Functional v.s Imperative Programs

• Imperative language
  – Basic constructs are *imperative statements*
    ◊ Change existing values, i.e., the “state” of a program
    ◊ e.g., \( x := x + 1 \)
  – Examples: C, Java, Fortran, Modula, Ada, Pascal, Algol, ...

• Functional language
  – Basic constructs are declarative
    ◊ Declare new values
    ◊ e.g., function \( f(int x) \) \{ return x+1 \}
  – Computation proceeds primarily by evaluating expressions
  – “*Pure*” if all constructs are strictly declarative, i.e., *No side-effects*
  – Examples: Lisp/Scheme, ML, Miranda, Haskell, ...
No side-effects languages test

Within the scope of specific declarations of $x_1, x_2, \ldots, x_n$, all occurrences of a given expression containing only $x_1, x_2, \ldots, x_n$ necessarily produce the same value.

Example

```c
int x = 3;
int y = 4;
int z = 5*(x+y) - 3;
... // no new declaration of x or y //
integer w = 4*(x+y) + z;
```
Turing lecture highlights functional programming

• Designer of Fortran, BNF, etc.
• Turing Award in 1977
  – Functional programming better than imperative programming
  – Easier to reason about functional programs
  – More efficient due to parallelism
  – Algebraic laws
    ◊ Reason about programs
    ◊ Optimizing compilers
Reasoning about programs

• To prove a program correct, must consider everything a program depends on

• In functional programs, dependence on any data structure is explicit

• Therefore, easier to reason about functional programs

• This is an argument on principles
  – This thesis must be tested in practice
  – Many who prove properties of programs believe this
  – Not many people really prove their code correct
Proving properties

Functional style supports reasoning

fun power \( (b, n) \) = if ( \( n = 0 \) ) then 1 else power\( (b, n-1) \) * b

Claim: For any integer \( n \geq 0 \) and any number \( b \),
\[
\text{power}(b, n) = b^n.
\]

How would you prove this???
Proving properties

Functional style supports reasoning

\[
\text{fun power (b, n) = if ( n = 0) then 1 else power(b, n-1) * b}
\]
Proving properties

Functional style supports reasoning

\[
\text{fun power (b, n) = if (n = 0) then 1 else power(b, n-1) * b}
\]

Compare to imperative style

```c
int power(int b, int n) {
    int result = 1;
    for (int i = 0; i < n; i++) {
        result *= b;
    }
    return result;
}
```

Claim: \(\text{power(b, n) = } b^n\)

How would you prove this claim for this program??
Proving properties

Functional style supports reasoning

$$\text{fun power}(b, n) = \text{if } (n = 0) \text{ then } 1 \text{ else power}(b, n - 1) \times b$$

Compare to imperative style

```java
int power(int b, int n) {
    int result = 1;
    for (int i = 0; i < n; i++) {
        result *= b;
    }
    return result;
}
```

Devise a loop invariant:

$$(n \geq i) \land (result = b^i)$$

◊ Prove that it is true at the start of the first loop iteration

◊ Prove that each loop iteration preserves it
Proving properties

Functional style supports reasoning

\[
\text{fun power (b, n) = if ( n = 0) then 1 else power(b, n \!\! \!\! \!\!\! \!\! \!\! \!\!\! - 1) * b}
\]

Compare to imperative style

\[
\text{int power(int b, int n) }
\begin{array}{c}
\text{int result = 1;}
\text{for (int i = 0; i < n; i++) { }
\text{result *= b;}
\text{}}
\text{return result;}
\end{array}
\]

At the start of the first loop iteration:

\[(result = 1) \land (i=0) \land (i<n)\]

So \[(n \geq i) \land (result = b^i)\]
Functional style supports reasoning

fun power (b, n) = if ( n = 0) then 1 else power(b, n-1) * b

Compare to imperative style

int power(int b, int n) {
    int result = 1;
    for (int i = 0; i < n; i++) {
        result *= b;
    }
    return result;
}

Loop preserves the invariant:

◊ Assume \((n \geq i) \land (result = b^i)\) here

◊ Prove \((n \geq i) \land (result = b^i)\) here
  (after incrementing \(i\))
Proving properties

Functional style supports reasoning:

\[
\text{fun power (b, n) = if ( n = 0) then 1 else power(b, n-1) * b}
\]

Compare to imperative style:

\[
\text{int power(int b, int n) {}
    int result = 1;
    for (int i = 0; i < n; i++) {
        result *= b;
    }
    return result;
}
\]

On exit from the loop we know:

◊ Loop invariant \((n \geq i) \land (result = b^i)\)
◊ We also know \((i \leq n)\)
◊ These imply \(result = b^n\)
Reasoning about programs

• Haskell quicksort

\[ qsort \; [] = [] \]
\[ qsort \; (x:xs) = (qsort \; \text{elts}_\lt\; x)++ [x] \]
\[ + + (qsort \; \text{elts}_\geq\; x) \]

where
\[ \text{elts}_\lt\; x = [y \mid y \leftarrow xs, y < x] \]
\[ \text{elts}_\geq\; x = [y \mid y \leftarrow xs, y \geq x] \]

• This is all there is ...
  – No assignment – just write expression for sorted list
  – No array indices, no pointers, no memory management, ...

Reasoning about programs

• Compare to C++

qsort( int a[], int lo, int hi )
{ int h, l, p, t;
  if (lo < hi) {
    l = lo; h = hi; p = a[hi];
    do {
      while ((l < h) && (a[l] <= p)) l = l+1;
      while ((h > l) && (a[h] >= p)) h = h-1;
      if (l < h) { t = a[l]; a[l] = a[h]; a[h] = t; }
    } while (l < h);
    t = a[l]; a[l] = a[hi]; a[hi] = t;
    qsort( a, lo, l−1 );
    qsort( a, l+1, hi );
  }
}
Interesting case study

Naval Surface Warfare Center experiment

- Different teams prototyped a (simplified) “geometric region server”
- Results were reviewed by a committee chosen by the Navy

<table>
<thead>
<tr>
<th>Language</th>
<th>Lines of code</th>
<th>Lines of documentation</th>
<th>Development time (hours)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) Haskell</td>
<td>85</td>
<td>465</td>
<td>10</td>
</tr>
<tr>
<td>(2) Ada</td>
<td>767</td>
<td>714</td>
<td>23</td>
</tr>
<tr>
<td>(3) Ada9X</td>
<td>800</td>
<td>200</td>
<td>28</td>
</tr>
<tr>
<td>(4) C++</td>
<td>1105</td>
<td>130</td>
<td>-</td>
</tr>
<tr>
<td>(5) Awk/Nawk</td>
<td>250</td>
<td>150</td>
<td>-</td>
</tr>
<tr>
<td>(6) Rapide</td>
<td>157</td>
<td>0</td>
<td>54</td>
</tr>
<tr>
<td>(7) Griffin</td>
<td>251</td>
<td>0</td>
<td>34</td>
</tr>
<tr>
<td>(8) Proteus</td>
<td>293</td>
<td>79</td>
<td>26</td>
</tr>
<tr>
<td>(9) Relational Lisp</td>
<td>274</td>
<td>12</td>
<td>3</td>
</tr>
<tr>
<td>(10) Haskell</td>
<td>156</td>
<td>112</td>
<td>8</td>
</tr>
</tbody>
</table>

- Concluded:
  “... the results clearly demonstrate the value of functional languages in prototyping ...”

*Haskell vs. Ada vs. C++ vs. Awk vs. ..., by P. Hudak and M. Jones, Yale University, Computer Science Department, 1994*
Disadvantage of (pure) functional programs

- Run-time efficiency: They often ...
  ... execute more slowly than imperative ones
  ... require more space than imperative ones

Why??
Disadvantage of (pure) functional programs

• Run-time efficiency: They often ...
  ... execute more slowly than imperative ones
  ... require more space than imperative ones
Von Neumann bottleneck

- Von Neumann
  - Responsible for idea of stored program

- Von Neumann Bottleneck
  - Backus’ term for limitation in CPU-memory transfer

- Related to sequentiality of imperative languages
  - Code must be executed in specific order
    
    ```
    function f(x) { if (x < y) then y = x; return y }
    g( f(i), f(j) );
    ```

Verdict is still out
Eliminating VN Bottleneck

• No side effects
  – Evaluate subexpressions independently

Example

```plaintext
function f(x) { if x < y return 1 else return 2; }
g(f(i), f(j), f(k), ...);
```

• Does this work in practice? Good idea but ... 
  – Too much parallelism
  – Little help in allocation of processors to processes

• Effective use of concurrency is a hard problem