Haskell, Part 3

CS 440: Programming Languages and Translators, Spring 2020

1/22: Add activity pp.11-15, 2/2 p.9

Prompts

- For the examples below, assume we've already declared

```
> :set prompt "> "        -- normal prompt
> :set prompt-cont "| " -- prompt for lines between :{ ... }:
```

A. Typechecking: Static, Dynamic, Strong, and Weak [partly LYaH Ch.3?]

Static [LYaH Ch.3 p1] vs Dynamic Typechecking [not in LYaH}

- 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 [not in LYaH}

- 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-specially, "**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].

Strong vs Weak Dynamic Typechecking [not in LYaH}

- 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.

B. Chapter 4: Syntax in Functions [LYaH Ch.4]

- What syntax do we use for defining functions? More than *id var = expr*
- There's **definition by case**. Below, when `f` is given 0, 1, or 2, we have a specific case that covers the situation. The `f x = 3` case handles things when `x < 0` or `x > 2`.

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

- The order of classes matters: If we move the f x = 3 case to the top, then it will handle all values of x, so the other cases will never apply.
- Haskell gives you an error message if you enter a case that can't be satisfied. E.g., with

---

[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.


```
> :{
| 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
```

- The definition for `fact n` below works for $n \geq 0$, but on $n < 0$, it produces infinite recursion. On unix, you can use ^C to break an infinite loop

```
> :{
| fact 0 = 1
| fact n = n * fact(n-1)
| :}
> -- f works ok on values ≥ 0
> fact (-1) -- infinite recursion
^C^C^C^CInterrupted.
```

- You can use underscore (_) instead of a variable if you don't care what value it has.

```
> f (x,y) = x -- same as fst function
> s (x,y) = y --- same as snd function
> :t (f, s, fst, snd)
(f, s, fst, snd)
  :: ((a1, b1) -> a1, (a2, b2) -> b2, (a3, b3) -> a3, (a4, b4) -> b4)
```

- Note `f` and `s` above only use one of their two arguments. To skip making a useless binding at runtime, you can use an underscore "_" in place of `x` or `y` above.

```
> f (x, _) = x
> s (_, y) = y
```

[Side discussion on laziness)

- Since Haskell uses lazy evaluation, the un-looked-at argument of `f` and `s` doesn't get evaluated.
- E.g., `inf` is the prototypical infinite loop function, but you can write a call of it in the useless field of the pair for `f` or `s`. The corresponding code in C / C++ / Java would cause an infinite loop regardless.

```
> inf x = inf x
> f("hi", inf 17)
"hi"
> s(inf "oops", 1234)
1234
```

End of side discussion]

- 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 analyze list arguments to functions

```
> :{
| 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
```

Multiple cases on a line

You can write multiple function cases on one line, but you have to separate them with semicolons. E.g.,

```
> f 0 = 1 ; f 1 = 2 ; f n = 0
```

C. Pattern Matching [LYaH Ch.4, p.1]

- Definitions of functions by cases relies on patterns and pattern matching.
  - When you write out the pattern during coding, you are specifying some arrangement you expect your data to have.
  - When a pattern is used at runtime, the pattern structure and data structure are checked for compatibility (a successful pattern match) or incompatibility (a failed pattern match).
  - In addition, when the data is inspected, you can bind names to parts of it or to all of it.
- Many patterns look like expressions; the _ pattern is an exception. **You can't evaluate a pattern** as an expression. E.g., the first line below defines k correctly as a function of one argument, which it ignores to return `0`.

```
> k _ = 0 -- k of anything is zero
> k 7
0
```


- If you try to use `k _` as an expression, you'll get an error message: `k _` is interpreted as the beginning of a function definition. If you're in single-line input mode of interpreter, hitting return produces an error message because you didn't finish off the definition. If you're in multi-line input, then ghci expects the rest of the function definition (and complains if you end the input).

```
> k _   -- error!
```

[... long error message that mentions a hole ...]

```
> :{
| k _
|  = 3
| :}
> k 17
3
> :{
| k _   -- error!
| :}
```

[... repeat of long error message that mentions a hole ...]

Patterns in list comprehensions [LYaH Ch.3, p.3]

- We've seen list comprehensions that let a variable range through the values of a list, like `[2*x | x <- y]`.
- More generally, we can use a pattern to match each value from the list.
  - E.g, `[left | (left, right) <- list_of_pairs]` takes a list of pairs and produces a list of the left elements. For a particular example, `[left | (left, _) <- [(1, 'a'), (2, 'b')]] = [1,2]`. (Since we didn't use the right elements, we didn't need to create a name for them.
- If a pattern match fails for some element, then no value is calculated for the new list.
  - E.g., `[x | (x : _) <- [[], [1], [2,3], [4,5,6]]] = [1,2,4]`. The pattern `(x : _)` will match any nonempty list and bind x to the head of the list. The match against `[]` failed, so no value was calculated. On the other hand, the heads of the other three lists existed and were collected.

D. Building Patterns

- Patterns are built using the *don't care* pattern `_`, *literal constants* (`1`, `2`, …, `True`, `False`), the *empty list* `[]`, nonempty *lists of patterns*, the *colon operator* (*value pattern* `:` *list pattern*), *tuples of patterns* `(…,, …)`, *as* patterns (see below), and various "data constructors" we'll see when we study `datatypes`.
- **You can't use general functions** within patterns, so no `+`, `-`, …, `sqrt`, `++`, for example.
- Sometimes you can write a pattern that does what you might have wanted with `++`. E.g., say we want a function that returns true iff a list starts off with `1`, `2`, `3`. We can't write this as the list `[1,2,3] ++` another list because we can't use `++`.

```
> :{
| f123 ([1,2,3] ++ _) = True -- bad definition
| f123 _ = False
:}
```

[error message about parse error in pattern]

- On the other hand, we can say we want to build a list using `:` and `_`.

```
> :{
| f123 (1 : 2 : 3 : _) = True  -- works
| f123 _ = False
:}
```

- If you use a variable in a pattern, then during matching, Haskell will bind the variable to the appropriate value. **You can only use a variable once** in a pattern, so writing a function that checks for a pair of the same value has to use two variables, not one. E.g., `doubled1` below causes an error; `doubled2` does not. (Also notice we only need one line to define `doubled2` because the test for equality is done in the body of `doubled2`.)

```
:{
> doubled1(x,x) = True
| doubled1(_,_) = False
:}
```

[error message about conflicting definitions for 'x']

```
> doubled2(x,y) = x == y
>
```

- **As patterns** (`@`). The pattern *name* @ *subpattern* lets you bind a name to an entire value while also trying to match it against the subpattern.
  - E.g., `f pair @ (x,y) = (x, y, pair)` takes an ordered pair and produces a triple consisting of the two values plus the pair. So `f("hi", 4) = ("hi", 4, ("hi", 4))`,

E. Definitions using Guards [LYaH Ch.4, p.5]

- There are times when you want to do boolean tests within a function definition.
- An example was the factorial function earlier where we defined fact n on $n \geq 0$ but didn't have a way to write a pattern for `fact n` when $n \leq 0$, so the definition below caused infinite recursion on `fact (-1)`.

```
> :{
| fact 0 = 1
| fact n = n * fact(n-1) -- for all n ≠ 0
:}
```

- A **guard** is a boolean expression test that tests to see if a function body should be used. (If the guard evaluates to `True`, we use it; if `False`, we continue on.) E.g., we can write the `doubled` function (did you give it a pair of the same value?). Notice that you put an equal sign between the guard and function body.

```
> :{
| doubled(x,y)
|      | x == y = True
|      | x /= y = False
:}
```

- A simplification: The second guard `x /= y` does a redundant test, since you only get there if `x == y` failed, so we can replace it by `True`. (You can also use `otherwise`, which is a guard that means `True`.)


```
> :{
| doubled(x,y)
|      | x == y = True
|      | otherwise = False
:}
```

- **Guards and indentation**: Don't put the vertical bars of the guards in column 1: You'll get an error message because Haskell will think you're defining a whole different thing.

```
> :{
| doubled(x,y)
| | x == y = True
| | otherwise = True
| :}
<interactive>:189:1: error: parse error on input '|'
```

- A list of guards does tests based on the pattern before the first guard. If none of the guards in a list are true, then testing continues with the next pattern.
  - E.g., here `doubled _` is used when `x == y` fails.

```
> :{
| doubled(x,y)
|      | x == y = True
| doubled _ = False
:}
```

- **More complicated versions of** `_` : Above, we could have written `doubled (_, _) = False`.
  - Stylistically, doing this would emphasize that `doubled` expects a pair as its argument, but that was made clear two lines above, so it's probably not needed.
- **Using the** `error` **function:** The function `error` is of type `String -> a`, so you can use a call to error in any context requiring an expression. Here's yet another version of factorial, this time producing a runtime error on negative arguments.

```
> :{
| fact 0 = 1
| fact n | n > 0 = n * fact(n-1)
| fact _ = error "factorial of negative number"
| :}
```

---------- 2020-01-21

Multiple guards inline [LYaH Ch.3 p.6]; Guards and cases on one line

- You can write multiple guarded clauses on one line; just be sure to include the vertical bars as separators.

```
> doubled(x,y) | x == y = True | otherwise = False
>
```

- You can also combine guards and cases, but remember to separate cases with semicolons.

```
> doubled(x,y) | x == y = True ; doubled _ = False
>
```


F. Let Expression [LYaH Ch3., p.8]

- A `let` expression lets you define a local variable; the scope of the variable is the body of the `let`.
- The syntax is `let` *var* = *value_expr* `in` *body_expr* .

```
> let x = 3 in x + 2
5
> let y = 3 in [y+2, y*y]
[5,9]
```

- Note the name `x` is not available after the first line; the name `y` is not available after the second line.

```
> x
<interactive>:32:1: error: Variable not in scope: x
```

- You can use `let` expressions within larger expressions

```
>  3 * let x = 7 in x*x
147
```

- More fully, the syntax for a let is `let` *pattern* = *value_expr* `in` *body_expr*. I.e., you can bind to patterns, not just variables. However, if the pattern match fails, you get a runtime error.

```
> let x:y = [1,2,3] in (x,y)
(1,[2,3])
> let x:y = [] in (x,y)
  [error message about non-exhaustive patterns]
```

- You can define **local functions** with let expressions and use them in the body. [LyaH Ch.3, p.10]

```
> :{
| let fact 0 = 1
|     fact n | n > 0 = n*fact(n-1)
|     fact _ = error "negative fact"
|  in [fact k | k <- [1..5]]
:}
[1,2,6,24,120]
```

- You can have multiple bindings in a `let` expression. If you put them on one line, separate them with semicolons. If you put them on successive lines, line up their left ends. The `in` keyword can be on the same line as the last binding or it can be on its own line. The body can be on the same line as the `in` keyword or on its own line. When on its own line, people generally indent it relative to the `l` in `let`, but it's not required.

```
> let x = 3 ; y = x*x in x+y
12
> :{
| let x = 3
|     y = x*x
| in
|     x+y
:}
```



```
12
```

G. Case Expressions

- **Case expressions** Let you take an expression and match it against various patterns. Here's an example of a function declared two ways. The first two calculate the length of a string and test it, the first using conditional expressions, the second using pattern matches against constants. The third function doesn't calculate the length explicitly, it uses list pattern matches. ~~For the last two functions, note that unlike function definition by cases, there are no vertical bars separating case clauses~~ [2/2 Sorry, don't know what I was thinking: Patterns don't involve vertical bars; that's for guards -- see Multiple guards inline about 3 pages back.]

```
> :{
| f1 x = let len = length x in
|            if len == 0 then "empty"
|            else if len < 3 then "short"
|            else "long"
| f2 x = case length x of
|            0 -> "empty"
|            1 -> "short"
|            2 -> "short"
|            _ -> "long"
| f3 x = case x of
|            [] -> "empty"
|            [_] -> "short"
|            [_,_] -> "short"
|            _ -> "long"
:}
[ (f1 x, f2 x, f3 x) | x <- ["", "a", "ab", "abc"]]
[("empty","empty","empty"),("short","short","short"),
("short","short","short"),("long","long","long")]
>
```

- Case patterns can bind local variables (their scope is the expression after the `->`). In the example below, we can run `first` on any list of printable values. Recall `show h` returns the string that represents `h` (assuming `h` has a string representation).

```
> :{
| first x = case x of [] -> "No first element"
|                      h : _ -> "First was " ++ show h
| :}
> first []
"No first element"
> first [17,18]
"First was 17"
> first [[], [3]]
"First was []"
```



```
> first [sqrt]
    • No instance for (Show (Double -> Double)) …
```

Function definitions by case

- Not surprisingly, functions defined using cases are implemented internally using `case` expressions. Functions `g1` and `g2` below are equivalent.

```
> :{
| g1 0 = 1
| g1 1 = 3
| g1 _ = 5
|
| g2 x = case x of 0 -> 1
|                  1 -> 3
|                  _ -> 5
| :}
> [ (g1 x, g2 x) | x <- [-1..3]]
[(5,5),(1,1),(3,3),(5,5),(5,5)]
```


Activity Questions, Lecture 3 [1/22: added]

1. Types and Typechecking

1. How are static and dynamic typechecking different? Must a language use just one or the other?
2. What does the strength or weakness of a typechecker refer to?
3. Haskell has strong static typechecking - what does this mean?
4. 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.
5. What is type inferencing? [1/22: Used to be Question 7]
6. How can type classes be seen as a generalization of operator overloading? Describe how (+) is an example.
7. How general are the types the Haskell compiler tries to infer? [1/22: Used to be Question 5]
8. What is the type of function inf defined via `inf x = inf x` ? Justify this, briefly.

2. Function Syntax

1. Translate the following function so that it's defined by cases. Also translate into one that uses guards.

```
:{
| f x =
|     if x == "" then ""
|     else if x == "a" then "a"
|     else if head x == 'b' then "B"
|     else if tail x == "" then ">" ++ [head x]
|     else f (tail x)
:}
>  [f x | x <- ["", "a", "b", "bc", "d", "efgh"]]
["","a","B","B",">d",">h"]
```

2. Rewrite each of the following incorrect function definitions to use patterns and/or guards, or argue that it's impossible. (The errors to fix might be syntactic or semantic.)

   a.
   ```
   sum n = n + sum(n-1)
   sum 0 = 0
   ```

   b.
   ```
   f[x, x, _] = True
   f[x, y, _] = x < y
   f _ = False
   ```

   c.
   ```
   g(x, y, x) = x < y
   g(x, x, y) = x < y
   g(y, x, x) = x < y
   ```



```
    g(x, y, z) = x < y && y < z
```

d.
```
    h x = False if x == [] or [_]
    h ([val] ++ _ ++ [val]) = true
        -- want true iff first and last values match
```

3. Consider the function `f x` = `0` if `x` ≤ `0`, `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, one pattern per line
   c. Write `f` using guards, one guard per line.

3. Let and Case expressions

1. What are the differences between (`let h1 : h2 : _ = x in h1+h2`) and `[h1+h2 | h1 : h2 : _ <-` [x]`]` ? [1/23]
2. Here is the same `let` expression with different indentations and `;` separators. Which ones cause errors in Haskell? (Hint: Enter them in and see.)

a.
```
    let x = 3 y = x*x in x+y
```

b.
```
    let x = 3
     y = x*x in x+y
```

c.
```
    let x = 3
         y = x*x in x+y
```

d.
```
    let x = 3;
        y = x*x
    in x+y
```

e.
```
    let x = 3;
      y = x*x in x+y
```

f.
```
    let x = 3;
         y = x*x in x+y
```


Answers to Activity Questions for Lecture 3

1. Types and Typechecking

1. Static checking is done before execution. Dynamic checking is done during execution. It's possible to use both (static checking for things like integers, but dynamic testing for OO inheritance).
2. When a typechecker okays the types of a program, the strength/weakness refers to level of guarantee that there actually won't be type errors at runtime. (Totally strong ("type-safe"): Complete confidence; Weak: Some confidence. Untyped languages don't even have typecheckers, so there's no formal guarantee of type correctness.
3. Strong static typechecking: Typechecking is type-safe and done at before execution (at compile time).
4. Strong static checking: Typically have restrictions on what kinds of language features you can have, compiling gets slowed down, but execution is faster. (An example from lecture is Haskell requires list values to all have the same type, so you can't have a list that mixes numbers and characters, for example. You have to create a type that embeds / hides / boxes the integer or character, then you make a list of that type of data.) At the other end, with untyped languages, you have no parse-time restriction on using different types of data, so parsing is faster, but you need dynamic typechecks to make sure you're using data correctly.
5. In type inferencing, we use the language's program structure and type system to reason about the types of variables and expressions.
6. In operator overloading (as in e.g., C), an overloaded symbol is a name that has different meanings depending on context. E.g., + can mean integer addition and it can mean floating-point addition, which are different operations (literally, in the hardware). A Haskell typeclass is a name for a set of types that are guaranteed to support some given set of operations / variables / types. E.g., any type that is an instance of `Num` (i.e., a member of the `Num` typeclass) has to support + and related operators. If you create your own types and provide them with arithmetic operations, you can make the type an instance of `Num` and use the usual +, -, etc. symbols. (Instead of prefix functions like `add`, `subtract`, etc.
7. The Haskell compiler tries to infer the most general type that meets the restrictions imposed by the structure / use of a variable or expression.
8. Function `inf` defined via `inf x = inf x` has type `a -> b`: You can pass any kind of argument to it and (pretend) that it produces any kind of result. The type inferencer starts by giving the parameter and result types names (`a` and `b` here) and then looks at the definition of the function for internal compatibility and inferred type information. So if `x :: a`, then in the body, we're passing a type `a` value to `inf`, so that's okay, so we're getting a type `b` value back, which we return, which is okay because we're supposed to return a type `b` value. We hit the end of the function definition and everything was okay from a type viewpoint, so `inf :: a -> b`.

2. Function Syntax

1. (Translate to use (definition by) cases) Leaving out the ghci prompts, the program itself is

```
f "" = ""
f "a" = "a"
f ('b' : _) = "B"           -- if head of arg is 'b'...
f (h : "") = ">" ++ [h]     -- if tail is empty, use head
f (_ : t)  = f t            -- otherwise use tail
```

2. (Take function definitions with incorrect syntax or semantics to use patterns and/or guards, if possible)

   a.
```
sum 0 = 0   -- (need specific case earlier than general case)
sum n = n + sum(n-1)   -- note we don't terminate if n < 0
```

   b.
```
f[x, y, _]
    | x == y = True
    | otherwise = x < y
f _ = False
```

   c.
```
g(x, y, z)
    | x == z = x < y              -- g(x,y,x) = x < y
    | x == y =  x < z             -- g(x,x,y) = x < y
    | y == z = y < x              -- g(y,x,x) = x < y
    | otherwise x < y && y < z
```

   d.
```
h [] = False
h [_] = False
h x == head x == last x    -- can't use pattern to get last x
```

   For the last line, we can also use
```
h (hd : tl) == hd == last (tl)
```

3. (Writing function definitions multiple ways)

   a. (with conditional expressions)
```
f x = if x <= 0 then 0 else if x == 1 then 1 else if x == 2 then
2 else f (x-2)
```

   b. (by cases) can't be done: there's no way to write a pattern for "f x where x <= 0" [that's what guards are for.) We can do this much:
```
f 1 = 1
```



```
f 2 = 2
f x = f (x-2)
```

c. (Using guards)

```
f x | x <= 0 = 0
    | x == 1 = 1
    | x == 2 = 2
    | otherwise = f (x-2)
```

3. Let and Case expressions

1. If `x` has length < 2, then (`let h1 : h2 : _ = x in h1+h2`) causes a runtime error, but `[h1+h2 | h1 : h2 : _ <- x]` is `[]`.

2. (Modify indentation and `;` separators in `let` expressions)
   Omitted (just try typing them into ghci!)

