Introduction to Functional Programming
Part 3: Types and Polymorphism

Jeremy Gibbons
University of Oxford

2.9 Algebraic datatypes

- eg people with names and ages

$$\begin{aligned}&\textbf{type}\ \mathit{Name} = \mathit{String}\\ &\textbf{type}\ \mathit{Age} = \mathit{Integer}\\ &\textbf{data}\ \mathit{Person} = P\ \mathit{Name}\ \mathit{Age}\end{aligned}$$

- then $P :: \mathit{Name} \to \mathit{Age} \to \mathit{Person}$
- such *constructor functions* do not simplify, they are in normal form
- so they can be used in pattern-matching:

$$\begin{aligned}&\mathit{showPerson} :: \mathit{Person} \to \mathit{String}\\ &\mathit{showPerson}\ (P\ n\ a) = \texttt{"Name: "} \mathbin{+\!\!+} n \mathbin{+\!\!+} \texttt{", Age: "} \mathbin{+\!\!+} \mathit{show}\ a\end{aligned}$$

2.9 Sum types

- datatypes can have multiple *variants*

$$\mathbf{data}\ \mathit{Maybe}\ a = \mathit{Just}\ a \mid \mathit{Nothing}$$

- eg $\mathit{Just}\ 13 :: \mathit{Maybe}\ \mathit{Int}$; so $\mathit{Just} :: a \to \mathit{Maybe}\ a$ and $\mathit{Nothing} :: \mathit{Maybe}\ a$
- useful for modelling exceptions

$$\begin{array}{l} \mathit{safeHead} :: [\,a\,] \to \mathit{Maybe}\ a \\ \mathit{safeHead}\ [\,]\ \ \ \ \ \ \ = \mathit{Nothing} \\ \mathit{safeHead}\ (x : xs) = \mathit{Just}\ x \end{array}$$

- the library definition of unfold:

$$\begin{array}{l} \mathit{unfoldr} :: (b \to \mathit{Maybe}\ (a, b)) \to (b \to [\,a\,]) \\ \mathit{unfoldr}\ f\ z = \mathbf{case}\ f\ z\ \mathbf{of} \\ \qquad\qquad \mathit{Nothing} \quad\ \to [\,] \\ \qquad\qquad \mathit{Just}\ (x, z') \to x : \mathit{unfoldr}\ f\ z' \end{array}$$

2.10 Arithmetic expressions

- algebraic datatypes may be recursive too
- eg datatype of arithmetic expressions

$$\mathbf{data}\ \mathit{Expr} = \mathit{Lit}\ \mathit{Integer} \mid \mathit{Add}\ \mathit{Expr}\ \mathit{Expr} \mid \mathit{Mul}\ \mathit{Expr}\ \mathit{Expr}$$

- an arithmetic expressions is either a literal,
  or two expressions added together, or two multiplied

2.10 Constructing expressions

- constructing expressions

$$\begin{aligned} &\mathit{expr1}, \mathit{expr2} :: \mathit{Expr} \\ &\mathit{expr1} = \mathit{Add}\ (\mathit{Mul}\ (\mathit{Lit}\ 4)\ (\mathit{Lit}\ 7))\ (\mathit{Lit}\ 11) \\ &\mathit{expr2} = \mathit{Mul}\ (\mathit{Add}\ (\mathit{Lit}\ 4)\ (\mathit{Lit}\ 7))\ (\mathit{Lit}\ 11) \end{aligned}$$

- note the difference between *syntax*

$$\begin{aligned} &\mathrm{Main}\rangle\ \mathit{Add}\ (\mathit{Lit}\ 4)\ (\mathit{Mul}\ (\mathit{Lit}\ 7)\ (\mathit{Lit}\ 11)) \\ &\mathit{Add}\ (\mathit{Lit}\ 4)\ (\mathit{Mul}\ (\mathit{Lit}\ 7)\ (\mathit{Lit}\ 11)) \end{aligned}$$

- and *semantics*

$$\begin{aligned} &\mathrm{Main}\rangle\ 4 + 7 \times 11 \\ &81 \end{aligned}$$

2.10 *Expr* definition pattern

- recursive definitions by pattern-matching

$$
\begin{aligned}
&\mathit{evaluate} :: \mathit{Expr} \rightarrow \mathit{Integer} \\
&\mathit{evaluate}\ (\mathit{Lit}\ i) &&= i \\
&\mathit{evaluate}\ (\mathit{Add}\ e1\ e2) &&= \mathit{evaluate}\ e1 + \mathit{evaluate}\ e2 \\
&\mathit{evaluate}\ (\mathit{Mul}\ e1\ e2) &&= \mathit{evaluate}\ e1 \times \mathit{evaluate}\ e2
\end{aligned}
$$

- the evaluator essentially replaces syntax (*Add* and *Mul*) by semantics ($+$ and $\times$)

2.10 *Expr* definition pattern

- remember: every datatype comes with a definition pattern
- *task:* define a function $f :: Expr \rightarrow S$
- *step 1:* solve the problem for literals

$$f\ (Lit\ n) = \ \ldots n \ldots$$

- *step 2:* solve the problem for addition;
  assume that you already have the solution for x and y at hand;
  extend the intermediate solution to a solution for $Add\ x\ y$

$$\begin{aligned} f\ (Lit\ n) &= \ \ldots n \ldots \\ f\ (Add\ x\ y) &= \ \ldots x \ldots y \ldots f\ x \ldots f\ y \ldots \end{aligned}$$

  you have to program only a *step*
- *step 3:* do the same for $Mul\ x\ y$

$$\begin{aligned} f\ (Lit\ n) &= \ \ldots n \ldots \\ f\ (Add\ x\ y) &= \ \ldots x \ldots y \ldots f\ x \ldots f\ y \ldots \\ f\ (Mul\ x\ y) &= \ \ldots x \ldots y \ldots f\ x \ldots f\ y \ldots \end{aligned}$$

2.10 Naturals

- *Peano* definition of natural numbers (non-negative integers)

$$\textbf{data}\ \mathit{Nat} = \mathit{Zero} \mid \mathit{Succ}\ \mathit{Nat}$$

- every natural is either *Zero* or the *Succ*essor of a natural
- eg $\mathit{Succ}\ (\mathit{Succ}\ (\mathit{Succ}\ \mathit{Zero}))$ corresponds to 3
- extraction

$$\begin{aligned}
&\mathit{nat2int} :: \mathit{Nat} \to \mathit{Integer} \\
&\mathit{nat2int}\ \mathit{Zero} \quad\quad = 0 \\
&\mathit{nat2int}\ (\mathit{Succ}\ n) = 1 + \mathit{nat2int}\ n
\end{aligned}$$

- addition

$$\begin{aligned}
&\mathit{plus} :: \mathit{Nat} \to \mathit{Nat} \to \mathit{Nat} \\
&\mathit{plus}\ \mathit{Zero} \quad\quad n = n \\
&\mathit{plus}\ (\mathit{Succ}\ m)\ n = \mathit{Succ}\ (\mathit{plus}\ m\ n)
\end{aligned}$$

(does this look familiar?)

2.10 Peano definition pattern

- remember: every datatype comes with a definition pattern
- *task:* define a function $f :: Nat \to S$
- *step 1:* solve the problem for $Zero$

$$f\ Zero = \ \ldots$$

- *step 2:* solve the problem for $Succ\ n$;
  assume that you already have the solution for n at hand;
  extend the intermediate solution to a solution for $Succ\ n$

$$\begin{aligned} f\ Zero &= \ \ldots \\ f\ (Succ\ n) &= \ \ldots n \ldots f\ n \ldots \end{aligned}$$

  you have to program only a *step*
- put on your problem-solving glasses
- (exercise: multiplication, exponentiation)

2.10 Lists

- built-in type of lists is not special (has only special syntax)

$$\textbf{data}\ \mathit{List}\ a = \mathit{Nil} \mid \mathit{Cons}\ a\ (\mathit{List}\ a)$$

- eg $[1,2,3]$ or $1:2:3:[\,]$ corresponds to $\mathit{Cons}\ 1\ (\mathit{Cons}\ 2\ (\mathit{Cons}\ 3\ \mathit{Nil}))$
- recursive definitions by pattern-matching

$$\begin{aligned}
&\mathit{mapList} :: (a \to b) \to (\mathit{List}\ a \to \mathit{List}\ b) \\
&\mathit{mapList}\ f\ \mathit{Nil} && = \mathit{Nil} \\
&\mathit{mapList}\ f\ (\mathit{Cons}\ x\ \mathit{xs}) && = \mathit{Cons}\ (f\ x)\ (\mathit{mapList}\ f\ \mathit{xs})
\end{aligned}$$

2.10 List definition pattern

- remember: every datatype comes with a definition pattern
- *task:* define a function $f :: List\ P \to S$
- *step 1:* solve the problem for the empty list

$$f\ Nil = \ \ldots$$

- *step 2:* solve the problem for the non-empty list;
  assume that you already have the solution for xs at hand;
  extend the intermediate solution to a solution for $Cons\ x\ xs$

$$\begin{aligned} f\ Nil &= \ \ldots \\ f\ (Cons\ x\ xs) &= \ \ldots x \ldots xs \ldots f\ xs \ldots \end{aligned}$$

  you have to program only a *step*
- put on your problem-solving glasses

2.10 Binary search trees

- definition

$$\textbf{data}\ BST\ k\ v = Leaf \mid Branch\ (BST\ k\ v)\ k\ v\ (BST\ k\ v)$$

- eg insertion

$$\begin{array}{lll}
\multicolumn{2}{l}{insert :: Ord\ k \Rightarrow k \rightarrow v \rightarrow BST\ k\ v \rightarrow BST\ k\ v} & \\
\multicolumn{2}{l}{insert\ k\ v\ Leaf = Branch\ Leaf\ k\ v\ Leaf} & \text{-- insert at leaf} \\
\multicolumn{2}{l}{insert\ k\ v\ (Branch\ l\ k'\ v'\ r)} & \\
\quad \mid k < k' & = Branch\ (insert\ k\ v\ l)\ k'\ v'\ r & \text{-- insert to the left} \\
\quad \mid k > k' & = Branch\ l\ k'\ v'\ (insert\ k\ v\ r) & \text{-- insert to the right} \\
\quad \mid otherwise & = Branch\ l\ k\ v\ r & \text{-- overwrite the root}
\end{array}$$

2.10 Binary search tree definition pattern

- remember: every datatype comes with a definition pattern
- *task:* define a function $f :: BST\ K\ V \rightarrow S$
- *step 1:* solve the problem for the empty tree

$$f\ Leaf = \ \ldots$$

- *step 2:* solve the problem for non-empty trees;
  assume that you already have solutions for l and r at hand;
  extend the intermediate solution to a solution for $Branch\ l\ k\ v\ r$

$$\begin{aligned} f\ Leaf &= \ \ldots \\ f\ (Branch\ l\ k\ v\ r) &= \ \ldots k \ldots v \ldots l \ldots f\ l \ldots r \ldots f\ r \ldots \end{aligned}$$

  you have to program only a *step*
- put on your problem-solving glasses

Part 3

Types and polymorphism

3.1 Strong typing

- Haskell is *strongly typed*: every expression has a unique type
- each type supports certain operations, which are meaningless on other types
- type checking guarantees that type errors cannot occur
- Haskell is *statically typed*: type checking occurs before runtime (after syntax checking)
- experience shows well-typed programs are likely to be correct
- Haskell can *infer types*: determine the most general type of each expression
- wise to specify (some) types anyway, for documentation and redundancy

3.2 Simple types

- booleans
- characters
- strings
- numbers

3.2 Booleans

- type *Bool* (note: type names capitalized)
- two constants, *True* and *False* (note: constructor names capitalized)
- eg definition by pattern-matching

$$
\begin{aligned}
&\mathit{not} :: \mathit{Bool} \to \mathit{Bool} \\
&\mathit{not}\ \mathit{False} = \mathit{True} \\
&\mathit{not}\ \mathit{True} = \mathit{False}
\end{aligned}
$$

- and $\&\&$, or $||$, both strict in first argument

$$
\begin{aligned}
&(\&\&) :: \mathit{Bool} \to \mathit{Bool} \to \mathit{Bool} \\
&\mathit{False}\ \&\&\ \_ = \mathit{False} \\
&\mathit{True}\ \&\&\ x = x
\end{aligned}
$$

- comparisons $==$, $\neq$, orderings $<$, $\leqslant$ etc

3.2 Characters

- type *Char*
- constants in single quotes: `'a'`, `'?'`
- special characters escaped: `'\''`, `'\n'`, `'\\'`
- ASCII coding: $ord :: Char \to Int$, $chr :: Int \to Char$ (defined in module *Data.Char*)
- comparison and ordering, as before

3.2 Strings

- type *String*, in fact $[\,Char\,]$
- constants in double quotes: `"Hello"`
- comparison and (lexicographic) ordering
- concatenation $+\!\!+$
- display function $show :: Integer \to String$
- (actually more general than this; see later)
- monadic $putStr :: String \to IO\ ()$ to print formatted text

```
Main⟩ putStr "abc\ndef\n"
abc
def
```

3.2 Numbers

- fixed-size (32-bit or 64-bit) integers *Int*
- arbitrary-size integers *Integer*
- single- and double-precision floats *Float*, *Double*
- others too: rationals, complex numbers, ...
- comparisons and ordering
- $+$, $-$, $\times$, $\uparrow$, $**$
- *abs*, *negate*
- $/$, *div*, *mod*, *quot*, *rem*
- etc
- operations are overloaded (more later)

3.3 Enumerations

- enumerations as degenerate algebraic datatypes

  data *Day* = *Mon* | *Tue* | *Wed* | *Thu* | *Fri* | *Sat* | *Sun*

- eg *Bool* is not built in (although **if** ... **then** ... **else** syntax is):

  data *Bool* = *False* | *True*

3.4 Tuples

- pairing types: eg $(\mathit{Char}, \mathit{Integer})$
- values in the same syntax: `('a', 440)`
- selectors fst, snd
- definition by pattern matching:

$$\mathit{fst}\ (x, y) = x$$

- nested tuples: $(\mathit{Integer}, (\mathit{Char}, \mathit{Bool}))$
- triples, etc: $(\mathit{Integer}, \mathit{Char}, \mathit{Bool})$
- nullary tuple $()$
- comparisons, (lexicographic) ordering

3.5 Polymorphism

- what is the type of *fst*?
- applicable at different types: $fst\ (1,2)$, $fst\ (\texttt{'a'}, True)$, ...
- what about strong typing?
- *fst* is *polymorphic* — it works for *any* type of pairs:

$$fst :: (a, b) \rightarrow a$$

- a, b here are *type variables* (uncapitalized)
- *implicit* universal quantification—really means something like

$$fst :: \forall\, a\ b.(a, b) \rightarrow a$$

- values can be polymorphic too: $\bot :: a$, $[\,] :: [\,a\,]$
- regain *principal types* for all expressions

3.5 A little game

- here is a little game: I give you a type, you give me a function of that type
  - $Integer \to Integer$
  - $a \to a$
  - $(Integer, Integer) \to Integer$
  - $(a, a) \to a$
  - $(a, b) \to a$
  - $[a] \to [a]$
- polymorphic functions: flexible to use, hard to define

3.5 Theorems for free

- you can tell a lot about a function from its (polymorphic) type
- consider an unknown function $h :: [\,a\,] \rightarrow [\,a\,]$
- h cannot manipulate list elements, or even invent them
- all it can do is rearrange or duplicate them
- as a consequence, for any f,

$$map\ f \circ h = h \circ map\ f$$

- the *free theorem* of (the type of) h
- similarly for any polymorphic type
- a kind of *representation independence*—a deep rabbithole!

3.5 Natural numbers as functions

Generalize $twice :: \forall a.(a \to a) \to a \to a$:

$$\mathbf{type}\ Natural = \forall a.(a \to a) \to (a \to a)$$

$$zero :: Natural$$
$$zero\ f = id$$

$$succ :: Natural \to Natural$$
$$succ\ n\ f = f \circ n\ f$$

The $\forall$ makes explicit that these functions are polymorphic.
These are called *Church numerals*. We could define:

$$one, two :: Natural$$
$$one = succ\ zero$$
$$two = succ\ one \quad \text{-- aka } twice$$

Caveat: here, explicit universal quantification is necessary.
Beyond plain Haskell; type inference no longer available. . .

Conversion from *Int* using *iter*:

$$
\begin{aligned}
&\mathit{iter} :: \mathit{Int} \to \mathit{Natural} \\
&\mathit{iter}\ 0\ f = \mathit{id} \\
&\mathit{iter}\ n\ f = f \circ \mathit{iter}\ (n-1)\ f
\end{aligned}
$$

How about back again?

$$\mathit{extract} :: \mathit{Natural} \to \mathit{Int}$$

such that *iter* and *extract* are inverses?

3.6 Type classes

- what about numeric operations?
- $(+) :: Integer \to Integer \to Integer$
- $(+) :: Float \to Float \to Float$
- cannot have $(+) :: \forall a . a \to a \to a$ (too general)
- the solution is *type classes* (sets of types)
- eg the type class *Num* is a set of numeric types; includes *Integer*, *Float*, etc
- now $(+) :: (Num\ a) \Rightarrow (a \to a \to a)$
- *ad hoc polymorphism* (different code for different types), as opposed to *parametric polymorphism* (same code for all types)

3.6 Some standard type classes

- *Eq*: $==$, $\neq$
- *Ord*: $<$ etc, *min* etc
- *Enum*: *succ*, ..
- *Bounded*: *minBound*, *maxBound*
- *Show*: $show :: a \to String$
- *Read*: $read :: String \to a$
- *Num*: $+$, $\times$ etc
- *Real* (ordered numeric types)
- *Integral*: *div* etc
- *Fractional*: $/$ etc
- *Floating*: *exp* etc
- more later

3.6 Derived type classes

- new **data**types not automatically instances of useful type classes
- possible to install as instances:

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Eq}\ \mathit{Day}\ \textbf{where} \\
\quad \begin{array}{lllll}
\mathit{Mon} & == & \mathit{Mon} & = & \mathit{True} \\
\mathit{Tue} & == & \mathit{Tue} & = & \mathit{True} \\
\mathit{Wed} & == & \mathit{Wed} & = & \mathit{True} \\
\mathit{Thu} & == & \mathit{Thu} & = & \mathit{True} \\
\mathit{Fri} & == & \mathit{Fri} & = & \mathit{True} \\
\mathit{Sat} & == & \mathit{Sat} & = & \mathit{True} \\
\mathit{Sun} & == & \mathit{Sun} & = & \mathit{True} \\
\_ & == & \_ & = & \mathit{False}
\end{array}
\end{array}
$$

- (default definition of $\neq$ obtained for free from $==$, more later)
- tedious for simple cases, which can be derived automatically:

$$
\begin{array}{l}
\textbf{data}\ \mathit{Day} = \mathit{Mon} \mid \mathit{Tue} \mid \mathit{Wed} \mid \mathit{Thu} \mid \mathit{Fri} \mid \mathit{Sat} \mid \mathit{Sun} \\
\quad \textbf{deriving}\ (\mathit{Eq}, \mathit{Ord}, \mathit{Enum}, \mathit{Bounded}, \mathit{Show}, \mathit{Read})
\end{array}
$$

3.7 Type-driven development

- types are a vital part of any program
- types are not an afterthought
- first specify the type of a function
- its definition is then driven by the type

$$f :: T \to U$$

- f consumes a T value: suggests case analysis
- f produces a U value: suggests use of constructors

- type safety and flexibility are in tension
- polymorphism partially releases the tension