Programming Abstractions

Lecture 36: Types

Stephen Checkoway

Announcements

Please fill out course evals

We only need 8 more students to fill out the evals to get the extra credit!

Recursive data types

Haskell has lists built in, but let's make our own list type

```
data List = Cons Int List
          | Empty
```

This is a recursive data type that says a list is either

- A `Cons` which contains an `Int` and a `List`; or it is
- `Empty`

Note that this is a sum of products:

- `Cons Int List` is a (named) product of an `Int` and a `List`, `Empty` is a (named) product of zero types
- `List` is a sum of these two products

Creating a list

```
reverseRange :: Int -> List
reverseRange n = if n == 0
                 then Empty
                 else Cons (n - 1) (reverseRange (n - 1))

ghci> reverseRange 5
Cons 4 (Cons 3 (Cons 2 (Cons 1 (Cons 0 Empty))))
```

MiniScheme

```
data Exp = Lit Int
         | Var String
         | App Exp [Exp]
         | IfThenElse Exp Exp Exp
         | Let [String] [Exp] Exp
         | Lambda [String] Exp
```

Another example of a sum of products

Result

```
data Result = Ok Int
            | Err String
  deriving (Show)

first :: List -> Result
first Empty = Err "This list is empty!"
first (Cons x xs) = Ok x

ghci> first (Cons 10 (Cons 20 Empty))
Ok 10
ghci> first Empty
Err "This list is empty!"
```

Note the two definitions for first
This is "pattern matching"

Let's write rest. Does this work?

```
rest :: List -> Result
rest Empty = Err "This list is empty!"
rest (Cons x xs) = Ok xs
```

A. Yes, Haskell is awesome!

B. No. Runtime error

C. No. Compile-time error

Limitations so far

Our List can only hold integers; our Result can only hold an integer or a string
We could make different list and result types for other types

```
data IntList = ICons Int IntList | IEmpty
data StringList = SCons String StringList | SEmpty

iLength :: IntList -> Int
iLength IEmpty = 0
iLength (ICons x xs) = 1 + iLength xs

sLength :: StringList -> Int
sLength SEmpty = 0
sLength (SCons x xs) = 1 + sLength xs
```

Polymorphic types to the rescue!

We can create **type constructors** which take in one (or more) types and produce a new type!

```
data List a = Cons a (List a) | Empty
```

Now, List is not a type, it's a **type constructor**

`a` (or anything starting with a lower case letter) is a **type variable**

If we apply `List` to a type like `Int` or `String`, we get a new type:

- `List Int`
- `List String`
- `List (List (Int, String))` — a list of lists of `(Int, String)` tuples

Defining functions that work on any type of list

Parametric polymorphism

```
length' :: List a -> Int
length' Empty = 0
length' (Cons x xs) = 1 + length' xs
```

(Haskell doesn't like when we shadow the built-in `length` because it doesn't know which one to use and it wants us to use `Main.length` to refer to ours so we used `length'` (apostrophes are valid characters in names))

What is the type of the map' function which behaves like map, but on our list type?

Here's the definition:

```
map' :: ???
map' f Empty = Empty
map' f (Cons x xs) = Cons (f x) (map' f xs)
```

A. `map' :: (a -> b) -> List -> List`

B. `map' :: a -> b -> List -> List`

C. `map' :: (a -> b) -> List a -> List b`

D. `map' :: a -> b -> List a -> List b`

The real list type

The real list type is a singly-linked list just like in Racket

Rather than using empty and cons for the empty list and the constructor, Haskell uses

- `[]` — empty list
- `x:xs` — constructor; : takes an element on the left and a list on the right

```
ghci> 1:2:3:[]
[1,2,3]
```

`:` is right-associative so this is the same as

```
ghci> 1:(2:(3:[]))
[1,2,3]
```

Revisiting Result

```
data Result a b = Ok a
                | Err b

first :: [a] -> Result a String
first [] = Err "List is empty"
first (x:xs) = Ok x

rest :: [a] -> Result [a] String
rest [] = Err "List is empty"
rest (x:xs) = Ok xs

ghci> rest [1, 2, 3]
Ok [2,3]
```

Aside: Types as documentation

Type signatures tell you a lot about what the function does

```
foo :: [a] -> a
```

foo takes a list of values of type a and returns a value of type a

What does this tell us about the return value of `foo [1, 2, 3, 4]`?

```
bar :: (a -> [b]) -> [a] -> [b]
```

So `bar` takes a function `a -> [b]` and a list of `a`s; it produces a list of `b`s

Where do the `b`s come from?

Examples matching those types

`head :: [a] -> a`

- takes a list as input
- returns the first element of the list (just like `first` in Racket)

last :: [a] -> a

- takes a list as input
- returns the last element of the list

`concatMap :: (a -> [b]) -> [a] -> [b]`

- takes a function `f :: a -> [b]` and a list of `a`s
- applies `f` to every element in the list and then concatenates all the lists together
- Equivalent to `(apply append (map f lst))` in Racket

Type inference

We don't have to give explicit types of functions

Haskell can figure out the most general types itself (in many cases)

```
reverseConcatMap f xs = reverse (concat (map f xs))

ghci> reverseConcatMap (\n -> [0..n]) [2, 3, 4]
[4,3,2,1,0,3,2,1,0,2,1,0]

ghci> :t reverseConcatMap
reverseConcatMap :: (a1 -> [a2]) -> [a1] -> [a2]
```

Type inference

(This is a bit hand-wavy)

```
map :: (a -> b) -> [a] -> [b]
concat :: [[a]] -> [a]
reverse :: [a] -> [a]
```

```
reverseConcatMap f xs = reverse (concat (map f xs))
```

Inference starts by assigning the types

```
reverseConcatMap :: t1 -> t2 -> t3
f :: t1
xs :: t2
```

Now it can start reasoning about the type of subexpressions

```
(map f xs) :: [t5]                          if t1 = t4 -> t5 and
                                               t2 = [t4]
(concat (map f xs)) :: [t6]                 if t1 = t4 -> t5,
                                               t2 = [t4], and
                                               t5 = [t6]
```

Type inference

(This is a bit hand-wavy)

```
map :: (a -> b) -> [a] -> [b]
concat :: [[a]] -> [a]
reverse :: [a] -> [a]
```

```
reverseConcatMap :: t1 -> t2 -> t3
f :: t1
xs :: t2
(map f xs) :: [t5]                         if t1 = t4 -> t5 and
                                              t2 = [t4]
(concat (map f xs)) :: [t6]                if t1 = t4 -> t5,
                                              t2 = [t4], and
                                              t5 = [t6]
(reverse (concat (map f xs))) :: [t7]      if t1 = t4 -> t5,
                                              t2 = [t4],
                                              t5 = [t6], and
                                              t6 = t7
```

Type inference

(This is a bit hand-wavy)

```
map :: (a -> b) -> [a] -> [b]
concat :: [[a]] -> [a]
reverse :: [a] -> [a]
```

```
reverseConcatMap :: t1 -> t2 -> t3
f :: t1
xs :: t2
(reverse (concat (map f xs))) :: [t7]   if t1 = t4 -> t5,
                                           t2 = [t4],
                                           t5 = [t6], and
                                           t6 = t7
```

So `t3 = [t7]` under a set of constraints giving us

```
t1 -> t2 -> t3 = (t4 -> t5) -> t2 -> t3        substituting t1
               = (t4 -> t5) -> [t4] -> t3      substituting t2
               = (t4 -> t5) -> [t4] -> [t7]    substituting t3
               = (t4 -> [t6]) -> [t4] -> [t7]  substituting t5
               = (t4 -> [t7]) -> [t4] -> [t7]  substituting t6
```

Type inference

(This is a bit hand-wavy)

Since `reverseConcatMap :: t1 -> t2 -> t3` and
`t1 -> t2 -> t3 = (t4 -> [t7]) -> [t4] -> [t7],` we have
```
reverseConcatMap :: (t4 -> [t7]) -> [t4] -> [t7]
```

We can rename `a1 = t4`, `a2 = [t7]` giving
```
reverseConcatMap :: (a1 -> [a2]) -> [a1] -> [a2]
```

which matches ghci
```
ghci> :t reverseConcatMap
reverseConcatMap :: (a1 -> [a2]) -> [a1] -> [a2]
```

Wrap up

Think of what you've done this semester

No Scheme knowledge → writing an interpreter for MiniScheme!

Key takeaways from the course

- Recursion!
- Functional programming
  - accumulators
  - tail recursion
- List manipulation functions (`map`, `filter`, `foldl`, `foldr`)
- Parsing and interpreting a language

Course evals

Remember to fill out course evals!

Final exam

Exam Format

Combination of problems (some or all of)

- True/false or multiple choice
- Short answer
- Code to write in DrRacket and uploaded to Blackboard

Exam will be available at 11:00 EDT on Wednesday, June 1, 2022

Your solutions are due by 11:00 EDT on Thursday, June 2, 2022

Late exams are *not* allowed by College policy (sorry, it's out of my control)

Note: that's 11 a.m.!

Final exam time

During the scheduled final exam time (09:00–11:00 EDT), I will be in my office

However, it's better to ask private questions on Piazza early instead since the scheduled time is the last two hours of the 24 you have to work on this

Possible question topics

Anything we have covered in the course from day 1 until today, including

- Basic Scheme/Racket procedures and special forms
  - cons, first, rest, list, append, empty?, filter, and all the others
  - define, lambda, if, cond, let, let*, letrec, etc.
- map, foldl, foldr
- apply
- Recursion
  - "Normal" recursion
  - Tail recursion
  - "Accumulator-passing style"
  - Continuation-passing style
- Closures
  - What they are, how we create them, and how we use them

Possible question topics

- Backtracking
  - Single solution
  - All solutions
- Environments
  - How and when they're created
- Lexical vs. dynamic binding
- Parameter passing mechanisms
  - Pass by value
  - Pass by reference
  - Pass by name

Possible question topics

- Interpreter project
  - Creating new structs
  - Implementation of the environment
  - Parsing expressions
  - Evaluating parse trees
  - Implementing new features/special forms
- Basic runtimes of procedures O(n), O(n log n), O(n^2), etc.
- Macros
  - How they work
  - How to write new ones
- Promises
- Streams

- Mutation and boxes
  - `set!` vs. `set-box!`
  - `unbox`

Practice questions

What is the run time of `(range1 n)` given the following definition?

```
(define (range1 n)
  (cond [(zero? n) empty]
        [else (append (range1 (sub1 n)) (list n))]))
```

A. $O(\log n)$

B. $O(n)$

C. $O(n \log n)$

D. $O(n^2)$

E. $O(2^n)$

What is the run time of `(range2 n)` given the following definition?

```
(define (range2 n)
  (letrec ([f (λ (m)
                (cond [(= m n) empty]
                      [else (cons m (f (add1 m)))]))])
    (f 0)))
```

A. O(log n)

B. O(n)

C. O(n log n)

D. O(n^2)

E. O(2^n)

Practice problems

Implement (range n) using accumulator-passing style

Implement (range n) using continuation-passing style

When you implemented MiniScheme, why was `(if …)` implemented as a special case in the parser and interpreter rather than implemented as a primitive procedure?

What fails if you try to implement it as a primitive procedure?

A. Got it!

Imagine you had implemented support for macros in MiniScheme, could `(if …)` be implemented as a built-in macro (similar to a primitive procedure except it's a macro rather than a procedure) rather than as a special case in the parser/interpreter?

Assume macros in MiniScheme would work similarly to macros in Racket where patterns are matched against expressions and the output of the macro is valid MiniScheme code.

A. Yes, it would be easy

B. Yes, it would be difficult

C. No, MiniScheme could never support macros

D. No, some conditional is needed

MiniScheme (and most programming languages including Racket) is a "strict" language. This means arguments to called functions must be evaluated before the body of the function is executed.

In contrast, a "lazy" language defers evaluation of arguments until the arguments are used.

If we made MiniScheme a lazy language by deferring evaluation of expressions until evaluating primitive procedures, could (if …) be implemented as a primitive procedure?

A. Yes. When evaluating `(if then-expr else-expr)`, only one of `then-expr` or `else-expr` would ever need to be evaluated

B. No. Like with macros, MiniScheme would still need a special case for conditionals