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

• 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?

Generally speaking:
• Functional programming is a style of programming in which the *primary method of computation is the application of functions to arguments*

• Define a function square:

\[
\text{square } x = x \times x
\]
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*

No parentheses: \texttt{square}(5)

Substitute the argument 5 into the body of the function

\begin{align*}
\text{Function application:} \\
\text{square 5} \\
= \{ \text{applying } \text{square} \} \\
= \{ \text{applying } * \} \\
= 5 * 5 \\
= 25
\end{align*}
Similarly an argument may itself be a function application:

\[
\begin{align*}
\text{square} ( \text{square} 3 ) & \quad = \{ \text{apply inner square} \} \\
\text{square} ( 3 * 3 ) & \quad = \{ \text{apply } * \} \\
\text{square} ( 9 ) & \quad = \{ \text{apply outer square} \} \\
9 * 9 & \quad = \{ \text{apply } * \} \\
81
\end{align*}
\]
- **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.

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

There is no loop!
Recursive, please!

Equations

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

[]-empty list;
“:”-cons a list

Variable assignments
**Imperative vs. Functional**

In C, the sequence of actions is:
- i = 1
- total = 1
- i = 2
- total = 3
- i = 3
- total = 6
- i = 4
- total = 10
- i = 5
- total = 15

Applying functions:

\[
\begin{align*}
\text{sum} [1,2,3,4,5] &= \{ \text{apply sum} \} \\
1 + \text{sum} [2,3,4,5] &= \{ \text{apply sum} \} \\
1 + (2 + \text{sum} [3,4,5]) &= \{ \text{apply sum} \} \\
1 + (2 + (3 + \text{sum} [4,5])) &= \{ \text{apply sum} \} \\
&\vdots \\
15 &= \{ \text{apply +} \}
\end{align*}
\]
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 \( 2 \times f(x) \)?

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.
• Imperative programs are not RT due to side effects.
• Consider the following C/Java function $f$:

```java
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

- 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.

```plaintext
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换成5
even’s definition,
[10/x]
Computation in FP

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

Can replace $f(x) + f(x)$ with $2 \cdot f(x)$
What’s so Good about FP?

• Referential Transparency and Equation Reasoning
• Executable Specification
• ...
Quick Sort in C

```c
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

McCarthy: Lisp  Landin: ISWIM  Steele: Scheme  Milner: ML  Backus: FP

Church: Lambda Calculus

Curry: Combinatory Logic
Background of Haskell
What is Haskell?

• 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/](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?

• 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> @p1 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)\n19:11 < autrijus> \n   Bool pre     system  (Str: List)\n19:11 < 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 :)
Haskell vs. Scheme/ML

- Haskell, like Lisp/Scheme, ML (Ocaml, Standard ML) and F#, is based on Church's lambda ($\lambda$) 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

Can Programming Be Liberated from the von Neumann Style?

1977 Turing Award Lecture

Late 1970s:

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

The Basics
Running Haskell Programs

- 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
  ```

  The Hugs > prompt means the Hugs system is ready to evaluate an expression.

  ![Diagram of Read Eval Print Loop](Read Eval Print Loop)
Hugs: a Haskell Interpreter

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
  - ...

06/30~07/04 FP & Types 33
Preparing Haskell Programs

• 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 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:

```haskell
> length [1,2,3,4]
4
```
The Standard Prelude ...

- Appending the elements of two lists:
  \[
  \text{> } [1,2,3] ++ [4,5,6] \\
  [1,2,3,4,5,6]
  \]

- Selecting the first element of a list:
  \[
  \text{> } \text{head} \ [1,2,3,4] \\
  1
  \]

- Removing the first element of a list:
  \[
  \text{> } \text{tail} \ [1,2,3,4] \\
  [2,3,4]
  \]
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 *.*

\[
f \ a \ b + c*d
\]
Function Application …

- 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
  \[ (f \ a) + b \quad \text{not} \quad f \ (a+b) \]
- since function application binds harder than plus.
Here’s a comparison between mathematical notations and Haskell:

<table>
<thead>
<tr>
<th>Math</th>
<th>Haskell</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f(x)$</td>
<td>$f \ x$</td>
</tr>
<tr>
<td>$f(x, y)$</td>
<td>$f \ x \ y$</td>
</tr>
<tr>
<td>$f(g(x))$</td>
<td>$f \ (g \ x)$</td>
</tr>
<tr>
<td>$f(x, g(y))$</td>
<td>$f \ x \ (g \ y)$</td>
</tr>
<tr>
<td>$f(x)g(y)$</td>
<td>$f \ x \ * \ g \ y$</td>
</tr>
</tbody>
</table>
A very simple functional program (also known as a *functional script*) in Haskell

- A set of definitions

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

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

main = print (square(smaller(5, 3+4)))
```
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:
  ```haskell```
  name :: type
  name = expression
  ```haskell```

- Examples:
  ```haskell```
  size :: Int
  size = (12+13)*4
  ```haskell```
Function Definitions

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

**average :: Float->Float->Float**

$$\text{average } x \ y = \frac{(x+y)}{2}$$

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

The **body** specifies how the result is computed. *No ‘return’*
Function Notation

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

Example:

\[
\text{square :: Float} \rightarrow \text{Float}
\]

\[
\text{square } x = x \times x
\]

Calls:

\[
\text{square } 2.5 \rightarrow 6.25
\]

\[
\text{Not square}(2.5)
\]

\[
\text{square } (1.2 + 1.3) \rightarrow 6.25
\]

Brackets are for grouping only!
### Simple Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>Unbounded integer numbers</td>
</tr>
<tr>
<td>Int</td>
<td>32-bit integer numbers</td>
</tr>
<tr>
<td>Rational</td>
<td>Unbounded rational numbers</td>
</tr>
<tr>
<td>Float, Double</td>
<td>Single- and double-precision floating point numbers</td>
</tr>
<tr>
<td>Bool</td>
<td>Boolean values: True and False</td>
</tr>
<tr>
<td>Char</td>
<td>Characters, e.g., 'a'</td>
</tr>
</tbody>
</table>
• **type** `Bool`

• **operations**

<table>
<thead>
<tr>
<th>&amp;&amp;</th>
<th>and</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>not</td>
<td>not</td>
</tr>
</tbody>
</table>

• `exOr :: Bool -> Bool -> Bool`

```haskell
exOr x y = (x || y) && not (x && y)
```
The integers

- **type Int**: range $-2147483648...2147483647$
- **type Integer**: range unbounded
- **operations**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>sum</td>
</tr>
<tr>
<td>*</td>
<td>product</td>
</tr>
<tr>
<td>^</td>
<td>raise to the power</td>
</tr>
<tr>
<td>-</td>
<td>difference</td>
</tr>
<tr>
<td>div</td>
<td>whole number division</td>
</tr>
<tr>
<td>mod</td>
<td>remainder</td>
</tr>
<tr>
<td>abs</td>
<td>absolute value</td>
</tr>
<tr>
<td>negate</td>
<td>change sign</td>
</tr>
</tbody>
</table>
Relational Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;</td>
<td>greater than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal to</td>
</tr>
<tr>
<td>==</td>
<td>equal to</td>
</tr>
<tr>
<td>/=</td>
<td>not equal to</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal to</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
</tr>
</tbody>
</table>

(==) 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

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

\[
\begin{align*}
> 4*12-6 \\
& 42 \\
> (<) 2 3 \\
& True \\
> \text{div} 126 3 \\
& 42 \\
> 126 \ '\text{div}' \ 3 \\
& 42
\end{align*}
\]
## Precedence & Associativity

<table>
<thead>
<tr>
<th>Op</th>
<th>Precedence</th>
<th>Associativity</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>^</td>
<td>8</td>
<td>right</td>
<td>Exponentiation</td>
</tr>
<tr>
<td>*, /</td>
<td>7</td>
<td>left</td>
<td>Mul, Div</td>
</tr>
<tr>
<td>'div'</td>
<td>7</td>
<td>free</td>
<td>Division</td>
</tr>
<tr>
<td>'rem'</td>
<td>7</td>
<td>free</td>
<td>Remainder</td>
</tr>
<tr>
<td>'mod'</td>
<td>7</td>
<td>free</td>
<td>Modulus</td>
</tr>
<tr>
<td>+, -</td>
<td>6</td>
<td>left</td>
<td>Add, Subtract</td>
</tr>
<tr>
<td>==, /=</td>
<td>4</td>
<td>free</td>
<td>(In-) Equality</td>
</tr>
<tr>
<td>&lt;, &lt;=, &gt;, &gt;=</td>
<td>4</td>
<td>free</td>
<td>Relational Comparison</td>
</tr>
</tbody>
</table>
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:

  toUpper `'a'` → `'A'`
  Ord `'a'` → 97
A list of values enclosed in square brackets.

\[ [1,2,3], [2] :: [\text{Int}] \]

A list of integers.

Some operations:

\[ [1,2,3] ++ [4,5] \rightarrow [1,2,3,4,5] \]

head \ [1,2,3] \rightarrow 1

last \ [1,2,3] \rightarrow 3

tail \ [1,2,3] \rightarrow [2,3]

We can have lists of lists:

\[ [ [1,3], [0, 5, 6], [4] ] :: [ [\text{Int}] ] \]
Quiz

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

\[
[1,2,3] ++ [4] \rightarrow [1,2,3,4]
\]

[4] not 4! ++ combines two lists, and 4 is not a list.
Types: Strings

Any characters enclosed in double quotes.

"Hello!" :: String

Some operations:

"Hello " ++ "World" → "Hello World"

First "Hello" → 'H'

List of Chars [Char]
Composite Types: Tuples

- 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 \textit{arity}.
- \textit{Arity} cannot be 1.
- The tuple of \textit{arity} zero () is called the \textit{empty tuple}, while a tuple of \textit{arity} 2 is called a \textit{pair}, one of \textit{arity} 3 a \textit{triple}, and so on.

Note that tuples are enclosed in parentheses, not square brackets.
You can have lists of tuples and tuples of lists

\[
[(1, \text{True}),(4, \text{False})] :: [(\text{Int, Bool})]
\]

\[
(1.4, [3, 5, 64, 7, 12], \text{True}) :: (\text{Float, [Int], Bool})
\]

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

• \( T_1 \rightarrow T_2 \) maps arguments of type \( T_1 \) to results of type \( T_2 \)

\(~ \rightarrow ~:: \) \( \text{Bool} \rightarrow \text{Bool} \)

\( \text{isDigit} :: \text{Char} \rightarrow \text{Bool} \)
A Note on Function Types

• Function types associate to right.

\[ \text{maxOf3} :: \text{Int} \to \text{Int} \to \text{Int} \to \text{Int} \]

means

\[ \text{maxOf3} :: \text{Int} \to (\text{Int} \to (\text{Int} \to \text{Int})) \]

• Functions are values, and \textit{partial application} is OK.

\[
\begin{aligned}
\text{let m = maxOf3 5} \\
\text{in let mm = m 8} \\
\text{in mm 12}
\end{aligned}
\]

\[
\rightarrow 12
\]
Multi-Parameter Functions

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

\[
\text{add} :: (\text{Int}, \text{Int}) \rightarrow \text{Int}\\
\text{add} (x, y) = x + y
\]

• Evaluate

\[
\text{add} (40, 2)
\]

• 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....
      - }
  - }


06/30--07/04  FP & Types  62
Function Definition by Cases and Recursion
The abs function

• The absolute value (abs) function:
  – abs x = |x|

• The definition is by cases (multiple equations):
  – abs x = \begin{cases} 
    x, & \text{if } x \geq 0 \\
    -x, & \text{if } x < 0 
  \end{cases}

• How to define in Haskell?

\[
\begin{align*}
\text{abs } x \mid x \geq 0 &= x \\
\text{abs } x \mid x < 0 &= -x
\end{align*}
\]

A guard. An equation is used if its guard is True.
Evaluating abs

Prelude> abs (-2)

- First equation, x = -2
- What is -2 \( \geq 0 \) ? \( \rightarrow \) False
- Second equation, x = -2 again
- What is -2 \( < 0 \) ? \( \rightarrow \) True
- Result is \(-x\), that is \(-(-2)\)

2

\[
\text{abs } x \mid x \geq 0 = x \\
\text{abs } x \mid x < 0 = -x
\]
Other Forms

• Fully explicit
  \[
  \text{abs } x \mid x \geq 0 = x \\
  \text{abs } x \mid x < 0 = -x
  \]

• Abbreviated left hand side
  \[
  \text{abs } x \mid x \geq 0 = x \\
  \mid x < 0 = -x
  \]

• Abbreviated last guard
  \[
  \text{abs } x \mid x \geq 0 = x \\
  \mid \text{otherwise } = -x
  \]

• "if" expression
  \[
  \text{abs } x = \\
  \quad \text{if } x \geq 0 \text{ then } x \text{ else } -x
  \]
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

\[
\begin{align*}
\text{fun} & \ v1\ v2\ \ldots\ vn \\
\quad & \mid g1 = e1 \\
\quad & \mid g2 = e2 \\
\quad & \quad \quad \ldots \\
\quad & \mid \text{otherwise} = er
\end{align*}
\]

\[
\begin{align*}
\text{fun} & \ v1\ v2\ \ldots\ vn = \\
& \quad \text{if}\ g1\ \text{then}\ e1 \\
& \quad \quad \text{else if}\ g2\ \text{then}\ e2 \\
& \quad \quad \quad \ldots \\
& \quad \text{else}\ e_r
\end{align*}
\]

\[
\begin{align*}
\text{max3} & :: \text{Int} \to \text{Int} \to \text{Int} \to \text{Int} \\
\text{max3} & \ i\ j\ k = \\
& \quad \text{if}\ \ (i \geq j)\ \&\&\ (i \geq k)\ \text{then}\ i \\
& \quad \quad \text{else if}\ \ (j \geq k)\ \text{then}\ j \\
& \quad \quad \quad \text{else}\ k
\end{align*}
\]
Recursive Functions

\[ \text{fac } n = 1 \times 2 \times \ldots \times n \]

\[
\text{fac} :: \text{Int} \to \text{Int} \\
\text{fac } n \\
\quad | \ n == 0 = 1 \\
\quad | \ \text{otherwise} = \text{fac} \ (n-1) \times n
\]

or

\[
\text{fac} :: \text{Int} \to \text{Int} \\
\text{fac } n = \text{if } n == 0 \ \text{then} \ 1 \\
\quad \text{else} \ \text{fac} \ (n-1) \times n
\]
Evaluating Factorials

fac :: Int → Int

fac 0 = 1

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

fac 4 ?? 4 == 0 → False

?? 4 > 0 → True

fac (4-1) * 4

fac 3 * 4

fac 2 * 3 * 4

fac 1 * 2 * 3 * 4

fac 0 * 1 * 2 * 3 * 4

1 * 1 * 2 * 3 * 4

24
Expensive to calculate...

\[
\text{fac 5} \\
5 \times (\text{fac 4}) \\
5 \times 4 \times (\text{fac 3}) \\
5 \times 4 \times 3 \times (\text{fac 2}) \\
5 \times 4 \times 3 \times 2 \times (\text{fac 1}) \\
5 \times 4 \times 3 \times 2 \times 1 \times (\text{fac 0}) \\
5 \times 4 \times 3 \times 2 \times 1 \times 1
\]

\[
\text{fac 0} = 1 \\
\text{fac n \mid n > 0} = n \times \text{fac (n-1)}
\]
Tail Recursion

• 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 \rightarrow tailfac 5 1
    \rightarrow tailfac 4 5*1
    \rightarrow tailfac 3 4*5*1
    ...```

06/30--07/04 FP & Types 72

FLOLAC 2008
A Better Process: Tail Recursion

Tail recursion is logically equivalent to a loop!

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

120

Stack

Time

06/30~07/04

FP & Types 73
Local Definitions: the where clause

•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

\[
f :: [\text{Int}] \rightarrow [\text{Int}]
f \ [ \ ] = [ \ ]
f \text{xs} = \begin{array}{l}
\text{let} \\
\text{square} \ a = a \ast a \\
\text{one} = 1 \\
\text{in} \\
(s\text{quare} \ (\text{head} \ \text{xs}) + \text{one}) : f \ (\text{tail} \ \text{xs})
\end{array}
\]

\[
f \ [3,2] \\
\Rightarrow (\text{square} \ 3 + \text{one}) : f \ [2] \Rightarrow \ldots \Rightarrow [10,5]
\]
The Layout Rule

Indentation determines where a definition ends:

circumference $r = 2 \times \pi \times r$

area $r$
  $= \pi \times r \times r$

bad $x = \text{area } x + \text{circumference } x$  -- Error: offside!
Example

\[
\begin{align*}
\text{let} & \quad y = x + 2 \\
& \quad x = 5 \\
\text{in} & \quad x / y
\end{align*}
\]

• same as:

\[
\text{let } y = \{x + 2; x = 5\} \text{ in } x / y
\]
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
```

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

Implicit grouping means 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

\[ 1 \quad 1 \quad 2 \quad 3 \quad 5 \quad 8 \quad 13 \quad 21 \ldots \]
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

Repetitive computation

Rewrite it as tail recursive!
Pattern Matching
Pattern Matching

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

$$~ :: \text{Bool} \rightarrow \text{Bool}$$

\[
\begin{align*}
~ \text{False} & = \text{True} \\
~ \text{True} & = \text{False}
\end{align*}
\]

Constant pattern; order matters
Pattern Matching

• We can also use pattern matching for functions that take more than one argument

• The library function (&&) returns the negation of a logical value

\[
\text{(&&)} :: \text{Bool} \to \text{Bool} \to \text{Bool}
\]

| \(\text{True} \&\& \text{True}\) | = | \text{True} |
| \(\text{True} \&\& \text{False}\) | = | \text{False} |
| \(\text{False} \&\& \text{True}\) | = | \text{False} |
| \(\text{False} \&\& \text{False}\) | = | \text{False} |
Pattern Matching

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

\[
&& : \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool}
\]

\[
\begin{array}{c c c}
\text{True} & \&& \text{True} & = & \text{True} \\
_ & \&& _ & = & \text{False}
\end{array}
\]

• This is also good because if the first argument is *False* then it doesn’t need to evaluate the second argument
• Haskell has a naming convention that means that we *cannot use the same variable name* for more than one argument in an equation, so

\[
\begin{align*}
  b \land \land b & = b \\
  \_ \land \land \_ & = False
\end{align*}
\]

would not be allowed, and needs to be rewritten as

\[
\begin{align*}
  b \land \land c & | b==c = b \\
  | otherwise = False
\end{align*}
\]
Tuple Patterns

• A tuple of patterns is itself a pattern which matches any tuple of the same arity whose components match the corresponding patterns \textit{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 \( \texttt{fst} \) and \( \texttt{snd} \) select the first and second components of a pair

<table>
<thead>
<tr>
<th>Function</th>
<th>Type</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \texttt{fst} )</td>
<td>((a,b) \rightarrow a)</td>
<td>( \texttt{fst} (x,_) = x )</td>
</tr>
<tr>
<td>( \texttt{snd} )</td>
<td>((a,b) \rightarrow a)</td>
<td>( \texttt{snd} (_,y) = y )</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\texttt{fst} (5, 'a') & \rightarrow 5 \quad -- (x \text{ binds to } 5) \\
\texttt{snd} (5, 'a') & \rightarrow 'a' \quad -- (y \text{ binds to } 'a')
\end{align*}
\]
More Selector Functions

• For pairs, we have
  \[ \text{fst} (x, y) = x \quad \text{snd} (x, y) = y \]

• For triples, we define
  \[ \text{fst3} (x, y, z) = x \]
  \[ \text{snd3} (x, y, z) = y \]
  \[ \text{trd3} (x, y, z) = z \]

• No general selectors such as:
  \[ \text{select 3} (x, y, z) = z \]

What would the type of the result be?
Selection using Pattern Matching

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

\[(x_1, x_2, x_3) = \text{a\_triple\_value}\]

Example:

\[(x_1, x_2, x_3) = (100, 'A', "Math")\]

Then \(x_1=100, \ x_2='A', \ x_3="Math"\).
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 precisely three characters and the first of these is the letter ‘a’

\[
\text{test} :: \text{[Char]} \rightarrow \text{Bool} \\
\text{test} [‘a’,_,_] = \text{True} \\
\text{test} _ _ _ = \text{False}
\]
List Patterns

- 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:

\[
\begin{align*}
[3,5,7] &= \{ \text{apply cons} \} \\
3 : [5,7] &= \{ \text{apply cons} \} \\
3 : (5 : [7]) &= \{ \text{apply cons} \} \\
3 : (5 : (7 : [])) &= \{ \text{apply cons} \}
\end{align*}
\]

- `cons` associates to the right:

\[
3 : 5 : 7 : []
\]
• 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

```hs
test :: [Char] -> Bool

test ('a':_) = True

test _ = False
```
Defining Functions with List Patterns

- *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

Head  Tail

2
3  []

1 : [2, 3]

2 : 3 : []  or  [2, 3]

1 : 2 : 3 : []  or  [1, 2, 3]
Lists are Homogenous

- Lists of lists:
  
  $[1]:[[2],[3]] \Rightarrow [[1],[2],[3]]$

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

  $[1, [1], 1] \Rightarrow \text{Illegal!}$

  $[[1], [2], [[3]]] \Rightarrow \text{Illegal!}$

  $[1, \text{True}] \Rightarrow \text{Illegal!}$
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

\[
\begin{align*}
\text{pred} & \;::\; \text{Int} \rightarrow \text{Int} \\
\text{pred} \;0 & \;=\; 0 \\
\text{pred} \;(n+1) & \;=\; n
\end{align*}
\]
Recursion over Lists

• 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

\[
\text{length} \ [\] = 0
\]
\[
\text{length} \ (x : xs) = 1 + \text{length} \ xs
\]

\[
\text{length} \ (1 : 2 : 4 : []) \\
\Rightarrow \ [ \ x \leftarrow 1 , \ xs \leftarrow 2 : 4 : [] ] \\
1 + \text{length} \ (2 : 4 : [])
\]
Evaluating Recursive Functions

\[
\begin{align*}
\text{length } [] &= 0 \\
\text{length } (x : xs) &= 1 + (\text{length } xs)
\end{align*}
\]

\[
\begin{align*}
\text{length } (1 : 2 : 4 : []) \\
\Rightarrow & \quad [x \leftarrow 1, \ x is \leftarrow 2 : 4 : []] \\
& \quad 1 + \text{length } (2 : 4 : []) \\
\Rightarrow & \quad [x \leftarrow 2, \ x is \leftarrow 4 : []] \\
& \quad 1 + 1 + \text{length } (4 : []) \\
\Rightarrow & \quad [x \leftarrow 4, \ x is \leftarrow []] \\
& \quad 1 + 1 + 1 + \text{length } [] \\
\Rightarrow & \quad [] \\
& \quad 1 + 1 + 1 + 0
\end{align*}
\]
• The \texttt{length} function does not care about the \textit{element type} of its list parameter.

\begin{align*}
\text{length} & \ [1, 2, 3] \quad \Rightarrow \quad 3 \\
\text{length} & \ [\text{True, False}] \quad \Rightarrow \quad 2 \\
\text{length} & \ [\text{‘a’, ‘b’, ‘c’, ‘d’}] \quad \Rightarrow \quad 4
\end{align*}

• Indeed, \texttt{length} is a polymorphic function, and its type is:

\begin{center}
\texttt{length :: [a] -> Int}
\end{center}

Here \texttt{a} is a \textit{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

- 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 `:type expression` asks Haskell to
- print the type of an expression:
  ```haskell
  > :type "hello"
  "hello" :: String
  > :type True && False
  True && False :: Bool
  > :t True && False :: Bool
  True && False :: Bool
  ```
Exercise

• 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

- $x, y, z, \cdots$ denote list elements.
- $x_s, y_s, z_s, \cdots$ denote lists of elements.
- $x_{ss}, y_{ss}, z_{ss}, \cdots$ denote lists of lists of elements.
List Concatenation

- $\text{xs} \: \text{++} \: \text{ys} \quad --\text{also known as} \quad \text{append} \: \text{xs} \: \text{ys}$

\[
\text{(+ +)} :: [a] \rightarrow [a] \rightarrow [a] \\
[] \: \text{++} \: \text{ys} = \text{ys} \\
(x : \: \text{xs}) \: \text{++} \: \text{ys} = x : (\text{xs} \: \text{++} \: \text{ys})
\]

\[
[1, 2, 3] \: \text{++} \: [4, 5, 6] \\
\qquad = \{ \text{apply} \: \text{++} \} \\
1: ([2, 3] \: \text{++} \: [4, 5, 6]) \\
\qquad = \{ \text{apply} \: \text{++} \} \\
1: (2: ([3] \: \text{++} \: [4, 5, 6])) \\
\ldots \\
1: (2: (3: [4, 5, 6]))) \\
\qquad = \{ \text{list notation} \} \\
[1, 2, 3, 4, 5, 6]
\]
List Concatenation

- Concatenate multiple lists in a list:

  \[
  \text{concat} :: [[a]] \rightarrow [a] \\
  \text{concat} [] = [] \\
  \text{concat} (xs:xss) = xs ++ \text{concat} xss
  \]

Examples:

  \[
  \text{concat} [] = [] \\
  \text{concat} [[]] = [] \\
  \text{concat} [[1], [3,5]] = [1,3,5]
  \]
More Polymorphic Recursive
List Functions: reverse

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

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

Example

```haskell
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

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

- 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:
  – \( \text{drop} \ 2 \ [1,2,3,4,5] = [3,4,5] \)

\[
drop :: Int \to [a] \to [a]
\]

• Defining the *init* function:
  – \( \text{init} \ [1,2,3,4,5] = [1,2,3,4] \) --remove the last element

\[
\text{init} :: [a] \to [a]
\]
Mutual Recursion

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

```plaintext

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

\[
\begin{align*}
evens \ "abcde" &= \{ \text{apply } evens \} \\
'a' : odds \ "bcde" &= \{ \text{apply } odds \} \\
'a' : evens \ "cde" &= \{ \text{apply } evens \} \\
'a' : 'c' : odds \ "de" &= \{ \text{apply } odds \} \\
'a' : 'c' : evens \ "e" &= \{ \text{apply } evens \} \\
\ldots
\end{align*}
\]
Mutual Recursion

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

\[
evens :: [a] \rightarrow [a] \\
evens [] = [] \\
evens (x : xs) = x : odds xs
\]

\[
odds :: [a] \rightarrow [a] \\
odds [] = [] \\
odds (_ : xs) = evens xs
\]
Arithmetic Sequences

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

\[ [1..3] \Rightarrow [1, 2, 3] \]
\[ [5..1] \Rightarrow [] \]

• A similar notation is used when the difference between consecutive elements is \( = 1 \): Examples:

\[ [1, 3..9] \Rightarrow [1, 3, 5, 7, 9] \]
\[ [9, 8..5] \Rightarrow [9, 8, 7, 6, 5] \]
\[ [9, 8..11] \Rightarrow [] \]

Or, in general:

\[ [m, k..n] \Rightarrow [m, m+(k-m)*1, m+(k-m)*2, \ldots, n] \]
List Comprehension

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

- Mathematical form
  \[ \{ x^2 \mid x \in \{1..5\} \} \]
  produces the set \{1,4,9,16,25\}

- Haskell
  \[
  > [ x^2 \mid x <- [1..5] ] \\
  [1,4,9,16,25]
  \]
  where
  | means “such that”
  <- means “is drawn from”; “for each element in”
Generators

- The expression \( x \leftarrow [1..5] \) is called a generator.
- Generators can also use patterns when drawing elements from a list.

Suppose \( ps \) is a list of pairs:

\[
[(1, \text{True}), (2, \text{False}), (5, \text{False}), (9, \text{True})]
\]

If we want to extract all pairs of the form \((x, \text{True})\) then we can do this using the generator:

\[
> [ x | (x, \text{True}) \leftarrow ps ]
\]

\[ [1,9] \]
Generators

- We can also use wildcards in generators
- If we take the same list of pair \( ps \)
  \[
  [(1,\text{True}), (2,\text{False}), (5,\text{False}), (9,\text{True})]
  \]
  then

  \[
  \text{\textgreater{} [ x | (x,\_)<-ps ]}
  \]

  \[
  [1,2,5,9]
  \]

  extracts the list of the first components of the pairs
Generators

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

```haskell
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

\[
\left\{ (x, y) \mid x \leftarrow [1, 2], \ y \leftarrow [8, 9] \right\}
\]

\[
[ (1, 8), (1, 9), (2, 8), (2, 9) ]
\]
• The second generator cycles faster than the first generator.
• Swap the order:

\[
\left\{ (x, y) \mid y \leftarrow [1, 2], \ x \leftarrow [8, 9] \right\}
\]
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 \textit{concat}, which concatenates lists, by using one generator to select each list then a second generator to select each element within the list:

\[
\text{concat} :: [[a]] \rightarrow [a] \\
\text{concat } xss = [x \mid xs<-xss, x<-xs]
\]
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

\[
\begin{align*}
\text{prime} & \quad :: \quad \text{Int} \rightarrow \text{Bool} \\
\text{prime } n & \quad = \quad \text{length } (\text{factors } n \quad == \quad 2)
\end{align*}
\]

So

\[
\begin{align*}
> \text{prime 15} & \quad > \text{prime 7} \\
\text{False} & \quad \text{True}
\end{align*}
\]
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

• 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*

\[ \text{P} \rightarrow \text{Q} \rightarrow \text{R} \rightarrow \text{S} \]
Representing a Point

**type** Point = (Float, Float)

**distance ::** Point -> Point -> Float

distance (x, y) (x', y') =

\[ \sqrt{(x-x')^2 + (y-y')^2} \]
Representing a Path

type Path = [Point]

eexamplePath = [p, q, r, s]

path_length = distance p q + distance q r + distance r s
Two Useful Functions

• `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 \textit{pathLength Function}

\begin{verbatim}
pathLength :: Path -> Float
pathLength xs = sum [ distance p q |
    (p,q) <- zip (init xs) (tail xs)]
\end{verbatim}

\textbf{Example:}

\begin{verbatim}
pathLength [p, q, r, s] ⇒
    distance p q + distance q r + distance r s
\end{verbatim}
Higher-Order Functions

- Functions take functions as *arguments*
- Functional values and Lambda Expressions
- Functions return functions as *results*.
Write a Haskell function `incAll` that adds 1 to each element in a list of numbers.

E.g., \( \text{incAll } [1, 3, 5, 9] = [2, 4, 6, 10] \)

\[
\begin{align*}
\text{incAll} & : [\text{Int}] \rightarrow [\text{Int}] \\
\text{incAll } [] &= [] \\
\text{incAll } (n : ns) &= n+1 : \text{incAll } ns
\end{align*}
\]
A Motivating Example, cont’d

• Write a Haskell function \texttt{lengths} that computes the lengths of each list in a \textit{list of lists}.

E.g.,
\[
\text{lengths } [[1,3], [], [5, 9]] = [2, 0, 2] \\
\text{lengths } ["but", "and", "if"] = [3, 3, 2]
\]

\[
\text{lengths} :: [[a]] \rightarrow [\text{num}] \\
\text{lengths } [] = [] \\
\text{lengths } (l : ls) \\
\quad = (\text{length } l) : \text{lengths } ls
\]
### Similarity and Abstraction

**incAll**

\[
\text{incAll} \; [] = [] \\
\text{incAll} \; (n : \text{ns}) = (+) \; n \; 1 : \text{incAll} \; \text{ns}
\]

**lengths**

\[
\text{lengths} \; (l : \text{ls}) = \text{length} \; l : \text{lengths} \; \text{ls} \\
\text{lengths} \; [] = []
\]

Let \( f \) be \(+\) or \( \text{length} \):

\[
f \; (\text{hd} \; l) : \text{recCall} \; (\text{tail} \; l)
\]

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

\[
[f \; l_1, f \; l_2, \ldots, f \; l_n]
\]
**List map function**

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

\[
\text{map} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b] \\
\text{map } f \ [\ ] = [\ ] \\
\text{map } f \ (x : xs) = (f \ x) : \text{map } f \ xs
\]

\[
\text{map } f \\
[l_1, \ l_2, \ ..., l_n] \quad \rightarrow \quad [f \ l_1, \ f \ l_2, \ ... \ f \ l_n]
\]
Using \textit{map}

\[
\text{map} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]
\]

\[
\text{incAll} = \text{map} \ (\text{plus} \ 1)
\]
\[
\quad \text{where} \ \text{plus} \ m \ n = m + n
\]

\[
\text{lengths} = \text{map} \ (\text{length})
\]

Note that \text{plus} :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int},

so

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

Functions of this kind are sometimes referred to as \textit{partially evaluated (applied)}. 
Any function may be called with fewer arguments than it was defined with.

The result is a function of the remaining arguments.

If \( f :: \text{Int} \rightarrow \text{Bool} \rightarrow \text{Int} \rightarrow \text{Bool} \)

then \( f \ 3 :: \text{Bool} \rightarrow \text{Int} \rightarrow \text{Bool} \)

\[ f \ 3 \ \text{True} :: \text{Int} \rightarrow \text{Bool} \]

\[ f \ 3 \ \text{True} \ 4 :: \text{Bool} \]
We say function application “brackets to the left” 
function types “bracket to the right”

If \( f :: \text{Int} \to (\text{Bool} \to (\text{Int} \to \text{Bool})) \)
then \( f \ 3 :: \text{Bool} \to (\text{Int} \to \text{Bool}) \)

\((f \ 3) \ True :: \text{Int} \to \text{Bool} \)

\(((f \ 3) \ True) \ 4 :: \text{Bool} \)

Functions really take only one argument, and return a function expecting more as a result.
Another HoF: List filtering

\[
\text{filtr} :: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]
\]

if \( p? \) \( w \), send \( w \) to output

\[
\begin{align*}
  a, b, c, \ldots, z & \quad \rightarrow \quad p? \\
  a', b', c' & \ldots
\end{align*}
\]

\[
\text{filter even \ [1,2,3,4,6] } = [2,4,6]
\]

\[
\text{even } x = x \ '\text{mod'} \ 2 \ == \ 0
\]
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 calculated from the arguments

• These are shown in Haskell using \ or mathematically using \( \lambda \)

  Example: \( \mathbf{x} \rightarrow (\mathbf{x}, \mathbf{x}, \mathbf{x}) \)

  \( \mathbf{\backslash \text{parameter}} \rightarrow \text{body} \)
Lambda Expressions

- The *square* function could also be implemented as a lambda expression:
  \[ x \rightarrow x \times x \]

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

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

  ```haskell
  filter (\x -> x `mod` 2 == 0) [2,3,5,6,7]
  [1,2,4]
  
  -> has lowest precedence, extends to the right
  ```
Lambda Expressions

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

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

can be understood as

\[ \text{add} = \lambda x \to (\lambda y \to x + y) \]

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
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

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]
• Another view of *partial application*: functions returning functions. Example:

• `makeAdder n`: creates a function add `n` to its argument

```haskell
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 \((A,B) \rightarrow C\) and \(A \rightarrow (B \rightarrow C)\).

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

\[
\text{curriedF} :: A \rightarrow B \rightarrow C
\]

\[
\text{curriedF } a \ b = f \ (a,b)
\]
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. \textit{product}: multiplies all the elements in a list of numbers together.
\[
\text{product } [2,5,26,14] = 2*5*26*14 = 3640
\]

\[
\text{product :: [Int] } \rightarrow \text{ Int}\\
\text{product } [] = 1\\
\text{product } (n : ns) = n * \text{ product } ns
\]

2. \textit{concat}: Concatenate multiple lists
\[
\text{concat } [[2,5], [], [26,14]] = [2,5,26,14]
\]

\[
\text{concat :: [[a]] } \rightarrow \text{ [a]}\\
\text{concat } [] = []\\
\text{concat } (xs:xss) = xs ++ xss
\]
A general pattern for the functions \texttt{product} and \texttt{concat} is \textit{replacing constructors with operators}. For example, \texttt{product} replaces : (cons) with * and [] with 1:

\[
1 : (2 : (3 : (4 : [])))
\]

\[
1 * (2 * (3 * (4 * 1)))
\]

\textbullet \ \texttt{concat} replaces : (cons) with ++ and [] with []:

\[
\]

\[
[2,5] ++ ([] ++ ([3,4] ++ []))
\]
Haskell has a built-in function, `foldr`, that does this replacement:

\[
\text{foldr} :: (a \to b \to b) \to b \to [a] \to b
\]

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

\[
(*) \ 1 \ (2 \ast (3 \ast (4 \ast 1)))
\]

\[
\text{product} = \text{foldr} \ (*) \ 1\\
\text{concat} = \text{foldr} \ (++) \ []
\]
Visualizing `foldr`

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

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

Example:

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

Another direction to fold: \texttt{foldl}:

\[
\text{foldl} :: (b \to a \to b) \to b \to [a] \to b
\]

\[
\text{foldl} \ f \ e \ [] = e \\
\text{foldl} \ f \ e \ (x : xs) = \text{foldl} \ f \ (f \ e \ x) \ xs
\]

- \textbf{product} = \text{foldl} \ (*) \ 1
- \textbf{concat} = \text{foldl} \ (++) \ []

\[
\text{foldl} \ \text{max} \ 0 \ [1,2,3] = 3
\]

\textbf{where} \ \text{max} \ a \ b = \text{if} \ a > b \ \text{then} \ a \ \text{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 \textit{foldr}

\begin{align*}
\text{reverser} & :: [a] \rightarrow [a] \\
\text{reverser} &= \text{foldr } \text{snoc } [] \\
\text{where } \text{snoc } x \ x & x s = x s \ ++ \ [x] \quad \text{--O} (N^2) \\
\end{align*}

\begin{itemize}
\item Add ‘x’ to the end of xs
\end{itemize}
Reversing a list using \textit{foldl}

\begin{verbatim}
reversel :: [a] -> [a]
reversel = foldl cons []
where cons xs x = x : xs
\end{verbatim}

\[\text{add a to the front of []}\]
Specialized fold

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

\[ \sum_{k=1}^{100} k = \frac{100 \times 101}{2} \]
\[ \sum_{k=1}^{100} k^2 = \frac{100 \times 101 \times 102}{6} \]
\[ \sum_{k=1, \text{odd}}^{101} \frac{1}{k^2} = \frac{\pi^2}{8} \]

In mathematics they are all captured by the notion of a \textit{sum}:

\[ \sum_{x \in I} f(x) \]

Can we express this abstraction directly?
Look at the three functions

\[ \sum_{k=1}^{100} k = \text{sum-integers } 1 \ 100 \]

\[ \sum_{k=1}^{100} k^2 = \text{sum-squares } 1 \ 100 \]

\[ \sum_{k=1, \text{odd}}^{101} k^{-2} = \text{pi-sum } 1 \ 101 \]

\[
\text{sumIntegers } k \ n = \\
\quad \text{if } k > n \text{ then } 0 \text{ else } \\
\quad k + (\text{sum-integers } (k+1) \ n)
\]

\[
\text{sumSquares } k \ n = \\
\quad \text{if } k > n \text{ then } 0 \text{ else } \\
\quad (\text{square } k) + (\text{sum-squares } (k+1) \ n)
\]

\[
\text{piSum } k \ n = \\
\quad \text{if } k > n \text{ then } 0 \text{ else } \\
\quad (1/(\text{square } k)) + (\text{pi-sum } (k+2) \ n)
\]
Abstraction from the three functions

\[ \sum_{x \in \mathbb{I}} f(x) \]

\[ \text{sum } f \_	ext{next } k n = \]
\[ \text{if } k > n \text{ then } 0 \text{ else } (f k) + \]  
\[ \text{sum } f \_	ext{next } (\text{next } k) n \]

• sumIntegers = \( \text{sum } (\lambda x \rightarrow x) (+1) \)

• sumSquares = \( \text{sum } (\lambda x \rightarrow x^2) (+1) \)

• piSum = \( \text{sum } \text{si} (+2) \)
  where \( \text{si } x = 1/(x \times x) \)

\[ \text{id } x = x \]

sumIntegers k n =
\[ \text{if } k > n \text{ then } 0 \text{ else } \]
\[ k + (\text{sum-integers } (k+1) n) \]

sumSquares k n =
\[ \text{if } k > n \text{ then } 0 \text{ else } \]
\[ (\text{square } k) + (\text{sum-squares } (k+1) n) \]

piSum k n =
\[ \text{if } k > n \text{ then } 0 \text{ else } \]
\[ (1/(\text{square } k)) + (\text{pi-sum } (k+2) n) \]
Using \textit{map} and \textit{reduce}

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

\[
\text{sum } f \ l = \text{foldl } (+) \ 0 \ (\text{map } f \ l)
\]

E.g.,
\[
\sum(x): \quad > \text{sum } (\lambda x \rightarrow x) \ [1, 2, 3] \\
\text{value: } 6
\]
\[
\sum(x^2): \quad > \text{sum } (\lambda x \rightarrow x \times x) \ [1, 2, 3] \\
\text{value: } 14
\]
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 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)
```

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

\[
\text{count} \ 2 \ [[1],[],[2,3],[4,5],[]] = 2
\]

Using recursion:

\[
\text{count} :: \text{Int} \rightarrow [[a]] \rightarrow \text{Int}\\
\text{count} \ [] = 0\\
\text{count} \ n \ (x:xs) \\
\quad | \ \text{length} \ x == n = 1 + \text{count} \ n \ xs\\n\quad | \ \text{otherwise} = \text{count} \ n \ xs
\]

Using functional composition:

\[
\text{count'} \ n = \text{length} \ . \ \text{filter} \ (==n) \ . \ \text{map} \ \text{length}
\]
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 \textit{parameters}. For example, given

\[
\text{type Pair } a = (a,a)
\]

we can define:

\[
\begin{align*}
\text{bits} &::= \text{Pair Int} \\
\text{bits} &= (0,1) \\
\text{copy} &::= a \rightarrow \text{Pair } a \\
\text{copy } x &= (x,x)
\end{align*}
\]

- Type declarations can be \textit{nested}:

\[
\begin{align*}
type \text{Pos} &= (\text{Int},\text{Int}) \\
type \text{Trans} &= \text{Pos} \rightarrow \text{Pos}
\end{align*}
\]

- However, they \textit{cannot be} recursive:

\[
\begin{align*}
type \text{Tree} &= (\text{Int},[\text{Tree}])
\end{align*}
\]
Defining New Types

- Enumerated
  
  ```haskell
  data Bool = False | True
  ```

- Parameterized (polymorphic)
  
  ```haskell
  data Maybe a = Nothing | Just a
  ```

- Recursive
  
  ```haskell
  Data List a = Nil | Cons a (List a)
  ```
Enumerated

Example:

```haskell
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

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

we can define:

```haskell
answers :: [Answer]
answers = [Yes, No, Unknown]
flip :: Answer -> Answer
flip Yes = No
flip No = Yes
flip Unknown = Unknown
```
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 \texttt{Float} \\
\hspace{1cm} \mid \hspace{1cm} \texttt{Rect Float Float}

- \textit{Shape} has values of the form \texttt{Circle} \hspace{1mm} r \hspace{1mm} \textit{where} \hspace{1mm} r \hspace{1mm} \textit{is a float}, and \hspace{1mm} \texttt{Rect} \hspace{1mm} x \hspace{1mm} y \hspace{1mm} \textit{where} \hspace{1mm} x \hspace{1mm} \text{and} \hspace{1mm} y \hspace{1mm} \textit{are floats}.

- \texttt{Circle} and \texttt{Rect} can be viewed as \underline{functions} that simply construct values of type \textit{Shape}:

\begin{align*}
\texttt{Circle} & : : \texttt{Float} \rightarrow \texttt{Shape} \\
\texttt{Rect} & : : \texttt{Float} \rightarrow \texttt{Float} \rightarrow \texttt{Shape}
\end{align*}
Not surprisingly, data declarations themselves can also have parameters. For example, given

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

we can define:

```haskell
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:
Modeling Arithmetic Expressions

\[ 1 + (2 \times 3) \]

Diagram:}

```
+  
/  
1   *  
   /  
  2   3
```
Arithmetic Expressions

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

\[
\text{data Expr} = \text{Val \ Int} \\
| \text{Add Expr Expr} \\
| \text{Mul Expr Expr}
\]

• So the tree for \(1 + 2 \times 3\) could be represented as

\[
\text{Add (Val 1) (Mul (Val 2) (Val 3))}
\]
Arithmetic Expressions

- We can define recursive functions to process expressions

\[
\begin{align*}
\text{size} & :: \text{Expr} \rightarrow \text{Int} \\
\text{size} \ (\text{Val} \ n) & = 1 \\
\text{size} \ (\text{Add} \ x \ y) & = \text{size} \ x + \text{size} \ y \\
\text{eval} & :: \text{Expr} \rightarrow \text{Int} \\
\text{eval} \ (\text{Val} \ n) & = n \\
\text{eval} \ (\text{Add} \ x \ y) & = \text{eval} \ x + \text{eval} \ y \\
\text{eval} \ (\text{Mul} \ x \ y) & = \text{eval} \ x \times \text{eval} \ y
\end{align*}
\]
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:

```haskell
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

\[
\begin{align*}
\text{flatten} & \quad : \text{Tree} \to [\text{Int}] \\
\text{flatten} \ (\text{Leaf} \ n) & = [n] \\
\text{flatten} \ (\text{Node} \ l \ n \ r) & = \text{flatten} \ l \\
& \quad + \ [n] \\
& \quad + \ \text{flatten} \ r
\end{align*}
\]

• If the tree flattens to an ordered list then the tree is a *search tree*

• Our example flattens to \([1,3,4,5,6,9]\)
We can define a function \( \text{find} \) that decides if a given integer occurs in a binary tree:

\[
\begin{align*}
\text{find} & : \text{Int} \rightarrow \text{Tree} \rightarrow \text{Bool} \\
\text{find} \ x \ (\text{Leaf} \ n) & = x == n \\
\text{find} \ x \ (\text{Node} \ l \ n \ r) & = x == n \\
& \quad \text{|| find} \ x \ l \\
& \quad \text{|| find} \ x \ r
\end{align*}
\]

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

\[
\begin{align*}
\text{find } x \ (\text{Leaf } n) &= x==n \\
\text{find } x \ (\text{Node } l \ n \ r) &= x==n = \text{True} \\
&\quad | \ x<n = \text{find } x \ l \\
&\quad | \ x>n = \text{find } x \ r
\end{align*}
\]

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

\[
\text{myFst} \ (x, \ 0) = x \\
\text{myFst} \ (x, \ y) = x
\]

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:

\[
\text{reverse} \ (1 : ((f \ 2) : [])) \quad \rightarrow [1, f \ 2]
\]
reverse \( n : ns \) = snoc \( n \) (reverse \( ns \))

snoc \( h \) \( tl \) = \( tl \) ++ \([h]\)

reverse \( 1 : ((f \ 2) : []) \)
⇒
(snoc 1 (reverse ((f \ 2) : [])))
⇒
(reverse ((f \ 2) : [])) ++ \([1]\)
⇒
((snoc (f \ 2) (reverse [])) ++ \([1]\))
⇒
((reverse [])) ++ \([(f \ 2)]\)) ++ \([1]\)
Topmost-Outermost

\[
((\text{reverse} \ []) \ ++ [(f \ 2)]) \ ++ \ [1]
\]
\[\Rightarrow\]
\[
([] \ ++ [(f \ 2)]) \ ++ \ [1]
\]
\[\Rightarrow\]
\[
[(f \ 2)] \ ++ \ [1]
\]
\[\Rightarrow\]
\[
[(f \ 2),1]
\]

\((f \ 2)\) 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, ... 
... 
2918,2919,2919<<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

The built-in function \texttt{takeWhile} returns the longest initial segment that satisfies a property \texttt{p}:

\begin{verbatim}
takeWhile :: (a -> Bool) -> [a] -> [a]
takeWhile p [] = []
takeWhile p (x : xs)
    | p x        = x : takeWhile p xs
    | otherwise  = []
\end{verbatim}

Main> \texttt{takeWhile (<10) [1, 2, 13, 3]}
[1,2]
Note that evaluation of \texttt{takeWhile} stops as soon as the given property doesn't hold, whereas evaluation of \texttt{filter} only stops when the end of the list is reached:

\begin{verbatim}
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]
\end{verbatim}

\texttt{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:

\[
\text{natsfrom} :: \text{num} \rightarrow \text{[num]}
\]
\[
\text{natsfrom} \ n = n : \text{natsfrom} \ (n+1)
\]

This function can be invoked in the usual way:

Main> \text{natsfrom} \ 0
[0,1,2,3,...] \hspace{1cm} \text{ERROR!}

Main> \text{take} \ 3 \ (\text{natsfrom} \ 0)
[0,1,2]
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", ",", ",", ...
Suggested Reading


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 $\lambda$ Calculus

The foundation of all FP languages.
The $\lambda$-Calculus

The $\lambda$-calculus was developed by the logician Alonzo Church in 1930’s as a tool to study functions and computability.
\(\lambda\)-calculus in Computer Science

- Computability
  - \(\lambda\)-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 $\lambda$-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

**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 pure lambda calculus (LC) is a *untyped* language composed entirely of functions
• The simply typed lambda calculus (SLC)
• The polymorphic typed lambda calculus (PLC)
• …
Pure Untyped $\lambda$-calculus

- Syntax is simple: $\mathcal{M},\mathcal{N}$ are called $\lambda$-terms or $\lambda$-expressions

\[
\mathcal{M},\mathcal{N} : ::= \ x \ \mid \ \lambda x.\mathcal{M} \ \mid \ \mathcal{M} \ \mathcal{N}
\]

variable abstraction application

- No types: e.g., $(\lambda x. x) y$; $(\lambda x. x) (\lambda x. x)$

- No numbers or operations
  - can be added
  - values are function abstractions

- Functions are nameless
  - No “let $f = \lambda x.\mathcal{M}$ in $\mathcal{N}$”
Syntax of $\lambda$-Terms

• Examples:
  – $\lambda x.x$ : the identity function
  – $(\lambda y. \lambda x. x) \ f \ g$: 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:
    • “$\lambda x. x \ z \ x$” is “$\lambda x.(x \ \lambda z.(x \ z \ x))$”
  – “$\lambda x. \lambda y. x \ y$” often abbreviates to “$\lambda x \ y. x \ y$”
Terminology

- Bound variables (parameters)
- Free variables
- Example:
  
  \( \lambda x.x \ y \)

  - \( x \) is bound in the term \( \lambda x.x \ y \)
  - \( y \) is free in the term \( \lambda x.x \ y \)
Terminology

- $\lambda x. M$
  - the scope of $x$ is the term $M$

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

FV(x) = \{x\}
FV(\lambda x. M) = FV(M) \setminus \{x\}
FV(M N) = FV(M) \cup FV(N)
<table>
<thead>
<tr>
<th>Open</th>
<th>Closed</th>
</tr>
</thead>
<tbody>
<tr>
<td>- FV(E) ≠ {}</td>
<td>- FV(E) = {}</td>
</tr>
<tr>
<td>- xz</td>
<td>- λx.x</td>
</tr>
<tr>
<td>- λx.xz</td>
<td>- λx.λy.xy</td>
</tr>
<tr>
<td>- (λx.x)y</td>
<td>- (λx.x)(λy.y)</td>
</tr>
<tr>
<td>- (λy.(λx.xz)y)w</td>
<td>- λf.λg.λx.f x (g x)</td>
</tr>
</tbody>
</table>

- Ex. Underline the bound variables
Evaluating $\lambda$-Terms

- Function application is straightforward:

$$\left( \lambda x. (f \; x) \right) \; y \rightarrow f \; y$$

substitute $y$ for $x$ in $(f \; x)$

- Reduce all applications $(\lambda x. L) \; N$

- Until none can be found
Evaluating λ-Terms

• \( \beta \)-reduction

\[
(\lambda x. \ x \ x) (\lambda y. \ y)
\]

\[
\xrightarrow{\beta} \ x \ x [\lambda y. \ y / x]
\]

\[
= (\lambda y. \ y) (\lambda y. \ y)
\]

\[
\xrightarrow{\beta} y [\lambda y. \ y / y]
\]

\[
= \lambda y. \ y
\]

\( M [N/x] \) 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 $\beta$-reduction

1. $(\lambda x . x) \ a \rightarrow_\beta a$
   
   $[a/x]$

2. $(\lambda x . \lambda y . x) \ a \ b \rightarrow_\beta (\lambda y . a) \ b \rightarrow_\beta a$
   
   $[a/x]$  $[b/y]$

3. $(\lambda x . x \ a) \ (\lambda x . x) \rightarrow_\beta (\lambda x . x) \ a \rightarrow_\beta a$
   
   $[\lambda x . x/x]$  $[a/x]$

4. $(\lambda x . \lambda y . x \ y) \ y \rightarrow_\beta (\lambda y . y \ y)$
   
   $[y/x]$  Name capturing error!

   $y \ Become \ bound$
A Similar Example in C Macro

- Name capturing problem in macro expansion

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

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

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

=> OK

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

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

=> oops! tmp got trapped
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:
    $$\lambda y. x \ y =_{\alpha} \lambda y. z \ y$$

- This is called *$\alpha$ conversion* in lambda calculus

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

  $$\lambda y. x \ y[x/y] \text{ will make the “free” } x \text{ captured.}$$
4. \((\lambda x . \lambda y . x y) \ y \rightarrow_\beta \lambda y . y y\) \(y \text{ Become bound}\)

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 $\beta$-reduction
• Terms $(\lambda x. L)N$ are called $\beta$-redexes

• $\beta$-normal form = no $\beta$-redexes

• $(\lambda x. xx)y \leftarrow a \beta$-redex

• $\rightarrow_\beta yy \leftarrow \beta$-normal form

• Not all $\lambda$-terms have $\beta$-nf
An example with no NF

\[(\lambda x. x x) (\lambda x. x x)\]
\[\rightarrow_\beta x x [\lambda x. x x/x]\]
\[== (\lambda x. x x) (\lambda x. x x)\]
\[\rightarrow \ldots \text{looping, no normal form}\]

\[\Omega = (\lambda x. x x)\]
\[\Omega\Omega \text{ has no } \beta\text{-nf}\]

- In other words, it is simple to write non-terminating computations in the lambda calculus
Evaluation Strategy (Order)

• A term may have many redexes:

\[ (\lambda x.(\lambda y.y)z) \ (\lambda z.z)w \]

• 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

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

\[
\lambda x. (\lambda y. y) z \quad (\lambda z. z) w \\
\rightarrow_\beta (\lambda x. z) \quad (\lambda z. z) w \\
\rightarrow_\beta z
\]
Normal Order Reduction

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

\[
(\lambda x. (\lambda y. y) z) \ (\lambda z. z) w \quad \rightarrow^\beta \quad (\lambda y. y) z \\
\rightarrow^\beta \quad z
\]
Why Not Normal Order?

- In most (all?) programming languages, functions are considered values (fully evaluated)
- Thus, no reduction is done inside of functions (under the lambda)
  \[ \lambda x. M \text{ is a value, not reducible} \]
- No popular programming language uses normal order
Call by Name; Call by Value

- Consider the application: \((\lambda x. E) \, e_1\)

- Call by value: evaluate the argument \(e_1\) to a value before \(\beta\) reduction

- Call by name: reduce the application, \textit{without} evaluating \(e_1\)

- In both cases: a lambda abstraction: \(\lambda x. E\) is a value.
Call-By-Name/Call-By-Value

- CBN example

\[\text{id}\ (\text{id}\ (\lambda z.\ \text{id}\ z))\]
\[\rightarrow_{\beta}\ \text{id}\ (\lambda z.\ \text{id}\ z)\]
\[\rightarrow_{\beta}\ \lambda z.\ \text{id}\ z\]

- CBV example

\[(\text{id}\ (\text{id}\ (\lambda z.\ \text{id}\ z)))\]
\[\rightarrow\ \text{id}\ (\lambda z.\ \text{id}\ z)\]
\[\rightarrow\ \lambda z.\ \text{id}\ z\]

where \(\text{id} = \lambda x. x\)
Order of Evaluation May Matter Much

- CBV (Inner redex):
  \[(\lambda y. \lambda z. z) ((\lambda x. x x) (\lambda x. x x)) \rightarrow_{\beta} (\lambda y. \lambda z. z) ((\lambda x. x x) (\lambda x. x x)) \rightarrow_{\beta} \ldots\]

- CBN (Outer redex):
  \[(\lambda y. \lambda z. z) ((\lambda x. x x) (\lambda x. x x)) \rightarrow_{\beta} (\lambda z. z) \]

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

If a $\lambda$-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 $\lambda$-expression, if it exists, is unique.
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:

\[ R = \{ X \mid X \notin X \}, \text{ is } R \in R? \]

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

• Church encounters similar issues in pure LC:

\[ \Omega = (\lambda x. x \ x), \Omega \Omega \text{ has no NF} \]

• 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
  \[ \text{let } x = e_1 \text{ in } e_2 \]
- Question: can we implement this construct in the lambda calculus?

\[ \text{source} = \text{lambda calculus + let} \]
\[ \downarrow \text{translate/compile} \]
\[ \text{target} = \text{lambda calculus} \]
• Given the let expressions in Haskell
  \( \text{let } x = e_1 \text{ in } e_2 \)

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

Example:

\[
\text{let } f = \lambda x. xz \text{ in } \lambda y. f( f \ y )
\]

\[
( \lambda f. \lambda y. f ( f \ y ) ) (\lambda x. xz)
\]
Compile the Let Expressions

• Given the let expressions in Haskell
  \[ \text{let } x = e_1 \text{ in } e_2 \]

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

Rule:
\[
\text{let } f = \lambda x. M \text{ in } N
\]
\[
(\lambda f. N)(\lambda x. M)
\]

• 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

\[
\begin{align*}
\text{if } true \ x \ y &= x \\
\text{if } false \ x \ y &= y
\end{align*}
\]

\[
\text{if } = \lambda \text{torf} . \lambda x . \lambda y . \text{torf} \ x \ y
\]

\[
\begin{align*}
true \ x \ y &= x \\
false \ x \ y &= y
\end{align*}
\]
Encoding Booleans

- the encoding:

\[ true = \lambda t. \lambda f. t \]

\[ false = \lambda t. \lambda f. f \]

\[ if = \lambda x. \lambda \text{then}. \lambda \text{else}. x \text{ then else} \]

If \( true \) \((\lambda x. t1) (\lambda x. t2)\)

\[ = (\lambda x. \lambda \text{then}. \lambda \text{else}. x \text{ then else}) (\lambda t. \lambda f. t) (\lambda x. t1) (\lambda x. t2) \]

\[ \rightarrow^{*} (\lambda t. \lambda f. t) (\lambda x. t1) (\lambda x. t2) \]

\[ \rightarrow^{*} \lambda x. t1 \]

\[ \rightarrow^{\beta} \text{ Zero or more steps of beta reduction} \]
Encoding Booleans

\[ \text{true} = \lambda t. \lambda f. t \quad \text{false} = \lambda t. \lambda f. f \]
\[ \text{and} = \lambda b. \lambda c. b \ c \ c \ \text{false} \]

\[ \text{and true true} \quad \rightarrow^* \text{true true false} \quad \rightarrow^* \text{true} \]
\[ \text{and false true} \quad \rightarrow^* \text{false true false} \quad \rightarrow^* \text{false} \]

\[ \beta \ \text{omitted} \]
A natural number is a function that given an operation $f$ and a starting value $s$, applies $f$ a number of times to $s$:

$$0 = \text{def } \lambda f. \lambda s. s$$
$$1 = \text{def } \lambda f. \lambda s. f \, s$$
$$2 = \text{def } \lambda f. \lambda s. f \,(f \, s)$$

...  

Church numerals

$$n = \text{def } \lambda f. \lambda s. f^n \, s$$
Computing with Natural Numbers

- The successor function
  \[ \text{succ } n = \text{def } \lambda f. \lambda s. f (n f s) \]

- Addition
  \[ \text{add } n_1 n_2 = \text{def } n_1 \text{ succ } n_2 \]

- Multiplication
  \[ \text{mult } n_1 n_2 = \text{def } n_1 (\text{add } n_2) 0 \]

- Testing equality with 0
  \[ \text{iszero } n = \text{def } n (\lambda b. \text{false}) \text{ true} \]
Computing with Natural Numbers. Example

Given: \( \text{succ } n = \text{def } \lambda f. \lambda s. f (n f s) \)

\[ 0 = \text{def } \lambda f. \lambda s. s \]

\[ 1 = \text{def } \lambda f. \lambda s. f s \]

\[
\text{succ } 0 = \\
(\lambda n. \lambda f. \lambda s. f (n f s)) \ 0 = \\
(\lambda n. \lambda f. \lambda s. f (n f s)) (\lambda f. \lambda s. s) \rightarrow \\
(\lambda f. \lambda s. f ((\lambda f. \lambda s. s) f s) \rightarrow \\
(\lambda f. \lambda s. f ((\lambda s. s) s) \rightarrow \\
\lambda f. \lambda s. f s = 1
\]
Computing with Natural Numbers. Example

\[
\begin{align*}
\text{mult} \ 2 \ 2 & \to \\
2 \ (\text{add} \ 2) \ 0 & \to \\
(\text{add} \ 2) \ ((\text{add} \ 2) \ 0) & \to \\
2 \ \text{succ} \ (\text{add} \ 2 \ 0) & \to \\
2 \ \text{succ} \ (2 \ \text{succ} \ 0) & \to \\
\text{succ} \ (\text{succ} \ (\text{succ} \ (\text{succ} \ 0))) & \to \\
\text{succ} \ (\text{succ} \ (\text{succ} \ (\lambda f. \lambda s. \ f \ (0 \ f \ s)))) & \to \\
\text{succ} \ (\text{succ} \ (\text{succ} \ (\lambda f. \lambda s. \ f \ s))) & \to \\
\text{succ} \ (\text{succ} \ (\lambda g. \lambda y. \ g \ ((\lambda f. \lambda s. \ f \ s) \ g \ y))) & \to \\
\text{succ} \ (\text{succ} \ (\lambda g. \lambda y. \ g \ (g \ y))) & \to \lambda g. \lambda y. \ g \ (g \ (g \ (g \ y))) = 4
\end{align*}
\]
Encoding pairs

- would like to encode the operations
  - \texttt{mkPair} \(e_1 e_2\)
  - \texttt{fst} \(p\)
  - \texttt{snd} \(p\)

- pairs will be \textit{functions}
  - when the function is used in the \texttt{fst} or \texttt{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

\[
\begin{align*}
\text{mkpair } x \; y & \stackrel{\text{def}}{=} \lambda \; b. \; x \; y \\
\text{fst } p & \stackrel{\text{def}}{=} p \; \text{true} \\
\text{snd } p & \stackrel{\text{def}}{=} p \; \text{false}
\end{align*}
\]

• Example:

\[
\text{fst (mkpair } x \; y) \rightarrow (\text{mkpair } x \; y) \; \text{true} \rightarrow \text{true } x \; y \rightarrow x
\]
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

• The Y combinator
  \( Y \equiv \lambda f. (\lambda x. f(x x)) (\lambda x. f(x x)) \)

• \( Y \) 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

\[
\begin{align*}
Y \ F & \\
\equiv & \ (\lambda f. (\lambda x. f(x \ x)) \ (\lambda x. f(x \ x))) \ F \\
\rightarrow_{\beta} & \ (\lambda x. F(x \ x)) \ (\lambda x. F(x \ x)) \\
\rightarrow_{\beta} & \ F \ ((\lambda x. F(x \ x)) \ (\lambda x. F(x \ x))) \ \\
\leftarrow_{\beta} & \ F \ ((\lambda f. (\lambda x. f(x \ x)) \ (\lambda x. f(x \ x))) \ F) \\
\equiv & \ F \ (Y \ F)
\end{align*}
\]

So, if we let \( X \equiv Y \ F \) then this tells us

\[ X = F \ X \]

in other words, \( X \) is a fixed point of \( F \).
Recursion

- Factorial in Haskell:
  \[
  \text{fact} = \lambda n \rightarrow \text{if } (n==0) \text{ then } 1 \text{ else } (n* (\text{fact} \ (n-1)))
  \]

- **Ex.** Write fact in \(\lambda\)-calculus by using the \(Y\) combinator.

- **Hint:** consider the term

- \(F \equiv \lambda f. \lambda n. \text{if } (\text{isZero } n) \ 1 \ (n*f \ (\text{pred } n))\)

- **Ex.** Evaluate \(\text{fact } 0\), \(\text{fact } 1\) and \(\text{fact } 2\).
Solution

\[
\text{fact} \equiv Y \ F \\
\equiv Y \ ( \lambda f. \lambda n. \text{if } (\text{isZero } n) \ 1 \ (n* (f \ (\text{pred } n))) )
\]

\[
\text{fact } 2 \\
= Y \ F \ 2 \\
= F \ (Y \ F) \ 2 \\
= (\lambda f. \lambda n. \text{if } (\text{isZero } n) \ 1 \ (n* (f \ (\text{pred } n)))) \ (Y \ F) \ 2 \\
= (\lambda n. \text{if } (\text{isZero } n) \ 1 \ (n* ((Y \ F) \ (\text{pred } n)))) \ 2 \\
= \text{if } (\text{isZero } 2) \ 1 \ (2* ((Y \ F) \ (\text{pred } 2))) \\
= 2* (Y \ F \ (\text{pred } 2)) \\
= 2* (Y \ F \ 1) \\
= 2* (\text{fact } 1) \quad \text{and so on...}
\]
Appendix: Formal Treatment of Substitutions
Name Capturing

\[(\lambda x.\lambda y.x)y \rightarrow_\beta \lambda y.y \times\]

• Replacing doesn’t always work
• But if we \(\alpha\)-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 $\lambda x$ guarantees that $x$ does not appear free in the term $(\lambda x. L)$, so there are no free occurrences to substitute for.

- $(\lambda x. L)[N/x] \equiv \lambda x. L$
Substitution $M[N/x]$

- The final case is the tricky one.

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

- If $x \notin \text{FV}(L)$ then there are no $x$’s to replace with $N$’s, so the term stays the same. If $y \notin \text{FV}(N)$ then there will be no $y$’s accidentally captured by the $\lambda y$ so we can keep $\lambda y$. But otherwise we must find a fresh variable $y'$ and replace $\lambda y$ by $\lambda y'$. 
Lambda Calculus with Constants and Types
Example: Extended LC

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

\[
E ::=\text{ constants: } 1, 2, 3, \ldots \\
\text{succ, iszero} \\
\text{true, false,} \\
\&\&\text{(and)}, \ ||\text{(or)}, \ !(\text{not}), \\
| \text{variable: } x, y, z, \ldots \\
| \lambda x. E \\
| E1 E2 \\
| \text{if } E1 \text{ then } E2 \text{ else } E3
\]
Evaluation Rules for the Extended LC

- Based on $\beta$-reduction
- Extended to Booleans and numbers
- Reduced to values:
  - $0, 1, 2, \ldots$
  - $\text{true, false}$
  - $\lambda x. E$
- Values are normal forms.

Some extended rules:

- $\text{iszero } 0 \to \text{true}$
- $\text{iszero } (\text{succ } n) \to \text{false}$
- $\text{pred } 0 \to 0$
- $\text{pred } (\text{succ } n) \to n$
- $\text{if true then } e_1 \text{ else } e_2 \to e_1$
- $\text{if false then } e_1 \text{ else } e_2 \to e_2$
- $e_1 \to e_2$
- $\text{succ } e_1 \to \text{succ } e_2$

...
Evaluation Rules for the Extended LC …

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

\[
(\lambda x. \text{succ } x) \, \text{true} \rightarrow \text{succ } \text{true} \rightarrow ??
\]

Reproduce it in LC:

\[
\text{succ true} = (\lambda n. \lambda f. \lambda s. f (n \, f \, s)) (\lambda t. f. t) \\
\rightarrow \lambda f. \lambda s. f ((\lambda t. f. t) \, f \, s) \\
\rightarrow \lambda f. \lambda s. f \, f \quad \text{--Not a number!}
\]
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: $A \to B$ is the type of functions with a parameter of type $A$ and a result of type $B$. Types are defined by this grammar:

$$T ::= \text{Int} \mid \text{Bool} \mid T \to T$$

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

Examples: $\text{Int} \to \text{Int} \to \text{Int}$ curried function of two arguments

$(\text{Int} \to \text{Int}) \to \text{Int}$ function which is given a function
Types and Type Errors

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

\[
\text{succ} : \text{Int} \rightarrow \text{Int} \\
\text{true} : \text{Bool}
\]

Then

\[
\text{"succ true"}
\]

is not acceptable!

We’ll introduce typing rules to filter out (type checking) such expressions.
Lambda Calculus with Types

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.

\[
\begin{align*}
\text{values} & : \quad \nu ::= \text{integer literal} \\
& \quad | \text{true} \quad | \text{false} \\
& \quad | \lambda x:T. e
\end{align*}
\]

\[
\begin{align*}
\text{expressions} & : \quad e ::= \nu \\
& \quad | x \\
& \quad | e + e \quad | e == e \\
& \quad | e \&\& e \quad | \text{if } e \text{ then } e \text{ else } e \\
& \quad | e \ e
\end{align*}
\]

\[
\begin{align*}
\text{types} & : \quad T ::= \text{Int} \\
& \quad | \text{Bool} \\
& \quad | T \rightarrow T
\end{align*}
\]

And more operators, such as ‘+’, ‘==’, ‘&&’

Type declaration for parameters

06/30~07/04

FP & Types 274
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

\[ \lambda x: \text{Int}. x + 2 \]

\[ \lambda b: \text{Bool}. \lambda x: \text{Int}. \text{if } b \text{ then } x \text{ else } -x \]

\[ \lambda f: \text{Int} \to \text{Int}. \lambda x: \text{Int}. f\ (f\ x) \]

\[ (\lambda f: \text{Int} \to \text{Int}. \lambda x: \text{Int}. f\ (f\ x))\ \text{succ} \]

\[ \lambda x: \text{Int}. \lambda f: \text{Int} \to \text{Int}. \lambda g: \text{Int} \to \text{Int}. \]

\[ \text{if } (x == 0) \text{ then } f \text{ else } g \]
• In function application, the type of the argument must be the same with that of the parameter.

\[
\begin{align*}
\text{e1} & : T_1 \rightarrow T_2 \\
\text{e2} & : T_1 \\
\hline \\
\text{e1 e2} & : T_2
\end{align*}
\]

\[
(\lambda f: \text{Int}\rightarrow\text{Int}. \lambda x: \text{Int}. f (f x)) : (\text{Int}\rightarrow\text{Int})\rightarrow\text{Int}
\]

\[
\text{succ} : \text{Int}\rightarrow\text{Int}
\]

\[
(\lambda f: \text{Int}\rightarrow\text{Int}. \lambda x: \text{Int}. f (f x)) \ \text{succ} : \text{Int}
\]
Determining the Type of an Expression

Type Checking: Does $e$ has a type $\tau$?

- $\tau$ is a meta-variable representing a type

$\tau ::= \text{Int} \lor \text{Bool} \lor \tau_1 \rightarrow \tau_2$
Type Judgments

• A *type judgment* has the form

\[ \Gamma |- \text{exp} : \tau \]

“exp has type \(\tau\) under TE \(\Gamma\)”

• \(\Gamma\) is a typing environment
  – Supplies the types of variables and functions
  – \(\Gamma\) is a list of the form \([x : \tau, \ldots]\)

• exp is a program expression

• \(\tau\) is a *type* to be assigned to exp

• |- pronounced “turnstyle”, or “entails” (or “satisfies”)
Example Valid Type Judgments

- \([\ ]\) |- true or false : Bool
- \([x : \text{Int}]\) |- x + 3 : Int
- \([p : \text{Int} \to \text{String}]\) |- (p 5) : String

Type judgments are derived via *typing rules.*
Format of Typing Rules

Assumptions:

$$
\Gamma_1 |- \text{exp}_1 : \tau_1 \ldots \Gamma_n |- \text{exp}_n : \tau_n
$$

Conclusion: $$\Gamma |- \text{exp} : \tau$$

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

|- n : Int (assuming $n$ is an integer constant)

|- true : Bool

|- false : Bool

- These rules are true with any typing environment
- $n$ is a meta-variable
Typing Environment

• A *typing environment* $\Gamma$ keeps track of the types of *free identifiers* occurred in expressions

  \[ \Gamma = [\ldots, x:\text{Int}, f:\text{Int} \rightarrow \text{Int}, \ldots] \]

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

  \[ \Gamma : \text{Ide} \rightarrow \text{Type} \]

  So, given $\Gamma$ as above, $\Gamma(x) = \text{Int}$

• No *multiple* bindings for any id:

  \[ \Gamma' = [\ldots, x:\text{Int}, f:\text{Int} \rightarrow \text{Int}, x:\text{Bool}, \ldots] \]
Axioms - Variables

• Typing rule for variables: (Var)

\[
\Gamma |- x : \tau \quad \text{if} \quad \Gamma(x) = \tau
\]

• We can also include the types for pre-defined identifiers (functions) in \( \Gamma \). For example:

  • \( \Gamma = [..., \text{succ: Int->Int, ...}] \)
Simple Rules - Arithmetic

Primitive operators \(( ⊕ ∈\{ +, -, *, \ldots \})\):

\[
\begin{align*}
\Gamma |- e_1 : \text{Int} \quad & \Gamma |- e_2 : \text{Int} \\
\Gamma |- e_1 ⊕ e_2 : \text{Int}
\end{align*}
\]

Relations \(( ∼ ∈\{ <, >, =, <=, >= \})\):

\[
\begin{align*}
\Gamma |- e_1 : \text{Int} \quad & \Gamma |- e_2 : \text{Int} \\
\Gamma |- e_1 ∼ e_2 : \text{Bool}
\end{align*}
\]
Logical Connectives:

\[
\begin{align*}
\Gamma |- e_1 : \text{Bool} & \quad \Gamma |- e_2 : \text{Bool} \\
\Gamma |- e_1 && e_2 : \text{Bool}
\end{align*}
\]

\[
\begin{align*}
\Gamma |- e_1 : \text{Bool} & \quad \Gamma |- e_2 : \text{Bool} \\
\Gamma |- e_1 || e_2 : \text{Bool}
\end{align*}
\]
Simple Example

• Let $\Gamma = [x: \text{Int}; y: \text{Bool}]$
• Show $\Gamma \vdash y \parallel (x + 3 > 6) : \text{Bool}$
• Start building the proof tree from the \textit{bottom up}

\[ \Gamma \vdash y \parallel (x + 3 > 6) : \text{Bool} \]
Simple Example

• Let $\Gamma = [x:\text{Int} ; y:\text{Bool}]$
• Show $\Gamma |- y \parallel (x + 3 > 6) : \text{Bool}$
• Which rule has this as a conclusion?

\[ ? \]

\[ \Gamma |- y \parallel (x + 3 > 6) : \text{Bool} \]
Simple Example

- Let $\Gamma = [\text{x: Int} ; \text{y: Bool}]$
- Show $\Gamma |- y || (x + 3 > 6) : \text{Bool}$
- *Booleans*: $||$

\[
\begin{align*}
\Gamma |- y : \text{Bool} & & \Gamma |- x + 3 > 6 : \text{Bool} \\
\hline
\Gamma |- y || (x + 3 > 6) : \text{Bool}
\end{align*}
\]
Simple Example

• Let $\Gamma = [x: \text{Int} ; y: \text{Bool}]$
• Show $\Gamma |- y \parallel (x + 3 > 6) : \text{Bool}$
• Pick an assumption to prove

?  

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

$\Gamma |- y \parallel (x + 3 > 6) : \text{Bool}$
Simple Example

- Let $\Gamma = [\text{x:Int}; \text{y:Bool}]$
- Show $\Gamma |- \text{y $|$ (x + 3 > 6)} : \text{Bool}$
- Which rule has this as a conclusion?

\[
\begin{align*}
? \\
\Gamma |- \text{y : Bool} & \quad \Gamma |- \text{x + 3 > 6 : Bool} \\
\hline
\Gamma |- \text{y $|$ (x + 3 > 6) : Bool}
\end{align*}
\]
Simple Example

• Let $\Gamma = [x:\text{Int}; y:\text{Bool}]$
• Show $\Gamma |- y \parallel (x + 3 > 6) : \text{Bool}$
• Axiom for variables

\[
\begin{align*}
\Gamma |- y : \text{Bool} & \quad \Gamma |- x + 3 > 6 : \text{Bool} \\
\hline
\Gamma |- y \parallel (x + 3 > 6) : \text{Bool}
\end{align*}
\]
Simple Example

- Let $\Gamma = [\text{x:} \text{Int}; \text{y:Bool}]$
- Show $\Gamma |- y || (x + 3 > 6) : \text{Bool}$
- Pick an assumption to prove

\[
\begin{align*}
\Gamma &|- y : \text{Bool} \\
\Gamma &|- x + 3 > 6 : \text{Bool} \\
\Gamma &|- y || (x + 3 > 6) : \text{Bool}
\end{align*}
\]
Simple Example

• Let $\Gamma = [x: \text{Int} ; y: \text{Bool}]$
• Show $\Gamma |- y \parallel (x + 3 > 6) : \text{Bool}$
• Which rule has this as a conclusion?

\[
\Gamma |- y : \text{Bool} \quad \Gamma |- x + 3 > 6 : \text{Bool} \quad ?
\]
\[
\Gamma |- y \parallel (x + 3 > 6) : \text{Bool}
\]
Simple Example

• Let $\Gamma = [\ x:\text{Int} ; \ y:\text{Bool}]$
• Show $\Gamma |- y \parallel (x + 3 > 6) : \text{Bool}$
• *Arithmetic relations*

\[
\frac{\Gamma |- y : \text{Bool} \quad \Gamma |- x + 3 : \text{Int} \quad \Gamma |- 6 : \text{Int}}{\Gamma |- y \parallel (x + 3 > 6) : \text{Bool}}
\]
Simple Example

• Let $\Gamma = [\text{x: Int}; \text{y: Bool}]$
• Show $\Gamma |- \text{y} \parallel (\text{x + 3 > 6}) : \text{Bool}$
• Pick an assumption to prove

\[
\begin{align*}
\Gamma |- \text{y} : \text{Bool} & \quad \Gamma |- \text{x + 3} : \text{Int} & \quad \Gamma |- \text{6} : \text{Int} \\
\Gamma |- \text{y} \parallel (\text{x + 3 > 6}) : \text{Bool}
\end{align*}
\]
Simple Example

• Let $\Gamma = \{ x: \text{Int} ; y: \text{Bool} \}$
• Show $\Gamma |- y || (x + 3 > 6) : \text{Bool}$
• Which rule has this as a conclusion?

\[
\begin{align*}
\Gamma |- y : \text{Bool} & \quad \Gamma |- x + 3 : \text{Int} \\
\Gamma |- 6 : \text{Int} & \quad \Gamma |- x + 3 > 6 : \text{Bool} \\
\hline
\hline
? & \quad \Gamma |- y || (x + 3 > 6) : \text{Bool}
\end{align*}
\]
Simple Example

• Let $\Gamma = [\ x:\text{Int} ; y:\text{Bool}]$
• Show $\Gamma |- y \ || (x + 3 > 6) : \text{Bool}$
• Axiom for constants

\[
\begin{align*}
\Gamma |- x + 3 : \text{Int} & \quad \Gamma |- 6 : \text{Int} \\
\Gamma |- y : \text{Bool} & \quad \Gamma |- x + 3 > 6 : \text{Bool} \\
\hline
\Gamma |- y \ || (x + 3 > 6) : \text{Bool}
\end{align*}
\]
Simple Example

• Let $\Gamma = [x: \text{Int} ; y: \text{Bool}]$
• Show $\Gamma |- y \parallel (x + 3 > 6) : \text{Bool}$
• Pick an assumption to prove

\[
\begin{align*}
? &\quad \Gamma |- x + 3 : \text{Int} & \Gamma |- 6 : \text{Int} \\
\Gamma |- y : \text{Bool} &\quad \Gamma |- x + 3 > 6 : \text{Bool} \\
\Gamma |- y \parallel (x + 3 > 6) : \text{Bool}
\end{align*}
\]
Simple Example

• Let \( \Gamma = [x: \text{Int} ; y: \text{Bool}] \)
• Show \( \Gamma |- y || (x + 3 > 6) : \text{Bool} \)
• Which rule has this as a conclusion?

\[
\begin{array}{c}
\begin{array}{l}
\Gamma |- y : \text{Bool} \\
\Gamma |- x + 3 : \text{Int} \\
\Gamma |- 6 : \text{Int}
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\begin{array}{l}
\Gamma |- x + 3 > 6 : \text{Bool}
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\begin{array}{l}
\Gamma |- y || (x + 3 > 6) : \text{Bool}
\end{array}
\end{array}
\]
Simple Example

- Let $\Gamma = \{ x: \text{Int} ; y: \text{Bool} \}$
- Show $\Gamma \vdash y \parallel (x + 3 > 6) : \text{Bool}$
- *Arithmetic operations*

\[
\begin{align*}
\Gamma \vdash x : \text{Int} & \quad \Gamma \vdash 3 : \text{Int} \\
\Gamma \vdash x + 3 : \text{Int} & \quad \Gamma \vdash 6 : \text{Int} \\
\Gamma \vdash y : \text{Bool} & \quad \Gamma \vdash x + 3 > 6 : \text{Bool} \\
\Gamma \vdash y \parallel (x + 3 > 6) : \text{Bool}
\end{align*}
\]
Simple Example

- Let $\Gamma = [x:\text{Int} ; y:\text{Bool}]$
- Show $\Gamma |- y || (x + 3 > 6) : \text{Bool}$
- Pick an assumption to prove

\[
\begin{align*}
\Gamma |- x : \text{Int} & \quad \Gamma |- 3 : \text{Int} \\
\Gamma |- x + 3 : \text{Int} & \quad \Gamma |- 6 : \text{Int} \\
\Gamma |- y : \text{Bool} & \quad \Gamma |- x + 3 > 6 : \text{Bool} \\
\Gamma |- y || (x + 3 > 6) : \text{Bool}
\end{align*}
\]
Simple Example

- Let $\Gamma = \{ x:\text{Int} ; y:\text{Bool} \}$
- Show $\Gamma |- y \parallel (x + 3 > 6) : \text{Bool}$
- Which rule has this as a conclusion?

\[
\begin{align*}
\Gamma |- x : \text{Int} & \quad \Gamma |- 3 : \text{Int} \\
\Gamma |- x + 3 : \text{Int} & \quad \Gamma |- 6 : \text{Int} \\
\Gamma |- y : \text{Bool} & \quad \Gamma |- x + 3 > 6 : \text{Bool} \\
\Gamma |- y \parallel (x + 3 > 6) : \text{Bool}
\end{align*}
\]
Simple Example

- **Let** $\Gamma = [\text{x:} \text{Int}; \text{y:} \text{Bool}]$
- **Show** $\Gamma |- y \parallel (x + 3 > 6) : \text{Bool}$
- **Axiom for constants**

\[
\begin{align*}
\Gamma |- x : \text{Int} & \quad \Gamma |- 3 : \text{Int} \\
\Gamma |- x + 3 : \text{Int} & \quad \Gamma |- 6 : \text{Int} \\
\Gamma |- y : \text{Bool} & \quad \Gamma |- x + 3 > 6 : \text{Bool} \\
\Gamma |- y \parallel (x + 3 > 6) : \text{Bool}
\end{align*}
\]
Simple Example

- Let $\Gamma = \{ x: \text{Int} ; y: \text{Bool} \}$
- Show $\Gamma |- y \parallel (x + 3 > 6) : \text{Bool}$
- Pick an assumption to prove

$$
\begin{array}{ll}
\Gamma |- x : \text{Int} & \Gamma |- 3 : \text{Int} \\
\Gamma |- x + 3 : \text{Int} & \Gamma |- 6 : \text{Int} \\
\Gamma |- y : \text{Bool} & \Gamma |- x + 3 > 6 : \text{Bool} \\
\Gamma |- y \parallel (x + 3 > 6) : \text{Bool}
\end{array}
$$
Simple Example

• Let $\Gamma = [x: \text{Int}; y: \text{Bool}]$
• Show $\Gamma |- y \parallel (x + 3 > 6) : \text{Bool}$
• Which rule has this as a conclusion?

$\Gamma |- x : \text{Int}$  $\Gamma |- 3 : \text{Int}$  $\Gamma |- x + 3 : \text{Int}$  $\Gamma |- 6 : \text{Int}$
$\Gamma |- y : \text{Bool}$  $\Gamma |- x + 3 > 6 : \text{Bool}$
$\Gamma |- y \parallel (x + 3 > 6) : \text{Bool}$
Simple Example

- Let $\Gamma = [ x:\text{Int} ; y:\text{Bool}]$
- Show $\Gamma |- y || (x + 3 > 6) : \text{Bool}$
- *Axiom for variables*

\[
\begin{align*}
\Gamma |- x : \text{Int} & \quad \Gamma |- 3 : \text{int} \\
\Gamma |- x + 3 : \text{Int} & \quad \Gamma |- 6 : \text{Int} \\
\Gamma |- y : \text{Bool} & \quad \Gamma |- x + 3 > 6 : \text{Bool} \\
\Gamma |- y || (x + 3 > 6) : \text{Bool}
\end{align*}
\]
Simple Example

- Let $\Gamma = [x:\text{Int}; y:\text{Bool}]$
- Show $\Gamma |- y \parallel (x + 3 > 6) : \text{Bool}$
- No more assumptions! DONE!

\[
\begin{align*}
\Gamma |- x : \text{Int} & \quad \Gamma |- 3 : \text{Int} \\
\Gamma |- x + 3 : \text{Int} & \quad \Gamma |- 6 : \text{Int} \\
\Gamma |- y : \text{Bool} & \quad \Gamma |- x + 3 > 6 : \text{Bool} \\
\Gamma |- y \parallel (x + 3 > 6) : \text{Bool}
\end{align*}
\]
If-Expressions

• If_then_else rule:

\[
\begin{array}{c}
\Gamma |- e_1 : \text{Bool} \\
\Gamma |- e_2 : \tau \\
\Gamma |- e_3 : \tau \\
\end{array}
\]
\[
\Gamma |- (\text{if } e_1 \text{ then } e_2 \text{ else } e_3) : \tau
\]

• \(\tau\) 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

if x==2 then 10 else 20  ✓

if x==2 then 10 else false  ✗
Function Application

- Application rule: (App)

\[
\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1
\]

\[
\Gamma \vdash (e_1 \ e_2) : \tau_2
\]

- If you have a function expression \( e_1 \) of type \( \tau_1 \rightarrow \tau_2 \) applied to an argument of type \( \tau_1 \), the resulting expression has type \( \tau_2 \)
Application Examples

\( \Gamma |- (\lambda f:\text{Int}->\text{Int}. \lambda x:\text{Int}. f\ (f\ x)) : (\text{Int}->\text{Int})->\text{Int}->\text{Int} \)

\( \Gamma |- \text{succ} : \text{Int}->\text{Int} \)

\( \Gamma |- (\lambda f:\text{Int}->\text{Int}. \lambda x:\text{Int}. f\ (f\ x)) \text{ succ} : \text{Int}->\text{Int} \)

\([f:\text{Int}->\text{Int}, g:\text{Int}->\text{Int}, b:\text{Bool}] |- \text{if}\ b\ \text{then}\ f\ \text{else}\ g : \text{Int}->\text{Int} \)

\([f:\text{Int}->\text{Int}, g:\text{Int}->\text{Int}, b:\text{Bool}] |- (\text{if}\ b\ \text{then}\ f\ \text{else}\ g)\ 5 : \text{Int} \)
Function Rule

- Rules describe types, but also how the environment $\Gamma$ may change
- $\lambda$-fun rule: (Abs)

\[
\begin{align*}
\frac{[x : \tau_1] \cup \Gamma |- e : \tau_2}{\Gamma |- \lambda x. e : \tau_1 \to \tau_2}
\end{align*}
\]

We often write $\Gamma.x:T = \Gamma \cup [x:T]$ -- extends $\Gamma$

- If $x \in \text{dom}(\Gamma)$, then $\Gamma.x:T$ means that the new binding of $x$ will replace the original one.
Function Example

\[
[y : \text{int}] \cup \Gamma \vdash y + 3 : \text{int} \\
\Gamma \vdash \lambda y. y + 3 : \text{int} \rightarrow \text{int}
\]

\[
[succ: \text{Int}\rightarrow\text{Int}] \vdash \text{x: Int} \vdash \text{succ: Int}\rightarrow\text{Int} \\
\vdash \text{x: Int} \vdash \text{x: Int} \\
\text{App} \vdash \text{x: Int} \vdash \text{(succ x): Int} \\
\vdash [\text{succ: Int}\rightarrow\text{Int}] \vdash \lambda x. (\text{succ x}) : \text{Int}\rightarrow\text{Int}
\]
Anther Fun Example

\[ \Gamma |- \lambda f : \text{Int} \to \text{Int}. \lambda x : \text{Int}. f \ (f \ x) : ? \]

*Move f and x to \( \Gamma \)*

\[
\begin{align*}
\Gamma &. f : \text{Int} \to \text{Int}. x : \text{Int} |- f : \text{Int} \to \text{Int} \quad \text{(Var)} \\
\Gamma &. f : \text{Int} \to \text{Int}. x : \text{Int} |- x : \text{Int} \quad \text{(Var)} \\
\Gamma &. f : \text{Int} \to \text{Int}. x : \text{Int} |- f \ x : \text{Int} \quad \text{(App)} \\
\Gamma &. f : \text{Int} \to \text{Int}. x : \text{Int} |- f \ (f \ x) : \text{Int} \quad \text{(App)} \\
\Gamma &. f : \text{Int} \to \text{Int}. x : \text{Int} |- \lambda x : \text{Int}. f \ (f \ x) : \text{Int} \to \text{Int} \quad \text{(Abs)} \\
\Gamma & |- \lambda f : \text{Int} \to \text{Int}. \lambda x : \text{Int}. f \ (f \ x) : (\text{Int} \to \text{Int}) \to \text{Int} \to \text{Int} \quad \text{(Abs)}
\end{align*}
\]
Typing Rules for the LC with Constants & Types

Γ |- i : Int if i is an integer literal
Γ |- true : Bool  Γ |- false : Bool

Γ |- x:T ∈ Γ
Γ |- -- x : T type judgement

Γ |- E1:Int  Γ |- E2:Int
Γ |- E1 + E2 : Int

Γ |- E1:Int  Γ |- E2:Int
Γ |- E1 == E2 : Bool

Γ.x:T1 |- E : T2
Γ |- \lambda x:T1.E : T1->T2

Γ |- E1:Bool  Γ |- E2:Boolean
Γ |- E1 && E2 : Boolean

Γ |- E1:Boolean  Γ |- E2:T  Γ |- E3:T
Γ |- if E1 the E2 else E3 : T

Γ |- E1:T1->T2  Γ |- E2:T1
Γ |- E1 E2 : T2
• Alternative: treat built-in operators like literal constants, and include their types in $\Gamma$

\[
\Gamma |- \&\& : \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool}
\]
\[
\Gamma |- + : \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}
\]
\[
\Gamma |- \text{succ} : \text{Int} \rightarrow \text{Int}
\]

• Then, no need to have special rules for them
Well-typed programs won’t get stuck!

Theorem: If $e$ is a closed expression of type $T$ ($\vdash e : T$), then for all $e'$ such that $e \rightarrow^* e'$, it is the case that either

(A) $e'$ is a value (say, $v'$) and $\vdash v' : t$, or

(B) exists $e''$ such that $e' \rightarrow e''$.

If $\vdash e_0 : T$, then $e_0 \rightarrow e_1 \rightarrow e_2 \rightarrow \ldots \rightarrow v$
The Simply Typed Lambda Calculus $\lambda \rightarrow$

- 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, $\beta$-reduction is Strong normalizing: all terms will be evaluated to a normal form.
Limitations of the SLC

• Types are monomorphic.

\[ \lambda x: \text{Int}. x + 1 : \text{Int} \rightarrow \text{Int} \] is OK

• But what is the type for the identity function?

\[ \lambda x: ? . x : ? \]

\[ \lambda x: \text{Int} . x : \text{Int} \rightarrow \text{Int}? \]

\[ \lambda x: \text{Bool} . x : \text{Bool} \rightarrow \text{Bool}? \]

\[ \lambda x: \text{Int} \rightarrow \text{Int} . x : (\text{Int} \rightarrow \text{Int}) \rightarrow (\text{Int} \rightarrow \text{Int})? \]

\[ \ldots \]
• Polymorphism: allow many types for a value (hence also for variable, expression)

• Introducing type variables and ∀ quantification to express parametric polymorphism.

• Let $\alpha$ be a type variables representing any types. We can type the id function as follows.

$$\|- \lambda x:\alpha. x : \forall \alpha. \alpha \rightarrow \alpha$$
Parametric Polymorphism...

Polymorphic type: $\forall \alpha . \alpha \to \alpha$

The $\alpha$ can be instantiated to any types:

Int $\to$ Int

Bool $\to$ Bool

(Int$\to$Int)$\to$(Int$\to$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
  - $\lambda x: T. E$
- Extend types with generic type variables and quantification
  - $\forall \alpha. \alpha \rightarrow \alpha$
- Enhance terms with types
  - Type generalization: $\Lambda \alpha. \lambda x: \alpha. E$, a polymorphic term
  - Type application: $(\Lambda \alpha. \lambda x: \alpha. E) (\text{Int}\rightarrow\text{Int})$
    - Replace $\alpha$ with $\text{Int}\rightarrow\text{Int}$
Types of the PLC

Syntax:

**Types**

\[ \tau ::= T \quad \text{type constants, (Int, Bool,...)} \]

\[ | \quad \alpha \quad \text{type variables} \]

\[ | \quad \tau \to \tau \quad \text{function types} \]

\[ | \quad \forall \alpha.\tau \quad \text{polymorphic types} \]

Examples:

\[ \text{Int, Int->Bool, Int->Int->Bool, ...} \]

\[ \alpha \to \beta \quad \forall \alpha.\alpha \to \alpha \]

\[ \forall \alpha.\alpha \to \forall \beta.\beta \quad \forall \alpha.\forall \beta.(\alpha \to \beta) \to \forall \gamma.\gamma \]
Terms of the PLC

Terms

\[ M ::= c \text{ constants} \]
\[ | \quad x \text{ variables} \]
\[ | \quad \lambda x: \tau. M \text{ function} \]
\[ | \quad M \, M \text{ function application} \]
\[ | \quad \Lambda \alpha(M) \text{ type generalization} \]
\[ | \quad M \, \tau \text{ type application} \]

Examples:

\[ \text{Id} = \Lambda \alpha(\lambda x: \alpha. x) \quad \text{-- type generalization (abstraction)} \]
\[ (\Lambda \alpha. \lambda x: \alpha. x)(\text{Int} \rightarrow \text{Int}) \quad \text{-- type application (specialization)} \]
Functions on Types

• In PLC, $\Lambda \alpha (M)$ is an anonymous notation for the function $F$ mapping each type $\tau$ to the value of $M[\tau/\alpha]$.

• I.e., computation in PLC involves $\beta$-reduction for such functions on types.

\[
(\Lambda \alpha (M)) \tau \rightarrow M[\tau/\alpha]
\]

e.g., $(\Lambda \alpha(\lambda x:\alpha.x))$ (Int->Int) $\rightarrow \lambda x:$$\text{Int}$$\rightarrow$$\text{Int}.x$

as well as the usual form of $\beta$-reduction from $\lambda$-calculus

\[
(\lambda x:\tau.M1) \ M2 \rightarrow M1[M2/x]
\]
In summary, we apply *substitution* on terms as well as types explicitly.

\[(\lambda x : \tau (M_1)) M_2 \rightarrow M_1[M_2/x] \]
\[(\Lambda \alpha (M)) \tau \rightarrow M[\tau/\alpha].\]
In this system of PLC:

- Two new kinds of terms (expressions):
  - $\Lambda \alpha (M)$ (typically, $\alpha$ is used in $M$)
  - Application with type operand: $M \tau$ ($\tau$ a type)

- The first kind of expression is also a value

- To the type language we add:
  - Type variables – $\alpha$
  - Universal types of the form $\forall$
Example: the identity function

\[
\text{Id} = \Lambda \alpha (\lambda x:\alpha.x) \quad \text{has type} \quad \forall \alpha.\alpha \rightarrow \alpha
\]

We can apply \text{Id} to many kinds of arguments:

- \text{Id Int 5} = \Lambda \alpha (\lambda x:\alpha.x) \text{ Int 5} \rightarrow (\lambda x:\text{Int}.x) \quad 5 \rightarrow 5

- \text{Id Bool true} = \Lambda \alpha (\lambda x:\alpha.x) \text{ Bool true} \rightarrow^* \text{ true}
Polymorphism in PLC, 2

Example: applying a function twice

\[
twice = \forall \alpha \, (\lambda f: \alpha \to \alpha. \lambda x: \alpha. f (f x)))
\]

has type \( \forall \alpha. (\alpha \to \alpha) \to \alpha \to \alpha \)

and can be applied to arguments of different types:

a) \( twice \) Int \((\lambda x: \text{Int}. x+2)\) 5 \( \to \) (\(\lambda f: \text{Int} \to \text{Int}. \lambda x: \text{Int}. f (f x)\)) \((\lambda x: \text{Int}. x+2)\) 5  
\( \to \) ((\(\lambda x: \text{int. } x+2\)) ((\(\lambda x: \text{int. } x+2\)) 5 ))  
\( \to^* \) 9

b) \( twice \) Bool \((\lambda x: \text{Bool. } x)\) false \( \to^* \) false
Polymorphism in PLC, 3

- Polymorphic function parameters
- Consider the following function application in LC:

\[(\lambda f. (f \ 5, f \ True)) \ (\lambda x. x) \quad --(,) \text{ is a pair}\]

Here the function parameter \(f\) is applied to two types of arguments: \textit{Int} and \textit{Bool}

In PLC, \((\lambda x. x)\) is \(\Lambda \alpha. \lambda x: \alpha. x\) with type \(\forall \alpha. \alpha \rightarrow \alpha\)
so we let \(f\) has the polymorphic type: \(\lambda f: \forall \alpha. \alpha \rightarrow \alpha\)
And rewrite the above example as:

\[(\lambda f: \forall \alpha. \alpha \rightarrow \alpha. (f \ Int \ 5, f \ Bool \ True)) \ (\Lambda \alpha. \lambda x: \alpha. x)\]
Polymorphism in PLC, 3

- Polymorphic function parameters
- Consider the following function application in LC:

\[(\lambda f. (f \, 5, f \, \text{True})) \, (\lambda x.x)\] --(,) is a pair

Write it in the PLC:

\[(\lambda f: \forall \alpha. \alpha \rightarrow \alpha. (f \ \text{Int} \, 5, f \ \text{Bool} \ \text{True})) \, (\Lambda \alpha. \lambda x: \alpha. x)\]

⇒\[((\Lambda \alpha (\lambda x: \alpha. x)) \ \text{Int} \, 5, (\Lambda \alpha (\lambda x: \alpha. x)) \ \text{Bool} \ \text{true})\]

⇒... ⇒ (5, true)
Re-visit the identity function

\[ \text{Id} = \Lambda \alpha (\lambda x: \alpha . x) \quad \text{has type} \quad \forall \alpha . \alpha \rightarrow \alpha \]

We can apply \textit{Id} to \textit{Id} in a similar way:

\[ > (\text{Id} \ (\forall \alpha . \alpha \rightarrow \alpha)) \quad \text{Id} = (\Lambda \alpha (\lambda x: \alpha . x) \ (\forall \alpha . \alpha \rightarrow \alpha)) \ (\Lambda \alpha (\lambda x: \alpha . x)) \]

\[ \rightarrow (\lambda x: \forall \alpha . \alpha \rightarrow \alpha . x) \ (\Lambda \alpha (\lambda x: \alpha . x)) \]

\[ \rightarrow \Lambda \alpha (\lambda x: \alpha . x) = \text{Id} \]

\[ \text{has type} \quad \forall \alpha . \alpha \rightarrow \alpha \]
Formal Typing Rules of PLC
## Syntax of PLC

### Types

\[ \tau ::= \ T \quad \text{type constants, (Int, Bool, ...)} \]
\[ \mid \alpha \quad \text{type variables} \]
\[ \mid \tau \rightarrow \tau \quad \text{function types} \]
\[ \mid \forall \alpha. \tau \quad \text{polymorphic types} \]

### Terms

\[ M ::= \ c \quad \text{constants} \]
\[ \mid x \quad \text{variables} \]
\[ \mid \lambda x : \tau. M \quad \text{function} \]
\[ \mid M M \quad \text{function application} \]
\[ \mid \Lambda \alpha . M \quad \text{type generalization} \]
\[ \mid M \tau \quad \text{type application} \]
Generic (Bound) vs. Free Type Variables

\[ \forall \alpha \tau = \forall \alpha \rightarrow \forall \beta \beta \]

\[ \text{ftv}(\tau) = [\beta] \]

\[ \forall \alpha \tau = \forall \alpha \rightarrow \beta \]

\[ \text{ftv}(\tau) = [\beta] \]

- Free type variables stand for *some* types;
- Generic type variables stand for *any* types.
Type Judgements of PLC

takes the form $\Gamma \vdash M : \tau$ where

- the typing environment $\Gamma$ is a finite function from variables to PLC types.
  (We write $\Gamma = \{ x_1 : \tau_1, \ldots, x_n : \tau_n \}$ to indicate that $\Gamma$ has domain of definition $\text{dom}(\Gamma) = \{ x_1, \ldots, x_n \}$ and maps each $x_i$ to the PLC type $\tau_i$ for $i = 1..n$.)

- $M$ is a PLC expression

- $\tau$ is a PLC type.

Source: Prof. A. Pitts
PLC Typing Rules

(var) \[ \Gamma |- x : \tau \quad \text{if } x : \tau \in \Gamma \]

(fn) \[ \frac{\Gamma . x : \tau_1 |- M : \tau_2}{\Gamma |- \lambda x : \tau_1. M : \tau_1 \rightarrow \tau_2} \]

(app) \[ \frac{\Gamma |- M_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma |- M_2 : \tau_1}{\Gamma |- M_1 \ M_2 : \tau_2} \]

(gen) \[ \frac{\Gamma |- M : \tau}{\Gamma |- \Lambda \alpha . M : \forall \alpha . \tau} \quad \text{If } \alpha \notin \text{ftv}(\Gamma) \]

(ty_app) \[ \frac{\Gamma |- M : \forall \alpha . \tau_1}{\Gamma |- M \ \tau_2 \ : \ \tau_2 [\tau_1 / \alpha]} \]
The Side-Condition in Gen

\[
\begin{align*}
&\quad \frac{x_1 : \alpha, x_2 : \alpha \vdash x_2 : \alpha}{x_1 : \alpha \vdash \lambda x_2 : \alpha (x_2) : \alpha \rightarrow \alpha} \quad \text{(fn)} \\
\end{align*}
\]

\[
\begin{align*}
&\quad x_1 : \alpha \vdash \forall \alpha (\lambda x_2 : \alpha (x_2)) : \forall \alpha (\alpha \rightarrow \alpha)
\end{align*}
\]

For \(\alpha \notin \text{ftv}(\Gamma)\)

\[
\begin{align*}
&\quad \frac{x_1 : \alpha, x_2 : \alpha' \vdash x_2 : \alpha'}{x_1 : \alpha \vdash \lambda x_2 : \alpha' (x_2) : \alpha' \rightarrow \alpha'} \quad \text{(fn)} \\
\end{align*}
\]

\[
\begin{align*}
&\quad x_1 : \alpha \vdash \forall \alpha' (\lambda x_2 : \alpha' (x_2)) : \forall \alpha' (\alpha' \rightarrow \alpha')
\end{align*}
\]
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 “∀”
  - 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.
  - \[ Γ \vdash E : τ \] where E has no type annotation at all
We can define a function `erase` on well-typed expressions, that removes all type-related information:

\[
\begin{align*}
\text{erase}(\lambda x:\tau. M) &= \text{erase}(\lambda x. M) \quad \text{--remove parameter type} \\
\text{erase}(\Lambda \alpha (M)) &= \text{erase}(M) \quad \text{--remove type abs} \\
\text{erase}(M \, \tau) &= \text{erase}(M) \quad \text{--remove type app}
\end{align*}
\]

This brings us back to extended LC (ELC without types)
The type reconstruction (inference) problem:

Given \( M \) without type information (in, say, \( ELC \)), find:
- \( M' \) with type information (annotations, abstractions, applications)
- \( \Gamma \) for \( \text{freevars}(M) \) (\( = \text{freevars}(M') \))
- a type \( \tau \)

s.t. \( \text{Erase}(M') = M \) and \( \Gamma |- M' : \tau \)

We then say that \( \Gamma |- M : \tau \)
Example of Type Reconstruction

Erase

\[
\left( (\lambda f: \forall \alpha. \alpha \rightarrow \alpha. (f \text{ Int 5}, f \text{ Bool True})) \ (\Lambda \alpha. \lambda x: \alpha. x) \right)
\]

\[
\left( (\lambda f. (f \text{ 5}, f \text{ True})) \ (\lambda x. x) \quad --(,\text{)} \text{ is a pair} \right)
\]
Type reconstruction

Theorem:
Given $M$ w/o type info, it is undecidable if well-typed $M'$ in PLC s.t. $\text{erase}(M') = M$ 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

```haskell
let id = \x -> x
    in (id 5, id True)
```

- *Function parameters* are monomorphic only.

```haskell
(\f -> (f 5, f True)) (\x -> x)
```
Mini-Haskell Expression

E ::= constants: 1, 2, 3, ...
   ‘a’, ’b’, …,
   True, False, &&, ||, !
   +, -, *, …, >, <, =,
   variable: x, y, z, …
   \x -> E
   E1 E2
   if E1 then E2 else E3
   let x = E1 in E2
   (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, ‘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 \( \tau \): adding type variables \( \alpha \)
  – \( \tau ::= \text{Int} | \text{Bool} | \ldots | \alpha | \beta \ldots | \tau_1 \rightarrow \tau_2 | (\tau_1, \tau_2) | [\tau] \)
Types in the HMTS

- No more general polymorphic types of PLC.
  - $\forall \alpha. \alpha \rightarrow \forall \beta. \beta \rightarrow \text{Int}$

- Adopts a two-layered types
  - Types with variables, but no quantifiers
  - Type Schemes that support only outermost quantification
    $$\forall \alpha. \forall \beta. (\alpha \rightarrow \beta) \rightarrow [\alpha] \rightarrow [\beta]$$
Types & Type Schemes

• Types τ: (mono)

  \[ \tau ::= \text{Int} \mid \text{Bool} \mid \ldots \mid \alpha \mid \beta \mid \ldots \mid \tau_1 \rightarrow \tau_2 \mid (\tau_1, \tau_2) \mid [\tau] \]

  two-layered types

  primitive types
  type variables
  function types  (Right-associative)
  pair (tuple) types
  list types

• Type schemes σ: (poly)

  \[ \sigma ::= \tau \mid \forall \alpha . \sigma \]

  generic type variable
## Examples of Type Schemes

<table>
<thead>
<tr>
<th>Type Scheme</th>
<th>Type Scheme</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Int], Bool, Char Æ Bool</td>
<td>∀α.α</td>
</tr>
<tr>
<td>(Char, Int) Æ Bool</td>
<td>∀α. [α] Æ α Æ Bool</td>
</tr>
<tr>
<td>[Int] Æ (Int-&gt;Bool) Æ Bool</td>
<td>∀α. ∀β. (α Æ β) Æ [α] Æ β</td>
</tr>
<tr>
<td>[Int] Æ β Æ Bool</td>
<td>∀α. α Æ β</td>
</tr>
</tbody>
</table>

**Invalid type schemes**

<table>
<thead>
<tr>
<th>Invalid Type Scheme</th>
<th>Invalid Type Scheme</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int Æ ∀α.α</td>
<td>∀α.α Æ ∀β.β</td>
</tr>
</tbody>
</table>

• Outermost quantification only
Generic (Bound) vs. Free Type Variables

\[ \sigma = \forall \alpha. \forall \beta. \alpha \to \beta \]
\[ \text{ftv}(\sigma) = \{\} \]
\[ \text{ftv}(\alpha \to \beta) = \{\alpha, \beta\} \]

• Free type variables stand for *some* types;
• Generic type variables stand for *any* types.

Notation: omit inner \( \forall \)

\[ \forall \alpha. \beta.(\alpha \to \beta) \to [\alpha] \to \beta \equiv \forall \alpha. \forall \beta. (\alpha \to \beta) \to [\alpha] \to \beta \]
Typing in Mini-Haskell

• A type judgment has the form

\[ \Gamma |- \text{exp} : \tau \quad --\text{not } \sigma \]

• \text{exp} is a Mini-Haskell expression
• \( \tau \) is a Mini-Haskell type to be assigned to \text{exp}

the typing environment \( \Gamma \) is a finite function from variables to type schemes.
(We write \( \Gamma = \{x_1 : \sigma_1, \ldots, x_n : \sigma_n\} \) to indicate that \( \Gamma \) has domain of definition \( \text{dom}(\Gamma) = \{x_1, \ldots, x_n\} \) and maps each \( x_i \) to the type scheme \( \sigma_i \) for \( i = 1..n \).)
Example Valid Type Judgments

- \([\ ]\) \ |- \ True \ or \ False : \text{Bool}
- \([x : \text{int}]\) \ |- \ x + 3 : \text{int}
- \([\text{len} : \forall \alpha. [\alpha] \rightarrow \text{Int}]\) \ |- \ \text{len} [1,3,5,7] : \text{Int}
- \([\text{len} : \forall \alpha. [\alpha] \rightarrow \text{Int}]\) \ |- \ \text{len} [\text{True, False}] : \text{Int}
- \([\text{len} : \forall \alpha. [\alpha] \rightarrow \text{Int}]\) \ |- \ \text{len} : [[\beta]] \rightarrow \text{Int} \quad \text{via} \ [[\beta]/\alpha]
Typing in Mini-Haskell

(Int) \( \Gamma |- n : \text{Int} \)  
(assuming \( n \) is an Integer constant)

(Boolean) \( \Gamma |- \text{True} : \text{Bool} \) \( \Gamma |- \text{False} : \text{Bool} \)

(nil) \( \Gamma |- [] : [\tau] \)  
--any type \( \tau \)

(cons) \( \frac{\Gamma |- e_1 : \tau_1 \quad \Gamma |- e_2 : [\tau_1]}{\Gamma |- (e_1:e_2) : [\tau_1]} \)

Note: \([e_1, e_2, e_3]\) is a syntactic sugar of \((e_1:(e_2:e_3))\)

(Pair) \( \frac{\Gamma |- e_1 : \tau_1 \quad \Gamma |- e_2 : \tau_2}{\Gamma |- (e_1, e_2) : (\tau_1, \tau_2)} \)
• A major change lies in typing a function.

• In PLC, we need to specify the type of a function’s parameter.

\[
\frac{- \Gamma. x: \tau_1 |- M : \tau_2}{\Gamma |- \lambda x: \tau_1. M : \tau_1 \to \tau_2}
\]

• In the HTMS, We guess a type for \( x \). No type annotation for parameters.

\[
\frac{- \Gamma. x: \tau_1 |- e : \tau_2}{\Gamma |- \lambda x. e : \tau_1 \to \tau_2}
\]

A type, not a type scheme, such as \( \forall \alpha. \alpha \), because fun Parameters are monomorphic.
• Guess as general as possible
• Consider the following two type derivations:

\[
\Gamma. \, x : \alpha \vdash x : \alpha \\
\Delta \vdash \lambda x. x : \alpha \to \alpha
\]

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:

\[ \forall \alpha. \alpha \rightarrow \alpha \prec \beta \rightarrow \beta \quad \text{via} \ [\beta/\alpha] \]

\[ \forall \alpha. \alpha \rightarrow \alpha \prec \text{Int} \rightarrow \text{Int} \quad \text{via} \ [\text{Int}/\alpha] \]

\[ \forall \alpha. \beta. \alpha \rightarrow \beta \rightarrow \beta \prec \text{Int} \rightarrow (\text{Bool} \rightarrow \text{Bool}) \quad \text{via} \ [\text{Int}/\alpha, \text{Bool}/\beta] \]
We say a type scheme $\sigma = \forall \alpha_1, \ldots, \alpha_n (\tau')$ generalises a type $\tau$, and write $\sigma \succ \tau$ if $\tau$ can be obtained from the type $\tau'$ by simultaneously substituting some types $\tau_i$ for the type variables $\alpha_i$ ($i = 1, \ldots, n$):

$$\tau = \tau'[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n].$$

(N.B. The relation is unaffected by the particular choice of names of bound type variables in $\sigma$.)

• Also called instantiation of a type scheme to a type. $\forall \alpha. \alpha \rightarrow \alpha \succ \beta \rightarrow \beta$ via $[\beta/\alpha]$
Orders between Type Schemes and Types, 3

- Not all type variables are equal!
- *Generic type variables vs. free type variables*

\[ \forall \alpha. \alpha \rightarrow \alpha \quad \beta \rightarrow \beta \]

- *Generic type variables* can be instantiated to any *types* \( \tau \), but free types variables are not!
- Generalization order between a type scheme and a type: \( \sigma \gg \tau \), this is required in typing rules
- Specialization between two types is derived during type reconstruction as interim results.
Typing in Mini-Haskell, 2

- Instantiate a type scheme to a type by guessing
  - From $\forall \alpha.[\alpha]\to\text{Int}$ to $[[\beta]]\to\text{Int}$
- Only when typing a variable:

  $(\text{Var } \neg\neg) \quad \Gamma \vdash x : \tau$

  if $\Gamma(x) = \sigma$ and $\sigma \supset \tau$

  Example:
  
  $[\text{len} : \forall \alpha.[\alpha]\to\text{Int}] \mid \vdash \text{len} : [[\beta]]\to\text{Int}$

  - In PLC,
  
  $[\text{len} : \forall \alpha.[\alpha]\to\text{Int}] \mid \vdash \text{len } \beta : [[\beta]]\to\text{Int}$
PLC vs. HTMS

• Recall that PLC has:
  – General polymorphic types: \( \tau \equiv \forall \alpha . \tau' \)
  – Application with type operand: \( M \tau \) (\( \tau \) a type)
  – Type generalization: \( \Lambda \alpha (M) \)

• By contrast, the HMTS
  – types \( \tau \) and type schemes \( \sigma \)
  – Instantiate a type scheme to a type
    • From \( \forall \alpha . [\alpha] -> \text{Int} \) to \( [[\beta]] -> \text{Int} \)
  – Generalize a type to a type scheme
    • From \( [\beta] -> \text{Int} \) to \( \forall \beta . [\beta] -> \text{Int} \)
• Function application remains the same, except that only monomorphic arguments ($\tau$).

\[
\frac{
\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \\
\Gamma \vdash e_2 : \tau_1
}{
\Gamma \vdash (e_1 \; e_2) : \tau_2}
\]

Example:

\[
\begin{array}{l}
\text{[ len : } \forall \alpha. [\alpha] \rightarrow \text{Int ] } \vdash \text{len} : \text{[Bool]} \rightarrow \text{Int} \\
\text{[ len : } \forall \alpha. [\alpha] \rightarrow \text{Int ] } \vdash [\text{True, False}] : \text{[Bool]} \\
\text{[ len : } \forall \alpha. [\alpha] \rightarrow \text{Int ] } \vdash \text{len} \; [\text{True, False}] : \text{Int}
\end{array}
\]

\[
\frac{
\Gamma \vdash e_1 : \text{Bool} \\
\Gamma \vdash e_2 : \tau \\
\Gamma \vdash e_3 : \tau
}{
\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau}
\]
A Function Example

Γ |- \( f \rightarrow \lambda x \rightarrow f \ (f \ x) \): ?

- Move \( f \) and \( x \) to \( \Gamma \)

\[
\begin{align*}
\Gamma &. f : \alpha \rightarrow \alpha. x : \alpha |- f : \alpha \rightarrow \alpha \\
\Gamma &. f : \alpha \rightarrow \alpha. x : \alpha |- x : \alpha \\
\hline
\alpha & \rightarrow \alpha \rightarrow \alpha \\
\hline
\end{align*}
\]

(App)

\[
\begin{align*}
\Gamma &. f : \alpha \rightarrow \alpha. x : \alpha |- f \ x : \alpha \\
\Gamma &. f : \alpha \rightarrow \alpha. x : \alpha |- f : \alpha \rightarrow \alpha \\
\hline
\alpha & \rightarrow \alpha \rightarrow \alpha \\
\hline
\end{align*}
\]

(App)

\[
\begin{align*}
\Gamma &. f : \alpha \rightarrow \alpha. x : \alpha |- f \ (f \ x) : \alpha \\
\hline
\alpha & \rightarrow \alpha \\
\hline
\end{align*}
\]

(Abs)

\[
\begin{align*}
\Gamma &. f : \alpha \rightarrow \alpha |- \lambda x \rightarrow f \ (f \ x) : \alpha \rightarrow \alpha \\
\hline
\alpha & \rightarrow \alpha \rightarrow \alpha \\
\hline
\end{align*}
\]

(Abs)

\[
\begin{align*}
\Gamma & |- \lambda f \rightarrow \lambda x \rightarrow f \ (f \ x) : (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha \\
\end{align*}
\]
Generalizing a type to a type scheme via LET-expr

\[ \Gamma \vdash \lambda f \rightarrow \lambda x \rightarrow f \ (f \ x) \ : \ (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha \]

\[ \forall \alpha. \ (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha \]

(Let)

\[
\frac{
\begin{align*}
\Gamma & \vdash e_1 : \tau_1 \\
\Gamma. \ x:\sigma & \vdash e_2 : \tau \\
\end{align*}
}{
\Gamma \vdash \text{let } x=e_1 \text{ in } e_2 : \tau}
\]

\[ x \notin \text{dom}(\Gamma) \]

\[ \sigma = \text{Gen}(\tau_1, \Gamma) = \forall \alpha_1...\alpha_n. \tau_1. \]

\[ \text{where } [\alpha_1,...,\alpha_n] = \text{ftv}(\tau_1) - \text{ftv}(\Gamma) \]
Generalization \textit{aka Closing}

\[
\text{Gen}(\Gamma,\tau) = \forall \alpha_1 \ldots \alpha_n. \tau
\]
where \([\alpha_1 \ldots \alpha_n] = \text{ftv}(\tau) - \text{ftv}(\Gamma)\)

- \textit{Generalization} introduces polymorphism
- Quantify type variables that are free in but not free in the type environment (TE)
- Captures the notion of \textit{new} type variables of \(\tau\) \text{ (introduced via the Var \rightarrow rule)}
Example of Let-Polymorphism

\[ E \equiv \text{let } \text{id} = \lambda x \rightarrow x \ \text{in} \ (\text{id} \ 5, \ \text{id} \ True) \]

(1) \( \Gamma |- \ \lambda x \rightarrow x : \alpha \rightarrow \alpha \) \quad \alpha \text{ is a fresh var, Gen called}

(2.1) \( \Gamma. \ \text{id} : \forall \alpha. \alpha \rightarrow \alpha |- \ \text{id} : \text{Int} \rightarrow \text{Int} \quad \Gamma. \ \text{id} : \forall \alpha. \alpha \rightarrow \alpha |- \ 5 : \text{Int} \)

\( \Gamma. \ \text{id} : \forall \alpha. \alpha \rightarrow \alpha |- \ \text{id} 5 : \text{Int} \)

(2.2) \( \Gamma. \ \text{id} : \forall \alpha. \alpha \rightarrow \alpha |- \ \text{id} : \text{Bool} \rightarrow \text{Bool} \quad \Gamma. \ \text{id} : \forall \alpha. \alpha \rightarrow \alpha |- \ True : \text{Bool} \)

\( \Gamma. \ \text{id} : \forall \alpha. \alpha \rightarrow \alpha |- \ \text{id} True : \text{Bool} \)

(2.1), (2.2) \quad \text{Pair}

\( \Gamma. \ \text{id} : \forall \alpha. \alpha \rightarrow \alpha |- \ (\text{id} \ 5, \ \text{id} \ True) : (\text{Int}, \ \text{Bool}) \)

\( \Gamma |- \ \text{let } \text{id} = \lambda x \rightarrow x \ \text{in} \ (\text{id} \ 5, \ \text{id} \ True) : (\text{Int}, \ \text{Bool}) \)
1. We can also have "id id" in the let-body:
   \[ \text{let } id = \lambda x \rightarrow x \text{ in } id \ id \]

2. Derive the type for the following lambda function:

\[
\lambda x. \begin{array}{l}
\quad \text{let } f = \lambda y \rightarrow x \\
\quad \text{in } (f \ 1, \ f \ True)
\end{array}
\]

\[ \Gamma_A = [ x : \alpha ] \]

(1) \[ \frac{\Gamma_A [ y : \beta ] \vdash x : \alpha}{\Gamma_A \vdash \lambda y \rightarrow x : \beta \Rightarrow \alpha} \]
HM Type Inference Rules

(App) \[ \frac{\Gamma \vdash e_1 : \tau \rightarrow \tau'}{\Gamma \vdash (e_1 e_2) : \tau'} \]

(Abs) \[ \frac{\Gamma \vdash [x : \tau] e : \tau'}{\Gamma \vdash \lambda x. e : \tau \rightarrow \tau'} \]

(Var) \[ \frac{(x : \sigma) \in \Gamma \quad \sigma \geq \tau}{\Gamma \vdash x : \tau} \]

(Const) \[ \frac{\text{typeof}(c) \geq \tau}{\Gamma \vdash c : \tau} \]

(Let) \[ \frac{\Gamma + [x : \tau] \vdash e_1 : \tau \quad \Gamma + [x : \text{Gen}(\text{TE}, \tau)] \vdash e_2 : \tau'}{\Gamma \vdash (\text{let } x = e_1 \text{ in } e_2) : \tau'} \]
Limitations of the HMTS:

\[ \lambda\text{-bound (monomorphic)} \text{ vs Let-bound Variables} \]

- Only *let-bound* identifiers can be instantiated differently.

\[
\begin{align*}
E_1 \equiv & \text{let } id = \lambda x \rightarrow x \text{ in } (id \ 5, \ id \ True) \\
E_2 \equiv & \ (\lambda f \rightarrow (f \ 5, \ f \ True)) (\lambda x \rightarrow x)
\end{align*}
\]

Semantically \( E_1 = E_2 \), but

\[
\boxed{\begin{align*}
E_2 & \equiv \ (\lambda f \rightarrow (f \ 5, \ f \ True)) \text{ is not typable:} \\
(f : \ ?) & \vdash (f \ 5, \ f \ True) : (\text{Int, Bool}) \\
\end{align*}}
\]

Recall the (Abs) rule

\[
\frac{
\begin{align*}
\Gamma \vdash e : \tau_2 \\
\end{align*}
}{
\Gamma \vdash \lambda x : \tau_1 \rightarrow e : \tau_1 \rightarrow \tau_2}
\]

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
• What type for “\f->\x->f x”?

\[
\begin{align*}
[ f: \text{Int} \to \text{Bool}, x: \text{Int} ] & \vdash f : \text{Int} \to \text{Bool} & [ f: \text{Int} \to \text{Bool}, x: \text{Int} ] & \vdash x : \text{Int} \\
\hline
& \quad \quad \quad \quad \text{App} \\
[ f: \text{Int} \to \text{Bool}, x: \text{Int} ] & \vdash f \ x : \text{Bool} & \quad \quad \quad \quad \text{Abs} \\
\hline
& \quad \quad \quad \quad \text{Abs} \\
[ f: \text{Int} \to \text{Bool} ] & \vdash \ \lambda x : \text{Int} \to \text{Bool} \\
\hline
[ ] & \vdash \ \lambda f : (\text{Int} \to \text{Bool}) \to (\text{Int} \to \text{Bool}) \\
\end{align*}
\]

Can we derive a more “general” type for this expression?
• A more general type for “\( \lambda f. \lambda x. f \ x \)”?

\[
\begin{align*}
[f : \alpha \to \beta, \ x : \alpha] & \vdash f : \alpha \to \beta \quad [f : \alpha \to \beta, \ x : \alpha] & \vdash x : \alpha \\
\hline 
[f : \alpha \to \beta, \ x : \alpha] & \vdash f \ x : \beta \\
\hline 
[f : \alpha \to \beta] & \vdash \lambda x. f \ x : (\alpha \to \beta) \\
\hline 
[\ ] & \vdash \lambda f. \lambda x. f \ x : (\alpha \to \beta) \to (\alpha \to \beta)
\end{align*}
\]

\textbf{Most general type}

Any instance of \( (\alpha \to \beta) \to (\alpha \to \beta) \) is a valid type.
E.g., \( (\text{Int} \to \text{Bool}) \to (\text{Int} \to \text{Bool}) \)
A type scheme $\sigma$ is the *principal* type scheme of a closed Mini-Haskell expression $E$ if

(a) $|- E : \tau$ is provable and $\sigma = \text{Gen}(\tau, \{\})$

(b) for all $\tau'$, if $|- E : \tau'$ is provable and $\sigma' = \text{Gen}(\tau', \{\})$

then $\sigma \succ \sigma'$

where by definition $\sigma \succ \sigma'$ if $\sigma' = \forall \alpha_1\ldots\alpha_n.\tau'$ and $\text{FV}(\sigma) \cap \{\alpha_1\ldots\alpha_n\} = \{\}$ and $\sigma \succ \tau'$.

E.g., $\lambda f.\lambda x.f \ x$ 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

- 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)

\[ W(\Gamma, e) \text{ returns } (S, \tau) \text{ such that } S(\Gamma) \vdash e : \tau \]

- \( \Gamma \) is a typing environment recording the most general type of each identifier that may occur in \( e \)
- \( e \) is an expression
- \( \tau \) is a type, may contain type variables to be generalized
- \( S \) is a type substitution recording the changes in the free type variables in \( \Gamma \), if any.
The W Algorithm

\[ W(\Gamma, e) \text{ returns } (S, \tau) \text{ such that } S(\Gamma) \vdash e : \tau \]

• Example: Open expression

\[ \Gamma = [f : \alpha \rightarrow \alpha, \ x : \beta], \quad e \equiv f \ x \]

\[ W(\Gamma, e) = ([\alpha/\beta], \beta) \text{ and } [\alpha/\beta](\Gamma) \vdash f \ x : \beta \]
The W Algorithm

\[ W(\Gamma, e) \text{ returns } (S, \tau) \text{ such that } S(\Gamma) \vdash e : \tau \]

- Example: closed expression

\[
\Gamma = [], \quad e \equiv \text{let } id=\lambda x\rightarrow x \text{ in } (id \ id)
\]

\[
W(\Gamma, e) = ([\beta\rightarrow\beta/\alpha], \beta\rightarrow\beta) \text{ and } [\beta\rightarrow\beta/\alpha](\Gamma) \vdash e : \beta\rightarrow\beta
\]
The W Algorithm: Syntax-Directed

The W algorithm is defined in terms of the syntactic structure of the expression to type.

\[ W(\Gamma, e) \text{ returns } (S, \tau) \text{ such that } S(\Gamma) \vdash e : \tau \]

**Syntax-directed**

\[
\text{Def } W(\Gamma, e) = \\
\quad \text{Case } e \text{ of } \\
\quad \quad x = \ldots \\
\quad \lambda x.e = \ldots \\
\quad (e_1 e_2) = \ldots \\
\quad \text{let } x = e_1 \text{ in } e_2 = \ldots 
\]
The W Algorithm: Variables

1. When \( e \) is a variable:

\[
\text{Def } \ W(\Gamma, e) = \begin{cases} 
\text{Case } e \text{ of } & \\
\quad x = \ldots & 
\end{cases}
\]

Recall the inference rule (axiom) for variables:

\[
(\text{Var}) \quad \frac{(x : \sigma) \in \Gamma \quad \sigma \geq \tau}{\Gamma \vdash x : \tau}
\]

We do not yet know which \( \tau \) to instantiate!

Let \( \forall \alpha. \alpha \rightarrow \alpha = \Gamma(x) \), we simply replace \( \alpha \) with fresh (new) type variable, say \( \beta \); and determine the type for \( \beta \) later when \( x \) is applied via unification.
1. When \( e \) is a variable:

Recall the inference rule (axiom) for variables: (Var)

\[
\begin{array}{c}
(x : \sigma) \in \Gamma \\
\sigma \geq \tau \\
\hline
\Gamma \vdash x : \tau
\end{array}
\]

We do not yet know which \( \tau \) to instantiate!

\[
\text{Def } W(\Gamma, e) =
\]

Case \( e \) of

\( x \) = if \( (x \notin \text{Dom}(\Gamma)) \) then Fail
else let \( \forall \alpha_1... \alpha_n. \tau = \Gamma(x); \)

in \( (\{\}, [\beta_i/\alpha_i] \tau) \)

\( \beta \)'s represent new type variables
The W Algorithm: Application

2. When e is an application:

\[ \text{Def } \ \mathcal{W}(\Gamma, e) = \]
\[ \text{Case } e \text{ of } \]
\[ (e_1 \ e_2) \quad = \]

Recall the inference rule for fun application:

\[
\frac{\Gamma \vdash e_1 : \tau \rightarrow \tau' \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash (e_1 \ e_2) : \tau'} \quad \text{(App)}
\]

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 substitution to unify them..
2. When $e$ is a function application:

\[
\frac{
\Gamma \vdash e_1 : \tau \to \tau' \quad \Gamma \vdash e_2 : \tau
}{
\Gamma \vdash (e_1 \, e_2) : \tau'
}
\]

\[\text{Def } W(\Gamma, e) = \]

\[\text{Case } e \text{ of} \]

\[(e_1 \, e_2) \quad \text{let } (S_1, \tau_1) = W(\Gamma, e_1);
(S_2, \tau_2) = W(S_1(\Gamma), e_2);
S_3 = \text{Unify}(S_2(\tau_1), \tau_2 \to \beta);
\text{in } (S_3 \, S_2 \, S_1, S_3(\beta))\]

\[\beta \text{ represents a new type variable}\]
Unification: \( \text{Unify}(\tau_1, \tau_2) \)

- \( \text{Unify}(\tau_1, \tau_2) = \) fail or a type substitution \( S \) such that \( S\tau_1 = S\tau_2 \).

\[
\text{Unify}(\alpha \to \alpha, \text{Int} \to \text{Bool}) = \) fail
\]
\[
\text{Unify}(\alpha \to \alpha, \text{Int} \to \text{Int}) = [\text{Int}/\alpha] \equiv S
\]
- Then \( S(\alpha \to \alpha) = S(\text{Int} \to \text{Int}) \)

\[
\text{Unify}([\alpha] \to \beta, [\gamma] \to \text{Int}) = [\gamma/\alpha, \text{Int}/\beta] \equiv S
\]

- And compute the Most General Unifier (MGU)
  - Let \( S' = [\text{Bool}/\alpha, \text{Int}/\beta] \).
  - \( S'([\alpha] \to \beta) = S'([\gamma] \to \text{Int}) \) and \( S \preceq S' \)
Unification: \( \text{Unify}(\tau_1, \tau_2) \)

\[
\text{def Unify}(\tau_1, \tau_2) = \\
\text{case } (\tau_1, \tau_2) \text{ of} \\
(\tau_1, \alpha) = [\tau_1/\alpha] \quad \text{-- } C_i \text{ constant type} \\
(\alpha, \tau_2) = [\tau_2/\alpha] \\
(C_1, C_2) = \text{if } (\text{eq? } C_1, C_2) \text{ then } [] \text{ else fail} \\
(\tau_{11} \mapsto \tau_{12}, \tau_{21} \mapsto \tau_{22}) \\
= \text{let } S1 = \text{Unify}(\tau_{11}, \tau_{21}) \\
S2 = \text{Unify}(S1 (\tau_{12}), S1 (\tau_{22})) \\
in \ S2 \circ S1 \\
\text{otherwise } = \text{fail}
\]

- Composition of substitution: \( s_2 \circ s_1 \)

Ex: \([\text{Int}/\beta] \circ [\beta/\alpha] = [\text{Int}/\beta, \text{Int}/\alpha]\)
3. When $e$ is a lambda function:

$$\text{Def } W(\Gamma, e) = \begin{cases} \text{Case } e \text{ of } \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ }$$

Recall the inference rule for lambda function:

$$\frac{\Gamma + \{ x : \tau \} \vdash e : \tau'}{\Gamma \vdash \lambda x.e : \tau \rightarrow \tau'}$$

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.
3. When e is a lambda function:

\[
\begin{align*}
\Gamma + [x : \tau] & \vdash e : \tau' \\
\Gamma & \vdash \lambda x. e : \tau \rightarrow \tau'
\end{align*}
\]

\[\text{Def \ } W(\Gamma, e) = \]
\[\text{Case } e \text{ of} \]
\[\lambda x \rightarrow e \quad = \text{let} \quad (S_1, \tau_1) = W(\Gamma + [x : \beta], e); \]
\[\text{in} \quad (S_1, S_1(\beta) \rightarrow \tau_1)\]

\(\beta\) is new
The W Algorithm: Let

4. When e is a let expression:

\[\text{Def } W(\Gamma, e) = \begin{cases} 
\text{Case } e \text{ of } \\
\text{let } x = e_1 \text{ in } e_2 = \ldots 
\end{cases}\]

Recall the inference rule for let expression:

\[
\frac{
\Gamma + [x : \tau] \vdash e_1 : \tau \quad \Gamma + [x : \text{Gen}(\mathcal{T}_E, \tau)] \vdash e_2 : \tau'}{
\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau'}
\]

\[\text{Def } W(\Gamma, e) = \begin{cases} 
\text{Case } e \text{ of } \\
\text{let } x = e_1 \text{ in } e_2 = \text{let } (S_1, \tau_1) = W(\Gamma, e_1); \\
\sigma = \text{Gen}(S_1(\Gamma), \tau_1); \\
(S_2, \tau_2) = W(S_1(\Gamma) + [x : \sigma], e_2); \\
in (S_2 S_1, \tau_2) 
\end{cases}\]
The W Algorithm

\textbf{Def} \( W(\Gamma, e) \) = \text{Case } e \text{ of } \\
\begin{align*}
x & = \text{if} \ (x \notin \text{Dom}(\Gamma)) \ \text{then} \ \text{Fail} \\
\text{else} \ & \ \text{let} \ \forall t_1...t_n.\tau = \Gamma(x) ; \\
& \text{in} \ (\{\}, [\beta_i / t_i] \tau) \\
\lambda x.e & = \text{let} \ (S_1, \tau_1) = W(\Gamma + [x : \beta], e) ; \\
& \text{in} \ (S_1, S_1(\beta) -> \tau_1) \\
(e_1 \ e_2) & = \text{let} \ (S_1, \tau_1) = W(\Gamma, e_1) ; \\
& \ (S_2, \tau_2) = W(S_1(\Gamma), e_2) ; \\
& \ S_3 \ = \text{Unify}(S_2(\tau_1), \tau_2 -> \beta) ; \\
& \text{in} \ (S_3 S_2 S_1, S_3(u)) \\
\text{let} \ x = e_1 \ \text{in} \ e_2 & = \text{let} \ (S_1, \tau_1) = W(\Gamma , e_1) ; \\
& \sigma \ = \text{Gen}(S_1(\Gamma), \tau_1) ; \\
& \ (S_2, \tau_2) = W(S_1(\Gamma) + [x : \sigma], e_2) ; \\
& \text{in} \ (S_2 S_1, \tau_2)
\end{align*}
The W Algorithm: Example

\[ \lambda x. \text{let } f = \lambda y. x \text{ in (} f \ 1, \ f \ True \) \]

\[ W(\emptyset, A) = ([], u_1 \to (u_1, u_1)) \]
\[ W(\{x : u_1\}, B) = ([], (u_1, u_1)) \]
\[ W(\{x : u_1, f : u_2\}, \lambda y. x) = ([], u_3 \to u_1) \]
\[ W(\{x : u_1, f : u_2, y : u_3\}, x) = ([], u_1) \]

Unify(u_2, u_3 \to u_1) = [(u_3 \to u_1) / u_2]
Gen(\{x : u_1\}, u_3 \to u_1) = \forall u_3. u_3 \to u_1

TE = \{x : u_1, f : \forall u_3. u_3 \to u_1\}

W(TE, (f \ 1)) = ([], u_1)

W(TE, f) = ([], u_4 \to u_1)
W(TE, 1) = ([], \text{Int})

Unify(u_4 \to u_1, \text{Int} \to u_5) = [\text{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 \( \Gamma \).

• It generates most general types of expressions. called Principal Type Scheme. Any verifiable type is inferred.

• Complexity
  PSPACE-Hard
  DEXPTIME-Complete
  Nested \( \text{let} \) 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) \[ \frac{\text{TE} \vdash e : \tau \quad \alpha \notin \text{FV(TE)}}{\text{TE} \vdash e : \forall \alpha. \tau} \]

(Spec) \[ \frac{\text{TE} \vdash e : \forall \alpha. \tau}{\text{TE} \vdash e : \tau [\tau'/\alpha]} \]

(Var) \[ \frac{(x : \tau) \in \text{TE}}{\text{TE} \vdash x : \tau} \]

(Let) \[ \frac{\text{TE} + \{x : \tau\} \vdash e_1 : \tau \quad \text{TE} + \{x : \tau\} \vdash e_2 : \tau'}{\text{TE} \vdash (\text{let } x = e_1 \text{ in } e_2) : \tau'} \]

(App) and (Abs) rules remain unchanged.

Sound but no direct inference algorithm!
Appendix: Haskell’s Type Classes
Polymorphism

Universal Polymorphism
- Parametric
- Subtyping

Ad Hoc Polymorphism
- Overloading
- Coercion

?
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: \[ \text{double} = \lambda x \rightarrow x+x \]

- What should be the proper type of double?
  - Int -> Int -- too specific
  - \forall 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?
  \( \forall a. a \rightarrow a \) -- 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 :: (\( \forall a \in \text{Num} \)) a -> a

• **Qualified types** generalize this by qualifying the type variable, as in \( (\forall a \in \text{Num}) a \rightarrow a \), which in Haskell we write as Num \( a \rightarrow a \rightarrow a \)

• Note that the type signature \( a \rightarrow a \) is really shorthand for \( \forall a.a \rightarrow a \)
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.
In Haskell, we use type classes and instance declarations to support parametric overloading systematically.

A type is made an instance of a class by an instance declaration.

Type \( a \) belongs to class \( \text{Num} \) if it has ‘+’, ‘-’, ‘*’, … of proper signature defined.

Type \( \text{Int} \) is an instance of class \( \text{Num} \).
In Haskell, the **qualified type** for `double` is:

\[ \forall a. \textbf{Num} \ a \Rightarrow a \rightarrow a \]

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
  \[ f :: a \rightarrow a \]
  "f is of type a -> a for any type a"

- Overloading using qualified types
  \[ f :: C a \Rightarrow a \rightarrow a \]
  "f is of type a -> a for any type a belonging to the type class C"

- Think of a Qualified Type as a type with a Predicate set, also called context in Haskell.
double :: ∀ a. Num a => a->a

The type predicate “Num a” will be supported by an \textit{additional (dictionary) parameter}.

In Haskell, the function \textit{double} is translated into

double \textit{NumDict} x =

\[(\text{select} \ (+) \ \text{from} \ \text{NumDict}) \ x \ x\]

Similar to

double \textit{add} x = x `add` x -- \textit{add} x x
Type Classes and Overloading

Dictionary for (type class, type) is created by the *Instance declaration*.

```haskell
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

- 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! >> (\(\rightarrow\) 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:

```haskell
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 instances (1)

- Make pre-existing classes instances of type class:

  ```haskell```
  instance Eq Integer where
  x == y = x `integerEq` y

  instance Eq Float where
  x == y = x `floatEq` y

- (assumes `integerEq` and `floatEq` functions exist)

  ```haskell```
  instance Eq Bool where
  True  == True  = True
  False == False = True
  _     == _     = False
  ```haskell```
• 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 (2)
• 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: \texttt{Eq a => ...}
Functions Requiring Context

Constraints

• Consider the following list element testing function:

\[
\text{elem} :: \text{Eq } a \Rightarrow a \rightarrow [a] \rightarrow \text{Bool}
\]

\[
\text{elem } x \ [ \ ] \quad = \quad \text{False}
\]

\[
\text{elem } x \ (y:ys) \quad = \quad (x == y) \lor \text{elem } x \ ys
\]

\[
\text{>elem 5 [1, 3, 5, 7]}
\]

True

\[
\text{>elem ‘a’ “This is an example”}
\]

False
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

- Comparable types:
  \texttt{Ord} \rightarrow \texttt{<} \texttt{ <= } \texttt{ >} \texttt{ >=}

- Printable types:
  \texttt{Show} \rightarrow \texttt{show} \texttt{ where}
  \texttt{show} \texttt{ ::} (\texttt{Show} \ a) \Rightarrow a \rightarrow \texttt{String}

- Numeric types:
  \texttt{Num} \rightarrow + - * \texttt{ negate} \texttt{ abs} \texttt{ etc.}
Show – Showable Types

- This class contains all those types whose values can be converted into character strings using
  \[
  \text{show} :: a \rightarrow \text{String}
  \]

- \text{Bool, Char, String, Int, Integer} and \text{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

> 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
  \[ \text{read} :: \text{String} \rightarrow a \]

• \text{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 "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

```haskell
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

  http://portal.acm.org/citation.cfm?id=582176

  http://hal.inria.fr/inria-00076025/en/

  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