha.(f\ \mathit{Int}\ 5, f\ \mathit{Bool}\ \mathrm{True}))\ \ (\Lambda\alpha.\lambda x{:}\alpha.x)$

Polymorphism in PLC, 3

- Polymorphic function parameters
- Consider the following function application in LC:

$(\lambda f.\ (f\ 5,\ f\ \mathrm{True}))\ \ (\lambda x.x)$ --(,) is a pair

---

Write it in the PLC:

$(\lambda f{:}\forall\alpha.\alpha\text{->}\alpha.(f\ \mathit{Int}\ 5,\ f\ \mathit{Bool}\ \mathrm{True}))\ \ (\Lambda\alpha.\lambda x{:}\alpha.x)$

$\rightarrow((\Lambda\alpha(\lambda x{:}\alpha.x))\ \mathit{Int}\ 5,\ (\Lambda\alpha(\lambda x{:}\alpha.x))\ \mathit{Bool}\ \mathrm{true})$

$\rightarrow\ldots\rightarrow(5,\ \mathrm{true})$

Polymorphism in PLC, 4

Re-visit the identity function

$\boxed{\text{Id} = \Lambda\alpha\ (\lambda x{:}\alpha.x) \quad \text{has type} \quad \forall\alpha.\alpha\text{->}\alpha}$

We can apply *Id* to *Id* in a similar way*:*

$> (\text{Id}\ \ (\forall\alpha.\alpha\text{->}\alpha))\ \ \text{Id} = (\Lambda\alpha(\lambda x{:}\alpha.x)\ (\forall\alpha.\alpha\text{->}\alpha))\ \ (\Lambda\alpha(\lambda x{:}\alpha.x))$

$\rightarrow (\lambda x{:}\forall\alpha.\alpha\text{->}\alpha.x)\ \ (\Lambda\alpha(\lambda x{:}\alpha.x))$

$\rightarrow \Lambda\alpha(\lambda x{:}\alpha.x)\ \ = \text{Id}$

$\boxed{\text{has type}\ \ \forall\alpha.\alpha\text{->}\alpha}$

Formal Typing Rules of PLC

Syntax of PLC

Types

τ ::=	**T**	type constannts, (Int, Bool,…)
|	α	type variables
|	$\tau \to \tau$	function types
|	$\forall\alpha.\tau$	*polymorphic types*

Terms

M ::=	**c**	constants
|	**x**	**v**ariables
|	$\lambda x:\tau.\,M$	function
|	M M	function application
|	$\Lambda\alpha\,.M$	*type generalization*
|	$M\ \tau$	*type application*

Generic (Bound) vs. Free Type Variables

FLOLAC 2008

$\tau = \forall\alpha.\alpha \rightarrow \forall\beta.\beta$

`ftv(`τ`) = []`

$\tau = \forall\alpha.\alpha \rightarrow \beta$

`ftv(`τ`) = [`β`]`

- Free type variables stand for *some* types;
- Generic type variables stand for *any* types.

Type Judgements of PLC

takes the form $\boxed{\Gamma \vdash M : \tau}$ where

- the *typing environment* Γ is a finite function from variables to PLC types.
  (We write $\Gamma = \{x_1 : \tau_1, \ldots, x_n : \tau_n\}$ to indicate that Γ has domain of definition $dom(\Gamma) = \{x_1, \ldots, x_n\}$ and maps each x_i to the PLC type τ_i for $i = 1..n$.)
- M is a PLC expression
- τ is a PLC type.

•ftv(Γ) = $\cup$ ftv(τ_i)

Source: Prof. A. Pitts

PLC Typing Rules

(var) $\Gamma \mid\!- x : \tau$ if $x{:}\tau \in \Gamma$

(fn)
$$\frac{\Gamma.x{:}\tau_1 \mid\!- M : \tau_2}{\Gamma \mid\!- \lambda x : \tau_1 . M : \tau_1 \text{->} \tau_2}$$

(app)
$$\frac{\Gamma \mid\!- M_1 : \tau_1 \text{->} \tau_2 \quad \Gamma \mid\!- M_2 : \tau_1}{\Gamma \mid\!- M_1\ M_2 : \tau_2}$$

(gen)
$$\frac{\Gamma \mid\!- M : \tau}{\Gamma \mid\!- \Lambda\alpha . M : \forall\alpha.\tau} \quad \text{If } \alpha \notin \text{ftv}(\Gamma)$$

(ty_app)
$$\frac{\Gamma \mid\!- M : \forall\alpha.\tau_1}{\Gamma \mid\!- M\ \tau_2 : \tau 2[\tau_1/\alpha]}$$

The Side-Condition in Gen

$$\cfrac{\cfrac{\cfrac{}{x_1 : \alpha, x_2 : \alpha \vdash x_2 : \alpha}\ (\mathrm{var})}{x_1 : \alpha \vdash \lambda\, x_2 : \alpha\,(x_2) : \alpha \to \alpha}\ (\mathrm{fn})}{x_1 : \alpha \vdash \Lambda\, \alpha\,(\lambda\, x_2 : \alpha\,(x_2)) : \forall\, \alpha\,(\alpha \to \alpha)}\ (\mathbf{wrong!})$$

If $\alpha \notin$ ftv(Γ)

$$\cfrac{\cfrac{\cfrac{}{x_1 : \alpha, x_2 : \alpha' \vdash x_2 : \alpha'}\ (\mathrm{var})}{x_1 : \alpha \vdash \lambda\, x_2 : \alpha'\,(x_2) : \alpha' \to \alpha'}\ (\mathrm{fn})}{x_1 : \alpha \vdash \Lambda\, \alpha'\,(\lambda\, x_2 : \alpha'\,(x_2)) : \forall\, \alpha'\,(\alpha' \to \alpha')}\ (\mathrm{gen})$$

PLC Typing Exercise

twice = $\Lambda\alpha.\lambda f{:}\alpha\to\alpha.\lambda x{:}\alpha\ f\ (f\ x))$

Type Inference (Type Reconstruction)

- Languages like Haskell differ somewhat from the pure polymorphic lambda calculus.
  - No type annotation for fun parameters
  - No need to declare types and put in the "$\forall$"
  - Not required to put in explicit type abstractions (Λ) or type specialization (applications).
- Instead, the compiler figures those out for you through the process of *type inference*.
  - Γ |-- E : τ where E has *no type annotation* at all

Type Reconstruction

- We can define a function *erase* on well-typed expressions, that removes all type-related information :

```
erase(λx:τ.M) = erase(λx.M)--remove parameter type

erase(Λα(M)) = erase(M) --remove type abs

erase(M τ) = erase(M)   --remove type app
```

This brings us back to extended LC (ELC without types)

Type reconstruction

The type reconstruction (inference) problem:

Given *M* without type information (in, say, *ELC*), **find**:

- *M'* with type information (annotations, abstractions, applications)
- Γ for *freevars(M)* (= *freevars(M')*)
- a type τ

s.t. *Erase (M') = M* and Γ |- M' : τ

We then say that Γ |- M : τ

Example of Type Reconstruction

Erase

$$\mathrm{Erase}\left(\boxed{(\lambda f{:}\forall\alpha.\alpha\text{->}\alpha.(f\ \mathit{Int}\ 5,\ f\ \mathit{Bool}\ \mathrm{True}))\ \ (\Lambda\alpha.\lambda x{:}\alpha.x)}\right)$$

$$\Big\Downarrow$$

$$\boxed{(\lambda f.\ (f\ \mathit{5},\ f\ \mathit{True}))\ \ (\lambda x.x) \qquad \text{--(,) is a pair}}$$

Type reconstruction

Theorem:

Given *M* w/o type info, it is undecidable if well-typed *M*' in PLC s.t. *erase*(M') = *M* exists

Corollary:

Type reconstruction in PLC is impossible

So, how is it done in Haskell or SML?

Let us proceed to the Hindley-Milner Type System.

The Hindley-Milner Type System

We'll use the Damas-Milner version

Damas and Milner, POPL 82,
Principal type-schemes for functional programs

Let-Polymorphism

- The HMTS is *weaker* than the PLC, but admits a *type reconstruction algorithm*.
- Parametric polymorphism is achieved via let-expressions

```
let  id=\x->x
  in  (id 5, id True)
```

✓

- *Function parameters* are monomorphic only.

```
(\f->(f 5, f True)) (\x->x)
```

✗

Mini-Haskell Expression

FLOLAC 2008

```
E  ::=   constants: 1, 2, 3, …
                     'a', 'b', …,
                    True, False,  &&, ||, !
                     +, -, *, …,    >, <. =,
      |    variable:  x, y, z, …
      |   \x -> E                              Function abstraction
      |    E1 E2                               Function application
      |    if E1 then E2 else E3               If-expr
      |    let x = E1 in E2                    Let-expr
      |    (E1, E2)  | [] | [E1, …, En] | fst | snd | : | head | tail
              pairs           lists                      cons
```

Expression Examples

```
3+5,      x>y+3,     not (x>y) || z>0
(1, ‘a’)      fst (‘a’, 5)   --pair
[True, False]    x:xs    tail xs  --list
\x -> if x>0 then x*x else 1
(\x -> x*x) (4+5)
\f -> \x -> f (f x)
let f = \x-> x in (f True,  f ‘a’)  --pair
```

Types in Mini-Haskell

- Simple types
  - Int, Bool, Char, …
- Functional types
  - Int→Int, (Int→Bool)→Int, (Int→Bool)→(Int→Int),…
- Pair types
  - (Int, Bool), (Int, (Bool, Char)),…
- List types
  - [Int], [Bool], [[Int]], [(Int, Bool)], …
- Generalized types τ: adding type variables α
  - τ ::= Int | Bool | … | α | β … | $\tau 1 \rightarrow \tau 2$ | ($\tau 1$, $\tau 2$) | [τ]

Types in the HMTS

- No more general polymorphic types of PLC.
  - $\forall\alpha.\alpha \to \underline{\forall\beta.\beta} \to \mathrm{Int}$ ✗

    Nested quantification
- Adopts a <u>two-layered types</u>
  - Types with variables, but no quantifiers
  - Type Schemes that support only *outermost quantification*

$$\boxed{\overbrace{\forall\alpha.\forall\beta.}\ (\alpha \to \beta) \to [\alpha] \to [\beta]}$$ ✓

Types & Type Schemes

FLOLAC 2008

two-layered types

- Types τ: (mono)

– τ ::= Int | Bool | …	primitive types
| α| β| …	type variables
| $\tau1 \rightarrow \tau2$	function types **(Right-associative)**
| ($\tau1$, $\tau2$)	pair (tuple) types
| [τ]	list types

- Type schemes σ: (poly)

$$\boxed{\sigma ::= \tau \mid \forall\alpha\,.\,\sigma}$$

generic type variable

Examples of Type Schemes

$[\text{Int}]$, Bool, $\text{Char} \rightarrow \text{Bool}$

$(\text{Char}, \text{Int}) \rightarrow \text{Bool}$

$[\text{Int}] \rightarrow (\text{Int->Bool}) \rightarrow \text{Bool}$

$[\text{Int}] \rightarrow \beta \rightarrow \text{Bool}$

$\forall\alpha.\alpha$

$\forall\alpha.[\alpha] \rightarrow \alpha \rightarrow \text{Bool}$

$\forall\alpha.\ \forall\beta.(\alpha \rightarrow \beta) \rightarrow [\alpha] \rightarrow \beta$

$\forall\alpha.\alpha \rightarrow \beta$

•Outermost quantification only

Invalid **type schemes**

$\text{Int} \rightarrow \forall\alpha.\alpha$ $\quad$ $\forall\alpha.\alpha \rightarrow \forall\beta.\beta$ ✗

Generic (Bound) vs. Free Type Variables

FLOLAC 2008

$\sigma = \forall\alpha.\forall.\beta.\alpha \rightarrow \beta$

ftv(σ) = {}

$\sigma = \forall\alpha.\alpha \rightarrow \beta$

ftv(σ) = {β}

ftv($\alpha\rightarrow\beta$)= {α,β}

- Free type variables stand for *some* types;
- Generic type variables stand for *any* types.

Notation: omit *inner* $\forall$

$\forall\alpha.\ \beta.(\alpha\rightarrow\beta)\rightarrow[\alpha] \rightarrow \beta \equiv \forall\alpha.\forall\beta.\ (\alpha\rightarrow\beta) \rightarrow [\alpha] \rightarrow \beta$

Typing in Mini-Haskell

- A *type judgment* has the form

  Γ |- exp : τ --not σ

- exp is a Mini-Haskell expression
- τ is a Mini-Haskell *type* to be assigned to exp

the *typing environment* $\mathbf{\Gamma}$ is a finite function from variables to *type schemes.*

(We write $\Gamma = \{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$ to indicate that Γ has domain of definition $dom(\Gamma) = \{x_1, \ldots, x_n\}$ and maps each x_i to the type scheme σ_i for $i = 1..n$.)

Example Valid Type Judgments

- [ ] |- True or False : Bool
- [x : int] |- x + 3 : int
- [len : $\forall\alpha.[\alpha]$->Int] |- len [1,3,5,7] : Int
- [len : $\forall\alpha.[\alpha]$->Int] |- len [True, False] : Int
- [len : $\forall\alpha.[\alpha]$->Int] |- len : $[[\beta]]$ ->Int

via $[[\beta]/\alpha]$

Typing in Mini-Haskell

(Int) $\Gamma \mid\!- n : \text{Int}$ (assuming *n* is an Integer constant)

(Bool) $\Gamma \mid\!- \text{True} : \text{Bool}$ $\Gamma \mid\!- \text{False} : \text{Bool}$

(nil) $\Gamma \mid\!- [\,] : [\tau]$ --any type τ

(cons)
$$\frac{\Gamma \mid\!- e1 : \tau 1 \qquad \Gamma \mid\!- e2 : [\tau 1]}{\Gamma \mid\!- (e1 : e2) : [\tau 1]}$$

Note: [e1, e2, e3] is a syntactic sugar of (e1:(e2:e3))

(Pair)
$$\frac{\Gamma \mid\!- e1 : \tau 1 \qquad \Gamma \mid\!- e2 : \tau 2}{\Gamma \mid\!- (e1, e2) : (\tau 1, \tau 2)}$$

Typing in Mini-Haskell, 1

FLOLAC 2008

- A major change lies in *typing a function*
- In PLC, we need to specify the type of a function's parameter.

(fn)
$$\frac{\Gamma.x{:}\boldsymbol{\tau_1} \mid\!- M : \tau_2}{\Gamma \mid\!- \lambda x{:}\boldsymbol{\tau_1}.M : \tau_1 \to \tau_2}$$

- In the HTMS, We *guess a type for x*. No type annotation for parameters.

(Abs)
$$\frac{\Gamma.x{:}\tau_1 \mid\!- e : \tau_2}{\Gamma \mid\!- \lambda x.e : \tau_1 \to \tau_2}$$

A type, not a type scheme, such as $\forall\alpha.\alpha$, because fun Parameters are monomorphic.

Typing in Mini-Haskell, 2

- Guess as general as possible
- Consider the following two type derivations:

$$\frac{\Gamma.x{:}\alpha \mid\!- \ x : \alpha}{\Gamma \mid\!- \lambda x.x : \alpha \to \alpha} \quad \succ \quad \frac{\Gamma.x{:}\mathrm{Int} \mid\!- \ x : \mathrm{Int}}{\Gamma \mid\!- \lambda x.x : \mathrm{Int} \to \mathrm{Int}}$$

Obviously, the one on *the left* is better for type reconstruction – it is the most general.

- We can define some kind of order ($\succ$) between a type scheme and type

Orders between Types and Type Schemes, 1

- Specialization order between types and type schemes:

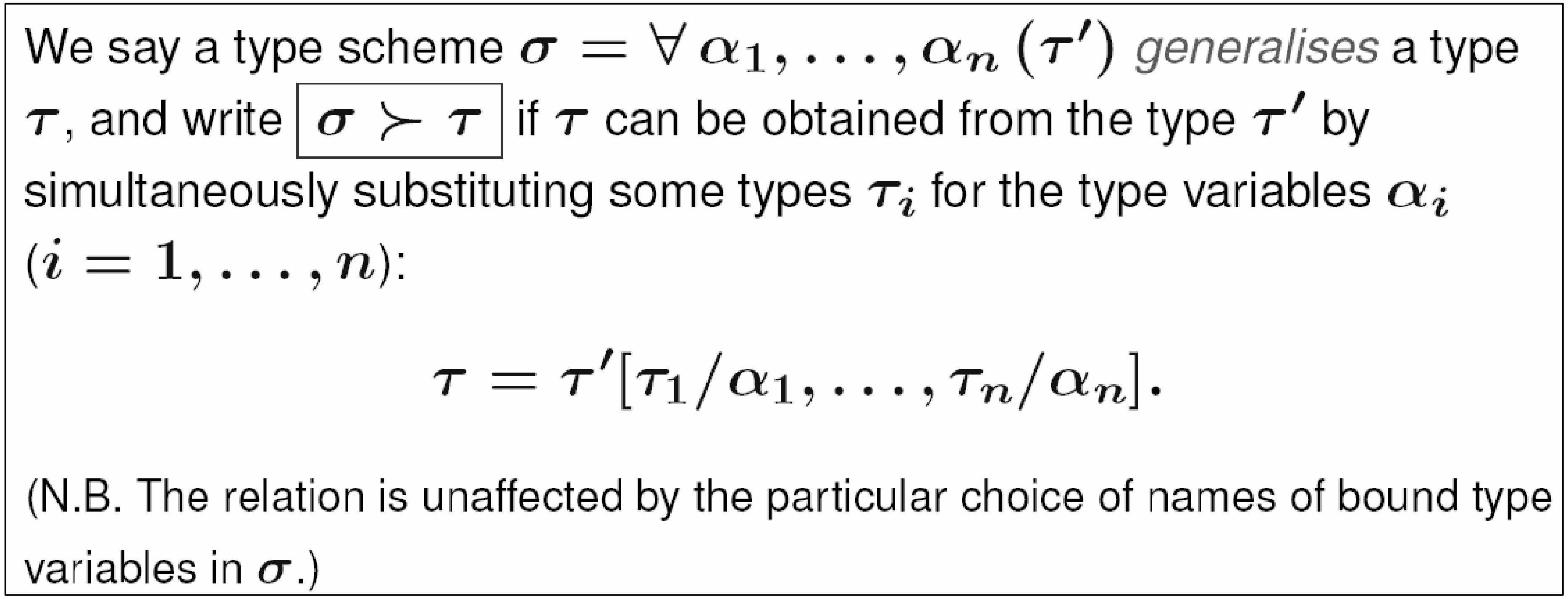
$\forall\alpha.\alpha\to\alpha \succ \beta\to\beta$ via $[\beta/\alpha]$

$\forall\alpha.\alpha\to\alpha \succ \mathrm{Int}\to\mathrm{Int}$ via $[\mathrm{Int}/\alpha]$

$\forall\alpha.\beta.\alpha\to\beta\to\beta \succ \mathrm{Int}\to(\mathrm{Bool}\to\mathrm{Bool})$

via $[\mathrm{Int}/\alpha, \mathrm{Bool}/\beta]$

Order between a Type Scheme and a Type, 2



We say a type scheme $\sigma = \forall\,\alpha_1, \ldots, \alpha_n\,(\tau')$ *generalises* a type τ, and write $\boxed{\sigma \succ \tau}$ if τ can be obtained from the type τ' by simultaneously substituting some types τ_i for the type variables α_i $(i = 1, \ldots, n)$:

$$\tau = \tau'[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n].$$

(N.B. The relation is unaffected by the particular choice of names of bound type variables in σ.)

•Also called instantiation of a type scheme to a type.

$\forall\alpha.\alpha \to \alpha \quad \succ \quad \beta \to \beta \quad$ via $[\beta/\alpha]$

Orders between Type Schemes and Types, 3

FLOLAC 2008

- Not all type variables are equal!
- *Generic type variables* vs. *free type variables*

$\forall\alpha.\alpha\rightarrow\alpha$ $\beta\rightarrow\beta$

•*Generic type variables* can be instantiated to any *types* τ, but free types variables are not!

•<u>Generalization order</u> between a type scheme and a type: $\sigma \succ \tau$, this is required in typing rules

•Specialization between two types is derived during type reconstruction as interim results.

Typing in Mini-Haskell, 2

FLOLAC 2008

- Instantiate a type scheme to a type *by guessing*
  - From $\forall\alpha.[\alpha]$->Int to $[[\beta]]$ ->Int
- Only when typing a variable:

$$(\text{Var} \succ) \quad \frac{}{\Gamma \mid\!\!- \; \mathbf{x} : \tau} \quad \text{if } \Gamma(x) = \sigma \text{ and } \sigma \succ \tau$$

Example：

[len : $\forall\alpha.[\alpha]$->Int] |- len : $[\beta]$ ->Int

•In PLC,

[len : $\forall\alpha.[\alpha]$->Int] |- len β : $[\beta]$ ->Int

PLC vs. HTMS

- Recall that PLC has:
  - General polymorphic types: $\tau \equiv \forall\alpha.\tau'$
  - Application with *type* operand: M τ (τ a type)
  - Type generalization: $\Lambda\alpha$ (M)
- By contrast, the HMTS
  - **types τ and type schemes σ**
  - **Instantiate a type scheme to a type**
    - **From $\forall\alpha.[\alpha]$->Int to [[β]] ->Int**
  - Generalize a type to a type scheme
    - From [β] ->Int to $\forall\beta$. [β] ->Int

Typing in Mini-Haskell, 3

- Function application remains the same, except that only *monomorphic arguments* (τ).

$$\text{(App)}\quad \boxed{\dfrac{\Gamma \mid\!- \, e1 : \tau 1 \to \tau 2 \qquad \Gamma \mid\!- \, e2 : \tau 1}{\Gamma \mid\!- \, (e1\ e2) : \tau 2}}$$

Example:

$$\dfrac{\begin{array}{l}[\,\text{len} : \forall\alpha.[\alpha]\text{->Int}\,]\ \mid\!-\ \text{len} : [\text{Bool}]\ \text{->Int} \\ [\,\text{len} : \forall\alpha.[\alpha]\text{->Int}\,]\ \mid\!-\ [\text{True,False}] : [\text{Bool}]\end{array}}{[\,\text{len} : \forall\alpha.[\alpha]\text{->Int}\,]\ \mid\!-\ \text{len}\ [\text{True,False}] : \text{Int}}$$

$$\text{(If)}\quad \boxed{\dfrac{\Gamma \mid\!- \, e1 : \text{Bool} \qquad \Gamma \mid\!- \, e2 : \tau \qquad \Gamma \mid\!- \, e3 : \tau}{\Gamma \mid\!- \, \text{if } e1 \text{ then } e2 \text{ else e3} : \tau}}$$

A Function Example

```
Γ |- \f->\x->f (f x)): ?
```

•Move f and x to Γ

```
Γ.f:α->α.x:α |- f: α->α
Γ.f:α->α.x:α |-  x: α
_____________________________________ (App)
Γ.f:α->α.x:α |- f x: α
Γ.f:α->α.x:α |- f: α->α
_____________________________________________ (App)
 Γ.f:α->α.x:α |- f (f x)): α
__________________________________________________ (Abs)
        Γ.f:α->α|- \x->f (f x)): α->α
 __________________________________________________ (Abs)
        Γ|- \f->\x->f (f x)):(α->α)->α->α
```

Typing in Mini-Haskell, 4

FLOLAC 2008

- Generalizing a type to a type scheme via LET-expr

Γ `|- \f->\x->f (f x)):` $(\alpha\text{->}\alpha)\text{->}\alpha\text{->}\alpha$

$$\Downarrow$$

$\forall\alpha.(\alpha\text{->}\alpha)\text{->}\alpha\text{->}\alpha$

$$\text{(Let)}\quad \frac{\Gamma \ \text{|-}\ e1 : \tau 1 \qquad \Gamma.\ x{:}\sigma\ \text{|-}\ e2 : \tau}{\Gamma\ \text{|-}\ \text{let } x{=}e1 \text{ in } e2 : \tau}\quad x \notin \text{dom}(\Gamma)$$

$\sigma = \text{Gen}(\tau 1, \Gamma) = \forall\alpha 1\ldots\alpha n.\tau 1.$.
where $[\alpha_1,\ldots,\alpha_n] = \text{ftv}(\tau 1) - \text{ftv}(\Gamma)$

Generalization *aka Closing*

$\text{Gen}(\Gamma,\tau) = \forall\alpha_1\ldots\,\alpha_n.\ \tau$
where $[\alpha_1\ldots\,\alpha_n\,] = \text{ftv}(\tau) - \text{ftv}(\Gamma)$

- *Generalization* introduces polymorphism
- Quantify type variables that are free in but not *free* in the type environment (TE)
- Captures the notion of *new* type variables of τ (introduced via the Var $\succ$ rule)

Example of Let-Polymorphism

E ≡ let id=\x->x in *(id 5, id True)*

(1) Γ |- \x->x : $\alpha \to \alpha$ α is a fresh var, Gen called

$$\frac{\text{(2.1) } \Gamma.\ \text{id}:\forall\alpha.\alpha \to \alpha \ \text{|-}\ \textit{id} : \text{Int->Int} \qquad \Gamma.\ \text{id}:\forall\alpha.\alpha \to \alpha \ \text{|-}\ \textit{5} : \text{Int}}{\Gamma.\ \text{id}:\forall\alpha.\alpha \to \alpha \ \text{|-}\ \textit{id 5} : \text{Int}}$$

$$\frac{\text{(2.2) } \Gamma.\ \text{id}:\forall\alpha.\alpha \to \alpha \ \text{|-}\ \textit{id} : \text{Bool->Bool} \quad \Gamma.\ \text{id}:\forall\alpha.\alpha \to \alpha \ \text{|-}\ \textit{True} : \text{Bool}}{\Gamma.\ \text{id}:\forall\alpha.\alpha \to \alpha \ \text{|-}\ \textit{id True} : \text{Bool}}$$

$$\frac{\dfrac{\text{(2.1), (2.2)}}{\Gamma.\ \text{id}:\forall\alpha.\alpha \to \alpha \ \text{|-}\ \textit{(id 5, id True)} : \text{(Int, Bool)}}\ \text{Pair}}{\Gamma\ \text{|-}\ \text{let id=\textbackslash x->x in}\ \textit{(id 5, id True)} : \text{(Int, Bool)}}\ \text{Let}$$

Exercises of Let-Polymorphism

FLOLAC 2008

1. We can also have "*id id*" in the let-body:

```
let id = \x->x in  id id
```

2. Derive the type for the following lambda function:

```
\x. let f = \y->x  B        A
    in (f 1, f True)
```

$\Gamma_A = [\, x : \alpha \,]$

(1) $$\frac{\Gamma_A.[\, y{:}\beta \,] \mid\!- \; x : \alpha}{\Gamma_A \mid\!- \; \backslash y\text{->}x : \beta \rightarrow \alpha}$$

HM Type Inference Rules

(App)
$$\frac{\Gamma \vdash e_1 : \tau \text{ -> } \tau' \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash (e_1\ e_2) : \tau'}$$

.

(Abs)
$$\frac{\Gamma + [x : \tau] \vdash e : \tau'}{\Gamma \vdash \lambda x.e : \tau \text{ -> } \tau'}$$

(Var)
$$\frac{(x : \sigma) \in \Gamma \qquad \sigma \geq \tau}{\Gamma \vdash x : \tau}$$

(Const)
$$\frac{\text{typeof}(c) \geq \tau}{\Gamma \vdash c : \tau}$$

(Let)
$$\frac{\Gamma+[x : \tau] \vdash e_1 : \tau \qquad \Gamma+[x:\text{Gen}(TE,\tau)] \vdash e_2 : \tau'}{\Gamma \vdash (\textit{let}\ x = e_1\ \textit{in}\ e_2) : \tau'}$$

Syntax-Directed

Limitations of the HMTS:

FLOLAC 2008

λ–bound (monomorphic) vs Let-bound Variables

•Only *let-bound* identifiers can be instantiated differently.

E1 ≡ let id=\x->x in *(id 5, id True)*
vs. E2 ≡ (\f->*(f 5, f True))(\x->x)*

$\Bigg\}$ Semantically E1 = E2, but

•E2≡ \f->*(f 5, f True) is not typable:*

Recall the (Abs) rule

$$\frac{\Gamma .x:\tau 1 \mid\!- e : \tau 2}{\Gamma \mid\!- \backslash x \to e : \tau 1 \to \tau 2}$$

[f : ?] |- (f 5, f True) : (Int, Bool)

$\uparrow$

a type only, not a type scheme to instantiate

Good Properties of the HMTS

- The HMTS for Mini-Haskell is *sound*.
  - Well-typed programs won’t get stuck!.
- The typeability problem of the HMTS is *decidable*: there is *a type reconstruction algorithm* which computes the *principal type scheme* for any Mini-Haskell expression.
  - The W algorithm using unification

Principle Type Schemes for Closed Expressions, 1

- What type for “\f->\x->f x”?

$$\dfrac{\dfrac{\dfrac{\text{[f:Int}\rightarrow\text{Bool, x:Int] |- f : Int}\rightarrow\text{Bool} \qquad \text{[f:Int}\rightarrow\text{Bool, x:Int] |- x : Int}}{\text{[f:Int}\rightarrow\text{Bool, x:Int] |- f x : Bool}}\,\text{App}}{\text{[f:Int}\rightarrow\text{Bool] |- \\x->f x : Int} \rightarrow \text{Bool}}\,\text{Abs}}{\text{[ ] |- \\f->\\x->f x : (Int} \rightarrow \text{Bool)} \rightarrow \text{(Int} \rightarrow \text{Bool)}}\,\text{Abs}$$

Can we derive a ***more “general” type*** for this expression?

Principle Type Schemes for Closed Expressions, 2

FLOLAC 2008

•A more general type for “\f->\x->f x”?

$$\dfrac{\dfrac{\dfrac{[\text{ f: }\alpha\to\beta\text{, x :}\alpha] \text{ |- f : }\alpha\to\beta \qquad [\text{f : }\alpha\to\beta\text{, x : }\alpha] \text{ |- x : }\alpha}{[\text{f : }\alpha\to\beta\text{, x : }\alpha] \text{ |- f x : }\beta}}{[\text{f : }\alpha\to\beta] \text{ |- \\x -> f x : }(\alpha\to\beta)}}{[\ ] \text{ |- \\f -> \\x -> f x : }\underbrace{(\alpha\to\beta)\to(\alpha\to\beta)}_{\textbf{\textit{Most general type}}}}$$

Any instance of $(\alpha\to\beta)\to(\alpha\to\beta)$ is a valid type.

E.g., (Int $\to$ Bool) $\to$(Int $\to$ Bool)

Principle Type Schemes for Closed Expressions

- A type scheme σ is the *principal* type scheme of a closed Mini-Haskell expression *E* if
  - (a) |- E : τ is provable and σ = Gen(τ, {})
  - (b) for all τ', if |- E : τ' is provable and σ' = Gen(τ', {})

    then $\sigma \succ \sigma$**'**

  where by definition $\sigma \succ \sigma$' if σ' = $\forall\alpha_1\ldots\alpha_n.\tau$' and FV($\sigma$) $\cap$ {$\alpha_1\ldots\alpha_n$} = {} and $\sigma \succ \tau$' .

E.g., \f->\x->f x has the PTS of $\forall\alpha.\beta.(\alpha\rightarrow\beta)\rightarrow(\alpha\rightarrow\beta)$
and $\forall\alpha.\beta.(\alpha\rightarrow\beta)\rightarrow(\alpha\rightarrow\beta) \succ \forall\gamma.(\gamma\rightarrow\text{Bool})\rightarrow(\gamma\rightarrow\text{Bool})$

Type Reconstruction Algorithm Based on Unification

The W Algorithm by Damas and Milner

Type Inference

FLOLAC 2008

- Type inference is typically presented in two different forms:
  - *Type inference rules:* Rules define the type of each expression
    - Clean and concise; needed to study the semantic properties, i.e., soundness of the type system
  - *Type inference (reconstruction) algorithm:* Needed by the compiler writer to deduce the type of each subexpression or to deduce that the expression is ill typed.
- Often it is nontrivial to derive an inference algorithm for a given set of rules. There can be many different algorithms for a set of typing rules.

The W Algorithm (Damas&Milner 82)

W(Γ, e) returns (S,τ) such that **S(Γ) $\vdash$ e : τ**

- Γ is a typing environment recording the most general type of each identifier that may occur in e
- *e* is an expression
- τ is a type, may contain type variables to be generalized
- S is a type substitution recording the changes in the free type variables in Γ, if any.

The W Algorithm

W(Γ, e) returns (S,τ) such that **S(Γ)** $\vdash$ **e : τ**

•Example: Open expression

Γ **= [f:α->α, x:β],** **e $\equiv$ f x**

W(Γ, e) = ([α/β], β) and

[α/β](Γ) $\vdash$ **f x : β**

The W Algorithm

W(Γ, e) returns (S,τ) such that **S(Γ)** $\vdash$ **e :** τ

- Example: closed expression

Γ `= [],` `e` $\equiv$ `let id=\x->x in (id id)`

`W(`Γ`, e) = ([`β`->`β`/`α`], `$\beta\to\beta$`)` and

`[`β`->`β`/`α`](`Γ`)` $\vdash$ `e :` β`->`β

The W Algorithm: Syntax-Directed

W(Γ, e) returns (S,τ) such that **S(Γ)** $\vdash$ **e : τ**

The W algorithm is defined in terms of the syntactic structure of the expression to type.

Syntax-directed

```
Def  W(Γ, e) =
    Case e of
      x                       =   …
      λx.e                    =   …
      (e₁ e₂)                 =   …
      let  x = e₁ in e₂       =   …
```

The W Algorithm: Variables

1. When e is a variable:

```
Def  W(Γ, e) =
          Case e of
              x        =   ...
```

Recall the inference rule (axiom) for variables:

$$\text{(Var)} \quad \boxed{\dfrac{(x : \sigma) \in \Gamma \qquad \sigma \geq \tau}{\Gamma \mid\!- \; x : \tau}}$$

We do not yet know which τ to instantiate!

Let $\forall\alpha.\alpha\text{->}\alpha = \Gamma(x)$, we simply replace α with fresh (new) type variable, say β; and determine the type for β later when x is applied via unification.

The W Algorithm: Variables

FLOLAC 2008

1. When e is a variable:

Recall the inference rule (axiom) for variables: (Var)

$$\frac{(x : \sigma) \in \Gamma \quad \sigma \geq \tau}{\Gamma \mid\!- \ x : \tau}$$

We do not yet know which τ to instantiate!

```
Def W(Γ, e) =
        Case e of
         x        = if (x ∉ Dom(Γ)) then Fail
                    else let ∀α_1… α_n.τ = Γ(x);
                         in ( { }, [β_i /α_i]τ)
```

β's represent new type variables

The W Algorithm: Application

FLOLAC 2008

2. When e is an application:

Def W(Γ, e) =
Case e *of*
(e_1 e_2) =

Recall the inference rule for fun application:

(App) $$\frac{\Gamma \vdash e_1 : \tau \to \tau' \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash (e_1\ e_2) : \tau'}$$

We have to ensure that *the type of parameter* is the same as *the type of the argument* (e_2)!

We apply the unification algorithm to compute a Type substiution to unify them..

The W Algorithm: Application

FLOLAC 2008

2. When e is a function application:

(App)
$$\frac{\Gamma \vdash e_1 : \tau \to \tau' \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash (e_1\ e_2) : \tau'}$$

β represents a new type variable

```
Def W(Γ, e) =
        Case e of
          (e₁ e₂)  let (S₁, τ₁) = W(Γ, e₁);
                       (S₂, τ₂) = W(S₁(Γ), e₂);
                          S₃    = Unify(S₂(τ₁), τ₂ ->β);
                   in  (S₃ S₂ S₁, S₃(β))
```

Unification: Unify(τ_1, τ_2)

FLOLAC 2008

•Unify(τ_1, τ_2) = fail or a type substitution S such that $S\tau_1 = S\tau_2$.

Unify(**α->α, Int->Bool) = fail**

Unify(**α->α, Int->Int) = [Int/α]** ≡ **S**

Then **S(α->α) = S(Int->Int)**

Unify(**[α]->β, [γ]->Int) = [γ/α, Int/β]≡S**

•And compute the Most General Unifier (MGU)

Let S' = **[Bool/α, Int/β]**.

S'(**[α]->β**) = S'(**[γ]->Int**) and **S** $\succ$ **S'**

Unification: Unify(τ_1, τ_2)

```
def Unify($\tau_1$ , $\tau_{t2}$ ) =
   case    ($\tau_1$ ,  $\tau_2$ ) of
     ($\tau_1$ , $\alpha$ )     = [$\tau_1$ / $\alpha$ ]              --$C_i$  constant type
     ($\alpha$ , $\tau_2$ )     = [$\tau_2$/ $\alpha$ ]
     ( $C_1$ , $C_2$ )  = if (eq? $C_1$ , $C_2$  ) then [ ] else  fail
     ($\tau_{11}$-> $\tau_{12}$, $\tau_{21}$ -> $\tau_{22}$)
       = let    S1 =Unify($\tau_{11}$, $\tau_{21}$  )
                S2 =Unify(S1 ($\tau_{12}$), S1 ($\tau_{22}$))
           in   S2$\circ$ S1
    otherwise  = fail
```

•Composition of substitution: **S2$\circ$S1**

Ex: **[Int/β]$\circ$[β/α]=[Int/β,Int/α]**

The W Algorithm: Function

3. When e is a lambda function:

```
Def  W(Γ, e) =
          Case e of
             \x->e     =
```

Recall the inference rule for lambda function:

$$(\text{Abs}) \quad \frac{\Gamma + [x : \tau] \vdash e : \tau'}{\Gamma \vdash \backslash x.e : \tau \to \tau'}$$

We have to guess a *type for the parameter!*

We use a new type variable to represent the *type of the parameter* and get a type for it later when the function is applied.

The W Algorithm: Function

FLOLAC 2008

3. When e is a lambda function:

(Abs)
$$\frac{\Gamma + [x : \tau] \vdash e : \tau'}{\Gamma \vdash \backslash x.e : \tau \to \tau'}$$

```
Def  W(Γ, e) =
        Case e of
          \x->e     =  let  $(S_1, \tau_1)$ = W(Γ + [x:β], e);
                       in   $(S_1, S_1(\beta) \to \tau_1)$
```

β is new

The W Algorithm: Let

4. When e is a let expression:

```
Def  W(Γ, e) =
  Case e of
        let  x = e₁ in e₂  =...
```

Recall the inference rule for let expression:

(Let) $$\frac{\Gamma + [x : \tau] \vdash e_1 : \tau \qquad \Gamma + [x : Gen(TE, \tau)] \vdash e_2 : \tau'}{\Gamma \vdash let\ \ x = e1\ in\ e2 : \tau'}$$

```
Def  W(Γ, e) =
  Case e of
        let  x = e₁ in e₂  = let (S₁, τ₁) = W(Γ , e₁);
                                  σ        = Gen( S₁(Γ), τ₁ );
                                  (S₂, τ₂) = W(S₁(Γ) + [x : σ], e₂);
                             in   (S₂ S₁, τ₂)
```

The W Algorithm

β's new type vars

```
Def W(Γ, e)    = Case e of
    x          = if  (x ∉ Dom(Γ)) then Fail
                 else  let  ∀t_1...t_n.τ  = Γ(x);
                          in   ( { }, [β_i / t_i] τ)
    λx.e       =     let  (S_1, τ_1) = W(Γ + [x : β], e);
                     in   (S_1, S_1(β) -> τ_1)
    (e_1 e_2)  =     let (S_1, τ_1) = W(Γ, e_1);
                         (S_2, τ_2) = W(S_1(Γ), e_2);
                             S_3    = Unify(S_2(τ_1), τ_2 -> β);
                     in   (S_3 S_2 S_1, S_3(u))
    let  x = e_1 in  e_2
               =     let (S_1, τ_1) = W(Γ , e_1);
                         σ          = Gen( S_1(Γ), τ_1 );
                         (S_2, τ_2) = W(S_1(Γ) + [x : σ], e_2);
                     in   (S_2 S_1, τ_2)
```

The W Algorithm: Example

FLOLAC 2008

```
λx. let f = λy.x      B       A
    in (f 1, f True)
```

W(∅, A) = ([ ] , u_1 **->** (u_1, u_1))

W({x : u_1}, B) = ([ ] , (u_1, u_1))

W({x : u_1, f : u_2}, λ**y.x**) = ([ ] , u_3 **->** u_1)

W({x : u_1, f : u_2, y : u_3}, **x**) = ([ ] , u_1)

Unify(u_2 , u_3 **->** u_1) = [(u_3 **->** u_1) **/** u_2]

Gen({x : u_1}, u_3 **->** u_1) = $\forall u_3.u_3$ **->** u_1

TE = {x : u_1, f : $\forall u_3.u_3$ **->** u_1}

W(TE, **(f 1)**) = ([ ] , u_1)

W(TE, **f**) = ([ ] , u_4 **->** u_1)

W(TE, 1) = ([ ] , **Int**)

Unify(u_4 **->** u_1 , **Int ->** u_5) = [**Int** / u_4 , u_1 / u_5]

. . .

Important Observations

- Do not generalize over type variables used elsewhere
- Let is the only way of defining polymorphic constructs
- Generalize the types of let-bound identifiers only after processing their definitions

Properties of HM Type Inference (W)

- It is sound with respect to the type system.
  An inferred type is verifiable using |-.
- It generates most general types of expressions.
  called *Principal Type Scheme*.
  Any verifiable type is inferred.
- Complexity
  PSPACE-Hard
  DEXPTIME-Complete
  Nested *let* blocks

Extensions

- Type Declarations
  Sanity check; can relax restrictions
- Incremental Type checking
  The whole program is not given at the same time, sound inferencing when types of some functions are not known
- Typing references to mutable objects
  Hindley-Milner system is unsound for a language with refs (mutable locations)
- Overloading Resolution

Puzzle: Another set of Inference rules **Not syntax-directed**

(Gen)
$$\frac{TE \vdash e : \tau \quad \alpha \notin FV(TE)}{TE \vdash e : \forall\alpha.\tau}$$

(Spec)
$$\frac{TE \vdash e : \forall\alpha.\tau}{TE \vdash e : \tau\,[\tau'/\alpha]}$$

(Var)
$$\frac{(x : \tau) \in TE}{TE \vdash x : \tau}$$

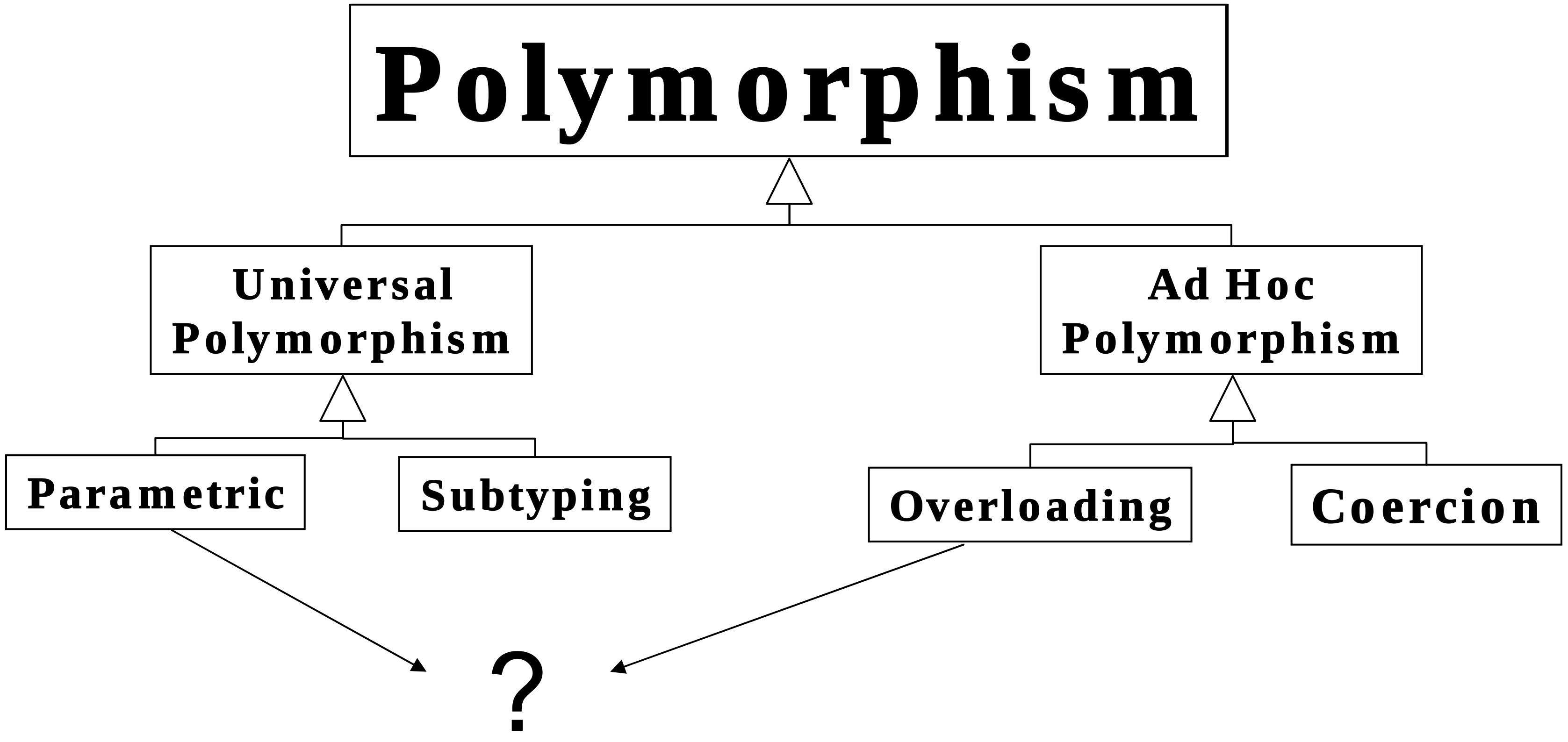
(Let)
$$\frac{TE+\{x:\tau\} \vdash e_1 : \tau \quad TE+\{x:\tau\} \vdash e_2 : \tau'}{TE \vdash (\textit{let}\ x = e_1\ \textit{in}\ e_2) : \tau'}$$

(App) and (Abs) rules remain unchanged.

Sound but no direct inference algorithm !

Appendix: Haskell’s Type Classes

Polymorphism



When Overloading Meets Parametric Polymorphism

- Overloading: some operations can be defined for *many different data types*
  - **`==, /=, <, <=, >, >=,`** defined for many types
  - **`+, -, *,`** defined for numeric types

- Consider the *double* function: *double* = \x-> *x*+*x*

  - What should be the proper type of double?
    - Int -> Int -- too specific
    - ∀a.a -> a -- too general

Indeed, this *double* function **is not** typeable in (earlier) SML!

Type Classes—a "middle" way

- What should be the proper type of double?
  ∀a.a -> a -- too general
- It seems like we need something "in between", that restricts "a" to be from the set of all types that admit *addition operation*, say
  Num = {Int, Integer, Float, Double, etc.}.—type class
  double :: (∀ a ∈ Num) a -> a
- *Qualified types* generalize this by qualifying the type variable, as in (∀ a ∈ Num) a -> a , which in Haskell we write as $\boxed{\text{Num a => a -> a}}$

•Note that the type signature a -> a is really shorthand for ∀a.a -> a

Type Classes

- "**Num**" in the previous example is called a *type class*, and should not be confused with a type constructor or a value constructor.
- "**Num T**" should be read "T is a member of (or an instance of) the type class Num".
- Haskell's type classes are one of its most innovative features.
- This capability is also called "overloading", because one function name is used for potentially very different purposes.
- There are many *pre-defined type classes*, but you can also *define your own*.

Defining Type Classes in Haskell, 1

- In Haskell, we use type classes and instance declarations to support parametric overloading systematically.

```
class  Num a where
     (+), (-), (*)   :: a -> a -> a
      negate         :: a -> a
      …
```

- Type a belongs to class Num if it has '+','-','*', …of proper signature defined.

A type is made an instance of a class by an *instance declaration*

```
Instance Declaration:
instance  Num Int  where
   (+) =  IntAdd   --primitive
   (*)  =  IntMul   -- primitive
   (-)  =  IntSub   -- primitive
   …
```

- Type Int is an instance of class Num

Defining Type Classes in Haskell, 2

In Haskell, the ***qualified type*** for double

type predicate

$$\text{double x = x + x ::}$$

$$\forall a.\ \underline{\text{Num a}} \Rightarrow a \text{->} a$$

I.e., we can apply *double* to only types which are instances of class Num.

```
double 12       --OK
double 3.4      --OK
double “abc”    --Error unless String is an instance
                --of class Num,
```

Constrained polymorphism

- Ordinary parametric polymorphism

  f :: a -> a

  "f is of type a -> a for any type a"

- Overloading using *qualified types*

  f :: C a => a -> a

  "f is of type a -> a for any type *a* belonging to the <u>*type class*</u> C"

- Think of a Qualified Type as a type with a Predicate set, also called context in Haskell.

Type Classes and Overloading

double :: ∀ a. <u>Num a</u> => a->a

The type predicate "Num a" will be supported by an *<u>additional (dictionary) parameter</u>*.

In Haskell, the function *double* is translated into

double *NumDict* x =
(select (+) from NumDict) x x

Similar to

double *add* x = x `*add*` x -- *add* x x

Type Classes and Overloading

Dictionary for (type class, type) is created by the *Instance declaration.*

```
instance  Num Int  where
    (+) =  IntAdd   --primitive
    (*)  =  IntMul          -- primitive
    (-)  =  IntSub          -- primitive
    …
```

Create a dictionary called *IntNumDict, and "double 3" will be translated to*
double intNumDIct 3

Another Example: Equality

FLOLAC 2008

- Like addition, *equality* is not defined on all types (how do we test the equality of two functions, for example?).
- So the equality operator (==) in Haskell has type Eq a => a -> a -> Bool. For example:

```
42 == 42              ➔ True
`a` == `a`            ➔ True
`a` == 42             ➔ << type error! >>
                        (types don't match)
(+1) == (\x->x+1)     ➔ << type error! >>
                        ((->) is not an instance of Eq)
```

- Note: the type errors occur *at compile time!*

Equality, cont'd

- Eq is defined by this *type class declaration:*

```
class Eq a  where
          (==), (/=)       :: a -> a -> Bool
          x /= y          =  not (x == y)
          x == y           =  not (x /= y)
```

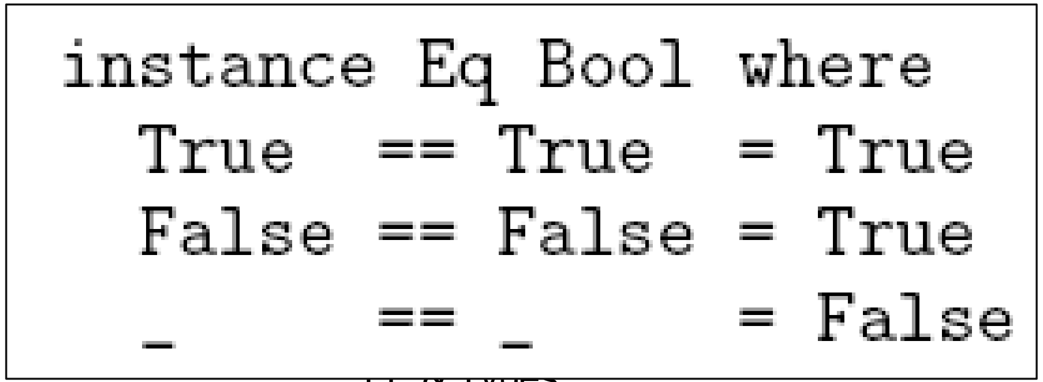
- The last two lines are *default methods* for the operators defined to be in this class.
- So the instance declarations for Eq only needs to define the "==" method.

Defining class <u>instances</u> (1)

- Make pre-existing classes instances of type class:

```
instance Eq Integer where
    x == y =  x `integerEq` y
instance Eq Float where
    x == y =  x `floatEq` y
```

- (assumes **`integerEq`** and **`floatEq`** functions exist)



```
instance Eq Bool where
  True  == True  = True
  False == False = True
  _     == _     = False
```

Defining class instances (2)

- Do same for composite data types, such as tuples (pairs).

```
instance  (Eq a, Eq b) =>  Eq (a, b) where
    (x1, y1) == (x2, y2) = (x1==x2) &&
                           (y1==y2)
```

- Note the context: **`(Eq a, Eq b) =>`** ...

Defining class instances (3)

- Do same for composite data types, such as lists.

```
instance Eq a =>  Eq [a] where
   [] == []             = True
   (x:xs) == (y:ys)  =  x==y && xs==ys
    _      == _        =  False
```

- Note the context: **Eq a =>** ...

Functions Requiring Context Constraints

FLOLAC 2008

•Consider the following list element testing function:

```
elem :: Eq a => a -> [a] -> Bool

elem x [ ]         =  False
elem x (y:ys)      =  (x == y) || elem x ys


>elem  5  [1, 3, 5, 7]
 True

>elem  ‘a’  “This is an example”
 False
```

Context Constraints (cont’d)

```
succ :: Int -> Int
succ = (+1)
```

`elem succ [succ]` causes an error

```
ERROR - Illegal Haskell 98 class constraint
  in inferred type
*** Expression : elem succ [succ]
*** Type       : Eq (Int -> Int) => Bool
```

which conveys the fact that **`Int->Int`** is not an instance of the **`Eq`** class.

Other useful type classes

FLOLAC 2008

- Comparable types:

`Ord` → **`< <= > >=`**

- Printable types:

`Show` → **`show`** where

```
show :: (Show a) => a -> String
```

- Numeric types:

`Num` → **`+ - * negate abs`** etc.

Show – Showable Types

FLOLAC 2008

- This class contains all those types whose values can be converted into character strings using

  show :: *a* -> *String*

- *Bool*, *Char, String*, *Int*, *Integer* and *Float*, are part of this class, as well as list and tuple types whose elements and components are part of the class

Show – Showable Types

FLOLAC 2008

```
> Show True
”True”

> show ’a’
”’a’”

> show 42
”42”

> show (´q´, 13)
”(’q’, 13)”
```

Read **– Readable Types**

- This class contains all those types whose values can be converted from character strings using

  $\boxed{\textit{read} :: \textit{String} \rightarrow \textit{a}}$

- *Bool*, *Char, String*, *Int*, *Integer* and *Float*, are part of this class, as well as list and tuple types whose elements and components are part of the class

Read – Readable Types

FLOLAC 2008

```
> read ”True” :: Bool
False

> read ”’a’” :: Char
’a’

> read ”42” :: Int
42

> read ”(´q´, 13)”
(’q’, 13)

> read ”[1,2,3]” :: [Int]
[1,2,3]
```

Super/Subclasses

- Subclasses in Haskell are more a *syntactic mechanism*.
- Class Ord is a subclass of Eq.

```
class Eq a => Ord a where
  (<), (>), (<=), (>=) :: a -> a -> Bool
  max, min :: a -> a -> a

  x < y = x <= y && x /= y
  x >= y = y <= x
  x > y = y <= x && x /= y

  max x y | x <= y    = y
          | otherwise = x
  min x y | x <= y    = x
          | otherwise = y
```

"=>" is misleading!

Note: If type T belongs to *Ord*, then T must also belong to *Eq*

Class hierarchies

- Classes can be hierarchically structured

```
class Eq a where  ...

class Eq a => Ord a where ...

class Ord a => Bounded a where
    minBound, maxBound :: a

class (Eq a, Show a) => Num a  where
  (+), (-), (*)  :: a -> a -> a       ...

class (Num a, Ord a) => Real a where
    toRational     :: a -> Rational

class (Real a, Enum a) => Integral a where
    quot, rem, div, mod :: a -> a -> a     ...
```

Source: D. Basin

Recommended Readings

•Luca Cardelli, Basic Polymorphic Typechecking.
http://research.microsoft.com/users/luca/Papers/**BasicTypechecking**.pdf

[DM82] Luis Damas and Robin Milner. Principal type schemes for functional programs. Proceedings of the 8th annual ACM symposium on Principles of Programming languages, Albuquerque, New Mexico, January 1982.
http://portal.acm.org/citation.cfm?id=582176

[CDK86] Dominique Clément, Joëlle Despeyroux, Thierry Despeyroux and Gilles Kahn. A simple applicative language: Mini-ML. ACM symposium on LISP and functional programming, 1986.
http://hal.inria.fr/inria-00076025/en/

Philip Wadler and Stephen Blott. How to make *ad-hoc* polymorphism less *ad-hoc*. Proceedings of the 16th annual ACM symposium on Principles of Programming Languages, Austin, Texas, January 1989.
http://portal.acm.org/citation.cfm?id=75283&dl=ACM&coll=GUIDE

Acknowledgements

- Parts of the materials presented here are taken from the slides prepared by :
- Dr. A.C. Daniels and Dr. S. Kahrs, Univ. of Kent, UK
- Professor. A. Pitts, Cambridge Univ., UK
- Professor E. Gunter, CS421, UIUC USA
- Professor Arvind, 6.827/F2006, MIT, USA