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

Shin-Cheng Mu

Autumn 2025

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://cool.ntu.edu.tw/courses/51303`
  - 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:

$$\begin{array}{ll} \mathit{square} & :: \mathit{Int} \to \mathit{Int} \\ \mathit{square}\ x & = x \times x \\ & \\ \mathit{smaller} & :: \mathit{Int} \to \mathit{Int} \to \mathit{Int} \\ \mathit{smaller}\ x\ y & = \mathbf{if}\ x \leq y\ \mathbf{then}\ x\ \mathbf{else}\ y \end{array}$$

- The GHCi interpreter evaluates expressions in the loaded context:

$$\begin{array}{l} ?\ \mathit{square}\ 3768 \\ 14197824 \\ ?\ \mathit{square}\ (\mathit{smaller}\ 5\ (3+4)) \\ 25 \end{array}$$

1 Values and Evaluation

Evaluation

One possible sequence of evaluating (simplifying, or reducing) $\mathit{square}\ (3+4)$:

$$\begin{array}{cl} & \mathit{square}\ (3+4) \\ = & \{\ \text{definition of } +\ \} \\ & \mathit{square}\ 7 \\ = & \{\ \text{definition of } \mathit{square}\ \} \\ & 7 \times 7 \\ = & \{\ \text{definition of } \times\ \} \\ & 49 \end{array}$$

Another Evaluation Sequence

- Another possible reduction sequence:

$$\begin{array}{cl} & \mathit{square}\ (3+4) \\ = & \{\ \text{definition of } \mathit{square}\ \} \\ & (3+4) \times (3+4) \\ = & \{\ \text{definition of } +\ \} \\ & 7 \times (3+4) \\ = & \{\ \text{definition of } +\ \} \\ & 7 \times 7 \\ = & \{\ \text{definition of } \times\ \} \\ & 49 \end{array}$$

- 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:

$$\begin{aligned} &\mathit{three} &&:: \mathit{Int} \to \mathit{Int} \\ &\mathit{three}\ x &&= 3 \end{aligned}$$

$$\begin{aligned} &\mathit{infinity} &&:: \mathit{Int} \\ &\mathit{infinity} &&= \mathit{infinity} + 1 \end{aligned}$$

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

$$\begin{aligned} & \mathit{three}\ \mathit{infinity} \\ = & \quad \{ \text{ definition of } \mathit{infinity} \ \} \\ & \mathit{three}\ (\mathit{infinity} + 1) \\ = & \ \mathit{three}\ ((\mathit{infinity} + 1) + 1) \ldots \end{aligned}$$

- If we start with simplifying *three*:

$$\begin{aligned} & \mathit{three}\ \mathit{infinity} \\ = & \quad \{ \text{ definition of } \mathit{three} \ \} \\ & 3 \end{aligned}$$

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 $f :: A \to B$ 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*:

$$\begin{aligned} &\mathit{smaller} &&:: \mathit{Int} \to \mathit{Int} \to \mathit{Int} \\ &\mathit{smaller}\ x\ y &&= \mathbf{if}\ x \leq y\ \mathbf{then}\ x\ \mathbf{else}\ y \end{aligned}$$

- 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 $\mathit{Int} \to (\mathit{Int} \to \mathit{Int})$.

Precedence and Association

- In a sense, all Haskell functions takes exactly one argument.
  - Such functions are often called *curried*.
- Type: $a \to b \to c = a \to (b \to c)$, not $(a \to b) \to c$.
- 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.

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

- Try evaluating *quad* 3:

$$\begin{aligned}
&\ \ \mathit{quad}\ 3\\
&= \mathit{twice}\ \mathit{square}\ 3\\
&= \mathit{square}\ (\mathit{square}\ 3)\\
&= \ldots
\end{aligned}$$

- Had we defined:

$$\begin{aligned}
&\mathit{twice} && :: (a \to a, a) \to a\\
&\mathit{twice}\ (f, x) && = f\ (f\ x)
\end{aligned}$$

  we would have to write

$$\begin{aligned}
&\mathit{quad} && :: \mathit{Int} \to \mathit{Int}\\
&\mathit{quad}\ x && = \mathit{twice}\ (\mathit{square}, x)
\end{aligned}$$

- 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 $\mathit{Int} \to \mathit{Int} \to \mathit{Int}$.

- Infix operator can be partially applied too.

$$\begin{aligned}
&(x\ \oplus)\ y && = x \oplus y\\
&(\oplus\ y)\ x && = x \oplus y
\end{aligned}$$

  - $(1\ +) :: \mathit{Int} \to \mathit{Int}$ increments its argument by one.
  - $(1.0\ /) :: \mathit{Float} \to \mathit{Float}$ is the "reciprocal" function.
  - $(/\ 2.0) :: \mathit{Float} \to \mathit{Float}$ 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:

$$\begin{aligned}
&(\cdot) :: (b \to c) \to (a \to b) \to (a \to c)\\
&(f \cdot g)\ x = f\ (g\ x)
\end{aligned}$$

- E.g. another way to write *quad*:

$$\begin{aligned}
&\mathit{quad} && :: \mathit{Int} \to \mathit{Int}\\
&\mathit{quad} && = \mathit{square} \cdot \mathit{square}
\end{aligned}$$

- Some important properties:
  - $\mathit{id} \cdot f = f = f \cdot \mathit{id}$, where $\mathit{id}\ x = x$.
  - $(f \cdot g) \cdot h = f \cdot (g \cdot h)$.

2.3 Definitions

Guarded Equations

- Recall the definition:

$$\begin{aligned}
&\mathit{smaller} && :: \mathit{Int} \to \mathit{Int} \to \mathit{Int}\\
&\mathit{smaller}\ x\ y && = \mathbf{if}\ x \leq y\ \mathbf{then}\ x\ \mathbf{else}\ y
\end{aligned}$$

- We can also write:

$$\begin{aligned}
&\mathit{smaller} && :: \mathit{Int} \to \mathit{Int} \to \mathit{Int}\\
&\mathit{smaller}\ x\ y && \mid x \leq y = x\\
& && \mid x > y = y
\end{aligned}$$

- Equivalently,

$$\begin{aligned}
&\mathit{smaller} :: \mathit{Int} \to \mathit{Int} \to \mathit{Int}\\
&\mathit{smaller}\ x\ y \mid x \leq y && = x\\
&\qquad\qquad\ \ \mid \mathbf{otherwise} && = y
\end{aligned}$$

- Helpful when there are many choices:

$$\begin{aligned}
&\mathit{signum} :: \mathit{Int} \to \mathit{Int}\\
&\mathit{signum}\ x \mid x > 0 && = 1\\
&\qquad\qquad \mid x == 0 && = 0\\
&\qquad\qquad \mid x < 0 && = -1
\end{aligned}$$

  Otherwise we'd have to write

$$\begin{aligned}
\mathit{signum}\ x\ = &\ \mathbf{if}\ x > 0\ \mathbf{then}\ 1\\
&\ \mathbf{else\ if}\ x == 0\ \mathbf{then}\ 0\ \mathbf{else}\ -1
\end{aligned}$$

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 \
- Yet another way to define *smaller*:

$$\begin{aligned} &smaller\ &&:: Int \rightarrow Int \rightarrow Int \\ &smaller\ &&= \lambda x\ y \rightarrow \mathbf{if}\ x \leq y\ \mathbf{then}\ x\ \mathbf{else}\ y \end{aligned}$$

- 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:

$$\begin{aligned} &f &&:: Float \rightarrow Float \rightarrow Float \\ &f\ x\ y &&= \mathbf{let}\ a = (x+y)/2 \\ & &&\qquad\ \ b = (x+y)/3 \\ & &&\quad\ \mathbf{in}\ (a+1) \times (b+2) \end{aligned}$$

- **where**-clause:

$$\begin{aligned} &f &&:: Int \rightarrow Int \rightarrow Int \\ &f\ x\ y &&\mid x \leq 10 = x + a \\ & &&\mid x > 10 = x - a \\ &\quad \mathbf{where}\ a = square\ (y+1) \end{aligned}$$

- **let** can be used in expressions (e.g. $1 + (\mathbf{let}..\mathbf{in}..)$), 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:
  - $(\cdot) :: (Int \rightarrow Int) \rightarrow (Int \rightarrow Int) \rightarrow (Int \rightarrow Int)$
  - $(\cdot) :: (Int \rightarrow Float) \rightarrow (Int \rightarrow Int) \rightarrow (Int \rightarrow Float)$
- 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 Programming 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)
:cd <dir>	change directory to <dir>
:cmd <expr>	run the commands returned by <expr>::IO String
:ctags[!] [<file>]	create tags file for Vi (default: "tags") (!: use regex instead of line number)
:def <cmd> <expr>	define command :<cmd> (later defined command has precedence, ::<cmd> is always a builtin command)
:edit <file>	edit file
:edit	edit last module
: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	reload the current module set
: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>] <l> [<col>]	set a breakpoint at the specified location
:break <name>	set a breakpoint on the specified function
:continue	resume after a breakpoint
:delete <number>	delete the specified breakpoint
:delete *	delete all breakpoints
:force <expr>	print <expr>, forcing unevaluated parts
:forward	go forward in the history (after :back)
:history [<n>]	after :trace, show the execution history
:list	show the source code around current breakpoint
:list identifier	show the source code for <identifier>
:list [<module>] <line>	show the source code around line number <line>
:print [<name> ...]	prints a value without forcing its computation
:sprint [<name> ...]	simplifed version of :print
:step	single-step after stopping at a breakpoint

`:step <expr>`	single-step into `<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>`	evaluate `<expr>` with tracing on (see `:history`)

Commands for changing settings

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

Options for `:set` and `:unset`

`+m`	allow multiline commands
`+r`	revert top-level expressions after each evaluation
`+s`	print timing/memory stats after each evaluation
`+t`	print type after evaluation
`-<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 bindings`	show the current bindings made at the prompt
`:show breaks`	show the active breakpoints
`:show context`	show the breakpoint context
`:show imports`	show the current imports
`:show modules`	show the currently loaded modules
`:show packages`	show the currently active package flags
`:show language`	show the currently active language flags
`:show <setting>`	show value of `<setting>`, which is one of [`args`, `prog`, `prompt`, `editor`, `stop`]
`:showi language`	show language flags for interactive evaluation