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:

```
sqr x = x * x
dbl x = x + x
main = sqr (dobl (2 + 3))
```

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`
- `dobl (2 + 3)`
- `sqr (dobl (2 + 3))`

Evaluation Orders (2)

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

- `sqr x = x * x`
- `dobl x = x + x`
- `main = sqr (dobl (2 + 3))`

Starting from `main`:

```
main ⇒ sqr (dobl (2 + 3)) ⇒ sqr (dobl 5)
⇒ sqr (5 + 5) ⇒ sqr 10 ⇒ 10 * 10 ⇒ 100
```

**Call-By-Value (CBV)** = AOR except no evaluation under λ (inside function bodies).

Evaluation Orders (3)

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

```
main ⇒ sqr (dobl (2 + 3))
⇒ dbl (2 + 3) * dbl (2 + 3)
⇒ ((2 + 3) + (2 + 3)) * dbl (2 + 3)
⇒ (5 + (2 + 3)) * dbl (2 + 3)
⇒ (5 + 5) * dbl (2 + 3) ⇒ 10 * dbl (2 + 3)
⇒ ... ⇒ 10 * 10 ⇒ 100
```

(Application of arithmetic operations only considered redexes once arguments are numbers.)

**Call-By-Name (CBN)** = NOR except no evaluation under λ.

Why NOR or CBN? (1)

NOR and CBN seem rather inefficient. Any use?

- Best possible termination properties.

A pure functional languages is just the λ-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 it can be found through NOR.
Why NOR or CBN? (2)

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

Why NOR or CBN? (3)

- More reuse. E.g. consider:
  - any :: (a -> Bool) -> [a] -> Bool
  - any p = or . map p
  Under AOR/CBV, we would have to inline all functions to avoid doing too much work:
  - any :: (a -> Bool) -> [a] -> Bool
  - any p [] = False
  - any p (y:ys) = y || any p ps
  (Assume (||) has “short-circuit” semantics.)
  No reuse.
  (See references for in-depth discussion.)

Strict vs. Non-strict Semantics (1)

- ⊥, or “bottom”, the undefined value, representing errors and non-termination.
- A function f is strict iff:
  \[ f \bot = \bot \]
  For example, + is strict in both its arguments:
  \[
  (0/0) + 1 = \bot + 1 = \bot \\
  1 + (0/0) = 1 + \bot = \bot 
  \]

Lazy Evaluation (2)

Recall:

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

\[
\Rightarrow \begin{align*}
\left((2+3) \times 2\right) \times 2 \\
\Rightarrow 10 \\
\Rightarrow 100
\end{align*}
\]

Exercise 2

Evaluate main using AOR, NOR, and lazy evaluation:

\[
\begin{align*}
\text{f} \ x \ y \ z &= x \times z \\
\text{g} \ x &= \text{f} \ (\text{x} \times \text{x}) \ (\text{x} \times 2) \\
\text{main} &= \text{g} \ (1 + 2)
\end{align*}
\]

(Only consider an application of an arithmetic operator a redex once the arguments are numbers.)

How many reduction steps in each case?

Answer: 7, 8, 6 respectively

Lazy Evaluation (1)

Lazy evaluation or Call-by-Need is a technique for implementing CBN 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.

Infinite Data Structures (1)

\[
\begin{align*}
\text{take} \ 0 \ xs &= [] \\
\text{take} \ n \ [] &= [] \\
\text{take} \ n \ (x:xs) &= x : \text{take} \ (n-1) \ xs
\end{align*}
\]

\[
\begin{align*}
\text{from} \ n &= n : \text{from} \ (n+1) \\
\text{nats} &= \text{from} \ 0 \\
\text{main} &= \text{take} \ 5 \ \text{nats}
\end{align*}
\]

Exercise 1

Consider:

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

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.)
Exercise 3

Given the following tree type

\[ \text{data Tree} = \text{Empty} \mid \text{Node Tree Int Tree} \]

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

\[ \text{treeOnes} = \text{Node treeOnes} 1 \text{ treeOnes} \]

\[ \text{treeFrom} n = \text{Node} \left( \text{treeFrom} \left( n + 1 \right) \right) \]

\[ \text{treeDepths} = \text{treeFrom} 0 \]

Circular Programming (1)

A type of non-empty trees:

\[ \text{data Tree} = \text{Leaf Int} \mid \text{Node Tree Tree} \]

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:

\[ \text{fmr} :: \text{Int} \to \text{Tree} \to \left( \text{Tree}, \text{Int} \right) \]

\[ \text{fmr} m \left( \text{Leaf} i \right) = \left( \text{Leaf} m, i \right) \]

\[ \text{fmr} m \left( \text{Node} \text{tl} \text{ tr} \right) = \left( \text{Node} \text{tl'} \text{ tr'}, \text{min} ml mr \right) \]

where

\[ \left( \text{tl'}, ml \right) = \text{fmr} m \text{ tl} \]

\[ \left( \text{tr'}, mr \right) = \text{fmr} m \text{ tr} \]

Circular Programming (3)

For a given tree, the desired tree is now obtained as the solution to the equation:

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

Thus:

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

where

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

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

Circular Programming (4)

Operational view:

\[ \text{fmr \ (snd \ } \_ \text{ \ )} = \left( \text{min} \ (\text{min} 3 \ 1) \ 2 \right) \]
A Simple Spreadsheet Evaluator

\[
\begin{array}{ccc}
  1 & c3 + c2 & 1 \\
  2 & a3 + b2 & 14 \\
  3 & 7 & 7 \\
\end{array}
\]

\[
\begin{array}{ccc}
  1 & 37 \\
  2 & 16 \\
  3 & 21 \\
\end{array}
\]

\[r = \text{array (bounds } s)\]
\[| \{(i,j) \mapsto \text{eval } r(s!(i,j))\}\]

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:

\[
\begin{array}{cccccccc}
  & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
 1 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \\
\end{array}
\]

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:

```haskell
data Tree a = Empty
    | Node (Tree a) a (Tree a)
```

Define:

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

Breadth-first Numbering (3)

The following system of equations defines breadth-first numbering:

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

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.
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 $S_s$ 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 $S_s$ 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 $S_s$ 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 $S_s$ 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 (4)

Let $C_s$ denote the minimal triangulation cost of $S_s$.

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 $s \geq 4$:
  $$C_s = \min_{k \in [1, s-2]} \left\{ C_{i,k+1} + C_{i+k,s-k} + D(v_i, v_{i+k}) + D(v_{i+k}, v_{i+s-1}) \right\}$$
  - For $s < 4$, $S_s = 0$.

The Triangulation Problem (5)

These equations can be transliterated straight into 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 <- [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 (1)


Reading (2)