Programming Languages: Functional Programming 1. Introduction to Haskell: Value, Functions, And Types

Shin-Cheng Mu

Autumn 2023

A Quick Introduction to Haskell

- We will mostly learn some syntactical issues, but there are some important messages too.
- Most of the materials today are adapted from the book *Introduction to Functional Programming using Haskell* by Richard Bird. Prentice Hall 1998.
- References to more Haskell materials are on the course homepage.

Course Materials and Tools

- Course homepage: https://scmu.github.io/ plfp/
 - Announcements, slides, assignments, additional materials, etc.
- We will be using the Glasgow Haskell Compiler (GHC).
 - A Haskell compiler written in Haskell, with an interpreter that both interprets and runs compiled code.
 - See the course homepage for instructions for installation and other info.

Function Definition

• A function definition consists of a type declaration, and the definition of its body:

square :: Int
$$\rightarrow$$
 Int
square x = x \times x
smaller :: Int \rightarrow Int \rightarrow Int
smaller x y = if x \leq y then x else y

- The GHCi interpreter evaluates expressions in the loaded context:
 - ? square 3768 14197824 ? square (smaller 5 (3+4)) 25

1 Values and Evaluation

Evaluation

One possible sequence of evaluating (simplifying, or reducing) square (3+4):

$$square (3+4)$$

$$= \{ \text{ definition of } + \}$$

$$square 7$$

$$= \{ \text{ definition of } square \}$$

$$7 \times 7$$

$$= \{ \text{ definition of } \times \}$$

$$49$$

Another Evaluation Sequence

• Another possible reduction sequence:

$$square (3+4)$$

$$= \{ definition of square \}$$

$$(3+4) \times (3+4)$$

$$= \{ definition of + \}$$

$$7 \times (3+4)$$

$$= \{ definition of + \}$$

$$7 \times 7$$

$$= \{ definition of \times \}$$

$$49$$

- In this sequence the rule for *square* is applied first. The final result stays the same.
- Do different evaluations orders always yield the same thing?

A Non-terminating Reduction

Consider the following program:

three :: Int \rightarrow Int three x = 3infinity :: Int infinity = infinity + 1

• Try evaluating *three infinity*. If we simplify *infinity* first:

three infinity
= { definition of infinity }
three (infinity + 1)
= three ((infinity + 1) + 1)...

• If we start with simplifying three:

Evaluation Order

- There can be many other evaluation orders. As we have seen, some terminates while some do not.
- normal form: an expression that cannot be reduced anymore.
 - 49 is in normal form, while 7×7 is not.
 - Some expressions do not have a normal form. E.g. *infinity*.
- A corollary of the *Church-Rosser theorem*: an expression has at most one normal form.
 - If two evaluation sequences both terminate, they reach the same normal form.

Evaluation Order

- Applicative order evaluation: starting with the innermost reducible expression (a redex).
- Normal order evaluation: starting with the outermost redex.
- If an expression has a normal form, normal order evaluation delivers it. Hence the name.
- For now you can imagine that Haskell uses normal order evaluation. A way to implement normal order evaluation is called *lazy evaluation*.

2 Functions

Mathematical Functions

- Mathematically, a function is a mapping between arguments and results.
 - A function $f :: A \rightarrow B$ maps each element in A to a unique element in B.
- In contrast, C "functions" are not mathematical functions:

- int y = 1; int f (x:int) { return
 ((y++) * x); }

- Functions in Haskell have no such *side-effects*: (unconstrained) assignments, IO, etc.
- Why removing these useful features? We will talk about that later in this course.

2.1 Using Functions

Curried Functions

• Consider again the function *smaller*:

smaller :: *Int* \rightarrow *Int* \rightarrow *Int smaller* $x y = \text{if } x \le y$ then x else y

- We sometimes informally call it a function "taking two arguments".
- Usage: smaller 3 4.
- Strictly speaking, however, *smaller* is a function returning a function. The type should be bracketed as $Int \rightarrow (Int \rightarrow Int)$.

Precedence and Association

- In a sense, all Haskell functions takes exactly one argument.
 - Such functions are often called *curried*.
- Type: $a \rightarrow b \rightarrow c = a \rightarrow (b \rightarrow c)$, not $(a \rightarrow b) \rightarrow c$.
- Application: f x y = (f x) y, not f (x y).
 - smaller 3 4 means (smaller 3) 4.
 - square square 3 means (square square) 3, which results in a type error.
- Function application binds tighter than infix operators. E.g. *square* 3+4 means (*square* 3)+4.

Why Currying?

• It exposes more chances to reuse a function, since it can be partially applied.

 $\begin{array}{ll} twice & :: (a \to a) \to (a \to a) \\ twice \ f \ x = f \ (f \ x) \\ quad & :: Int \to Int \\ quad & = twice \ square \end{array}$

• Try evaluating quad 3:

quad 3 = twice square 3 = square (square 3) = ...

· Had we defined:

twice :: $(a \rightarrow a, a) \rightarrow a$ *twice* (f, x) = f(f x)

we would have to write

quad :: Int \rightarrow Int *quad* x = twice (square, x)

• There are situations where you'd prefer not to have curried functions. We will talk about coversion between curried and uncurried functions later.

2.2 Sectioning

Sectioning

- Infix operators are curried too. The operator (+) may have type $Int \rightarrow Int \rightarrow Int$.
- Infix operator can be partially applied too.

$$(x \oplus) y = x \oplus y$$
$$(\oplus y) x = x \oplus y$$

- $(1 +) :: Int \rightarrow Int$ increments its argument by one.
- (1.0 /) :: *Float* \rightarrow *Float* is the "reciprocal" function.
- (/ 2.0) :: *Float* \rightarrow *Float* is the "halving" function.

Infix and Prefix

- To use an infix operator in prefix position, surrounded it in parentheses. For example, (+) 3 4 is equivalent to 3+4.
- Surround an ordinary function by back-quotes (not quotes!) to put it in infix position. E.g. 3 'mod' 4 is the same as mod 3 4.

Function Composition

• Functions composition:

$$(\cdot) :: (b \to c) \to (a \to b) \to (a \to c)$$
$$(f \cdot g) x = f (g x)$$

• E.g. another way to write quad:

quad :: Int \rightarrow Int quad = square \cdot square

· Some important properties:

-
$$id \cdot f = f = f \cdot id$$
, where $id x = x$.
- $(f \cdot g) \cdot h = f \cdot (g \cdot h)$.

2.3 Definitions

Guarded Equations

• Recall the definition:

smaller :: *Int* \rightarrow *Int* \rightarrow *Int smaller* $x y = \text{if } x \le y \text{ then } x \text{ else } y$

• We can also write:

smaller :: Int \rightarrow Int \rightarrow Int smaller x y | $x \le y = x$ | x > y = y

• Equivalently,

smaller :: Int \rightarrow Int \rightarrow Int smaller x y | $x \le y = x$ | otherwise = y

• Helpful when there are many choices:

signum :: Int \rightarrow Int signum x | x > 0 = 1 | x = 0 = 0 | x < 0 = -1

Otherwise we'd have to write

signum x = if x > 0 then 1 else if x = 0 then 0 else -1

λ Expressions

- Since functions are first-class constructs, we can also construct functions in expressions.
- A λ expression denotes an anonymous function.
 - $\lambda x \rightarrow e$: a function with argument *x* and body *e*.
 - $\lambda x \rightarrow \lambda y \rightarrow e$ abbreviates to $\lambda x y \rightarrow e$.
 - In ASCII, we write λ as \setminus
- Yet another way to define *smaller*:

smaller :: *Int* \rightarrow *Int* \rightarrow *Int smaller* = $\lambda x y \rightarrow$ **if** $x \le y$ **then** x **else** y

- Why λs? Sometimes we may want to quickly define a function and use it only once.
- In fact, λ is a more primitive concept.

Local Definitions

There are two ways to define local bindings in Haskell.

• let-expression:

$$f ::: Float \rightarrow Float \rightarrow Float$$
$$f x y = let a = (x+y)/2$$
$$b = (x+y)/3$$
$$in (a+1) \times (b+2)$$

• where-clause:

$$f :: Int \to Int \to Int$$

$$f x y | x \le 10 = x + a$$

$$| x > 10 = x - a$$

where $a = square (y+1)$

 let can be used in expressions (e.g. 1 + (let..in..)), while where qualifies multiple guarded equations.

3 Types

Types

- The universe of values is partitioned into collections, called *types*.
- Some basic types: Int, Float, Bool, Char...
- Type "constructors": functions, lists, trees ... to be introduced later.

- Operations on values of a certain type might not make sense for other types. For example: *square square 3*.
- Strong typing: the type of a well-formed expression can be deducted from the constituents of the expression.
 - It helps you to detect errors.
 - More importantly, programmers may consider the types for the values being defined before considering the definition themselves, leading to clear and well-structured programs.

Polymorphic Types

- Suppose *square* :: *Int* \rightarrow *Int* and *sqrt* :: *Int* \rightarrow *Float*.
 - square \cdot square :: Int \rightarrow Int
 - $sqrt \cdot square :: Int \rightarrow Float$
- The (\cdot) operator has different types in the two expressions:

$$\begin{array}{l} - \ (\cdot) :: (Int \to Int) \to (Int \to Int) \to (Int \to Int) \\ - \ (\cdot) :: (Int \to Float) \to (Int \to Int) \to (Int \to Float) \\ \end{array}$$

• To allow (\cdot) to be used in many situations, we introduce type variables and let its type be: $(b \rightarrow c) \rightarrow$ $(a \rightarrow b) \rightarrow (a \rightarrow c)$.

Summary So Far

- Functions are essential building blocks in a Haskell program. They can be applied, composed, passed as arguments, and returned as results.
- Types sometimes guide you through the design of a program.
- · Equational reasoning: let the symbols do the work!

Recommanded Textbooks

- Introduction to Functional Programming using Haskell [Bir98]. My recommended book. Covers equational reasoning very well.
- Programming in Haskell [Hut07]. A thin but complete textbook.

Online Haskell Tutorials

- *Learn You a Haskell for Great Good!* [Lip11], a nice tutorial with cute drawings!
- Yet Another Haskell Tutorial [DI02].
- A Gentle Introduction to Haskell by Paul Hudak, John Peterson, and Joseph H. Fasel: a bit old, but still worth a read. [HPF00]
- *Real World Haskell* [OSG98]. Freely available online. It assumes some basic knowledge of Haskell, however.

References

- [Bir98] Richard S. Bird. *Introduction to Functional Pro*gramming using Haskell. Prentice Hall, 1998.
- [DI02] Hal Daume III. Yet another haskell tutorial. http://en.wikibooks.org/wiki/ Haskell/YAHT, 2002.
- [HPF00] Paul Hudak, John Peterson, and Joseph Fasel. A gentle introduction to haskell, version 98. http://www.haskell.org/tutorial/, 2000.
- [Hut07] Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2007.
- [Lip11] Miran Lipovača. Learn You a Haskell for Great Good! No Starch Press, 2011. Available online at http://learnyouahaskell.com/.
- [OSG98] Bryan O'Sullivan, Don Stewart, and John Goerzen. Real World Haskell. O'Reilly, 1998. Available online at http://book. realworldhaskell.org/.

A GHCi Commands

```
(statement)
                                      evaluate/run (statement)
                                      repeat last command
:\{n ..lines.. \n:\}\n}
                                      multiline command
:add [*]<module> ...
                                      add module(s) to the current target set
:browse[!] [[*]<mod>]
                                      display the names defined by module <mod> (!: more details; *: all
                                      top-level names)
                                      change directory to <dir>
:cd <dir>
:cmd <expr>
                                      run the commands returned by <expr>::IO String
                                      create tags file for Vi (default: "tags") (!: use regex instead of line
:ctags[!] [<file>]
                                      number)
                                      define command :<cmd> (later defined command has precedence,
:def <cmd> <expr>
                                      ::<cmd> is always a builtin command)
:edit <file>
                                      edit file
                                      edit last module
:edit
:etags [<file>]
                                      create tags file for Emacs (default: "TAGS")
:help, :?
                                      display this list of commands
:info [<name> ...]
                                      display information about the given names
:issafe [<mod>]
                                      display safe haskell information of module <mod>
:kind <type>
                                      show the kind of <type>
:load [*]<module> ...
                                      load module(s) and their dependents
:main [<arguments> ...]
                                      run the main function with the given arguments
:module [+/-] [*]<mod> ...
                                      set the context for expression evaluation
:quit
                                      exit GHCi
                                      reload the current module set
:reload
:run function [<arguments> ...]
                                      run the function with the given arguments
:script <filename>
                                      run the script <filename>
:type <expr>
                                      show the type of <expr>
:undef <cmd>
                                      undefine user-defined command :<cmd>
:!<command>
                                      run the shell command <command>
```

Commands for debugging

:abandon	at a breakpoint, abandon current computation
:back	go back in the history (after :trace)
:break [<mod>] <1> [<col/>]</mod>	set a breakpoint at the specified location
:break <name></name>	set a breakpoint on the specified function
:continue	resume after a breakpoint
:delete <number></number>	delete the specified breakpoint
:delete *	delete all breakpoints
:force <expr></expr>	print <expr>, forcing unevaluated parts</expr>
:forward	go forward in the history (after :back)
:history [<n>]</n>	after :trace, show the execution history
:list	show the source code around current breakpoint
:list identifier	show the source code for <identifier></identifier>
:list [<module>] <line></line></module>	show the source code around line number <line></line>
:print [<name>]</name>	prints a value without forcing its computation
<pre>:sprint [<name>]</name></pre>	simplifed version of :print
:step	single-step after stopping at a breakpoint

:step <expr></expr>	single-step into <expr></expr>
:steplocal	single-step within the current top-level binding
:stepmodule	single-step restricted to the current module
:trace	trace after stopping at a breakpoint
:trace <expr></expr>	evaluate <expr> with tracing on (see :history)</expr>

Commands for changing settings

<pre>:set <option></option></pre>	set options
:seti <option></option>	set options for interactive evaluation only
:set args <arg></arg>	set the arguments returned by System.getArgs
:set prog <progname></progname>	set the value returned by System.getProgName
:set prompt <prompt></prompt>	set the prompt used in GHCi
:set editor <cmd></cmd>	set the command used for :edit
:set stop [<n>] <cmd></cmd></n>	set the command to run when a breakpoint is hit
:unset <option></option>	unset options

Options for :set and :unset

1	allou multilina annonda
+m	anow multime commands
+r	revert top-level expressions after each evaluation
+s	print timing/memory stats after each evaluation
+t	print type after evaluation
- <flags></flags>	most GHC command line flags can also be set here (eg. $-v2$, -fglasgow-exts, etc). For GHCi-specific flags, see User's Guide, Flag reference, Interactive-mode options.

Commands for displaying information

show l	bindings	show the current bindings made at the prompt
show l	breaks	show the active breakpoints
:show o	context	show the breakpoint context
:show :	imports	show the current imports
:show r	modules	show the currently loaded modules
show]	packages	show the currently active package flags
:show]	language	show the currently active language flags
:show <	<setting></setting>	show value of <setting>, which is one of [args, prog, prompt,</setting>
		editor, stop]
:showi	language	show language flags for interactive evaluation