Overview

- **Today**: Look at the Haskell programming language
- Like LISP, Haskell is a functional language
- Unlike LISP, Haskell is statically typed and has an expressive type system
- Integrates many concepts we looked at: static typing, type inference, polymorphism, higher-order functions, ...
- At the cutting edge of PL research: designed by researchers, lots of new research done in the context of Haskell

Little Bit of History

- Haskell borrows ideas from both LISP and ML
- Like LISP and ML, functional language
- Like ML, has a strong, expressive static type system
- But also different from both in some ways: e.g., lazy functional language

History of Haskell

- Designed in the early 90’s by a committee of researchers
- Goal was to unify research efforts in pure, lazy functional languages
- Feeling was that widespread use of lazy functional languages was being hampered by lack of standard language
- Goal of the committee was to design a lazy functional language that would become standard
- People who played key role in designing Haskell: Simon Peyton Jones (Microsoft Research), Philip Wadler (Edinburgh), Paul Hudak (Yale), John Hughes (Chalmers)

Why Called Haskell?

- The Haskell programming language is named after logician Haskell B. Curry
- Haskell curry well-known for Curry-Howard isomorphism
- Curry-Howard isomorphism sheds light on relationship between constructive logic and functional programming languages
- Specifically, it says:
  1. Propositions are types
  2. Proofs are programs
  3. Proof checking is type checking

Overview of Haskell Features

- Lazy evaluation
- Expressive static type system
- Polymorphism and type classes
- Tagged unions, type constructors
- Pattern matching
- List comprehension
Lazy vs. Strict Programming Languages

- Haskell is different from many other functional languages in that it’s lazy.
- Lazy (or call-by-need, non-strict) languages delay evaluation of an expression until its value is actually required.
- In contrast, eager (or strict, call-by-value) languages, an expression is evaluated as soon as it is bound to a variable, regardless of whether the variable is used or not.
- The default behavior in many languages used today is eager evaluation: Java, C, C++, ML, Python, Lisp, Scheme . . .

Example of Lazy vs. Eager

- Suppose you pass a divergent (i.e., non-terminating) expression as an argument to a function.
- In an eager language, the program will definitely not terminate.
- In a lazy language, the program will terminate if the value of that argument is not needed in the called function.
- There are both advantages and disadvantages of lazy evaluation.

Advantages and Disadvantages of Lazy Evaluation

Advantages of Lazy Evaluation:
- Unused arguments are not evaluated – can be a big win if evaluation of expression is expensive.
- Can avoid divergent or buggy behavior in some cases.
- Can create infinite data structures.

Disadvantages of Lazy Evaluation:
- Can be difficult to predict program behavior.
- Doesn’t work well in the presence of side effects.
- Even in purely functional languages, things like IO become much more difficult.

Expressive Static Type System

- Haskell is a statically typed language with a sound type system.
- Does not allow ways to subvert the type system, such as casting.
- Thus, if the Haskell compiler assigns type \( T \) to expression \( e \), run-time value of \( e \) will be in the set of values defined by \( T \).
- Haskell performs type inference, so you don’t have to explicitly annotate types of expressions.
- Furthermore, Haskell has polymorphism: Types of variables can contain (universally-quantified) type variables.

Example of Polymorphism in Haskell

- Consider the id function in Haskell: \( \text{let id} \ x = x \)
- This function has the inferred polymorphic type \( \text{a} \to \text{a} \)
- For any type \( \text{a} \), if the value of the input is \( \text{a} \), the output is also of type \( \text{a} \).
- The Haskell type \( \text{a} \to \text{a} \) is the same as \( \forall \text{a} . \text{a} \to \text{a} \).

Another Example

- Let’s look at a more interesting example:
  \[
  \text{map} \ f \ [] = [] \\
  \text{map} \ f \ (x:xs) = f \ x : \text{map} \ f \ xs
  \]
- Here, we define a higher-order map function which applies a function \( f \) to every element in the list.
- This function definition illustrates pattern matching in Haskell: here, we pattern match on the second argument.
- First line: if the input list is empty, then return the empty list.
- Second line corresponds to case where list has at least one element.
Example, cont.

\[
\begin{align*}
\text{map } f \, [] & = [] \\
\text{map } f \, (x:xs) & = f \, x : \text{map } f \, xs
\end{align*}
\]

- By using the notation \((x:xs)\), we bind first element of input list to \(x\) and rest of the list to \(xs\)
- Second line: Applies function \(f\) to head \(x\) of the list
- Then, recursively invokes \(\text{map}\) with function \(f\) on the remainder \(xs\) of the list
- Finally, returns a list which is the concatenation of \(f \, x\) and result of recursive call

Haskell compiler infers type of this function to be:

\[ (a \to b) \to [a] \to [b] \]
- Again, this is a polymorphic type
- Says: Given any function of type \(a \to b\) and a list of values of type \(a\), return value is a list of values of type \(b\)
- Here, \(a\) and \(b\) are (implicitly) universally-quantified type variables

Type Classes

- Consider the Haskell function: \(f \, x = x \times 2\)
- What is the type of this function?
- We can’t give it the polymorphic type \(a \to a\) because it only works on types for which multiplication is defined
- But assigning type \(\text{Integer} \to \text{Integer}\) also too restrictive because works for floats or other types for which \(\times\) is defined
- Haskell solves this problem through type classes

Programming with Type Classes

- To use this feature of Haskell, first need to declare type classes
- A type class is simply a set of types that support a common set of operations
- To define the type class, declare the common operations and give type class a name
- Example:

```haskell
class Num a where
    (+) :: a -> a -> a
    (*) :: a -> a -> a
    ...
```
- This declares a type class called \(\text{Num}\) that supports operations \(+\) and \(\times\)
Algebraic Data Types

Another useful feature of Haskell is algebraic data types, also called tagged unions or discriminated unions.

Suppose we want to have a type to represent students.

For undergraduates, we want to track their name and department.

For masters students, we want to track their name, department, and specialty.

In Haskell, you can have a data type to represent both undergraduate and masters students:

```
data Student = BS (Name, Dept) | MS (Name, Dept, Area)
```

Here, BS and MS are constructors.

The BS constructor must be applied to a pair consisting of a name and department.

The MS constructor must be applied to tuple consisting of name, department, and specialty.

Thus, Student is really a union of the two types Name*Dept and Name*Dept*Area.

Here, A*B is a product type which corresponds to type of tuple whose elements are of type A and B.

Pattern Matching on Data Types

Now, suppose we want to write a name function that gives the name of any student.

We can conveniently do this in Haskell through pattern-matching:

```
name (BS(n,d)) = n
name (MS(n,d,a)) = n
```

Haskell compiler will infer type of name as Student -> Name.

This is another example of pattern matching in Haskell.

First line matches on students with type constructor BS.

Second line matches on students with type constructor MS.

Recursive Data Types

Data types in Haskell can also be recursive.

i.e., data type being declared can appear as arguments of type constructors.

Example:

```
data Tree = Leaf Int | Node (Tree, Tree)
```

Here is how you would write a function in Haskell that checks if a given integer is in the tree:

```
inTree x (Leaf y) = x==y
inTree x (Node(y,z)) = inTree x y || inTree x z
```

List Comprehension

Haskell has another very convenient feature called list comprehension.

List comprehension allows conveniently building data structures from existing data structures.

List comprehension is similar to set-builder notation in math:

```
E = \{ x | x \in A, x^2 = 0 \}
```

This says set E contains all even numbers in set A, i.e., build new set from existing set.

List comprehension in Haskell has similar notation and is very convenient.

List Comprehension Example

Here is an example of list comprehension in Haskell:

```
myList = [1,2,3,4,5,6,7,8]
twiceMyList = [2*x | x<-myList]
```

This code creates new list twiceMyList by going over every element in myList and multiplying it by two.

Just syntactic sugar, but very convenient.

Another example:

```
mysteryList = [(x,y) | x<-list1, y<-list2]
```

What does this code snippet do? takes cross product of two lists.
Anonymous Functions in Haskell

- Just like L and Lisp, can write anonymous functions in Haskell, but slightly different syntax
- Here is an anonymous Haskell function that adds one to its argument: \( x \rightarrow x+1 \)
- Anonymous functions especially useful for simple functions passed as arguments to higher order functions
- Example: map \((x \rightarrow x+1)\) myList
- Here, argument to map is an anonymous function that adds one to its argument
- Result of this expression is a list where every element is one greater than corresponding element in myList

Lazy Evaluation in Haskell Example

- Now, let’s write a function to get n’th element from a list
  
  \[
  \begin{align*}
  \text{get_nth} \; [] & \; _\; = \; 0 \\
  \text{get_nth} \; (x:xs) \; 1 & \; = \; x \\
  \text{get_nth} \; (x:xs) \; n & \; = \; \text{get_nth} \; xs \; (n-1)
  \end{align*}
  \]
- Will get_nth (magic 1 1) 3 terminate?
  - Yes – let’s see why
  - First, recall that since Haskell is lazy, magic 1 1 will not be evaluated until it is needed in get_nth
  - In get_nth, we need to figure out which pattern is matched
  - This forces one step in the evaluation of magic 1 1

Revisiting Lazy Evaluation in Haskell

- Now that we are more familiar with Haskell syntax, let’s revisit lazy evaluation is Haskell
- Consider the following function magic:
  
  \[
  \begin{align*}
  \text{magic} \; 0 \; _\; = \; [] \\
  \text{magic} \; m \; n \; = \; m \:\cdot\:\text{(magic} \; n \; (m+n))
  \end{align*}
  \]
- What is \text{magic} 1 1?
  - The list of Fibonacci numbers: \([1,1,2,3,5,8, \ldots]\)
  - Clearly, \text{magic} 1 1 does not terminate since this list is infinite

Example, cont.

\[
\begin{align*}
\text{magic} \; 0 \; _\; = \; [] \\
\text{magic} \; m \; n \; = \; m \:\cdot\:\text{(magic} \; n \; (m+n))
\end{align*}
\]

- Thus, after one step in evaluation of \(\text{magic} \; 1 \; (1+1)\), we get: \(1:\text{(magic} \; (1+1) \; (1+(1+1)))\)
- Thus, our expression is now:
  
  \[
  \text{get_nth} \; (1:\text{(magic} \; (1+1) \; (1+(1+1)))) \; (3-1)
  \]
- Now, to figure out if we match second or third case, we evaluate 3-1:
  
  \[
  \text{get_nth} \; (1:\text{(magic} \; (1+1) \; (1+(1+1)))) \; (2)
  \]
Example, cont.

```haskell
magic 0 _ = []
magic m n = m : (magic n (m+n))
get_nth [] _ = 0
get_nth (x:xs) 1 = x
get_nth (x:xs) n = get_nth xs (n-1)
```

```
get_nth ((magic (1+1) (1+(1+1)))) (2-1)
```

▶ Now, to figure out which pattern matches in `magic`, we need to evaluate first argument; this yields: `magic 2 (1+(1+1))`

▶ Now, second case matches, thus we have:

```
2 : (magic 1+(1+1)) (2+(1+(1+1)))))
```

▶ Now, we continue evaluating:

```
get_nth (2 : (magic 1+(1+1)) (2+(1+(1+1))))) (2-1)
```

▶ This forces evaluation of 2-1

▶ This means we match on second case!

▶ Thus, the whole expression evaluates to 2!

▶ Although we wrote a function to generate infinite list, expression to extract element from this infinite list terminates!

▶ This is one of the nice aspects of lazy evaluation

---

Haskell Summary

- Haskell is a lazy, pure-functional language
- Integrates a lot of research from PL community: polymorphism, type classes, type inference, . . .
- Statically typed, no escape hatches (e.g., casts) from type system
- Considered by many to be a very elegant language