FP

Functional programming - programming with functions instead of variables
Define a function as a combination of others using various combinig forms. Think of the function being defined as a set of filters or pipes each transforming its argument object into an output object
Each function takes a single argument object and produces a single object result
There are no declarations other than function definitions. In fact, there are no variables. You program with functions, not values
Programs should be read from right to left, mathematical order, except for conditionals
The set of combining forms is limited, unlike the λ-calculus
IFP (Illinois FP interpreter)

Example FP function
Def IP = (l+) (α+) TRANS
- +, * and TRANS are functions
- /, □, and α are combining forms
Inner product is the result of three operations on a pair of input vectors
- First transpose the input pair of vectors, using TRANS, into a vector of pairs
- Then apply the multiplication operator, using "apply to all" (α), to each pair of the vector, producing a vector of results
- Finally, add up the elements of the vector, using INSERT (l+), producing a scalar result

Motivations

Word-at-a-time programming: "unit" of traditional programming is the assignment statement that alters one variable
von Neumann I/O bottleneck: the architectural limitations of early machines lead to the imperative style of language
Single accumulator, data in memory, instruction fetch, decode, and execute
Small modifications made frequently to the state are hard to comprehend; programs are static representations of dynamic processes
- Structured programming was a reaction to this
- but it did not go far enough
Use of names for parameters engenders complex semantics for argument passing (call by name/value/etc.)
Hard to reason about programs because of the variety of features
Backus’ 3 tiers of complexity

Simply functional language (fp): no state, limited names, closed set of functional forms, simple substitution semantics, algebraic laws, no eval/apply available to programmers

Formal fp system (fpl): extensible set of functional forms, functions can be represented by objects; conversion from object representation to applicable form; formal semantics

Applicative state transition system (ast): fpl with the addition of a state that can be modified with coarse-grained operations

An FP System

A set of objects; the data to be transformed
A set of functions for computing the transforms (built-in & user-defined)
A set of combining forms, fixed for the system
A set of definitions for extending the set of functions
The operation of application (·) for combining objects and functions

FP Objects

Atoms: numbers, True and False, characters, LISP atoms, `nil`
Sequences (tuples): elements are objects, similar to LISP lists
⊥: ("bottom" or "undefined")
- If ⊥ is part of a sequence, then the sequence itself is ⊥. This property is called strictness

FP Functions

Built-in or user-defined
Map an object into another object
Are strict (⊥ preserving) f: ⊥ == ⊥
Are applied to an object with the colon operator f : x (where x is some object and not a name)
The user can define functions with Def; recursion is allowed

FP Built-in Functions

Arithmetic: +, -, *, /
List processing:
- tl, atom, null, reverse, equals, length, apndl, apndr, rotl, rotr, tr
Selection: 1 (car), 2 (cadr), ..., 1r, 2r, ...
Logical: and, or, not
Restructuring: trans, distl, distr
Identity: id
Iota (integers up to): (iota: 5 == <1, 2, 3, 4, 5>)

FP Functional Forms

<table>
<thead>
<tr>
<th>Form</th>
<th>Syntax</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Composition</td>
<td>f o g</td>
<td>(f o g) : x == f (g : x)</td>
</tr>
<tr>
<td>Constant</td>
<td>x y = x</td>
<td>x y = y → ⊥, x (not strict)</td>
</tr>
<tr>
<td>Construction</td>
<td>f g = f (g : )</td>
<td>f (g : ) : x == x</td>
</tr>
<tr>
<td>Conditional</td>
<td>(p → f : g)</td>
<td>(p → f : g) : x == (p : x) True → f : x ;</td>
</tr>
<tr>
<td>(strict)</td>
<td></td>
<td>(p : x) False → g : x ;</td>
</tr>
<tr>
<td>Insert (reduce)</td>
<td>/ f x</td>
<td>/ f x : x == x \ x = x ;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>x = x1, ..., x = x and</td>
</tr>
<tr>
<td></td>
<td></td>
<td>n = 1 →</td>
</tr>
<tr>
<td></td>
<td></td>
<td>f : &lt;x1, f (x1) : &lt;x2, ..., x_n&gt;) ; ⊥</td>
</tr>
</tbody>
</table>
### Functional forms (continued)

<table>
<thead>
<tr>
<th>Form</th>
<th>Syntax</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apply to all (mapcar)</td>
<td>( \alpha f )</td>
<td>( \alpha f : x \mapsto y \mapsto x \mapsto \phi )</td>
</tr>
<tr>
<td>Binary to unary (Currying)</td>
<td>( \text{bind } f x )</td>
<td>( \text{bind } f x : y \mapsto f &lt;x, y&gt; )</td>
</tr>
<tr>
<td>while (while p f)</td>
<td>while p f : x \mapsto p : x = True \mapsto [while p f : f : x]; p : x = False \mapsto x; \bot</td>
<td></td>
</tr>
</tbody>
</table>

### Example Definitions

- Def last \( \equiv \) null \( \square \) tl \( \rightarrow \) 1; last \( \square \) tl  
  \( \equiv (\text{null } \square \text{tl}) \rightarrow 1; (\text{last } \square \text{tl}) \)
- Def sub1 \( \equiv \) \( \_ \square \left[ \text{id}, 1 \right] \)
- Def eq0 \( \equiv \) eq \( \square \left[ \text{id}, 0 \right] \)
- Def ! \( \equiv \) eq0 \( \rightarrow \) 1; \( \_ \square \left[ \text{id}, ! \square \text{sub1} \right] \)
- Def ! \( \equiv \) *\( \square \) iota

### FP Semantics

- To apply a function, replace its use by the right-hand side of its definition.
- Non-terminating executions yield \( \bot \).
- The semantics rely on the primitive functions and combining forms.
- Hence, it is better to think of an fp system instead of the fp system.
- Semantics for the primitive functions and combining forms were given informally in the paper.

### Algebraic manipulation of programs

fp's simple semantics enable program manipulations and optimizations. These are expressed as algebraic laws.

For example, the following law indicates how composition distributes on the right over construction:

\[
(f \circ g) \circ h)x = (f \circ (g \circ h))x = f \circ (g \circ h)x
\]

### Example laws

\[
(p \rightarrow f; g) \circ h = p \circ h \rightarrow f \circ h; g \circ h
\]

\[
h \circ (p \rightarrow f; g) = p \rightarrow h \circ f; h \circ g
\]

\[
p \rightarrow (p \rightarrow f; g); h = p \rightarrow f; h
\]

\[
\alpha (f \circ g) \equiv \alpha f \circ \alpha g
\]
FP Limitations

Set of definitions is fixed; programs cannot compute programs
No concept of state; no history sensitivity
Cannot construct new functional forms
Notation is so concise that it is hard to read programs

FP Advantages

No need for arguments or binding forms
Reduced need for control statements
Does not over-constrain the order of evaluation
Programs can be algebraically manipulated (for efficiency or readability)
Simple semantics

FP Concepts

No names for variables, hence, no parameter passing semantics to understand
No state; Applicative programs
Combining forms for constructing programs
Only data structure is the tuple; type checking up to the user
Algebra of programs; reasoning about programs in the programming language itself

FFP Systems

Definition of new functional forms
Objects can be used to represent functions
Ability to convert the object representation of a function into a function
Syntax:
  – \((x:y)\) - \(x\) is the operator; \(y\) is the operand
  – All objects are also expressions

Power of FFP systems

In an FFP system, we can define a “function” apply, as in LISP:

\[
\text{apply} : \langle x, y \rangle = (x : y)
\]

Notes:
  – Result (i.e., \((x : y)\) ) meaningless in FP system
  – Formally, result of apply is an expression, which can be manipulated using replacement to yield a function application