Imperative vs. Declarative (1)

- **Imperative Languages**:
  - Implicit state.
  - Computation essentially a sequence of side-effecting actions.
  - Examples: Procedural and OO languages

- **Declarative Languages** (Lloyd 1994):
  - No implicit state.
  - A program can be regarded as a theory.
  - Computation can be seen as deduction from this theory.
  - Examples: Logic and Functional Languages.

Imperative vs. Declarative (2)

Another perspective:

- **Algorithm = Logic + Control**
  - Declarative programming emphasises the logic (“what”) rather than the control (“how”).
  - Strategy needed for providing the “how”:
    - Resolution (logic programming languages)
    - Lazy evaluation (some functional and logic programming languages)
    - (Lazy) narrowing: (functional logic programming languages)

No Control?

Declarative languages for practical use tend to be only **weakly declarative**, i.e., not totally free of control aspects. For example:

- Equations in functional languages are directed.
- Order of patterns often matters for pattern matching.
- Constructs for taking control over the order of evaluation. (E.g. **cut** in Prolog, **seq** in Haskell.)
Relinquishing Control

Theme of this lecture: *relinquishing control by exploiting lazy evaluation*.

- Evaluation orders
- Strict vs. Non-strict semantics
- Lazy evaluation
- Applications of lazy evaluation:
  - Programming with infinite structures
  - Circular programming
  - Dynamic programming
  - Attribute grammars

Evaluation Orders (1)

Consider:

\[
\begin{align*}
sqr \ x &= x \times x \\
dbl \ x &= x + x \\
main &= \text{sqr (dbl (2 + 3))}
\end{align*}
\]

Roughly, any expression that can be evaluated or reduced by using the equations as rewrite rules is called a *reducible expression* or *redex*.

Assuming arithmetic, the redexes of the body of main are: 2 + 3
dbl (2 + 3)
sqr (dbl (2 + 3))

Evaluation Orders (2)

Thus, in general, many possible reduction orders. Innermost, leftmost redex first is called *Applicative Order Reduction* (AOR). Recall:

\[
\begin{align*}
sqr \ x &= x \times x \\
dbl \ x &= x + x \\
main &= \text{sqr (dbl (2 + 3))}
\end{align*}
\]

Starting from main:

\[
\begin{align*}
\text{main} & \Rightarrow \text{sqr (dbl (2 + 3))} \\
& \Rightarrow \text{sqr (dbl 5)} \\
& \Rightarrow \text{sqr (5 + 5)} \\
& \Rightarrow \text{sqr 10} \\
& \Rightarrow 10 \times 10 \\
& \Rightarrow 100
\end{align*}
\]

This is just *Call-By-Value*.

Evaluation Orders (3)

Outermost, leftmost redex first is called *Normal Order Reduction* (NOR):

\[
\begin{align*}
\text{main} & \Rightarrow \text{sqr (dbl (2 + 3))} \\
& \Rightarrow \text{dbl (2 + 3)} \times \text{dbl (2 + 3)} \\
& \Rightarrow ((2 + 3) \times (2 + 3)) \times \text{dbl (2 + 3)} \\
& \Rightarrow (5 + (2 + 3)) \times \text{dbl (2 + 3)} \\
& \Rightarrow (5 + 5) \times \text{dbl (2 + 3)} \\
& \Rightarrow ... \\
& \Rightarrow 10 \times 10 \\
& \Rightarrow 100
\end{align*}
\]

(Applications of arithmetic operations only considered redexes once arguments are numbers.) Demand-driven evaluation or *Call-By-Need*
Why Normal Order Reduction? (1)

NOR seems rather inefficient. Any use?
- Best possible termination properties.
  A pure functional language is just the \( \lambda \)-calculus in disguise. Two central theorems:
  - Church-Rosser Theorem I: No term has more than one normal form.
  - Church-Rosser Theorem II: If a term has a normal form, then NOR will find it.

Why Normal Order Reduction? (2)

- More expressive power; e.g.:
  - “Infinite” data structures
  - Circular programming
- More declarative code as control aspects (order of evaluation) left implicit.

Exercise 1

Consider:

\[
\begin{align*}
  f & \ x = 1 \\
  g & \ x = g \ x \\
  main & = f \ (g \ 0)
\end{align*}
\]

Attempt to evaluate `main` using both AOR and NOR. Which order is the more efficient in this case? (Count the number of reduction steps to normal form.)

Strict vs. Non-strict Semantics (1)

- `⊥`, or “bottom”, the undefined value, representing errors and non-termination.
- A function \( f \) is strict iff:
  \[
  f \ ⊥ = ⊥
  \]

For example, + is strict in both its arguments:

\[
\begin{align*}
  (0/0) + 1 & = ⊥ + 1 = ⊥ \\
  1 + (0/0) & = 1 + ⊥ = ⊥
\end{align*}
\]
Strict vs. Non-strict Semantics (2)

Again, consider:
\[ f \ x = 1 \]
\[ g \ x = g \ x \]
What is the value of \( f \ (0/0) \)? Or of \( f \ (g \ 0) \)?
- **AOR:** \( f \ (0/0) \Rightarrow \bot \); \( f \ (g \ 0) \Rightarrow \bot \)
  Conceptually, \( f \bot = \bot \); i.e., \( f \) is strict.
- **NOR:** \( f \ (0/0) \Rightarrow 1 \); \( f \ (g \ 0) \Rightarrow 1 \)
  Conceptually, \( f\circ \bot = 1 \); i.e., \( f\circ \) is non-strict.
Thus, NOR results in non-strict semantics.

Lazy Evaluation (1)

Lazy evaluation is a **technique for implementing NOR** more efficiently:
- A redex is evaluated **only if needed**.
- **Sharing** employed to avoid duplicating redexes.
- Once evaluated, a redex is **updated** with the result to avoid evaluating it more than once.

As a result, under lazy evaluation, any one redex is evaluated at most once.

Lazy Evaluation (2)

Recall:
\[ \text{sqr} \ x = x \times x \]
\[ \text{dbl} \ x = x + x \]
\[ \text{main} = \text{sqr} \ (\text{dbl} \ (2+3)) \]

Evaluate \text{main} using AOR, NOR, and lazy evaluation:
\[ f \ x \ y \ z = x \times z \]
\[ g \ x = f \ (x \times x) \ (x \times 2) \ x \]
\[ \text{main} = g \ (1 + 2) \]
(Only consider an applications of an arithmetic operator a redex once the arguments are numbers.)

How many reduction steps in each case?

**Answer:** 7, 8, 6 respectively
**Infinite Data Structures (1)**

\[
take\ n\ \text{[]}\ =\ \text{[]} \\
 takeaway\ n\ \text{[]}\ =\ \text{[]} \\
 take\ n\ (x:\text{xs})\ =\ x\ :\ takeaway\ (n-1)\ \text{xs}
\]

\[
\text{from}\ n\ =\ n\ :\ \text{from}\ (n+1)
\]

\[
nats\ =\ \text{from}\ 0
\]

\[
\text{main}\ =\ takeaway\ 5\ nats
\]

**Infinite Data Structures (2)**

\[
\text{main} \Rightarrow^1 \text{take 5} (\bullet) \Rightarrow^4 0 : \text{take 4} (\bullet) \\
\Rightarrow^6 0:1: \text{take 3} (\bullet) \Rightarrow^8 \ldots \\
\Rightarrow 0:1:2:3:4: \text{take 0} (\bullet) \Rightarrow [0,1,2,3,4]
\]

\[
nats \Rightarrow^2 \text{from 0} \Rightarrow^3 0 : \text{from 1} \\
\Rightarrow^5 0:1: \text{from 2} \Rightarrow^7 \ldots \Rightarrow 0:1:2:3:4: \text{from 5}
\]

**Circular Data Structures (2)**

\[
take\ 0\ \text{xs}\ =\ \text{[]} \\
 take\ n\ \text{[]}\ =\ \text{[]} \\
 take\ n\ (x:\text{xs})\ =\ x\ :\ takeaway\ (n-1)\ \text{xs}
\]

\[
\text{ones}\ =\ 1\ :\ \text{ones}
\]

\[
\text{main}\ =\ \text{take 5 ones}
\]

**Circular Data Structures (2)**

\[
\text{main} \Rightarrow^1 \text{take 5} (\bullet) \Rightarrow^3 1 : \text{take 4} (\bullet) \\
\Rightarrow^4 1:1: \text{take 3} (\bullet) \Rightarrow^5 \ldots \\
\Rightarrow 1:1:1:1:1: \text{take 0} (\bullet) \Rightarrow [1,1,1,1,1]
\]

\[
\text{ones} \Rightarrow^2 1 : \bullet
\]
Exercise 3

Given the following tree type

```haskell
data Tree = Empty
           | Node Tree Int Tree
```

define:

- An infinite tree where every node is labelled by 1.
- An infinite tree where every node is labelled by its depth from the root node.

Exercise 3: Solution

```haskell
treeOnes = Node treeOnes 1 treeOnes

treeFrom n = Node (treeFrom (n + 1)) n (treeFrom (n + 1))

treeDepths = treeFrom 0
```

Circular Programming (1)

A non-empty tree type:

```haskell
data Tree = Leaf Int | Node Tree Tree
```

Suppose we would like to write a function that replaces each leaf integer in a given tree with the **smallest** integer in that tree.

How many passes over the tree are needed?

*One!*

Circular Programming (2)

Write a function that replaces all leaf integers by a given integer, and returns the new tree along with the smallest integer of the given tree:

```haskell
fmr :: Int -> Tree -> (Tree, Int)
fmr m (Leaf i) = (Leaf m, i)
fmr m (Node tl tr) = (Node tl' tr', min ml mr)
  where
    (tl', ml) = fmr m tl
    (tr', mr) = fmr m tr
```
**Circular Programming (3)**

For a given tree \( t \), the desired tree is now obtained as the *solution* to the equation:

\[(t', m) = \text{fmr } m \ t\]

Thus:

\[
\text{findMinReplace } t = t'
\]

where

\[(t', m) = \text{fmr } m \ t\]

Intuitively, this works because \( \text{fmr} \) can compute its result without needing to know the *value* of \( m \).

**A Simple Spreadsheet Evaluator**

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th></th>
<th>1</th>
<th>37</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>c3 + c2</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>14</td>
</tr>
<tr>
<td>2</td>
<td>a3 + b2</td>
<td>2a2 + b2</td>
<td></td>
<td></td>
<td>2</td>
<td>16</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>a2 + a3</td>
<td></td>
<td></td>
<td>3</td>
<td>21</td>
</tr>
</tbody>
</table>

\( r = \text{array} (\text{bounds } s) \)

\[
[ ((i,j), \text{eval } r (s!(i,j))) \\
| (i,j) <- \text{indices } s ]
\]

The evaluated sheet is again simply the *solution* to the stated equation. No need to worry about evaluation order. *Any caveats?*

**Breadth-first Numbering (1)**

Consider the problem of numbering a possibly infinitely deep tree in breadth-first order:

![Tree Diagram]

**Breadth-first Numbering (2)**

The following algorithm is due to G. Jones and J. Gibbons (1992), but the presentation differs.

Consider the following tree type:

\[
\text{data Tree } a = \text{Empty} \\
| \text{Node } (\text{Tree } a) \ a \ (\text{Tree } a)
\]

Define:

- \( \text{width } t \ i \) The width of a tree \( t \) at level \( i \) (0 origin).
- \( \text{label } t \ i \ j \) The \( j \)th label at level \( i \) of a tree \( t \) (0 origin).
Breadth-first Numbering (3)

The following system of equations defines breadth-first numbering:

\[
\begin{align*}
\text{label } t \ 0 \ 0 &= 1 \\
\text{label } t \ (i+1) \ 0 &= \text{label } t \ i \ 0 + \text{width } t \ i \\
\text{label } t \ i \ (j+1) &= \text{label } t \ i \ j + 1
\end{align*}
\] (1) (2) (3)

Note that label \( t \ i \ 0 \) is defined for all levels \( i \) (as long as the widths of all tree levels are finite).

Breadth-first Numbering (4)

The code that follows sets up the defining system of equations:

- **Streams** (infinite lists) of labels are used as a mediating data structure to allow equations to be set up between adjacent nodes within levels and between the last node at one level and the first node at the next.
- Idea: the tree numbering function for a subtree takes a stream of labels for the first node at each level, and returns a stream of labels for the node after the last node at each level.

Breadth-first Numbering (5)

- As there manifestly are no cyclic dependencies among the equations, we can entrust the details of solving them to the lazy evaluation machinery in the safe knowledge that a solution will be found.

Breadth-first Numbering (6)

\[
bfn :: \ Tree \ a \ -> \ Tree \ Integer
\]
\[
bfn t = t' \\
\ \\
\ \\
\text{where}
\]
\[
(n', t') = bfnAux (1 : ns) t
\]
\[
bfnAux :: [Integer] -> Tree a \\
\ \\
\ \\
\text{where}
\]
\[
(bnAux ns Empty = (ns, Empty)
\ \\
(bnAux (n : ns) (Node tl _ tr) = ((n + 1) : ns', Node tl' n tr')
\]
\[
\text{where}
\]
\[
(ns', tl') = bnAux ns tl
\ \\
(ns'', tr') = bnAux ns' tr
\]

Eqns (1) & (2)

Eqn (3)
**Dynamic Programming**

*Dynamic Programming:*

- Create a *table* of all subproblems that ever will have to be solved.
- Fill in table without regard to whether the solution to that particular subproblem will be needed.
- Combine solutions to form overall solution.

*Lazy Evaluation* is a perfect match as saves us from having to worry about finding a suitable evaluation order.

**The Triangulation Problem (1)**

Select a set of *chords* that divides a convex polygon into triangles such that:

- no two chords cross each other
- the sum of their length is minimal.

We will only consider computing the minimal length.

See Aho, Hopcroft, Ullman (1983) for details.
The Triangulation Problem (2)

Let $S_{is}$ denote the subproblem of size $s$ starting at vertex $v_i$ of finding the minimum triangulation of the polygon $v_i, v_{i+1}, \ldots, v_{i+s-1}$ (counting modulo the number of vertices).

Subproblems of size less than 4 are trivial.

Solving $S_{is}$ is done by solving $S_{i,k+1}$ and $S_{i+k,s-k}$ for all $k, 1 \leq k \leq s-2$.

The obvious recursive formulation results in $3^{s-4}$ (non-trivial) calls.

But for $n \geq 4$ vertices there are only $n(n-3)$ non-trivial subproblems!

The Triangulation Problem (3)

Let $C_{is}$ denote the minimal triangulation cost of $S_{is}$.

Let $D(v_p, v_q)$ denote the length of a chord between $v_p$ and $v_q$ (length is 0 for non-chords; i.e. adjacent $v_p$ and $v_q$).

For $s \geq 4$:

$$C_{is} = \min_{k \in [1,s-2]} \left\{ C_{i,k+1} + C_{i+k,s-k} + D(v_i, v_{i+k}) + D(v_{i+k}, v_{i+s-1}) \right\}$$

For $s < 4$, $S_{is} = 0$. 

The Triangulation Problem (4)

The Triangulation Problem (5)
The Triangulation Problem (6)

These equations can be transliterated straight into Haskell:

```haskell
triCost :: Polygon -> Double
triCost p = cost!(0,n) where
    cost = array ((0,0), (n-1,n))
        [(i,s),
            minimum [ cost!(i, k+1)
                + cost!((i+k) `mod` n, s-k)
                + dist p i ((i+k) `mod` n)
                + dist p ((i+k) `mod` n)
                | k <- [1..s-2] ]
            | i <- [0..n-1], s <- [4..n] ] ++
            [(i,s), 0.0]
            | i <- [0..n-1], s <- [0..3] ]

n = snd (bounds b) + 1
```

Attribute Grammars (1)

Lazy evaluation is also very useful for evaluation of Attribute Grammars:

- The attribution function is defined recursively over the tree:
  - takes inherited attributes as extra arguments;
  - returns a tuple of all synthesised attributes.

- As long as there exists some possible attribution order, lazy evaluation will take care of the attribute evaluation.

Attribute Grammars (2)

- The earlier examples on Circular Programming and Breadth-first Numbering can be seen as instances of this idea.

Reading

Reading
