Haskell: Flow of Control

CS 331 Programming Languages
Lecture Slides
Wednesday, February 26, 2025

Glenn G. Chappell
Department of Computer Science
University of Alaska Fairbanks
ggchappell@alaska.edu

Unit Overview
The Haskell Programming Language

Topics

- ✓ PL feature: type system
- ✓ PL category: functional PLs
- ✓ Introduction to Haskell
- ✓ Haskell: functions
- ✓ Haskell: lists
- Haskell: flow of control
- Haskell: I/O
- Haskell: data

Review

Review
Haskell: Functions

Haskell identifiers begin with a letter or underscore, and contain only letters, underscores, digits, and **single quotes** (`'`).

Haskell has two kinds of identifiers.

- **Normal identifiers** begin with a lower-case letter or underscore. These name variables—including functions.
- **Special identifiers** begin with an UPPER-CASE letter. These name modules, types, and constructors. This is *not* merely a convention!

```
myVariable          -- Normal
_my_Function'_33    -- Normal
MyModule            -- Special
```

For code from this topic, see `func.hs`.

Review
Haskell: Lists [1/2]

Cons: construct a list from its first item and a list of the rest of its items. In Haskell, infix colon (`:`) operator.

```
isEmpty [] = True
isEmpty (x:xs) = False
```

A pattern that matches any nonempty list.
Parentheses are necessary due to precedence.

For code from this topic, see `list.hs`.

DONE

- Write a function `myMap` that does the same thing as `map`.
- Write a function `myFilter` that does the same things as `filter`.

A **predicate** is a function that returns a Boolean value.

A useful construction: `if` *COND* `then` *EXPR1* `else` *EXPR2*.

- *COND* is an expression of type `Bool`. If it is `True`, then *EXPR1* is returned. If it is `False`, then *EXPR2* is returned.
- Expressions *EXPR1* and *EXPR2* must have the same type.

Haskell: Lists [2/2]

Lists can be dealt recursively. Base case: empty list. Recursive case: nonempty list—do a computation with the **head** (first item); make a recursive call on the **tail** (list of all other items).

Sometimes other kinds of recursion are used.

DONE

- Write a function `lookInd` that does item lookup by index (zero-based) in a list.

See `list.hs`.

Useful

- The pattern "`_`" matches any single entity but cannot be used in the right-hand side of a definition. So it means *unused value*.
- `error` is an overloaded function that takes a `String` and returns any type at all. When it executes, it crashes, printing an error message, which will include the given `String`.
- `undefined` is like `error`, but it takes no arguments.

Haskell: Flow of Control

Haskell: Flow of Control
Introduction

Flow of control refers to the ways a PL determines what code is executed.

For example, flow of control in Lua includes:

- Selection (`if` … `elseif` … `else`).
- Iteration (`for` … `=`, `for` … `in`, `while`, `repeat` … `until`).
- Function calls.
- Coroutines.
- Exceptions.

Haskell has very different flow-of-control facilities from PLs that are oriented toward imperative programming.

For code from this topic, see `flow.hs`.

Haskell: Flow of Control
Pattern Matching, Recursion, Lazy Evaluation [1/5]

We have seen that Haskell has a **pattern matching** facility, which allows us to choose one of a number of function definitions. The rule is that the first definition with a matching pattern is used.

```
isEmpty [] = True
isEmpty (_:_) = False


-- sfibo: SLOW Fibonacci
sfibo 0 = 0
sfibo 1 = 1
sfibo n = sfibo (n-2) + sfibo (n-1)
```

In many of the places we would use an if … else construction in other PLs, we can use pattern matching in Haskell.

Haskell: Flow of Control
Pattern Matching, Recursion, Lazy Evaluation [2/5]

Haskell also makes heavy use of recursion.

```
lookInd 0 (x:_) = x
lookInd n (_:xs) = lookInd (n-1) xs
lookInd _ [] = error "lookInd: index out of range"
```

In places where we would use a loop in an imperative PL, we use recursion in Haskell.

Recursion can be less costly in Haskell than in PLs like C++, because of Haskell's required **tail-call optimization** (**TCO**). TCO means that a tail call does not use additional stack space.

By default, Haskell does **lazy evaluation**.

We have seen that this allows for infinite lists. There is syntax for constructing these (involving ". ."), but we can actually make infinite lists without using this syntax. How? Here is an idea.

```
-- listFrom n
-- Returns the infinite list [n, n+1, n+2, ...].
listFrom n = n:listFrom (n+1)
```

Is the above code acceptable? It does recursion with no base case.

But this is not a problem, thanks to lazy evaluation. A recursive call is only made if list items are needed. Using only a finite number of list items guarantees that the recursion terminates.

Haskell: Flow of Control
Pattern Matching, Recursion, Lazy Evaluation [4/5]

Something else we can do: write our own if … else, as a function.

```
-- myIf condition tVal fVal
-- Returns tVal if condition is True, fVal otherwise.
myIf True tVal _ = tVal
myIf False _ fVal = fVal
```

Thanks to lazy evaluation, no more than one of `tVal`, `fVal` is ever evaluated. So `myIf` is efficient.

Here is the slow Fibonacci algorithm using `myIf`.

```
sfibo' n = myIf (n <= 1) n (sfibo' (n-2) + sfibo' (n-1))
```

Haskell: Flow of Control
Pattern Matching, Recursion, Lazy Evaluation [5/5]

And here is a reimplementation of `myFilter`, using `myIf`.

```
myFilter' p [] = []
myFilter' p (x:xs) = myIf (p x) (x:rest) rest  where
    rest = myFilter' p xs
```

It turns out that the combination of pattern matching, recursion, and lazy evaluation, together with function calls, are all we need. We can build any flow-of-control construct out of these—as long as the construct does not require eager evaluation.

However, Haskell has other flow-of-control facilities, for convenience. Next we look at a few of these.

Selection — Introduction

Selection refers to flow-of-control constructs that allow us to choose one of multiple options to execute.

Selection constructions in C++ include `if … else`, `switch`, and virtual function dispatch.

In Haskell, pattern matching works as a selection mechanism. Other selection constructions include `if … then … else`, *guards* (covered shortly), and a `case` construction (not covered).

Haskell: Flow of Control
Selection — if … then … else

We have seen Haskell's `if … then … else` construction. This is much like our `myIf`.

```
myIf condition tVal fVal
if condition then tVal else fVal  -- Same as above
```

Some people consider `if … then … else` to be un-Haskell-ish. I have found it natural to use when doing I/O (discussed at another time); otherwise, I generally prefer to use *guards*.

Haskell: Flow of Control
Selection — Guards [1/3]

Guards are the Haskell equivalent of mathematical notation like the following.

$$myAbs(x) = \begin{cases} x, & \text{if } x \geq 0; \\ -x, & \text{otherwise.} \end{cases}$$

In Haskell:

```
myAbs x
    | x >= 0      = x
    | otherwise   = -x
```

We can use guards in situations that pattern matching cannot handle. For example, there is no pattern that matches only nonnegative numbers.

Haskell: Flow of Control
Selection — Guards [2/3]

```
myAbs x
    | x >= 0      = x
    | otherwise   = -x
```

Note that there is no equals sign after the first line above.

Each vertical bar is followed by a Boolean expression. The first `True` expression tells which value is used.

To handle all remaining cases, we *could* use "`True`" as our final expression. "`otherwise`" is a variable with value `True`.

Here is the slow Fibonacci algorithm reimplemented using guards.

```
sfibo'' n
    | n <= 1      = n
    | otherwise   = sfibo'' (n-2) + sfibo'' (n-1)
```

Haskell: Flow of Control
Selection — Guards [3/3]

Here is `myFilter` reimplemented using guards.

```
myFilter'' p [] = []
myFilter'' p (x:xs)
    | p x         = x:rest
    | otherwise   = rest  where
        rest = myFilter'' p xs
```

Haskell: Flow of Control
Error Handling

We have seen Haskell's fatal-error facilities: `error` and `undefined`.

```
lookInd 0 (x:xs) = x
lookInd n (x:xs) = lookInd (n-1) xs
lookInd _ [] = error "lookInd: index out of range"
```

We use these in situations when a program needs to crash because it has detected a bug in its code. It crashes with an explanatory message, so that a developer can fix the bug.

Haskell has an exception-catching mechanism, for dealing with error conditions that may be handled at runtime. We will not cover this.

If you look into Haskell exceptions, then be aware that the Haskell Standard Library and documentation toss around "error" and "exception" rather loosely. Sometimes the terms are used in inconsistent ways.

Haskell: Flow of Control
Encapsulated Loops — Introduction

A very important idea in functional programming:

> **Many flow-of-control constructs can be encapsulated as functions**.

We have already done this with if ... else, in the form of function `myIf`. Next we look at four ways to encapsulate loops:

- `map`
- `filter`
- `zip`
- *Fold* operations

Haskell: Flow of Control
Encapsulated Loops — map & filter [1/3]

Consider the following C++ code. `v` is a `vector<int>`.

```
vector<int> w;
for (auto n : v)
{
    w.push_back(n % 3);
}
```

The same computation in Haskell is done without a loop.

```
w = map (\ n -> n `mod` 3) v
w = [ n `mod` 3 | n <- v ]  -- Alternate form, using a
                            --  list comprehension
```

Haskell: Flow of Control
Encapsulated Loops — map & filter [2/3]

Another C++ snippet:

```
vector<int> w;
for (auto n : v)
{
    if (n > 6)
        w.push_back(n);
}
```

And the equivalent Haskell:

A nice syntax that can be used with any infix binary operator, turning it into a function with one argument.

```
w = filter (> 6) v
w = [ n | n <- v, n > 6 ]  -- Alternate form, using a
                           --  list comprehension
```

Haskell: Flow of Control
Encapsulated Loops — map & filter [3/3]

So Haskell's `map` and `filter` functions—or the equivalent list comprehensions—perform operations that we would write as loops in other PLs.

In particular, `map` and `filter` encapsulate loops that process a sequence of values, constructing another sequence of values.

Haskell: Flow of Control
Encapsulated Loops — zip

Haskell has functions that encapsulate other kinds of loops. For example, `zip` takes two lists and returns a list of 2-item tuples.

```
> zip [8,3,7] "sun"
[(8,'s'),(3,'u'),(7,'n')]
```

`zip` stops when either of the given lists runs out.

```
> zip [8,3,7,4] "sunshine"
[(8,'s'),(3,'u'),(7,'n'),(4,'s')]
```

Haskell: Flow of Control
Encapsulated Loops — Fold Operations [1/3]

Yet another kind of loop involves processing a sequence of values and returning a single value. The result might be the sum of all the numbers in the sequence, the greatest value in the sequence, etc. The operation performed by a loop like this is called a **fold** (or sometimes **reduce**).

Here is an example of something that is conceptually a fold operation, implemented in a traditional manner in C++.

```
int result = 0;  // Will hold sum of items in v
for (auto n : v)
{
    result += n;
}
```

Haskell: Flow of Control
Encapsulated Loops — Fold Operations [2/3]

Haskell has a number of fold functions. These include `foldl`, `foldr`, `foldl1`, and `foldr1` (the "`l`" and "`r`" stand for "left" and "right").

Below, I show a call to each function, with a comment showing what the call computes.

```
foldl (+) 0 [1,2,3,4] -- (((0+1)+2)+3)+4

foldr (+) 0 [1,2,3,4] -- 1+(2+(3+(4+0)))

foldl1 (+) [1,2,3,4]  -- ((1+2)+3)+4

foldr1 (+) [1,2,3,4]  -- 1+(2+(3+4))
```

Haskell: Flow of Control
Encapsulated Loops — Fold Operations [3/3]

Here are more practical examples of Haskell folds.

```
> commafy str1 str2 = str1 ++ ", " ++ str2
> foldr1 commafy ["parsley","sage","rosemary","thyme"]
"parsley, sage, rosemary, thyme"

> bigger a b = if (b > a) then b else a
> maxVal list = foldr1 bigger list
> maxVal [5,2,7,3,9,8,4,9,2,1]
9
```

Haskell: Flow of Control
Other — seq [1/2]

`seq` is a primitive function that acts *almost* as if it is defined as:

```
seq x y = y
```

Except that the first argument (`x` above) is always evaluated.

`seq` is the unique Haskell primitive that breaks the lazy-evaluation rule. It evaluates something whose value may not be needed.

Why use `seq`? To control evaluation. Sometimes we can improve resource usage (time, stack space). *See the next slide.*

You will not need to use `seq` in this class. It is rarely used directly.

Haskell: Flow of Control
Other — seq [2/2]

Our `listLength` can crash with stack overflow for large lists.

```
> listLength [1..50000000]
*** Exception: stack overflow
```

But we can fix this as follows:

1. Make the function tail-recursive.
2. Use `seq` to prevent the construction of increasingly complex unevaluated expressions.

For details, see `flow.hs`.

You will not need to use `seq` in this class. It is rarely used directly.

Haskell: Flow of Control
Other — Do-Expression [1/2]

A final flow-of-control structure, which we will look at when we study Haskell I/O, is the *do-expression*. An example:

```
reverseIt = do
    putStr "Type something: "
    hFlush stdout  -- Requires import System.IO
    line <- getLine
    putStr "What you typed, reversed: "
    putStrLn (reverse line)
```

When a program returns this expression's value, this happens:

```
> reverseIt
Type something: Howdy!
What you typed, reversed: !ydwoH
```

Typed by user

Haskell: Flow of Control
Other — Do-Expression [2/2]

```
reverseIt = do
    putStr "Type something: "
    hFlush stdout  -- Requires import System.IO
    line <- getLine
    putStr "What you typed, reversed: "
    putStrLn (reverse line)
```

A do-expression is syntactic sugar around a pipeline of functions. Roughly, each function takes the current state and returns it, possibly modified by an *I/O action*. Each of the lines above, except the first line, represents an I/O action.

More on this in our next topic: Haskell I/O.