FP_k Tutorial

Kyle Ross
<kyle@cs.chalmers.se>

Introduction

FP is a beautiful, untyped functional language defined by John Backus in his Turing Award lecture "Can Programming Be Liberated from the von Neumann Style?". The present interpreter, FP_k, is based on that paper, and the reader is encouraged to follow that reference for more information.

Basics

FP provides a single operation: Application. A function is applied to an object; the result is an object. No function is an object and no object a function.

Objects

FP objects belong to one of following classes (types): Number (integers), Boolean, (untyped) List, Bottom (failure).

Numbers

{a series of digits} (FP, FP_k)

e.g., `5`

Booleans

T and F (FP)

`T` and `F` (FP_k)

Lists

$\langle e_1..e_n \rangle$ (FP)

`<e1..en>` (FP_k)

e.g., `<1, T, <3>>`

ϕ denotes the empty list in FP. `<>` is used in FP_k, but `{phi}` may be used in some output.

Bottom

$\perp$ (FP)

{there is no syntactic way to directly refer to Bottom} (FP_k)

Primitive Functions

Backus's FP paper describes a number of primitive functions. These are implemented in FP_k, along with some useful extensions.

Selectors

s (FP)

`s` (FP_k)

e.g., `2 : <3, 4, 5>` $\Rightarrow$ `4`

Right Selectors

s r (FP)

`s r` (FP_k)

e.g., `1 r : <1, 2, 3>` $\Rightarrow$ `3`

(Left) Tail

tl (FP)

`tl` (FP_k)

e.g., `tl : <1, 2, 3>` $\Rightarrow$ `<2, 3>`

Right Tail

tlr (FP)

`tlr` (FP_k)

e.g., `tlr : <1, 2, 3>` $\Rightarrow$ `<1, 2>`

Head

`hd` (FP_k)

e.g., `hd : <1, 2, 3>` $\Rightarrow$ `1`

This is an FP_k extension; of course `hd` is equivalent to `1`.

Identity

id (FP)

`id` (FP_k)

e.g., `id : 3` $\Rightarrow$ `3`

Atom?

atom (FP)

`atom` (FP$_k$)

`atom : <3>` $\Rightarrow$ `F`

Equals?

eq (FP)

`eq` (FP$_k$)

e.g., `eq : <2, 3>` $\Rightarrow$ `F`

Less?

`less` (FP$_k$)

e.g., `less : <2, 3>` $\Rightarrow$ `T`

This is an FP$_k$ extension.

Great?

`great` (FP$_k$)

e.g., `great : <2, 3>` $\Rightarrow$ `T`

This is an FP$_k$ extension.

Null?

null (FP)

`null` (FP$_k$)

e.g., `null : <>` $\Rightarrow$ `T`

Reverse

reverse (FP)

`reverse` (FP$_k$)

e.g., `reverse : <1, 2, 3>` $\Rightarrow$ `<3, 2, 1>`

Distribute from Left

distl (FP)

`distl` (FP$_k$)

e.g., `distl : <1, <1, 2, 3>>` $\Rightarrow$ `<<1, 1>, <1, 2>, <1, 3>>`

Distribute from Right

distr (FP)

`distr` (FP$_k$)

`distr : <<1, 2, 3>, 1>` $\Rightarrow$ `<<1, 1>, <2, 1>, <3, 1>>`

Length

length (FP)

`length` (FP_k)

e.g., `length : <4, 5, 6>` ⇒ `3`

Add, Subtract, Multiply, Divide

+, −, ∗, ÷ (FP)

`+`, `-`, `*`, `/` (FP_k)

e.g., `+ : <1, 2>` ⇒ `3`

(Matrix) Transpose

trans (FP)

`trans` (FP_k)

e.g., `trans : <<1, 2, 3>, <4, 5, 6>>` ⇒ `<<1, 4>, <2, 5>, <3, 6>>`

And, Or, Not

and, or, not (FP)

`and`, `or`, `not` (FP_k)

e.g., `and : <T, F>` ⇒ `F`

Append Left

apndl (FP)

`appendl` (FP_k)

e.g., `appendl : <1, <2, 3>>` ⇒ `<1, 2, 3>`

Append Right

apndr (FP)

`appendr` (FP_k)

e.g., `appendr : <<1, 2>, 3>` ⇒ `<1, 2, 3>`

Rotate Left

rotl (FP)

`rotl` (FP_k)

e.g., `rotl : <1, 2, 3>` ⇒ `<2, 3, 1>`

Rotate Right

rotr (FP)

`rotr` (FP_k)

e.g., `rotr : <1, 2, 3>` ⇒ `<2, 3, 1>`

Functional Forms

This section describes the FP functional forms as implemented in FP_k. The original FP syntax is shown alongside the FP_k syntax.

Composition

$f \circ g$ (FP)

`f o g` (FP_k)

e.g., `+ o [1, 2] : <2, 3>` $\Rightarrow$ `5`

Construction

$[f_1..f_n]$ (FP)

`[f1..fn]` (FP_k)

e.g., `[id, id, id] : 3` $\Rightarrow$ `<3, 3, 3>`

Condition

$(p \rightarrow f; g)$ (FP)

`(p -> f ; g)` (FP_k)

e.g., `(eq o [1, 2] -> ~3 ; ~4)` $\Rightarrow$ `4`

Constant

$\tilde{x}$ (FP)

`~x` (FP_k)

e.g., `~3 : 4` $\Rightarrow$ `3`

Insert (fold)

$/f$ (FP)

`/ f u` (FP_k, where `u` is the right unit of `f`)

e.g., `/ + 0 : <1, 2, 3>` $\Rightarrow$ `6`

This is a slight departure from the FP semantics—the right unit of the function must be provided explicitly. This seems a more general solution, and it was simpler to implement.

Apply to All (map)

αf (FP)

`alpha f` (FP_k)

e.g., `alpha (+ o [id, ~1]) : <1, 2, 3>` $\Rightarrow$ `<2, 3, 4>`

Binary-to-Unary (Curry)

bu f x (FP)

`bu f x` (FP$_k$)

e.g., `bu + 2 : 3` $\Rightarrow$ `5`

While (iteration)

while p f (FP)

`while p f` (FP$_k$)

e.g., `while (not o (bu eq 5)) (bu + 1) : 2` $\Rightarrow$ `5`

Definitions

FP allows recursive definitions, binding a name to a functional form. FP$_k$ implements this, and provides the `show` extension. All functions are assumed to be mutually recursive (a "call" is simply a right-hand-for-left-hand-side substitution; this is safe since there are no variables and, thus, no name problems).

Definition

Def $l = r$ (FP)

`def l = r` (FP$_k$)

e.g., `def last = (null o tl -> 1 ; last o tl)`

then `last : <1, 2, 3>` $\Rightarrow$ `3`

Show

`show l` (FP$_k$)

e.g., `show last`

This is an FP$_k$ extension.

Comments

To allow documented code, FP$_k$ ignores any text beginning with the first occurrence of "***" and extending until the end of the line.

Commands

FP$_k$ provides several utility commands to enrich the FP programming experience. A command must be the sole non-whitespace text on the input line.

Environment Reset

`reset`

Removes all definitions.

Exit

`exit` or `quit`

Exits the interpreter.

Input Echo

`echo`

Toggles term echo. When enabled, each line of input is pretty-printed after it is parsed.

Reduction Display

`debug`

Toggles display of each step in term reduction. When enabled, the interpreter prints extensive, step-by-step output as the term is reduced.

File Loading

`load f`

e.g., `load t` (loads from the file "t.fp")

Loads the definitions from a file. Because this is a bit of a hack to the parser, the filename must be a valid FP_k identifier followed by the extension ".fp" (which is *not* used in the argument to `load`).