FP

"A variable in a proposition of logic is, after all, nothing but a token that characterizes certain argument places and operators as belonging together; thus it has the status of a mere auxiliary notion that is really inappropriate to the constant, eternal essence of the propositions of logic."

-- Moses Schönfinkel

FP

Functional programming - programming with functions instead of variables

Define a function as a combination of others using various *combining 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

The 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 = (/+) □ (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 (/+), producing a scalar result

IP SAMPLE EVALUATION

```
IP:<<1, 2, 3>, <6, 5, 4>>
(/+) □ (α*) □ TRANS:<<1, 2, 3>,<6, 5, 4>>     -- def of IP
(/+):((α*):(TRANS:<<1, 2, 3>, <6, 5, 4>>))    -- def of composition
(/+):((α*):(<<1, 6>, <2, 5>, <3, 4>>))        -- def of TRANS
(/+):(< *:<1, 6>, *:<2, 5>, *:<3, 4>>)        -- def of α
(/+):<6, 10, 12>                              -- def of *
+:<6, (/+):<10, 12>>                          -- def of /
+:<6, +:<10, (/+):<12>>>                      -- def of /
+:<6, +:<10, 12>>                             -- def of /
+:<6, 22>                                     -- def of +
28                                            -- def of +
```

Motivations

Word-at-a-time programming; "unit" of traditional programming is the assignment statement that alters one variable

von Neumann 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 (ffp): 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): ffp 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, ϕ (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, tlr
- 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

Form	Syntax	Semantics
Composition	f □ g	(f □ g) : x == f : (g : x)
Constant	$\underline{x}$, $\underline{1}$, $\underline{2}$, ...	$\underline{x}$: y == y = ⊥ → ⊥; x (not strict)
Construction	[f, g]	[f, g] : x == <f : x, g : x>
Conditional (strict)	(p → f; g)	(p → f; g) : x == (p : x) = True → f : x; (p : x) = False → g : x; ⊥
Insert (reduce)	/ f	(/ f) : x == x = $\langle x_1 \rangle \to x_1$; x = $\langle x_1, ..., x_n \rangle$ and n > 1 → f : $\langle x_1, (/f) : \langle x_2,...,x_n \rangle \rangle$; ⊥

Functional forms (continued)

Form	Syntax	Semantics
Apply to all (mapcar)	α f	α f : x == x = ϕ → ϕ; x = <x_1, ..., x_n> → <f : x_1, ..., f : x_n>; ⊥
Binary to unary (Currying)	(bu f x)	(bu f x) : y == f : <x, y>
while	(while p f)	(while p f) : x == p : x = True → (while p f) : (f : x); p : x = False → x; ⊥

Example Definitions

Def last == null □ tl → 1; last □ tl
== (null □ tl) → 1; (last □ tl)

Def sub1 == - □ [id, $\underline{1}$]

Def eq0 == eq □ [id, $\underline{0}$]

Def ! == eq0 → $\underline{1}$; * □ [id, ! □ sub1]

Def ! == /* □ iota

FP Semantics

To apply a function, replace its use by the right hand side of its definition

Non-terminating executions yield ⊥

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

FP Semantics

To obtain the meaning of a functional application (f:x), consider the following four possibilities.

1. f is a primitive function ⇒ evaluate it immediately;
2. f is a functional form ⇒ use the definition of the form to obtain a new program and apply it. The functional forms are defined using McCarthy's conditional expression;
3. f is a user-defined function ⇒ plug in the definition for the occurrence;
4. If none of the above ⇒ the result of the application is ⊥

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, g]  □  h):x
    == [f, g]:(h:x)              -- def of composition
    == <f:(h:x), g:(h:x)>        -- def of construction
    == <(f □ h):x, (g □ h):x>    -- def of composition
    == [f □ h, g □ h]:x          -- def of construction
```

Example laws

(p → f; g) □ h == p □ h → f □ h; g □ h

h □ (p → f; g) == p → h □ f; h □ g

p → (p → f; g); h == p → f; h

α (f □ g) == α f □ α 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:

```
apply:<x,y> = (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