CMPT 383

Haskell Lists

Anders Miltner
This Lecture

- Describe how to construct Haskell lists
- Describe Haskell lists’ memory model
- Describe functions over Haskell lists
- Haskell lists + pattern matching
Empty List

- The empty list is written `[]`, and pronounced “nil”

- Has type `[a]`, meaning it is a list with elements of any type
  - But the empty list can be specialized to work with concrete types
  - Lists are polymorphic

- Lists can have type `[t]`, where
  - `t` is either a variable (like `[a]`)
  - `t` is some concrete type (ie Int)
Lists with elements

You can prepend single elements to a list using the “cons” operator, :

• “cons” is short for “construct”

• Notice the type of cons uses a type variable, so cons can work over Ints, Bools, etc

• But the type of the list and the type of the element must be the same, or the types must be able to unify

• For now, you can just think of it like normal generics with fewer annotations
What do lists look like in memory?

example :: [Int]
example = 2:3:5:[[]] -- 2:(3:(5:[]))

- Haskell lists are linked lists
- Can manipulate linked lists nondestructively
- Instance of a more general pattern we will go into later this semester
Example: Cons2

```haskell
cons2 :: a -> [a] -> [a]
cons2 h t = h:h:t

x :: [Int]
x = 0:[]

y :: [Int]
y = cons2 2
```

![Diagram of list structure](image-url)
How do we extract elements of a list?

• Pattern matching of course!

**Extract Head**

\[ \text{hd} :: [a] \to a \to a \]

\[ \text{hd} \; [] \; x \; = \; x \]

\[ \text{hd} \; (h:_:\;) \; \_ \; = \; h \]

**Extract Tail**

\[ \text{tl} :: [a] \to [a] \]

\[ \text{tl} \; [] \; = \; [] \]

\[ \text{tl} \; (_:t) \; = \; t \]

**Extract nth Element**

\[ \text{nth} :: [a] \to a \to \text{Int} \to a \]

\[ \text{nth} \; [] \; x \; _ \; = \; x \]

\[ \text{nth} \; (h:_:\;) \; \_ \; 0 \; = \; h \]

\[ \text{nth} \; (h:t) \; v \; i \; = \; \text{nth} \; t \; v \; (i-1) \]
Some Haskell subtleties!

- Haskell functions are TOTAL functions
- Output must be defined for all inputs
  - You can throw errors, but it is bad style. Don’t do it.
- Hence the inclusion of default values!
But what about removing?

How can we remove elements from a list while being non-destructive
Think, what does non-destructive mean?

• If you put a pointer somewhere, you never remove it
• If you put a pointer somewhere, you never move it

• We want the function:
  remove_at :: [a] -> Int -> [a]
remove_at function

remove_at :: [a] -> Int -> [a]
remove_at [] _ = []
remove_at (h:t) 0 = t
remove_at (h:t) i = h:(remove_at t (i-1))
Fundamental Issues with FP

• Doubly-linked lists are not possible in pure FP
• Arrays are not possible in pure FP
• No O(1) access to list/array indices
  • No expected O(1) dictionary lookups
• And more…
• This is why I personally don’t use Haskell, but it’s still great for learning FP
update at :: [a] -> Int -> a -> [a]
update_at [] _ _ = []
update_at (h:t) 0 v = v:t
update_at (h:t) i v = h:(update_at t (i-1) v)
List append

- Would like to be able to combine lists, rather than just prepend elements to list

- \([1,2] \text{ ++ } [3,4] \text{ ++ } [5] = [1,2,3,4,5]\)

- \((++): [a] \rightarrow [a] \rightarrow [a]\)

\[
\begin{align*}
(++) & : [a] \rightarrow [a] \rightarrow [a] \\
(++) & \quad \text{[]} \quad y = y \\
(++) & \quad (xh, xt) \quad y = xh: (xt ++ y)
\end{align*}
\]
List reverse

• reverse \([1,2,3,4,5]\) = \([1,2,3,4,5]\)

• reverse :: \([a]\) \rightarrow \([a]\)

• (++) :: \([a]\) \rightarrow \([a]\) \rightarrow \([a]\)

```haskell
reverse :: [a] -> [a]
reverse [] = []
reverse h:t = (reverse t) ++ [h]
```
Correct, but inefficient

• Call append every time

• $O(n^2)$

```
reverse :: [a] -> [a]
reverse [] = []
reverse h:t = (reverse t) ++ [h]
```

```
(++): [a] -> [a] -> [a]
(++)[] y = y
(++)(xh,xt)y = xh:(xt ++ y)
```
Faster Solution: Tail Recursion

• In tail recursion, you use an “accumulator”

• Accumulate the result, then return it when finished with the list

• Try building using a helper `reverse' :: [a] -> [a] -> [a]`

```haskell
reverse :: [a] -> [a]
reverse l = reverse' l []

where reverse' [] acc = acc
    reverse' h:t acc = reverse' t (h:acc)
```
Aside: Tail Call Optimization

• One negative thing about recursion is that it pops the call stack

• Does that need to be the case?

• If recursion does no computation after recursive call, what is the stack useful for

• Optimizing Haskell compiler (-O2 and above) will remove call stack

• So does gcc!

reverse :: [a] -> [a]
reverse l = reverse' l []
where reverse' [] acc = acc
      reverse' h:t acc = reverse' t (h:acc)
List Syntactic Sugar

• Can be annoying to write a:b:c:d:[]
  • \([a,b,c,d] = a:b:c:d:[]\)

• Can be annoying to write [1,2,3,4,5]
  • \([1..5] = [1,2,3,4,5]\)

• Can be annoying to write [0,2,4,6,8]
  • \([0,2..8] = [0,2,4,6,8]\)

• Also \([x..]\) gives infinitary list starting from x

• \([x,y..]\) give infinitary list starting from x, with increments y-x
Strings!

• Strings are just lists of characters
  
  • String = [Char]

• So, operations on strings are just list operations
  
  • String concat: (++)

• One additional syntactic sugar:
  
  • “abc” = ['a', 'b', 'c'] = ‘a’:'b':'c':[]
List Comprehensions

- Originally inspired from math set notation: \{ x \mid x \text{ in Nat}, x \% 2 == 0 \}
- Also used in Python
- Actually syntactic sugar
  - What for? The list monad! — see the desugaring later in the semester!
Generators

- Contain the source values for the comprehension
  - $x \leftarrow [1,2,3,4]$
  - $y \leftarrow [5,6,7,8]$
  - $z \leftarrow ['a'..'z']$
Generator + Expression = Comprehension

• The generator binds elements of the list to variables
• The expression shows how to use the elements of the list
• What do you think the following expression evaluates to?
  • \([ x^2+1 \mid x \leftarrow [1,2,3,4,5] \] )
  • \([2,5,10,17,26]\)
Multi-Generator Comprehensions

• You can build comprehensions from more than one generator

• Corresponds to the “cartesian product” of the two lists

• What do you think the following expression evaluates to?
  
  • \[ [\ 10x + y \mid x \leftarrow [1,2,3], y \leftarrow [1,2,3] \ ] \]
  
  • \[ [11,12,13,21,22,23,31,32,33] \]
Guards

- You can filter down to some subset of the elements
  - \[ 10x + y \mid x \leftarrow [1,2,3], \ y \leftarrow [1,2,3], \ x \ % \ 2 = 0 \]
  - \([21,22,23]\)
Using Comprehensions to Flex on Imperative Languages

```
public void quickSort(int [] arr, int begin, int end) {
    if (begin < end) {
        int partitionIndex = partition(arr, begin, end);
        quickSort(arr, begin, partitionIndex-1);
        quickSort(arr, partitionIndex+1, end);
    }
}
```

```java
private int partition(int [] arr[], int begin, int end) {
    int pivot = arr[end];
    int i = (begin-1);
    for (int j = begin; j < end; j++) {
        if (arr[j] <= pivot) {
            i++;
            int swapTemp = arr[i];
            arr[i] = arr[j];
            arr[j] = swapTemp;
        }
    }
    int swapTemp = arr[i+1];
    arr[i+1] = arr[end];
    arr[end] = swapTemp;
    return i+1;
}
```

```
qs :: [a] -> [a]
qs (x:xs) = smaller ++ [x] ++ larger
    where smaller = qs [a | a <- xs, a <= x]
          larger = qs [a | a <- xs, a > x ]
```