Plumber - A Higher Order Data Flow Visual Programming Language in Lisp

Seika Abe
MSI
Outline

1. What is DFVPL
2. Plumber
3. Streams
4. Higher order functions
5. Finding monads
1/5. What is DFVPL
Dataflow model

1 + 2 * 3

Print
DFVPL is a nice notation

(print (* (+ 1 2) (* 2 3)))
Flow chart is not a DFVPL

Flowchart:

1. **SUM ← 0**
2. **I ← 1**
3. **SUM ← SUM + I**
4. **I ← I + 1**
5. **I <= 100**
   - **Y**: **SUM := 0**
   - **I := 1**
   - **do**
     - **SUM := SUM + I**
     - **I := I + 1**
   - **while I <= 100**
   - **print SUM**
6. **N**: **Print SUM**
LabVIEW
DFVPL in the literature

• Jack B. Dennis, First version of a data flow procedure language, 1974.
  – Early work of DFVPL

• M. Mosconi and M. Porta, Iteration constructs in data-flow visual programming languages, 2000.
  – A study of loop in DFVPL and also a good survey

• W. M. JOHNSTON, Advances in Dataflow Programming Languages, 2004.
  – A good survey on DFVPL
Dennis’s factorial in his DFVPL
Mosconi’s factorial in his DFVPL
2/5. Plumber
Plumber

- Implemented in Lisp (Allegro CL)
- Consists of a graph editor for DFVPL
- And a compiler from DFVPL to Lisp
- Nodes can be categorized into two
  - One is functional and the other is special
  - Functional nodes have the dataflow semantics
  - Functional nodes can be any Lisp functions
  - Special nodes are built-in and its own semantics
- Each node corresponds to a thread
- Provides higher-order function feature
Behavior of function nodes

1. Wait until all of input data are available
2. Then, make a data list \( args \) of the all input
3. Apply function of the node to \( args \)
4. Send the result to the output port
5. \( N \)-values result are sent to \( N \) output port
6. GOTO Step 1.
addmul

constant node

function node

special node
(Shown in slant bold)
Demonstration 1
3/5. Streams
Stream operations in Plumber

• Stream
  – Converts a list to a stream

• UnStream
  – Converts a stream to list

• Repeat
  – Make a stream by repeating a data
Demonstration 2
stream-unstream
flipflop
sortnet
4/5. Higher order functions
Previous approaches for higher order functions in DFVPL

• Special node with function hole approach
  – Easy to understand for non-programmers
  – Seems awkward for Lispers
  – Used in: DataVis, CUBE, ESTL, YUBA

• Function as a value approach
  – Seems difficult for non-programmers
  – Needs a node for funcall or apply
  – Used in: ShowAndTell, VPL, PHF
  – Plummer also implemented this automatically
Special node with function hole approach

SORT

(3 1 2) → NumCmpFunc → (1 2 3)
Function as a value approach
Drawbacks of the simple version of function as a value approach

1. It depends the power of Lisp too much
2. Making a function value from a previously defined function is impossible

What is the most suitable notation for DFVPL?

Our answer is the currying based notation!
(lambda (x0) (lambda (x1) (+ x0 x1)))
curry2

\[\text{list} \to (\lambda (x_0) (\lambda (x_2) (\text{list } x_0 \ x_1 \ x_2 \ x_3)))\]
(lambda ()
  (lambda (x0)
    (lambda (x1)
      (< x0 x1))))

(defun uncurry (fun)
  (lambda (x y)
    (funcall (funcall fun x) y)))
5/5. Finding monads
Monads in the literature


• J. Hughes, *Generalizing monads to arrow*, 2000.
Value with effect \( M \alpha \)

\[
M \alpha = (\alpha \quad \text{ListOfDebugDumpString})
\]

(defun Print! (Ma)
  ;; prints purely part value
  (format t "value = ~s~\%~\%" (car Ma))
  ;; prints debug dump output
  (dolist (str (cadr Ma))
    (format t "~a~\%" str)))
\[(\text{sqrt!} (1+! 1)) \, ?\]

\[1+! : \text{Integer} \rightarrow \text{M Integer} \]
returns argument+1 with debug dump

\[(\text{defun} \ 1+! \ (x)\)
  (\text{let} ((y (1+ x)))
    (\text{list} y (\text{list} (\text{format} \ \text{nil} \ "\text{~s} \rightarrow 1+!\rightarrow \text{~s}\" \ x \ y))))))\]

\[\text{sqrt!} : \text{Integer} \rightarrow \text{M Real} \]
returns sqrt of argument with debug dump

\[(\text{defun} \ \text{sqrt!} \ (x)\)
  (\text{let} ((y (\text{sqrt} x)))
    (\text{list} y (\text{list} (\text{format} \ \text{nil} \ "\text{~s} \rightarrow \text{sqrt!}\rightarrow \text{~s}\" \ x \ y))))))\]
Demonstration 3
monademo2
monademo3
monademo5
We have found re-usable patterns without any knowledge of category theory!
monademo8
monademo9
Correspondence to the definition in the literature

\[ \alpha \rightarrow M\beta \]

\[ \text{unit}_M : \alpha \rightarrow M\alpha \]

\[ \text{bind}_M : M\alpha \rightarrow (\alpha \rightarrow M\beta) \rightarrow M\beta \]
Final remarks

- Plumber is firstly designed for non-programmers
- And intended to provide a customize features for MSI products, as in Emacs Lisp of Emacs
- It is still incomplete as a general-purpose programming language
- This project is being suspended over two years
Thank you !
Monad laws

unitM : \( \alpha \to M\alpha \)

bindM : \( M\alpha \to (\alpha \to M\beta) \to M\beta \)

\((\text{unitM } a) \ `\text{bindM}` k = k a\)

\(m \ `\text{bindM}` \text{ unitM} = m\)

\(m \ `\text{bindM}` (\lambda a. (k a `\text{bindM}` h)) = (m `\text{bindM}` \lambda a.k a) `\text{bindM}` h\)
\[(\text{unitM } a) \ `\text{bindM}\` k = ka\]

```
value = ((f a) ("string"))
value = ((f a) (append nil ("string"))

(append nil x) = x
```
\( m \ `\text{bindM}\` \ \text{unitM} = m \)

\[
\text{(append x nil)} = x
\]
\[ m \ `\text{bindM}` (λa.(ka `\text{bindM}` h)) \\
= (m \ `\text{bindM}` λa.ka) \ `\text{bindM}` h \\
\]

\[
(\text{append } x (\text{append } y z)) \\
= (\text{append } (\text{append } x y) z))
\]
Monad laws of $M\alpha$

(リスト, nil, append)

はモノイドでなければならない！