Haskell, Part 3, Version 2

CS 440: Programming Languages and Translators, Fall 2019

8/28: v2 up, 8/31: p.11-12, 9/1: p.12

Topics

• Types and Type Classes (Chapter 3)
  • Primitive types, constructed types, type variables, parametric polymorphism
  • Type classes (aren’t classes). Built-in classes; instances of classes.

Prompts

• For the examples below, assume we’ve already declared
  > :set prompt "> "       -- normal prompt
  > :set prompt-cont "|  " -- prompt for lines between :{ ... }:

A. Types in Haskell

Types

• Historically a type was a name used to identify a set of values implemented by hardware. (Integer, float, etc.). Nowadays, primitive types still do that, but we also have complex or constructed types that are more like logical descriptions of a collection of values and less about the internal representation of values. E.g., a list of integers might be implemented in different ways, but regardless, it will support functions like length, head, and so on.

• So a type is a name for a collection of values, either primitive (such as Int, Float, Double, Char, Bool, ... in Haskell) or built using type constructors, which are operators that built up complex types from simpler types. In Haskell we have [...], (... , ...), and -> (“arrow”) to build lists, n-tuples, and functions respectively. Soon, we’ll also see algebraic datatypes, such as user-defined trees.

Type Expressions, Constants, and Variables:

• A type expression is a syntactic representation of the name of a type.

• A name like Int is a type constant and names a primitive type (one that isn’t composed from other types).

• More complex type expressions are built using constructors from simpler types. Examples include [Int] and (Char, Int) and Int -> Int.

• Type Variables: Haskell (and various other languages) also support type variables, which are identifiers that can stand for various different types.

• E.g., the (built-in) identity function \( \text{\texttt{id \ x = x}} \) is said to have type \( \text{\texttt{a -> a}} \), which simply means that \( \text{\texttt{id}} \) can be used on any type of value (the first \( \text{\texttt{a}} \)) and produces a result of the same type (the second \( \text{\texttt{a}} \)).
• We can use any type for \(a\) (but we have to be consistent and use the same type for both \(a\)'s).
• E.g., \(\text{id}\) can be used as an \(\text{Int} \rightarrow \text{Int}\) function or \(\text{Char} \rightarrow \text{Char}\) function and so on.
• Note the type for \(a\) can even be a functional type, so \(\text{id \ sqrt}\) is legal (and its result behaves just like \(\text{sqrt}\), as in \(\text{id \ sqrt \ 16.0} = 4.0\)).

**Parametric Polymorphism; PP vs Templates**

• Type variables are similar to function parameters like the \(x\) in \(\text{id} \ x = \ldots\), and the type of polymorphism (multiple typing) they support is called **parametric polymorphism**.
• Note this is a different kind of polymorphism from the one used in object-oriented inheritance.
• **Parametric polymorphism vs templates**: In Haskell, a parametrically polymorphic function like \(\text{id} \ x = x\) is compiled into one piece of code that can be used on any type of input.
  • This is different from templates in, say C++, where the compiler generates different pieces of code for different kinds of uses of a template.
  • E.g., a templated definition of \(\text{id}\) would generate different code to use for \(\text{id}\) on an integer vs \(\text{id}\) on a float, and so on.
  • Conceptually, a template is a syntactic way to describe a collection of things that differ by the template type variables. (They're basically a more-civilized version of **syntactic macro**.)

**Type Inference and Explicit Type Declarations in Haskell**

• In Haskell, you can declare the type of a function or expression; the typechecker will make sure that your declared type is consistent with the code. (If it isn’t you’ll get an error message of course.) It’s quite common to include an explicit type declaration.

  ```haskell
  > :{
  |   makePalindrome :: \[Char\] \rightarrow \[Char\]
  |   makePalindrome s = s ++ "_" ++ s
  | :}
  > makePalindrome "hello"
  "hello_olleh"
  ```

• We’ve been omitting these explicit type declarations because Haskell is very good at **type inferencing** — deducing the type of an expression or function.

  ```haskell
  > makePalindrome2 s = s ++ "_" ++ s
  > :t makePalindrome2
  makePalindrome2 :: \[Char\] \rightarrow \[Char\]
  ```

• The \(\text{makePalindrome}\) function has a **monomorphic type** (just the one type), but for polymorphic code, Haskell will figure out the **most general type** that can be used.

• Below, Haskell infers the type \(b \rightarrow (b, b)\) for \(\text{mkpair}\) infers because it finds nothing to restrict the type of the first \(x\) (hence the \(b\) in \(b \rightarrow \ldots\)) and then uses that type to generate the most general type
for the result (the \((b, b)\) in \(\ldots b \rightarrow (b, b)\)). But for the application \texttt{mkpair 'x'}, Haskell infers the type \((\text{Char, Char})\) because \texttt{'x' :: Char}. Similarly, for \texttt{mkpair "hi"}, Haskell infers \((\text{[Char]}, \text{[Char]})\) as the result type.

\begin{verbatim}
> mkpair x = (x, x)
> :t mkpair
mkpair :: b -> (b, b)
> mkpair 'x'
('x', 'x')
> :t mkpair 'x'
mkpair 'x' :: (Char, Char)
> mkpair "hi"
mkpair "hi" :: ([Char], [Char])
\end{verbatim}

- The inferred type doesn't have to be monomorphic. Below, \texttt{mkpair} is used on a list value; we don't have a restriction on what kind of list, so the list and the result have a polymorphic type.

\begin{verbatim}
> mkdouble_pair x = mkpair [x]
> :t mkdouble_pair
mkdouble_pair :: a -> ([a], [a])
\end{verbatim}

- You can use explicit type declarations to restrict what would otherwise be a more general type for an expression or function. Here are three versions of \texttt{mkpair} that require a list argument; the different function declarations are equivalent, as you can see from their types. Note \texttt{mkpair_list} creates a different piece of code from \texttt{mkpair} (it takes \(x\) and calls \texttt{mkpair} on it). In contrast, \texttt{mkpair_list2} and \texttt{mkpair_list3} name exactly the same piece of code as \texttt{mkpair_list}.

\begin{verbatim}
> :{
|  mkpair_list :: [a] -> ([a], [a])
|  mkpair_list x = mkpair x  -- declaration with parameter
|  mkpair_list2 :: [a] -> ([a], [a])
|  mkpair_list2 = mkpair  -- declaration without parameter [8/28]
|  mkpair_list3 = mkpair :: [a] -> ([a], [a])  -- on 1 line
|  :}
> :t mkpair_list
mkpair_list :: [a] -> ([a], [a])
> :t mkpair_list2
mkpair_list2 :: [a] -> ([a], [a])
> :t mkpair_list3
mkpair_list3 :: [a] -> ([a], [a])
\end{verbatim}

- (If you don't like typing \texttt{:t} all the time, you can cheat :-)

\begin{verbatim}
> x = (mkpair_list, mkpair_list2, mkpair_list3)
> :t x
\end{verbatim}
B. Operator Overloading and Type Classes

Type Classes Generalize Operator Overloading

- In Haskell, as in other languages, you can use + on various different types of data — integers, floats, and so on. In (e.g.) C, this is called operator overloading. There, we say that + can take two integers and return an integer or two floats and return a float and so on.

- Overloading is a kind of polymorphism (“+” has multiple types because it has multiple meanings). There are different pieces of code for each meaning, so C has to figure out which meaning you want by looking at the types of the arguments / result; the code for adding integers is different from the code for adding floats.

- Haskell has a feature called type classes that formalizes and generalizes operator overloading by providing constraints on types. A type class is a collection of types that are guaranteed to support some particular operations. A type becomes an instance of a class when it’s shown to support all of the required operations.

- E.g., the type class Num is for types that support +, -, *, and some related functions. The built-in types Int, Float, and Double are all instances of Num. The code denoted by + is different for the different types, which is why type classes support operator overloading.

- User-defined types can also be instances of Num. You’re required to write an instance declaration, which simply shows that you’ve defined all the operators / functions required by the class.

- A function supported by a type class doesn’t have to have the same implementation across different instances. E.g., Int and Double are both instances of Num, so they both have + operators, but the two types can (and do) have different implementations of +. This is similar to operator overloading in (e.g.) C, where there are different versions of + depending on which types of values you pass to it.

Primitive Types and Some Standard Type Classes

- Haskell’s primitive types include Int, Float, Double, Char, Bool, and Integer. (Int is for 64-bit integers; Integer is for unbounded (“infinite precision”) integers.)

- All five of these types support equality testing and comparisons <, >, etc. Class Eq is for == and /=; class Ord extends this to include <, >, etc. (“Ord” is short for “ordinal”, which means “can be ordered”.)
  - The basic function for Ord is compare :: a -> a -> Ordering, where Ordering is a type with three values, LT, EQ, and GT. Then \( x < y \) means \( \text{compare} \ x \ y = \text{LT} \), etc.

- Integer, Int, Float, and Double differ from Char and Bool by being numeric types; as instances of type class Num, they support addition, subtraction, multiplication, absolute value, etc.
Float and Double are also instances of Fractional, so they support floating-point division (/). They are also instances of Floating, so they support \texttt{sqrt}, ** (exponentiation), log, and the trigonometric functions.

<table>
<thead>
<tr>
<th>Class</th>
<th>Subclass(es)</th>
<th>Supports</th>
<th>Instances Include Built-In Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Num</td>
<td></td>
<td>+ - * abs ...</td>
<td>Integer Int Float Double</td>
</tr>
<tr>
<td>Fractional</td>
<td>Num</td>
<td>/</td>
<td>Float Double</td>
</tr>
<tr>
<td>Floating</td>
<td>Fractional</td>
<td>sqrt ** log sin ...</td>
<td>Float Double</td>
</tr>
<tr>
<td>Integral</td>
<td>...</td>
<td>quot rem div mod toInteger</td>
<td>Integer Int</td>
</tr>
<tr>
<td>Eq</td>
<td>Eq</td>
<td>== /=</td>
<td>Integer Int Float Double</td>
</tr>
<tr>
<td>Ord</td>
<td>Eq</td>
<td>&lt; &gt; compare</td>
<td>Integer Int Float Double Char Bool</td>
</tr>
</tbody>
</table>

- **Polymorphic constants**: In Haskell, the constants 0, 1, 2, etc. have polymorphic type: They are Num values, not plain integers. If you want one to have a monomorphic type, you have to control their usage (or use explicit type declarations).

```
> :t 17
17 :: Num p => p
```

- **No automatic type conversions**: Haskell does not have the kinds of automatic type conversion you might expect, such as integer to floating point or integer to longer integer. E.g., to convert an Int to Integer, there’s a class-supported function \texttt{toIntInteger} :: Integral a => a -> Integer

```
> int17 = 17 :: Int
> integer17 = 17 :: Integer
> sum = toInteger int17 + integer17 -- works ok
> :t sum
sum :: Integer
> int17 + integer17 -- fails!
```

- **The Show class**: This class supports a function \texttt{show} :: a -> [Char] that takes a value and returns a printable representation of it. The types Integer Int Float Double Char Bool are all instances of Show. If a types a, a1, a2 ... are showable, then so are \texttt{[a]}, \texttt{(a1, a2)}, \texttt{(a1, a2, a3)}, ... (up to 15-tuples). To print a value, the interpreter prints the string \texttt{show value}; since no function types are instances of Show, the call \texttt{show sqrt} (e.g.) fails, which is why you can’t print functions.
C. Typechecking: Static, Dynamic, Strong, and Weak

Static vs Dynamic Typechecking
- Haskell uses static types: Type information is calculated using syntactic analysis, without running the program. (I.e., at “compile time”.) With dynamic types, type information needs to be determined as the program runs.
- Both static and dynamic typechecking use type information to determine type safety of a program. E.g., to make sure we are always adding together two things that can be added. If you can’t guarantee the type safety of an operation, you have to add a runtime check for it.
- Typically, static typechecking makes compilation slower but execution faster than dynamic typechecking, which can avoid compile-time calculations but might have to include safety tests at runtime.
- Note a language can have do some typechecking statically and some dynamically; it doesn’t have to do just one or the other. Turns out Haskell needs no dynamic type checks.

Strong vs Weak Static Typechecking
- The strength or weakness of a typechecker refers to how much type safety the typechecker guarantees. E.g., in C, typechecking is pretty weak because it allows things that aren’t necessarily pointer values to be cast as pointer values.
- In Haskell, typechecking is strong (or more-specialy, “type-safe”): Passing typechecking guarantees that there won’t be any runtime errors caused by incorrect types, so we don’t need to run type-safety tests. Since Haskell typechecking is also static, it’s the compiler that can avoid generating code for runtime type checks. (With dynamic type-safe code, it’s the programmer who doesn’t have to add the code.)
- With strong static typechecking, when you say “x is of type integer”, you’re guaranteeing that at runtime, no matter what the history of program execution has been, the value of x meets certain criteria, and you can determine this without actually doing any program execution.
- To ensure this, language designers have to restrict the ways that values can be created and manipulated. People who like static typechecking are willing to put up with restrictions on how they write code and with extra work getting code to compile because they get better type safety as a result.

---

1 When learning Haskell, people often find the typechecking to be frustrating because it’s sooo picky. On the other hand, once you get your program to pass typechecking, it seems to have a very good chance of working correctly. With Haskell, all those compile-time errors it detects are bugs that you have to get rid of before you can run the program. As a result, there are fewer bugs left for finding during runtime.
**Strong vs Weak Dynamic Typechecking**

- Languages with dynamic typechecking generally *have fewer restrictions* in how you write your code. People who like dynamic typechecking like this aspect of a language and are willing to *put up with execution requiring safety tests*.
- With strong dynamic typechecking, you perform enough runtime tests before every operation to guarantee that the operation is type-safe. (So, you want to add \(x\) and \(y\)? Let’s make sure they both support addition first.)
- Note in the best case, the safety tests aren’t needed because your program always creates type-safe data. In that case, you can turn off the safety tests and get the speed of statically typechecked code along with faster compilation. This is weak dynamic typechecking (well, non-existent dynamic typechecking). Of course, in the non-best case, turning off the safety tests causes unexpected runtime errors.
- Since full type safety only happens if all possible code execution sequences are type-safe, just having a program run correctly for ten years doesn’t guarantee that it will run correctly tonight: Some never-before-seen input might cause a never-before-used sequence of execution that causes an error. Life being the way it is, this sort of thing really does happen.

**D. Chapter 4: Syntax in Functions**

- What syntax do we use for defining functions? More than *id paramvar = expr*
- There’s definition by case. Below \(f\) is defined on all values (the \(f\ x\) case takes care of things that aren’t 0, 1, 2).

```plaintext
> :{
  | f 0 = 0
  | f 1 = 1
  | f 2 = 2
  | f x = 3
  | :}
> f 17
3
> f 0
0
```

- The order of classes matters:

```plaintext
> :{
  | g x = 2
  | g 1 = 3
  | :}
```

<interactive>:110:1: warning: [-Woverlapping-patterns]

Pattern match is redundant
In an equation for ‘g’: g 1 = ...

```
> g 1
```
> 2
>   > -- Simple factorial
>   > :{  
>   | fact 0 = 1  
>   | fact n = n * fact(n-1)  
>   | :}  
>   > -- f works ok on values >= 0
>   > fact -1
>   > (Oops!  I forgot that - is a function; I asked for (fact -) 1
> <interactive>:118:1: error:
>   • Non type-variable argument in the constraint: Num (p -> p)
>   (Use FlexibleContexts to permit this)
>   • When checking the inferred type
>     it :: forall p. (Eq p, Num p, Num (p -> p)) => p -> p
>   . On unix, ^C breaks an infinite loop
>   > fact (-1)
>   ^C^C^C^CInterrupted.
>
>   > You can use underscore (_) instead of a variable if you don't care what value it has.
>   > f (x, _) = x
>   > :t f
>   > f :: (a, b) -> a
>   
>   > If you declare a variable like y here, the user might think you plan to use it. Not necessarily a big problem.
>   > f (x, y) = x
>   > :t f
>   > f :: (a, b) -> a
>   > fst (3, 'a')
>   > 3
>   > snd (3, 'a')
>   > 'a'
>   > s (_, y) = y --- our own version of snd
>   > :t s
>   > s :: (a, b) -> b
>   > :t snd
>   > snd :: (a, b) -> b
>   > s (3, 'a')
>   > 'a'
>   
>   > The parameters can have more complicated types than is obvious
>   > h p = (snd p, fst p) -- reverse an ordered pair
>   > :t h
>   > h :: (b1, b2) -> (b2, b1)
>   > let x = (3, 'a') in h x
>   > ('a',3)
>   > h (3, 'a')
>   > ('a',3)
• You can use [ ] and _ to build functions by cases on lists.

```haskell
> :{
| j [] = 0
| j [_] = 5
| j [_,_] = 7
| j [1,_,_] = 8 -- exactly 3 elements, first one being 1
| j [_,_,_] = 9 -- other 3-long lists return 9
| j (x : y) = x + j y -- if the list is of length 4 or greater...
| :}
> :t j
j :: (Num p, Eq p) => [p] -> p
> j [1,2,3]
8
> j [3,2,1]
9
> j [10,1,2,3]
18
```

**Pattern Matching**

• These definitions above all used pattern matching.
  • Many patterns look like expressions; the _ pattern is an exception
    ```haskell
    > k _ = 0 -- k of anything is zero
    > k 7
    0
    ```
  • Here's an error: you can't evaluate a pattern. Haskell is actually trying to parse this as a function definition and it thinks you've left off the rest of the definition.
    ```haskell
    > k _
    <interactive>:156:3: error:
    • Found hole: _ :: p20
    Where: 'p20' is an ambiguous type variable
    • In the first argument of 'k', namely '_'
    [...]
    ```
  • You can use literal constants (1, 2, ...) and the empty list [ ] and you can build up later patterns using [...] and colon (:), and you can use tuples (...,...,...), and also various "data constructors" that we'll see in a bit.
  • Things you can't use in patterns: regular functions like +,-, etc., sqrt, ++ (list concatenation).
    ```haskell
    > f ([1,2,3] ++ y) = 2 -- fails
    <interactive>:161:4: error: Parse error in pattern: [1, 2, 3] ++ y
    > f ( 1 : 2 : 3 : y ) = 2
    > -- colon can be used in patterns,
    > -- so we can specify "list of length ≥ 3 beginning with
    > -- 1, 2, 3" that way
    ```
> -- Can also have boolean test exprs as part of function definitions
> -- these are "guards"
> :{
| f n | n < 0 = 1
|     | n == 1 = 1
|     | otherwise = n * f(n-1)
| :}

> --should've included n == 0 = 1
> f (-3)
  1
> f 6
  720
> f 0 -- last case makes this 0 * f (0-1) = 0 * 1 = 0
  0

• Otherwise just means True
> :{
| g n | n <= 0 = 1
|     | True = n * g(n-1)
| :}
> g 5
  120

• Do have some issues with where the vertical bars can be:
  Prelude> :{
  Prelude| g n | n <= 0 = 1
  Prelude|     | True = n * g(n-1)
  Prelude| :}

  <interactive>:189:1: error: parse error on input ‘|’
• (Haskell thought the | meant we were defining a whole different thing.)
Activity Questions, Lecture 3

1. What is a type? What are its original and modern uses?
2. What are type expressions, type constants, type variables, and type constructors? How are they related? Give examples.
3. How are type variables used in Haskell to implement parametric polymorphism? Give examples.
4. How is parametric polymorphism different from templates? Give an example.
5. How are static and dynamic typechecking different? Must a language use just one or the other?
6. What does the strength or weakness of a typechecker refer to?
7. What kind of typechecking does Haskell have? (Strong vs weak?, static vs dynamic? Or a mix?)
8. Haskell has strong static typechecking - what does this mean?
9. You tend to have to put up with different problems when you have strong static typechecking versus strong dynamic and weak dynamic typechecking. Enumerate briefly and explain.
10. How general are the types the Haskell compiler tries to infer? [8/31]
11. How can type classes be seen as a generalization of operator overloading? Describe how (+) is an example.
12. What is type inferencing?

Function syntax questions [added 8/31]

13. What’s wrong with the declaration. Can we fix it without getting more information?
   \[
   \begin{align*}
   & f \ x = x \ * \ x \\
   & f \ y = y \ + \ y
   \end{align*}
   \]
14. Consider the function \( f \ x = 1 \), if \( x = 1 \), \( 2 \) if \( x = 2 \), and \( f \ (x-2) \) otherwise.
   a. Write \( f \) using conditional expressions (if-else...)
   b. Write \( f \) using cases (i.e., one pattern per line).
   c. Write \( f \) using guards (with vertical slashes).
   d. Try to add the clause \( f \ 0 \) if \( x \leq 0 \) to the three definitions above. Any failures? Briefly explain.

Answers

1 – 12 omitted (but if you’re really stuck on one, ask in class or on Piazza).

13. The second clause \( f \ y = y \ + \ y \) is redundant. Can’t fix the function unless we know more about which arguments are supposed to be squared and which are supposed to be doubled.
14. (Writing function definitions multiple ways)
14a. (with conditional expressions)
   \[
   f \ x = \text{if } x == 1 \text{ then } 1 \text{ else if } x == 2 \text{ then } 2 \text{ else } f \ (x-2)
   \]
14b. (by cases)
\[ f \ 1 = 1 \]
\[ f \ 2 = 2 \]
\[ f \ x = f \ (x-2) \]

14c. (using guards) Note the vertical bars should line up
\[ f \ x \ | \ x == 1 = 1 \]
\[ | \ x == 2 = 2 \]
\[ | \ otherwise = f \ (x-2) \]

15. (Add \( f \ x = 0 \) if \( x \leq 0 \) clause) [9/1; typos, tweaks]

15a. Just add a test of \( x \leq 0 \) then 0:
\[ f \ x = \text{if} \ x \ <= \ 0 \ \text{then} \ 0 \ \text{else} \ \text{if} \ x \ == \ 1 \ \text{then} \ 1 \ \text{else} \ \text{if} \ x \ == \ 2 \ \text{then} \ 2 \ \text{else} \ f \ (x-2) \]

15b. Can’t be done; we can test for specific \( x \) values (like 1 and 2), and we can test for no specific property (the \( f \ x \) case) but not for \( x \) with some property. (That’s what guards are for.)

15c. Add a \( | \ x \leq 0 = 0 \) clause. Note the first three clauses can be in any order but they all need to go before the catch-all otherwise clause. (If you change the otherwise to \( x > 2 \), then all four clauses can go in any order you like.)
\[ f \ x \ | \ x <= 0 = 0 \]
\[ | \ x == 1 = 1 \]
\[ | \ x == 2 = 2 \]
\[ | \ otherwise = f \ (x-2) \]

(Wasn’t asked for): Turns out you can combine cases and clauses.
\[ f \ 1 = 1 \]
\[ f \ 2 = 2 \]
\[ f \ x \ | \ x <= 0 = 0 \]
\[ | \ otherwise = f \ (x-2) \]

(For study purposes, go off and repeat problem 15 on your own functions.)