**Advanced Programming**

**Handout 11**

**Programming With Streams**
(SOE Chapter 14)

---

### Streams
- A stream is an infinite sequence of values.
- We could define a special data type for them:
  ```haskell
data Stream a = a :^ Stream a
```
  but in practice it's easier to use conventional lists, ignoring `[]`, so that we can reuse the many operations on lists.
- Streams are often defined recursively, such as:
  ```haskell
twos = 2 : twos
```
- By calculation:
  ```haskell
twos \rightarrow 2 : twos \rightarrow 2 : 2 : twos \rightarrow ... \n```
- This calculation does not terminate – yet it is not the same as `_|_`, in that it yields useful information.
- [Another example: `numsfrom n = n : numsfrom (n+1)`]

---

### Lazy Evaluation
- Two ways to calculate "head twos":
  ```haskell
head twos \rightarrow head (2 : twos) \rightarrow head (2 : 2 : twos) \rightarrow head (2 : 2 : 2 : twos) \rightarrow ...
```
- One strategy terminates, the other doesn't.
- Normal order calculation guarantees finding a terminating sequence if one exists.
- Normal order calculation: always choose the outermost calculation (e.g., unfolding `head` above instead of unfolding `twos`).
- Also called lazy evaluation, or non-strict evaluation.
- (In contrast to eager or strict evaluation.)

---

### Example: Fibonacci Sequence
- Well-known sequence:
  ```plaintext
  1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...
  ``
- Here is a Haskell program that mimics the mathematical definition:
  ```haskell
  fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```
- Unfortunately, this program is terribly inefficient (perform the calculation to see this). Indeed, it has an exponential blow-up.
- Perhaps surprisingly, it is more efficient to create the infinite stream of Fibonacci numbers first, then select to the one we need.

---

### Fibs, cont’d
- Note this relationship:
  ```plaintext
  fibs 1   2   3   5   8   13   21   34
+ tail fibs 2   3   5   8   13   21   34   55
  ```
- This is easily transcribed into Haskell:
  ```haskell
  fibs 1   2   3   5   8   13   21   34
+ tail fibs 2   3   5   8   13   21   34   55
  ```
- This is easily transcribed into Haskell:
  ```haskell
  fibs = 1 : add fibs (tail fibs)
  tail = fibs !! n
  ```
  where add = zipWith (+)
- And then finally:
  ```haskell
  fib n = fibs !! n
  ```

---

### Chasing One’s Tail
- Notice in:
  ```haskell
  fibs 1 : add fibs (tail fibs)
  ```
  that "tail fibs" starts right here.
- Introduce a name for that value so it can be shared:
  ```haskell
  fibs 1 : if where if1 fibs 1 : add fibs (tail fibs)
  ```
  where add = zipWith (+)
- Doing this again for the tail of the tail yields:
  ```haskell
  1 : if where if1 if2 (1 : add if2)
  ```
- Finally, unfold add:
  ```haskell
  1 : if where if1 if2 where if2 = 1 : add if2
  ```
Garbage Collection

- Because of sharing, exponential blowup is avoided.
- In a few more steps we have:
  \[ \text{fibs} \Rightarrow 1 : \text{if} \]
  
  \[ \text{where } \text{if} = 1 : \text{if1} \]
  
  \[ \text{where } \text{if1} = 2 : \text{if2} \]
  
  \[ \text{where } \text{if2} = 3 : \text{if3} \]
  
  \[ \text{where } \text{if3} = 3 : \text{add if2 if3} \]

- Now note that "if" is only used in one place, and thus might as well be eliminated, yielding:
  \[ \Rightarrow 1 : 1 : \text{if} \]
  
  \[ \text{where } \text{if} = 2 : \text{if1} \]
  
  \[ \text{where } \text{if1} = 3 : \text{add if2 if3} \]

- Think of this as "garbage collection" of names.

Stream Diagrams

- An alternative (perhaps better) way to depict sharing is graphically using a stream diagram.
- Another example: client-server interactions.