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:
  
  ```text
twos \rightarrow 2 : twos \rightarrow 2 : 2 : twos \rightarrow 2 : 2 : 2 : twos \rightarrow \ldots
```

- 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”:
  - head twos
  - head (2 : twos)
  - 2
  - head twos
  - head (2 : twos)
  - head (2 : 2 : twos)
  - head (2 : 2 : 2 : twos)
  - ...

- 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:
  1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

- Here is a Haskell program that mimics the mathematical definition:
  \[
  \begin{align*}
  \text{fib 0} & \quad = 1 \\
  \text{fib 1} & \quad = 1 \\
  \text{fib n} & \quad = \text{fib (n-1)} + \text{fib (n-2)}
  \end{align*}
  \]

- Unfortunately, this program is \textit{terribly inefficient} (perform the calculation to see this). Indeed, it has an exponential blow-up.

- Perhaps surprisingly, it is more efficient to create the \textit{infinite stream} of Fibonacci numbers first, then select to the one we need.
Fibs, cont’d

- Note this relationship:

  \[
  \begin{array}{cccccccc}
  \text{fibs} & 1 & 1 & 2 & 3 & 5 & 8 & 13 & 21 & 34 \\
  + \quad \text{tail fibs} & 1 & 2 & 3 & 5 & 8 & 13 & 21 & 34 & 55 \\
  \hline
  \text{tail (tail fibs)} & 2 & 3 & 5 & 8 & 13 & 21 & 34 & 55 & 89
  \end{array}
  \]

- This is easily transcribed into Haskell:

  \[
  \text{fibs} = 1 : 1 : \text{add fibs (tail fibs)}
  \]

  where \(\text{add} = \text{zipWith (+)}\)

- And then finally:

  \[
  \text{fib n} = \text{fibs} !! n
  \]
Chasing One’s Tail

- Notice in:
  
  \[
  \text{fibs } \rightarrow 1 : 1 : \text{add fibs (tail fibs)}
  \]
  
  that “tail fibs” starts right here.

- Introduce a \textit{name} for that value so it can be \textit{shared}:
  
  \[
  \text{fibs } \rightarrow 1 : \text{tf where tf} = 1 : \text{add fibs (tail fibs)}
  \rightarrow 1 : \text{tf where tf} = 1 : \text{add fibs tf}
  \]

- Doing this again for the tail of the tail yields:
  
  \[
  \rightarrow 1 : \text{tf where tf} = 1 : \text{tf2 where tf2} = \text{add fibs tf}
  \]

- Finally, unfold \textit{add}:
  
  \[
  \rightarrow 1 : \text{tf where tf} = 1 : \text{tf2 where tf2} = 2 : \text{add tf tf2}
  \]
Garbage Collection

- Because of sharing, exponential blowup is avoided.
- In a few more steps we have:
  \[
  \text{fibs} \Rightarrow 1 : \text{tf} \\
  \quad \text{where } \text{tf} = 1 : \text{tf2} \\
  \quad \quad \text{where } \text{tf2} = 2 : \text{tf3} \\
  \quad \quad \quad \text{where } \text{tf3} = 3 : \text{add tf2 tf3}
  \]
- Now note that “tf” is only used in one place, and thus might as well be eliminated, yielding:
  \[
  \Rightarrow 1 : 1 : \text{tf2} \\
  \quad \text{where } \text{tf2} = 2 : \text{tf3} \\
  \quad \quad \text{where } \text{tf3} = 3 : \text{add tf2 tf3}
  \]
- 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*.