Functional Programming Languages

Benjamin Goldberg

Department of Computer Science
New York University

goldberg@cs.nyu.edu

λ

Why use Functional Languages?

<u>Adherents Claim:</u>

- **Faster production of software**
- **Shorter programs**
- **More readable code**
- **Code more easily verified (formally or informally)**
- **More appropriate for parallel computing (research issue)**

What is it about functional languages that makes this so?

- **Declarative Language (describes what is to be computed, rather than how)**

```
fun  fac(0) = 1
 |   fac(x) = x * fac(x-1)
```

vs. (imperative)

```
j := 0;
for i := 1 to x do
        j := j * i;
```

- **Firm mathematical foundation**
    - **the lambda calculus**
    - **denotational semantics**
- **Higher-level, more mathematical, notation**
- **Provides referential transparency, due to the absence of side-effects (i.e. assignment)**
- **Supports equational reasoning**

- **Supports functions as values - greater abstraction mechanisms.**

- **Very flexible (and in most case, static) type systems.**

- **Some FL's exhibit non-strict semantics for greater independence from order of evaluation issues and infinite data structures.**

- **Not an inherently sequential computation model, in fact, implicitly parallel.**

The History of Functional Languages

The Lambda calculus (Church, 1930's)

- Still the most important influence, forms the foundation of functional languages.

- Functional languages can be thought of as the lambda calculus (in various forms) with a lot of syntactic sugar.

- A simple calculus for modeling computation

- Syntactic rules for creating expressions and converting them into other expressions.

- Not intended to be a programming language.
    - predated computers!

LISP (McCarthy, late 1950’s)

- First popular programming language to (attempt to) represent functions as values.

- Adopted some syntax from the lambda calculus, but, according to McCarthy, was not influenced greatly by the lambda calculus.

- Scheme (Steele & Sussman’75), a relatively recent dialect of LISP, has a purely functional subset (i.e. the subset without SET! and other side-effect operators).

ISWIM (Landin, mid-1960’s)

- syntax for mutually recursive function definitions

- emphasis on equational reasoning

- simple abstract machine (the SECD machine) for executing

- functional programs

FP (Backus, 1970's)

- Described in Backus's 1978 Turing Award Lecture,
- received great attention from, and had great influence on, the programming languages community
- Syntax and higher-order combining forms (fixed, limited number) similar to APL (Iverson, 1960's).
- Backus actually argued that user-defined higher order functions would lead to too much complexity (he used the term "chaos").

ML (Milner mid-70's)

- Strict functional language (includes a non-functional component, references)
- Supported higher-order functions with currying
- Static polymorphic type system with type inference
- Originally designed as the command language (hence metalanguage) for LCF, a proof system for reasoning about recursive functions.

SASL, KRC, MIRANDA (Turner, mid-70’s to mid-80’s)

- Non-strict functional languages
- Had great impact on the “standardized” lazy functional language, Haskell.
- Miranda is one of the few commercially available functional languages.

Dataflow languages

- Languages for programming dataflow (parallel) machines
- Val (Dennis, late 1970’s),
- SISL (McGraw, 1980s),
- ID (Arvind, late 1970’s)
    - many dialects of ID since!

Others

- HOPE, FEL, ALFL, LML (Lazy ML), Ponder, Orwell, . . .

The two recent functional languages generating the most interest:

<u>Standard ML</u> (Milner and others 1982 - present)

- **strict functional language (+ non-functional "references")**
- **descendent of ML (added pattern matching, for instance)**
- **parametric polymorphic type system with type inference**
- **several implementations available**

<u>Haskell</u> (by committee 1987- present)

- **non-strict functional language**
- **extension of polymorphic type system with dynamic overloading based on classes**
- **several implementations also available**

But first....

The Lambda Calculus

We'll only be talking about the *untyped* lambda calculus augmented with constants - there are many others versions.

- **Just a set of rules describing what constitutes a legal expression and conversions between expressions.**

Lambda Expressions

e ::=	c	**constant (including operators +, -, if, etc.)**
	$\mid x$	**variable**
	$\mid e1\ e2$	**application**
	$\mid \lambda x.e$	**lambda abstraction (models functions)**

Application is left associative, so

$$(e1\ e2\ e3)$$

is equivalent to

$$((e1\ e2)\ e3)$$

Examples:

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

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

$$(+\ ((\lambda x.\ +\ x\ 3)\ 4)\ 5)$$

Computation is modeled by conversions using *textual substitution* on lambda expressions.

Free variables and substitution

- **Intuitively, the *free variables* in an expression are the “non-local” variables.**

- **The free variables of an expression are defined as follows:**

$$
\begin{aligned}
fv(x) &= \{x\} \\
fv(e1\ e2) &= fv(e1) \cup fv(e2) \\
fv(\lambda x.e) &= fv(e) - \{x\}
\end{aligned}
$$

The notation $e[M/x]$ denotes the result replacing all free occurrences of the variable x with the expression M in e.

- **One has to be careful, though, to avoid name conflicts.**

$$
\begin{aligned}
x[M/x] &= M \\
y[M/x] &= y \qquad \text{where } y \text{ is a variable, } y \neq x \\
(e1\ e2)\ [M/x] &= (e1[M/x])\ (e2[M/x]) \\
(\lambda x.e)\ [M/x] &= \lambda x.e \\
(\lambda y.e)\ [M/x] &= \lambda y.(e[M/x]) \text{ where } y \neq x,\ y \notin fv(M) \\
(\lambda y.e)\ [M/x] &= (\lambda z.e[z/y])\ [M/x] \text{ otherwise,} \\
&\quad \text{where } z \neq y,\ z \neq x,\ z \notin (fv(e) \cup fv(M))
\end{aligned}
$$

Conversions between Lambda Expressions

- **α-conversion (renaming of bound variables)**

$$\lambda x.e \Leftrightarrow_{\alpha} \lambda y.e[y/x] \qquad \textit{where } y \notin fv(e)$$

- **β-conversion (application)**

$$(\lambda x.e)\ M \Leftrightarrow_{\beta} e[M/x]$$

- **η-conversion**

$$\lambda x.(e\ x) \Leftrightarrow_{\eta} e \qquad \textit{where } x \notin fv(e)$$

For the pre-defined operators, there are conversions, called δ-conversions, between an application of the operator and the result. For example,

$$(+\ 1\ 2) \Leftrightarrow_{\delta} 3$$

$$(\textit{if true e1 e2}) \Leftrightarrow_{\delta} e1$$
$$(\textit{if false e1 e2}) \Leftrightarrow_{\delta} e2$$

We write

$$e1 \overset{*}{\Leftrightarrow} e2$$

if *e1* and *e2* can be converted to one another by zero or more applications of the conversion rules (i.e. the reflexive transitive closure).

Although conversion is both ways (⇔ above) we are mainly interested in β-, δ–, and η-*reduction*, in which the conversion is only ⇒.

- **β–Reduction**

$$(\lambda x.e)\ M \Rightarrow_{\beta} e[M/x]$$

- **η–Reduction**

$$\lambda x.(e\ x) \Rightarrow_{\eta} e \qquad\qquad \textit{where } x \notin fv(e)$$

Similarly

$$e1 \overset{*}{\Rightarrow} e2$$

denotes the reduction of *e1* to *e2* by zero or more applications of the reduction rules.

We model computation as the process of taking an expression and reducing it as far as possible, to a *normal form*

- **An expression that cannot be reduced further**

Not all expressions can be reduced to a normal form.

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

has no normal form:

$$\begin{aligned}(\lambda x.\ (x\ x))\ (\lambda x.\ (x\ x)) &\Rightarrow_\beta (\lambda x.\ (x\ x))\ (\lambda x.\ (x\ x))\\ &\Rightarrow_\beta \ldots\end{aligned}$$

Reduction Order

- An expression may contain several reducible expressions, called *redex*es. For example,

$$\overbrace{((\lambda x.\ (+\ x\ x))\ \underbrace{(+\ 3\ 2)}_{\text{inner redex}})}^{\text{outer redex}}$$

  can be reduced to

$$(+\ (+\ 3\ 2)\ (+\ 3\ 2))$$

  by reducing the outer redex first, or to

$$((\lambda x.\ +\ x\ x)\ 5)$$

  by reducing the inner redex first.

- In general, there may be many redexes to choose from.

Does the order in which redexes are chosen matter?

Sure!

Consider

$$\overbrace{(\lambda y.\ 3)\ \underbrace{((\lambda x.\ (x\ x))\ (\lambda x.\ (x\ x)))}_{\textbf{inner redex}}}^{\textbf{outer redex}}$$

Reducing the outer redex first gives us

$$3$$

Reducing the inner redex first gives us

$$\begin{aligned}(\lambda y.\ 3)\ ((\lambda x.\ (x\ x))\ (\lambda x.\ (x\ x))) &\Rightarrow_\beta (\lambda y.\ 3)\ ((\lambda x.\ (x\ x))\ (\lambda x.\ (x\ x))) \\ &\Rightarrow_\beta \cdots\end{aligned}$$

The reduction of the argument never terminates, but its value isn't needed.

- **In this case, one reduction order terminated and the other didn't.**

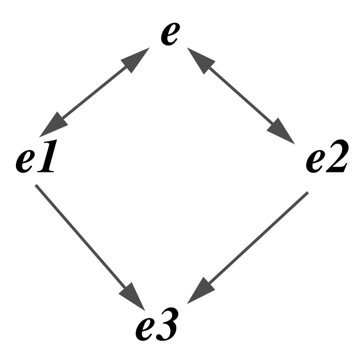
Can two terminating reductions give different answers?

Church-Rosser Theorem I

If $e1 \overset{*}{\Leftrightarrow} e2$ then there exists an $e3$ such that $e1 \overset{*}{\Rightarrow} e3$ and $e2 \overset{*}{\Rightarrow} e3$

Corollary

No lambda expression can be converted to two distinct normal forms.



- **So, all terminating reduction sequences give the same answer**

Common Evaluation Orders

- *Applicative order evaluation*: **reduce the leftmost innermost redex first.**
    - **intuitively, evaluate the arguments first**
    - **used by most programming languages, including "strict" functional languages**

- *Normal Order evaluation*: **reduce the leftmost outermost redex first.**
    - **intuitively, evaluate the body of the function first and the arguments when necessary.**
    - **used by "non-strict" functional languages**

Which is better? Well... stay tuned!

But, here is the first data point:

<u>Church Rosser Theorem II</u>
If $e1 \overset{*}{\Rightarrow} e2$ and $e2$ is in normal form, then there exists a normal-order reduction from $e1$ to $e2$.

This says that if any reduction sequence terminates, then normal order reduction will.

- **normal order reduction is the most likely to terminate!**

Recursion in the lambda calculus

It appears impossible to define recursion functions, since the functions aren't named.

- **Can't write**

  $$fac = \lambda x.\ (if\ (= x\ 0)\ 1\ (*\ x\ (fac\ (-\ x\ 1))))$$

- **So what can we do?**

First, some terminology:

- **The *fixpoint* of a function *f* is the value *e* such that**

  $$f\ e = e$$

- **For recursion in the lambda calculus, on can use the f*ixpoint combinator* Y, defined as**

  $$Y\ f = f\ (Y\ f)$$

- **For any function *f*, *(Y f)* computes *f*'s fixpoint.**

Why is this useful? Because now fac can be written as

$$Y\ (\lambda fac.\ \lambda x.\ (if\ (=x\ 0)\ 1\ (*\ x\ (fac\ (-\ x\ 1)))))$$

- **To see that this has the desired behavior, let**

$$F = \lambda fac.\ \lambda x.\ (if\ (=x\ 0)\ 1\ (*\ x\ (fac\ (-\ x\ 1))))$$

- **Notice that**

$$\begin{aligned}
(Y\ F)\ 3 &\overset{*}{\Rightarrow} Y\ (\lambda fac.\ \lambda x.\ (if\ (=x\ 0)\ 1\ (*\ x\ (fac\ (-\ x\ 1)))))\ 3 \\
&\overset{*}{\Rightarrow} (\lambda fac.\ \lambda x.\ (if\ (=x\ 0)\ 1\ (*\ x\ (fac\ (-\ x\ 1)))))\ (Y\ F)\ 3 \\
&\overset{*}{\Rightarrow} (\lambda x.\ (if\ (=x\ 0)\ 1\ (*\ x\ ((Y\ F)\ (-\ x\ 1)))))\ 3 \\
&\overset{*}{\Rightarrow} (*\ 3\ ((Y\ F)\ 2)) \\
&\overset{*}{\Rightarrow} \ldots
\end{aligned}$$

- **In general, if you want to write a recursive function of the form**

$$f = \lambda x.\ e$$

where *f* occurs free in *e*, write it in the lambda calculus as

$$Y\ (\lambda f.\ \lambda x.\ body)$$

But, hey!, the Y combinator was defined recursively!

- **No, Y is just**

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

- **To see this, for any expression e,**

$$\begin{aligned} Y\ e &= (\lambda h.\ ((\lambda x.\ (h\ (x\ x)))\ (\lambda x.(h\ (x\ x)))))\ e \\ &\Rightarrow \underline{((\lambda x.(e\ (x\ x)))\ (\lambda x.\ (e\ (x\ x))))} \\ &\Rightarrow (e\ \underline{((\lambda x.\ (e\ (x\ x)))\ (\lambda x.\ (e\ (x\ x))))}) \Leftrightarrow e\ (Y\ e) \end{aligned}$$

Church showed that the lambda calculus is a consistent mathematical system.

- **Scott and Strachey (and others) gave a mathematical semantics to the lambda calculus, showing that lambda abstractions do indeed denote values in domains of functions.**

- **Non-trivial result, since self-application cannot be described representing functions the traditional way as sets.**

Modern functional languages are essentially the lambda calculus (in some cases, a typed version) with nicer syntax!

- **Thus, the simplicity, consistency, Church-Rosser theorems, etc. all come along for free!**

Higher Order Functions

One of the elegant features of the Lambda calculus is that functions (lambda abstractions) are values. This leads to the notion of *higher order functions*

- **functions that manipulate other functions**

Functions in functional languages (as in the lambda calculus) are first class objects, they can be

- **passed as parameters to other functions,**
- **returned as results of function calls, and**
- **stored in aggregates.**

This is attractive for philosophical reasons,

- **functions are values, thus should be treated like any other value**

and for pragmatic reasons.

- **gives an additional mechanism of abstraction.**

```
fun quadrature(f, x, end, interval) =
        if x = end then 0
        else    ((f(left) + f(x+interval))/2) * interval +
                quadrature(f, x+interval, end, interval)
```

In languages without higher-order functions (or generics), you would have to write a different quadrature routine for each function.

Referential Transparency

In functional languages, as in mathematics, there is no notion of a variable being modified.

- **No assignment statement. The equation**

$$x = x + 1$$

has no solution in mathematics.

The lack of assignment ("side-effect") leads to the notion of referential transparency

- **"equals can be replaced by equals"**

If we say

```
let   x = f(a)
in    ... x + x ...
```

then we can be sure that the meaning of `x + x` **is the same as** `f(a) + f(a)`.

Assuming there is no intervening declaration of a new `x`.

Pragmatically, this is beneficial for understanding and debugging code. We simply need to look at the declaration of a variable to understand its behavior.

Standard ML.

- **Strict functional language (applicative order reduction)**
- **Statically typed.**

Primitive Types

int, real, bool, string, unit (a type with one value, ( ))

Aggregate Types

lists

[1,2,3] : int list

[true, false, true] : bool list

[[3.2, 4.5],[2.1]] : real list list

tuples

(true, 3, [4.2]) : bool * int * real list

records

{a=3,b=3.2, c=”hello”} : {a:int,b:real,c:string}

Function Types

int -> bool   real -> real-> bool-> bool

-> is right associative

- **There are also type variables - deferred until polymorphism discussion.**

Expressions

Arithmetic

x+y

3*2

Logical

a=3

4 > b

c andalso (d = 6)

Conditional

if z = 0 then 1 else f 6

Function Application

f 3

(g 4) 5

g 4 5

equivalent

List construction and selection (similar to LISP):

- **All elements of a list must be of the same type.**

<u>Cons</u>

```
x::xs
```

forms the list whose first element is `x` **and the other elements come from the list** `xs`**.**

```
3::[4,5]
```

returns [3,4,5]

Append

```
xs @ ys
```

forms the list consisting of the elements of both `xs` **and** `ys`

```
[3,4,5] @ [6,7,8]
```

returns [3,4,5,6,7,8]

Head, Tail:

```
hd [3,4,5]
```

returns 3

```
tl [3,4,5]
```

returns [4,5]

Function expressions (i.e. lambda abstractions)

```
fn x => x + 1
fn a => fn b => a + b
```

=> is right associative

Let expressions

```
let   <declaration1>
      <declaration2>
       . . .
      <declarationN>
in
      <exp>
end
```

where each *<declaration>* introduces a new name and gives it a value.

For example

```
let   val x = 6
      val g = fn z => z + 2
      fun fac n = if n = 0 then 1 else n * fac (n-1)
in
      fac (g x)
end
```

introduces the new names x, g, **and** fac **whose scope ranges from where they are introduced up to the** end **keyword.**

Recursive functions can only be defined using fun

- **or a seldom-used form:** val rec f = fn x =>...

Mutually recursive functions are defined using the and **keyword:**

```
fun    f x y = if x = 0 then y
                 else g (x-1) (y+2)
and
       g a b = f a (b* 2)
```

All ML functions take a single parameter.

The declaration

```
fun f x y = x + y + 1
```

is equivalent to

```
fun f x = fn y => x + y + 1
```

and thus can be used in

```
let   fun f x y = x + y + 1
      val g = f 2
in
      g 3 + g 4
end
```

Such function definitions are called *curried* definitions (after the logician Haskell Curry).

The function `f` above has the type

```
int -> int -> int
```

For efficiency when currying isn't needed, it is common practice to define a function as taking a single tuple as a parameter.

```
let   fun f(x,y) = x + y + 1
in    f(3,4) + f(5,6)
end
```

In this case f **has type**

```
int * int -> int
```

ML Patterns

Standard ML has a pattern matching facility to support

- **An equational style for function definitions**
- **Selecting components of aggregate values**

Equational Style

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

Selecting Components of Aggregate Values

```
val (x,y) = (3.4, 5)
```

introduces the variables x **and** y**, and**

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

binds x **to the head of the argument and** xs **to the tail.**

User-defined types in ML

Type Synonyms

type *<name>* = *<type expr>*

introduces a new name for the type described by *<type expr>*

- **It does not create new type, just a synonym for an existing one.**
- **Examples:**

```
type foo = int * bool * real
type bar = string
```

New types

Defined using the datatype **construct.**

In its simplest form, like an enumerated type in Pascal or Ada.

```
datatype stoplight = red | green | yellow
```

defines a new type stoplight **whose values are** red, green, **and** yellow.

In the more general form, the elements of an ML datatype can be *value constructors*, which can take a parameter.

```
datatype tree = empty | leaf of int | node of tree * tree
```

Here, leaf **is a value constructor taking an integer parameter and** node **is a value constructor taking a tuple of two** trees.

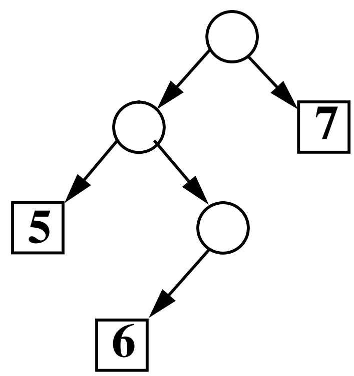
- **They are called value constructors because, when applied to their argument, they construct a value (of type** tree **in this case).**

- **The values** red, green, **and** yellow **above are simply nullary value constructors.**

The expression

node (node (leaf 5, node (leaf 6, empty)),(leaf 7))

constructs the tree



Value constructors can be used in patterns:

```
fun  drive red = “stop”
 |   drive green = “go”
 |   drive yellow = “accelerate”
```

The patterns can be used to select out the arguments to the value constructors

```
fun  fringe empty = []
 |   fringe (leaf x) = [x]
 |   fringe (node (t1,t2)) = fringe t1 @ fringe t2
```

So

fringe (node (node (leaf 5, node (leaf 6, empty)),(leaf 7)))

would return [5,6,7]

Type Variables and Parametric Polymorphism

Consider the length **function again:**

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

What is its type?

- **It can take a list of any type and returns an int.**

That is, its type is

$$\forall\alpha.\ \alpha\ \textit{list -> int}$$

- **In ML, this type is written**

  ‘a -> int

  where the ‘ signifies that the a **is a universally quantified type variable (rather than a type named** a**).**

- **Thus, length can take many different types of arguments (an infinite number, in theory). We say that length is *polymorphic* (“many shaped”).**

  length [1,2,3] + length [[4,5],[6]] + length [true, false, true]

- **Any object whose type contains a type variable is polymorphic. All others are said to be *monomorphic*.**

- **This kind of polymorphism is called *parametric polymorphism*.**
  - **In some theoretical type models, the type variable is written as a formal *parameter* in a polymorphic definition.**

Here is another example of a polymorphic function:

```
let
    fun    map f [] = []
     |     map f (x::xs) = f x :: map f xs
in
    (map (fn n => n+1) [1,2,3]) @
    (map (fn l => length l) [[2.2, 3.3],[4.4]])
end
```

where map has type

```
 ('a -> 'b) -> 'a list -> 'b list
```

and the result of the entire expression is [2,3,4,2,1].

Polymorphic Type Inference

Notice that we never specified the types of functions or variables that we declared.

- **The ML compiler figures out their types by the way they are used.**

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

$\alpha \rightarrow \beta$ $\quad \alpha$ $\quad \alpha$ list $\quad \beta$ $\quad \beta$ list

This is called type inference. The ML type system, based on work by Hindley, Milner, and others, is designed so that type inference can be performed.

- **The type it infers for an object is always the most general possible type, allowing it to be used as polymorphically as possible.**

In order to support type inference, there is a restriction that must be followed:

- **Formal parameters in a function definition must be used monomorphically within the function.**

That is, all occurrences of the formal parameter must have the same type.

```
fun f g = g 3 + g 4 + 2
```

This is fine. All uses of g **are of type** int -> int -> int

```
fun h g = (g 3.2, g true)
```

This is a type error, due to the application of g **to arguments of different types.**

Type Constructors

Type variables can also be used to parameterize datatype declarations.

Before, we defined a tree type with integer labels at the leaves:

```
datatype tree = empty | leaf of int | node of tree * tree
```

Instead, we can say

```
datatype ‘a tree = empty | leaf of ‘a | node of (‘a tree * ‘a tree)
```

allowing many different types of trees to be created. In this case, tree **is a *type constructor*, because it takes a parameter and constructs a new type (at compile time, of course).**

```
node (leaf 3.2, empty) : real tree
node (leaf [3,4,5], leaf [4,5,6]]) : int list tree
```

The type variable ‘a **can only be instantiated a one way within a single type, so**

```
node (leaf 4, leaf true)
```

is a type error.

Nullary value constructors are an example of polymorphic <u>non-function</u> values

empty: ‘a tree

Polymorphic functions work well with type constructors:

```
fun  fringe empty = []
 |   fringe (leaf x) = [x]
 |   fringe (node (t1,t2)) = fringe t1 @ fringe t2
```

has type ‘a tree -> ‘a list**.**

ML References

A non-functional component of the language

- **Provides aliasing and assignment (think "pointer")**

```
let   val x = ref 6        (* x points to cell containing 6 *)
      val y = x            (* y points to same cell as x *)
      x := 7               (* cell is modified to contain 7 *)
in
      !y                   (* returns 7 *)
end                        (* ! is dereference operator *)
```

An ML program without the use of references is a purely functional program (ignoring issues of I/O).

Non-Strict Functional Languages

Normal order reduction in the lambda calculus, revisited:

Advantages:

- **Most likely evaluation order to terminate**
- **Doesn't evaluate parameters that aren't needed.**

Disadvantage: Consider

$$
\begin{aligned}
((\lambda x.\ (+\ x\ x))\ (+\ 3\ 2)) \quad & \Rightarrow_{\beta} (+\ (+\ 3\ 2)\ (+\ 3\ 2)) \\
& \Rightarrow_{\delta} (+\ 5\ (+\ 3\ 2)) \\
& \Rightarrow_{\delta} (+\ 5\ 5) \\
& \Rightarrow_{\delta} 10
\end{aligned}
$$

Duplication of effort, due to textual substitution of each occurrence of the formal parameter with the actual parameter.

- **notice resemblance to call by name in Algol60**

Implementation Solution: Lazy Evaluation

- **Also referred to as “call by need”**

Instead of replicating the actual parameter for each occurrence of the formal parameter, have each occurrence of the formal parameter *point* to the actual.

$$((\lambda x.\ (+\ x\ x))\ (+\ 3\ 2)) \Rightarrow_{\beta} (+\quad)\ (+\ 3\ 2))$$

When the actual is evaluated, overwrite it with the result.

$$\Rightarrow_{\beta} (+\quad)\ \ 5$$

Thus, the actual is only evaluated once. Subsequent references to the formal simply retrieve the computed value.

- **β–reduction becomes primarily graph manipulation, rather than textual manipulation.**

Another disadvantage of non-strict languages:

- **Overhead cost of building object to represent delayed actual parameter.**
    - **may be a "thunk" (parameterless procedure) that will be called when the formal parameter is referenced.**
    - **may be a graph representing the delayed expression (in systems which use "graph reduction" to implement the graphical form of β-reduction.**

The theoretical property of normal order evaluation is nice:

- **Most likely reduction order to terminate**

But, we've managed to live with applicative order languages for a long time, and non-termination is seldom a concern in correct programs.

What is the practical benefit?

Frees programmers from worrying about some control issues:

- **“How much of this result should I compute”**
- **“What is the best order for the results to be computed in”**
- **“I’m not sure I’ll need this value, but I’ll compute in anyway just in case”**

Example: Attribute Grammars with Synthesized and Inherited Attributes.

- **“How do I get the data dependencies right?”**

<u>Allows use of infinite data structures!</u>

cons **(::) does not evaluate its arguments. They will be evaluated when their value is demanded by** hd **and** tl**.**

For example,

```
let
      fun numsfrom n = n :: numsfrom (n+1)
in
      numsfrom 1
end
```

would never terminate (until it ran out of memory) in ML.

In a non-strict language, the result of this expression would be a list whose head is 1 and whose tail is the delayed value of numsfrom (n+1)

The value of the above let **expression is the infinite list** [1,2,3, ...]

- **These infinite, but delayed, lists are generally called *streams*.**

Given

```
fun sumstream s 0 = 0
|    sumstream s n = hd s + sumstream (tl s) (n-1)
```

the evaluation of

```
sumstream (numsfrom 1) 10
```

would cause the first 10 elements of the stream to be computed.

The section of code producing the stream does not need to know how much of the stream to evaluate. That is done on demand.

Typical stream-based program: The infinite list of primes

```
let
    fun numsfrom n = n :: numsfrom (n+1)
    fun filter f (x::xs) = if f x then x :: filter f xs
                                  else filter f xs
    fun remove-mults (x :: xs) =
        let
            fun is-mult n = (n mod x) <> 0
        in
            x :: remove-mults (filter is-mult xs)
        end
in
    remove-mults (numsfrom 2)
end
```

Haskell

- Modern non-strict functional language
- normal order semantics ("demand driven" evaluation)
- extends ML's type system for dynamic overloading

The syntax of Haskell differs somewhat from ML, but has a similar look.

A few important syntactic differences:

- Identifiers representing specific types and value constructors are capitalized. Identifiers representing type variables and values are not capitalized.
- Definitions do not begin with a keyword (ML uses val or fun)
- ML and Haskell exchanged their uses of : and ::
- Indentation can be used to begin and end new blocks.

```
let   y   = a * b
      f x = (x+y)/y
in
      f c + f d
```

The Haskell Class system

<u>Overloading</u>: Giving the same name to distinct entities in a program.

- **Ada: Two or more functions can share the same name.**
  - **Function calls are disambiguated by the types of the arguments and the result type.**
  - **Overload resolution occurs at compile type, hence is called *static overloading*.**

Mixing static overloading and parametric polymorphism can cause trouble.

Consider the following definition in ML:

```
fun f x y = x + y
```

this is a type error!

- **The overloading of + cannot be resolved.**
- `x` **and** `y` **are either integers or reals, but you can't tell which.**
- **f is clearly not of type** `‘a -> ‘a -> ‘a`

You would like to say that f **is of type**

$$\forall\ \alpha\ \textit{for which + is defined.}\ \alpha \to \alpha \to \alpha$$

- **Of course, + may be very different for the different types that it is defined on.**
- **When** f **is called, the choice of which + to use in the body of** f **depends on the arguments passed in, and must be determined dynamically.**
  - **this is called** ***dynamic overloading*****.**

Haskell's <u>type classes</u> support dynamic overloading.

A type class specifies what operations a type must support.

- **Types are then declared to be instances of that class.**

For example, the Equality class, Eq**, defined by**

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

specifies that any type a **in class** Eq **must provide a definition for the infix == operator of type** a -> a -> Bool**.**

One can then write a polymorphic function that uses ==

```
f :: (Eq a) => a -> a -> Int
f x y = if x == y then 1 else 2
```

- **The** (Eq a) **is called a *context*.**

- **The type of** f**, written above its definition, is**

  $\forall\ \alpha$ in class Eq. $\alpha \to \alpha \to$ Int

User Defined Types

New types in Haskell are defined using the data **construct.**

- **Almost identical to ML's** datatype

```
data IntTree = Leaf Int | Node IntTree IntTree
```

Declaring a Type to be an Instance of a Class

To declare a type to be an instance of a class, the definitions of the required operations must be provided.

- **For example, class EQ requires the == operation to be defined.**

Instance Declaration:

```
instance Eq IntTree where
        Leaf x == Leaf y                 =  x == y
        (Node l1 r1) == (Node l2 r2 )    =  l1 == l2 && r1 == r2
        _ == _                           =  False
```

Now IntTree can be passed to any function expecting a type in class Eq.

- **There are a large number of predefined types in class Eq.**

User-define type constructors

Like ML, Haskell has type constructors parameterized by type variables.

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
```

Could the following instance declaration be used to declare Tree an instance of Eq?

```
instance Eq (Tree a) where
    Leaf x == Leaf y                 = x == y
    (Node l1 r1) == (Node l2 r2 )    = l1 == l2 && r1 == r2
    _ == _                           = False
```

No! The types of x and y might not be in class Eq, thus == might not be defined on them.

Solution: Need a context in the instance declaration

```
instance (Eq a) => Eq (Tree a) where
    Leaf x == Leaf y                 = x == y
    (Node l1 r1) == (Node l2 r2 )    = l1 == l2 && r1 == r2
    _ == _                           = False
```

This says that if a type a is an instance of Eq, then so is (Tree a).

Default Methods

A class can give a default definition for an operation

```
class Eq a where
    (==)              :: a -> a -> Bool
    x /= y            = not (x == y)
```

Any instance declaration not providing a definition of /= would use the one above.

Class Inclusion

Haskell provides class inclusion, a form of inheritance

- **One class definition can be used to define another one.**

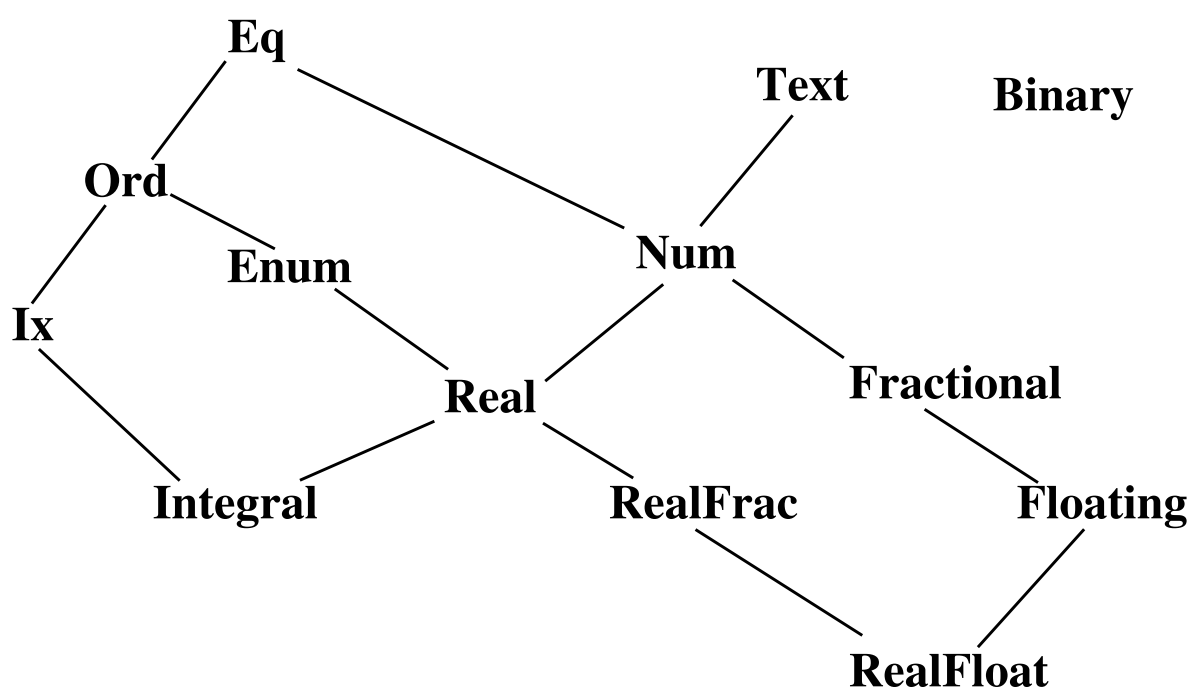
For example,

```
class (Eq a) => Ord a where
    (<), (<=), (>=), (>)          :: a -> a -> Bool
    max, min                      :: a -> a -> a
```

defines the class of ordered types in terms of the class of equality types.

- **Eq is a *superclass* of Ord**
- **Ord is a *subclass* of Eq**

Here is a diagram of the hierachy of Haskell predefined classes



List Comprehensions in Haskell

These are concise expressions for constructing entire lists

- **Resembles set notation in mathematics**

The expression

```
[ f x | x <- xs ]
```

computes the list of all (f x) such that x is taken from the list xs.

List comprehensions can also include guards:

```
quicksort []        = []
quicksort(x:xs)     =     quicksort [y | y <- xs, y<x]
                       ++ [x]
                       ++ quicksort [y | y <- xs, y>=x]
```

Research Issues in Functional Languages

Strictness Analysis

The advantage of using a lazy functional language is that an unneeded argument is never evaluated.

- **Delaying the evaluation of the argument incurs some overhead, though.**

What if the argument is always going to be needed?

- **i.e. if the function is "strict" in its argument.**

Compiler Optimization:

- **Transform call-by-need into call-by-value**
    - **That is, go ahead and evaluate the argument first.**

Strictness Analysis determines when this is safe to do.

Formal definition of strictness: A function f of the form

$$f\ x_1\ \ldots\ x_n = e$$

is strict in its *i*^th argument if, for all values y_j ,

$$f\ y_1\ \ldots\ y_{(i-1)}\ \perp\ y_{(i+1)}\ \ldots\ y_n = \perp$$

where ⊥ denotes non-termination. That is, if the *i*^th argument to the function doesn't terminate, then the function call won't either. Intuitively, this means that the function always needs the value of the *i*^th argument.

Not quite, though:

```
f x y = if x = 0 then y else f (x-1) y
```

is strict in its second argument, even though it may never access it. That is, if y doesn't terminate, then either the call to f won't terminate because y is accessed or it won't terminate because the recursion never ends.

- **Either way, f v ⊥ = ⊥, for all possible values of v.**

Strictness Analysis is quite well understood now. It typically uses an analysis framework called Abstract Interpretation.

- **Has been used to determine the strictness of higher-order functions, and for arguments that are aggregate structures (such as lists).**

Parallel Functional Programming

Church-Rosser Theorem I says that all terminating reduction sequences give the same result.

- **Even parallel reduction sequences, where multiple redexes are reduced simultaneously.**

Therefore, functional languages are obvious candidates for parallel programming.

If there is an expression

```
f x y + g a b
```

then we know that `(f x y)` **and** `(g a b)` **can be evaluated in parallel.**

- **Not true in imperative languages, where** `f` **and** `g` **might perform side-effects that affect each other.**

Likewise, in a strict functional language,

```
f(e1, e2, e3)
```

could be evaluated by computed e1, e2, and e3 in parallel.

In a non-strict functional language, strictness analysis can be used.

- **The strict arguments can be computed in parallel.**

A sizeable number of parallel functional programming systems have been built.

The difficult part of parallel functional programming is finding the right *granularity*

- **The size (in terms of # of instructions) of the smallest pieces that the program is decomposed into.**

How do you know, given

```
f x y + g a b
```

if it is worthwhile computing (`f x y`) **and** (`g a b`) **in parallel?**

- **The cost of spawning the tasks, sending data, etc. might outweigh any benefits.**
    - **Especially on distributed-memory machines.**

There are a number of granularity analyses, but it is still an open (and undecidable!) problem.

State in Functional Languages

Criticism of the functional programming paradigm:

- **State, and changes of state, occur in the real-world.**
- **Thus, can't easily solve real-world problems in a "stateless" language**
    - **i.e. a language without variables that can be modified**
- **However, with the addition of modifiable variables, referential transparency is lost.**

There has been much recent work on adding constructs to functional languages that add the notion of state to functional programs, but limit the way the state is accessed and updated.

- **Referential transparency is preserved.**