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

Shin-Cheng Mu

Autumn 2023

National Taiwan University and Academia Sinica

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:

$$\begin{aligned} &\mathit{square} && :: \mathit{Int} \rightarrow \mathit{Int} \\ &\mathit{square}\ x && = x \times x \end{aligned}$$

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

- The GHCi interpreter evaluates expressions in the loaded context:

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

Values and Evaluation

Evaluation

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

$$\mathit{square}\ (3 + 4)$$

Evaluation

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

$$
\begin{aligned}
& \mathit{square}\ (3 + 4) \\
= \quad & \{\ \text{definition of } +\ \} \\
& \mathit{square}\ 7
\end{aligned}
$$

Evaluation

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

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

Evaluation

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

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

Another Evaluation Sequence

- Another possible reduction sequence:

$$\mathit{square}\ (3 + 4)$$

Another Evaluation Sequence

- Another possible reduction sequence:

$$
\begin{aligned}
& \mathit{square}\ (3+4) \\
= \quad & \{\ \text{definition of } \mathit{square}\ \} \\
& (3+4) \times (3+4)
\end{aligned}
$$

Another Evaluation Sequence

- Another possible reduction sequence:

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

Another Evaluation Sequence

- Another possible reduction sequence:

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

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:

$$
\begin{aligned}
&\mathit{three} &&:: \mathit{Int} \to \mathit{Int}\\
&\mathit{three}\ x &&= 3\\
&\mathit{infinity} &&:: \mathit{Int}\\
&\mathit{infinity} &&= \mathit{infinity} + 1
\end{aligned}
$$

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

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

- If we start with simplifying *three*:

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

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*.

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.

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: $\mathit{smaller}\; 3\; 4$.
- 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 $f\ (x\ y)$.
  - $\mathit{smaller}\ 3\ 4$ means $(\mathit{smaller}\ 3)\ 4$.
  - $\mathit{square}\ \mathit{square}\ 3$ means $(\mathit{square}\ \mathit{square})\ 3$, which results in a type error.
- Function application binds tighter than infix operators. E.g. $\mathit{square}\ 3 + 4$ means $(\mathit{square}\ 3) + 4$.

Why Currying?

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

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

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

- Try evaluating $\mathit{quad}\ 3$:

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

Sectioning

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

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

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

$$quad :: Int \to Int$$
$$quad = square \cdot square$$

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

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:

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

- Helpful when there are many choices:

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

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 $\backslash$
- 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:

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

- **where**-clause:

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

- **let** can be used in expressions (e.g. $1 + (\mathbf{let..in..})$), while **where** qualifies multiple guarded equations.

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 $square :: Int \to Int$ and $sqrt :: Int \to Float$.
  - $square \cdot square :: Int \to Int$
  - $sqrt \cdot square :: Int \to Float$
- The $(\cdot)$ operator has different types in the two expressions:
  - $(\cdot) :: (Int \to Int) \to (Int \to Int) \to (Int \to Int)$
  - $(\cdot) :: (Int \to Float) \to (Int \to Int) \to (Int \to Float)$
- To allow $(\cdot)$ to be used in many situations, we introduce type variables and let its type be: $(b \to c) \to (a \to b) \to (a \to c)$.

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.* My recommended book. Covers equational reasoning very well.
- *Programming in Haskell.* A thin but complete textbook.

Online Haskell Tutorials

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