The Haskell Programming Language

Greg J. Badros
badros@cs.washington.edu
University of Washington, Seattle
CSE-341, Summer 1999
(C) 1999, Greg J. Badros—All Rights Reserved

An imperative language

```
int i = 7;

printf("%d\n",7);
```

- Can we substitute `i` for `7` in the `printf`?

Scheme is still imperative

```
(let ([i 7])
  i
 )
```

- Can we substitute 7 for i?

A more realistic example

```
(map + (window-size) (window-position))
(map + (window-size) (window-position))
```

- Want to factor out the duplicated code!

Factoring out an expression

```
(let ([se-corner
       (map + (window-size) (window-position))]
      se-corner
      se-corner
```

- Works ***as long as*** window-size, window-position always return same value, and no set! procedures affecting se-corner

Haskell: Works with fewer ifs and buts

- Almost no restrictions on when you can move expressions around without changing the meaning of the code
- Haskell permits "equals to be substituted for equals"
- This is called **referential transparency**

Overview of Haskell

- Purely functional
  No set!, no assignment— only bindings
- Fewer parentheses, more special syntax
- Pattern matching
- Static, polymorphic types
- Type inference
- Lazy evaluation of arguments
- Currying

Haskell's History

- Designed by committee in 1987
- Named after logician Haskell B. Curry
- Descends from Miranda
- Miranda is purely functional also
- Evolved into Haskell 1.4, Haskell 98
- Especially popular in Great Britain, Australia
  Used as introductory programming language!

The brevity of Haskell

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

List notation

List concatenation

Read as in mathematical notation:
list of values y, drawn from list xs, with y>=x

- Recognize this procedure?

Quicksort in two lines!

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


Haskell syntax

- More syntactic features than Scheme
- Generally more concise than Scheme
- Both infix operators and functions
- Whitespace and indentation can matter
- Very readable code,
  but sometimes harder to write
- Sometimes quirky precedence rules
  Use parentheses when in doubt, e.g. (-9)

Syntax: Haskell vs. Scheme

Haskell		Scheme	
--	comment	;	
{- -}	nesting comments	#| |#	
( )	tuple	#( )	only similar
[ ]	list (homogeneous)	( )	heterogenous
:	cons	cons	
=	bindings	let	
!!	selection	vector-ref, list-ref	
::	has type	*not applicable*	

Hugs98
Haskell Users Gofer System

- Nearly complete implementation of Haskell98
- Less fancy than Dr. Scheme—basically just a repl (read eval print loop)
- Commands to the interpreter begin with ":"; e.g. :?, :q, :load, :reload
- Other lines are treated as expressions to evaluate

One gotcha of Hugs98

- Files must be used to hold the definitions of functions that you write
  You cannot define functions at the repl!
- Use a separate text editor to create and edit those files
- Then :load the file in and use the functions you defined

Learning Haskell

- Try things at the repl
- Generate hypotheses, test them, re-evaluate the hypotheses based on the evidence you collect
- Ask questions!

Some simple expressions

1 + 2 ⇒ 3	number 3
1+2 ⇒ 3	number 3
"foo" ⇒ "foo"	string "foo"
'a' ⇒ 'a'	character 'a'
[1,2,3]++[4,5] ⇒ [1,2,3,4,5]	list of nums
1:[2,3,4] ⇒ [1,2,3,4]	list of nums
let x = 2 in x*3 ⇒ 6	number 6

When things do not go according to plan...

1:2 ⇒
ERROR: Unresolved overloading
*** Type : (Num a, Num [a]) => [a]
*** Expression : 1 : 2

1:[2] ⇒ [1,2]

- Haskell does not allow improper lists

More errors...

1:'a' ⇒
ERROR: Type error in application
*** Expression : 1 : 'a'
*** Term : 'a'
*** Type : Char
*** Does not match : [a]

- Haskell does not allow heterogeneous lists—all elements must be same kind of value

A simple example

```
-- length is built-in… this is just an example
length [] = 0
length (x:xs) = 1 + (length xs)
```

- Looks like we have defined the function "length" twice…
- Not really—just using pattern matching

Fibonnaci numbers Mathematical notation

f(0) = 1
f(1) = 1
f(n) = f(n-1) + f(n-2)

f(n) = { n==0? 1 ; n==1? 1 ; otherwise f(n-1) + f(n-2) }

Fibonnaci numbers in Haskell Two different approaches

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

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


Indentation matters!

ERROR "l.hs" (line 10): Syntax error in declaration (unexpected `;', possibly due to bad layout)

- **;** and **{ }** can be used as explicit grouping constructs
- Haskell's "layout" syntax is two-dimensional
- **;** and **{ }** are inserted automatically based on code indentation

Implicit vs. explicit grouping

```
let   y   = a*b
      f x = (x+y)/z
in f c + f d
-- rewritten internally to:
let {y   = a*b
   ; f x = (x+y)/z
  }
in f c + f d
```


Explicit grouping is useful at the repl

- repl only accepts a single line of input
- Use: `let x = 1; y =2 in x+y`
- There's more than one way to do it (TMTOWTDI):
  `x+y where x=1; y=2`

```
module MyPlayStuff where
-- definitions here
```

Function application

- We wrote (proc arg1 arg2 ...) in Scheme
- Haskell uses "juxtaposition"—writing the argument next to the name of the procedure:
  f x y
  or
  f (x,y)

Functions with multiple arguments

- Two ways to define mymul:

Takes two arguments

```
mymul_c x y = x * y
```

Takes one argument that is a tuple

```
mymul (x,y) = x * y
```


Currying

```
mymul_c x y = x * y
```

- mymul_c 5 2 ⇒ 10
- mymul_c 5 ⇒ <<function>>

This function could be written:

```
mymul_c_5 y = 5 * y
```

or:

```
\ y -> 5 * y
```

\ is Haskell syntax for λ, an anonymous function constructor

Partial application

```
mymul_c 5 2   ⇔   (mymul_c 5) 2
                  mumul_c_5 2
                  (\ y -> 5 * y) 2
              ⇒ 10
```

- Name "specializations" via partial application—omit trailing argument(s):

```
mulby5 = mymul_c 5
```


Specializations

- Consider map: (map fn items answer)
  :type map → :: (a -> b) -> [a] -> [b]
- Specializations let us bind leading arguments, giving them pre-specified values:

```
negator = map negate
:type negator → :: [Integer] -> [Integer]
                   items      answer
negator [1,2,3] ⇒ [-1,-2,-3]
```


Lambda expression syntax

- \ (looks most like λ) introduces anonymous function

```
\ arg1 arg2 arg3 -> expression
```

```
(\ x y -> x + y) 4 3 ⇒ 7
```

Anonymous function

Functions vs. operators

- mymul is a function, * is an operator
  mymul 2 3 -- prefix for function application
  -- but
  2 * 3 -- infix for operator application
- Can use function names as operators…
  2 `mymul` 3
- … and vice versa
  (*) 2 3

Other operators

==, <=, <, >, >=
/= not equal to
^ raise to the power (e.g., 2^3 ⇒ 8)
`div` whole number division
`mod` whole number remainder
. function composition
(f . g) args = f(g(args))

Why use tuples for arguments?

- Sometimes the argument really is a tuple
- Pattern matching can let you decompose the arguments from inside a tuple:
  -- suppose multiplication were a
  -- very expensive operation…
  mymul (x,0) = 0
  mymul (0,y) = 0
  mymul (x,y) = x * y

Ackermann function
(Wilhelm Ackermann, 1928)

A(0,y) = y+1
A(x,0) = **A**(x-1,1)
A(x,y) = **A**(x-1,**A**(x,y-1))

A(2,3) ⇒ 9

- Grows really quickly—**A**(4,2) is about 10^{19728} (but do not try to compute even **A**(4,1)!)

Patterns

- Used to match structural properties of values (often arguments)
- First (from top to bottom in program text) matching pattern to succeed provides the definition
  -- return first n elements of items
  take 0 items = []
  take n [] = []
  take n (x:xs) = x : take (n-1) xs

Wildcards in patterns

mymul (x,0) = 0
-- could have written
mymul (_,0) = 0
Just a placeholder, makes no binding

-- another example
isZero 0 = True
isZero _ = False

Patterns and case expressions

- Patterns are just sugar for a more general case expression

```
take m ys  = case (m,ys) of
               (0,_)    -> []
               (_,[])   -> []
               (n,x:xs)-> x : take (n-1) xs
```


Conditionals

- Patterns have eliminated much of the need for conditionals!
- Still have if-then-else expression syntax:
  if *test* then *exprA* else *exprB*
- Just sugar for:
  case *test* of True -> *exprA*
  False -> *exprB*

List shorthands

[1,2..5] ⇒ [1,2,3,4,5]
[1,3..9] ⇒ [1,3,5,7,9]
[0.0,0.3 .. 1.0] ⇒ [0.0,0.3,0.6,0.9]

➪Even can make infinite lists!
[1,2..] ⇒ [1,2,3,4,5,6,7,...]
[1,3..] ⇒ [1,3,5,7,9,11,...]

List comprehension

```
[x | x <- xs, x<3 ]    -- all x<3 from list xs
[ 2*x | x <- xs ]      -- double elements in xs

-- double elements y<x from list xs
[ 2*x | x <- xs, x<3 ]
-- double even elements y<x from list xs
[ 2*x | x <- xs, x<3, isEven x ]
```


Compare to using higher order functions

```
[x | x <- xs, x<3 ]   ⇔ filter (\y -> y<x) xs
[ 2*x | x <- xs ]     ⇔ map (\x -> 2*x) xs
[ 2*x | x <- xs, x<3 ] ⇔
  map (\x -> 2*x) (filter (\x -> x < 3) xs)

[ 2*x | x <- xs, x<3, isEven x ] ⇔
  map (\x -> 2*x)
      (filter (\x -> x < 3 && isEven x) xs)
```


Comprehension instead of nested loops

y is the inner loop "index"

```
[ (x*2,y) | x<-[1..2], y<-[1..3] ]
    ⇒ [(2,1),(2,2),(2,3),(4,1),(4,2),(4,3)]
```

x is the inner loop "index"

```
[ (x*2,y) | y<-[1..3], x<-[1..2] ]
    ⇒ [(2,1),(4,1),(2,2),(4,2),(2,3),(4,3)]
```

Values have types

Prelude> :set +t ← Ask Hugs to report types of values

5 ⇒ 5 :: Integer
"Hi" ⇒ "Hi" :: String
0==1 ⇒ False :: Bool
('a',2.0) ⇒ ('a',2.0) :: (Char,Double)

:: is read "has type"

Functions have types too

length ⇒ <<function>> :: [a] -> Int

(*) ⇒ <<function>>
:: Integer -> Integer -> Integer

head ⇒ <<function>> :: [a] -> a

map ⇒ <<function>> :: (a->b) -> [a] -> [b]

Built in types

- Int vs. Integer
  - Int is fixed-precision integer
  - Integer is arbitrary precision integer
- Double vs. Float
  - Double is double-precision
  - Float is single precision

Dynamic vs. Static typing

- Scheme used dynamic types:
  (define (f y) (* y "hi"))
  ;;; No error in above definition
  (f 5) ⇒ Error: expects type <number> as second arg.
- Haskell catches the error earlier:
  f y = y * "hi" ⇒
  Instance of Num [Char] required for definition of f

Advantages of static typing

- Catch errors earlier
- Find errors in untested code
- Avoid run-time argument checking
  - code that has passed the "type-checker" is guaranteed to have no improper applications (such as multiplying a number by a string)
  - code executes faster
- Better program documentation

Static typing rules

[From Thompson, Ch. 9]

- Application
- Cancellation
- Addition
- Conditional
- Function
- Tuple
- List

Application rule

If f has type s->t **and** e has type s
then f can be applied to e, and the result
f e :: t
has type t.

Cancellation rule

If f has type t_1->t_2->...->t_n->t **and**
e_1 has type t_1
⋮
e_k has type t_k, where k ≤ n
then f can be applied to e_1, ..., e_k
and the result, f e_1 ... e_k, has type
t_{k+1} -> t_{k+2} -> ... -> t_n -> t

Function rule

If x has type s and e has type t,
then \ x -> e has type s->t.

List rule

If e_1, ..., e_k have type t
then [e_1,...,e_k] has type [t].

➪Only homogeneous lists are permitted

Tuple rule

If e_1 has type t_1, ..., ek has type t_k
then (e1, ..., ek) has type:

Length of list of integers

```
len :: [Int] -> Int  -- you can declare a type
len [] = 0
len (x:xs) = 1 + len xs

len ⇒ <<function>> :: [Int] -> Int
len [1,2,3] ⇒ 3 :: Int
```

Applying len to [Char]

```
len ['a','b'] ⇒
  ERROR: Type error in application
  *** Expression     : len ['a','b']
  *** Term           : ['a','b']
  *** Type           : [Char]
  *** Does not match : [Int]
```


What does len rely on?

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

- All that matters is that the argument is a list of something
- With above definition:
  len ⇒ [a] -> Integer

Type variables

len ⇒ [a] -> Integer

"a" stands for any type
the type is a template for many types

head ⇒ <<function>> :: [a] -> a

map ⇒ <<function>> :: (a->b) -> [a] -> [b]

Polymorphic functions

- length can operate on any kind of list ([a]) and return an Integer
- Polymorphic = many shaped
- Many functions in Haskell are polymorphic
- You can (and do!) write polymorphic functions too

Type inference

- You often need not specify types
- Haskell looks inside your function and figures it out:
  f (x,y) = (x, ['n' .. y])
  f :: (a, Char) -> (a, [Char])

Type unification

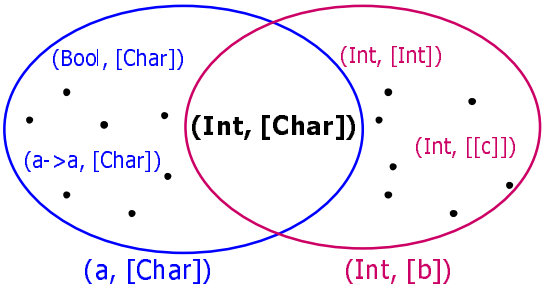
```
g (m,zs) = m + length zs
g :: (Int , [b]) -> Int
```

- Suppose we want h = g . f
  (the output of f is applied as input to g)

```
        g                        f
(Int, [b]) -> Int      (a, Char) -> (a, [Char])
```

- Must "unify" types:
  (a, [Char]) and (Int, [b])

Unification is the intersection of the described types



Evaluation in Haskell

- Recall Scheme:
  (multiply (sum 1 2) (sum 3 4)) →
  (multiply 3 7) → (* 3 7) ⇒
  21
- multiply (sum 1 2) (sum 3 4) →
  (sum 1 2) * (sum 3 4) →
  3 * 7 ⇒
  21

Not all arguments get evaluated

- In Scheme:
  (car (list (+ 1 2) (/ 5 0))) ⇒ ERROR!
- In Haskell:
  head [1+2, 5/0] ⇒ 3.0
  -- only evaluate the arguments we use

➪No error, because we discard the second element of the list before using it!

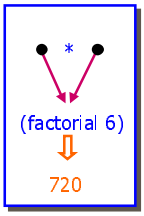
Eager evaluation vs. lazy evaluation

- Eager evaluation — "applicative-order"
  - Proceeds "inside-out"
  - All arguments are evaluated before function is applied
- Lazy evaluation
  - Proceeds "outside-in"
  - Arguments are passed in to function as full expressions

Lazy evaluation is efficient

square x = x * x

square (factorial 6) →
(factorial 6) * (factorial 6) →
720 * 720 ⇒ 518400



- factorial 6 is only invoked once!

No need for special forms

myif True exprTrue _ = exprTrue
myif False _ exprFalse = exprFalse

- Only one of the two expressions gets evaluated
- Remember, since Haskell has no side effects, this is simply a matter of efficiency

Outside-in evaluation avoids intermediate lists

```
sumFourthPowers n = sum (map (^ 4) [1 .. n])
```

sumFourthPowers 3 →
  sum (map (^ 4) [1..3]) →
  sum ((^ 4) 1 : map (^ 4) [2..3])) →
  (1^4) + sum (map (^ 4) [2..3])) →
  1 + sum (map (^ 4) [2..n]) →
  ...
  1 + (16 + (81)) ⇒ 98