Concatenative programming

Lalginar, 15. marec 2012

Jure Mihelič

Literature

- Why concatenative programming matters?, Jon Purdy
  - http://evincarofautumn.blogspot.com/2012/02/why-concatenative-programming-matters.html
- Mathematical foundations of Joy, Manfred von Thun
  - http://www.latrobe.edu.au/phimvt/joy/j02maf.html
- Concatenative programming: An overlooked paradigm in functional programming, Dominikus Herzberg, Tim Reichert
- Itd.

Applicative language

- Uses ***function application***.
- Well-known.
- λ-calculus:
  - variables, lambdas, applications;
  - name binding, scope, closures.

f x := x + 1	f := λx. plus x 1
f 41 → 41 + 1 → 42	f 41 → plus 41 1 → 42

Concatenative language

- Uses ***function composition***.
  - There is no (explicit) function application.
- Composition – monoid:
  - Operation: f . g = f(g(...))
  - Unit: e . x = x = x . e
  - Associativity: (x . y) . z = x . (y . z)

Syntax

- Words.
- Concatenation.
- Quotation.

Words

- Denotes functions.
- Functions have no (explicit) arguments.

```
f g h
dup dip bi tri
* - + /
0 1 2 3
“foo” “bar” “baz”
```

Words

- Primitives
  - dup: $(A\ b \rightarrow A\ b\ b)$
  - pop: $(A\ b \rightarrow A)$
  - swap: $(A\ b\ c \rightarrow A\ c\ b)$
  - apply: $(A\ (A \rightarrow B) \rightarrow B)$
  - quote: $(A\ b \rightarrow A\ (C \rightarrow C\ b))$
  - compose: $(A\ (B \rightarrow C)\ (C \rightarrow D) \rightarrow A\ (B \rightarrow D))$

Concatenation

- Denotes a composition.
- Only binary operation:
  - no explicit infix operator → juxtaposition.
- “Reverse” order.
  - Consider data flow.
  - It is actually forward.
- Associative operation:
  - no parentheses needed.

f g h ↔ h . g . f
↔ h(g (f (…)))

Homomorphism

- Words and concatenation.
  - Syntactic monoid.
- Functions and composition.
  - Semantic monoid.
- Meaning function.
  - Homomorphism:
    - syntactic monoid → semantic monoid.

Literals

- Literals are functions.
  - Take no arguments.
  - Return a value.
  - Types:
    - 42: () → (int)
    - “moo”: () → (str)
  - Ok, but how to compose 2 3?

Row polymorphism

- Functions have a polymorphic type.
- Input:
  - take any arguments,
  - followed by what is actually needed.
- Output:
  - return arguments not used,
  - followed by actual return value.

Types

- Types.
  - int, str, bool, …
  - Functions:
    - type → type
- Type variables.
  - Single type element.
  - Small letters.
  - a, b, c, …., 'a , 'b, 'c, ...

Types

- Stack variables.
  - Sequence of zero or more types.
  - Big letters.
  - A, B, C, …, 'A, 'B, 'C, ...
- Implicit stack variables – row variables.
  - Where the left-most variable is not a stack variable.
- Universal quantification.
  - At the outermost level.

Types

- 42:
  - (A → A, int)
  - (→ int)
- +:
  - (A int int → int)
  - (int int → int)
- dup:
  - (A b → A b b)
  - (b → b b)

Types

- apply:
  - $(A\ (A \to B) \to B)$
- quote:
  - $(A\ b \to A\ (C \to C\ b))$
  - $(b \to (C \to C\ b))$
- compose:
  - $(A\ (B \to C)\ (C \to D) \to A\ (B \to D))$
  - $((B \to C)\ (C \to D) \to (B \to D))$

2 3: ?

- How to compose 2 3?
  - 2: (A) → (A, int)
  - 3: (B) → (B, int)
  - 2 3: ?
  - A, int = B
  - 2 3: (A) → (A, int, int)

2: (A) → (A, int)
3: (B) → (B, int)

2 3 +: ?

- 2 3: ?
  - A, int = B
  - 2 3: (A) → (A, int, int)
- (2 3) +: ?
  - A, int, int = C, int, int
  - A = C
  - (2 3) +: (A) → (A, int)

2: (A) → (A, int)
3: (B) → (B, int)
+: (C, int, int) → (C, int)

2 3 +: ?

- 3 +: ?
  - B, int = C, int, int
  - B = C, int
  - 3 +: (C, int) → (C, int)
- 2 (3 +): ?
  - A, int = C, int
  - A = C
  - 2 (3 +): (A) → (A, int)

2: (A) → (A, int)
3: (B) → (B, int)
+: (C, int, int) → (C, int)

Quotation

- Denotes an abstraction / anonymous function.
- Unary operation:
  - outfix operator [*...*]

[f g h]

- f …. function f
- [f] …. function that returns function f

Quotation

- 2 >
  - function that returns true if argument greater than 2.
- [2 >]
  - function that returns function 2 >.

```
{ 1 2 3 4 5 } [ 2 > ] filter → { 3 4 5 }

4 [ 2 > ] call → 4 2 > → true

[ f ] [ g ] compose → [ f g ]
```

Type reconstruction

- Concatenation.

$$\frac{f : (A \to B) \quad g : (B \to C)}{fg : (A \to C)}$$

- Quotation.

$$\frac{f : (A \to B)}{[f] : (C \to C\ (A \to B))}$$

- [dup] apply

$$\frac{\dfrac{dup{:}A\ \ a{\to}A\ \ a\ \ a}{[dup]{:}B{\to}B\ \ (A\ \ a{\to}A\ \ a\ \ a)} \qquad \dfrac{}{apply{:}C(C{\to}D){\to}D}}{[dup]apply : A\ a \to A\ a\ a}$$

Partial function application

- Function composition is associative.
  - f g h = (f g) h = f (g h)
- Partial function application is trivially represented.
- Currying.

1 +
2 <

Refactoring

- Program is a sequence of words.
- Concatenation of programs is a new program.
- Any (word-)substring of a program is a program.
- Refactoring.
- Compiler optimization.
- Parallelism.

Point-free expressions

- f x y = x + y
  - f returns the sum of two arguments.
- f = (+)
  - f is addition function.

```
countWhere :: (a → Bool) → [a] → Int

countWhere predicate list = length (filter predicate list)
ali
countWhere = (length .) . filter

countWhere (>2) [1, 2, 3, 4, 5]
```

```
countWhere: (seq quot → newseq)
countWhere := filter length

{ 1 2 3 4 5 } [ 2 > ] countWhere
```

Examples

```
: palindrome ( x -- y )
    [ Letter? ] filter >lower dup reverse = ;

#! fun gcd a b is if b == 0 then a else gcd b (a % b)
: gcd ( a b -- c )
    dup zero? [ drop ] [ [ mod ] keep swap gcd ] if ;

#! fun fac n is if n <= 1 then 1 else n * fac (n-1)
: fac ( n -- m )
    dup 1 <= [ drop 1 ] [ dup 1 - fac * ] if ;
```

The dark side

- $f\ x\ y\ z = y^2 + x^2 - |y|$

- …

- f = drop dup dup × swap abs rot3 dup × swap − +

Stack shufflers

- drop, 2drop, 3drop, (pop, zap), nip, 2nip
  - 1 2 3 4 3drop → 1
  - 1 2 3 nip → 1 3
- dup, 2dup, 3dup, dupd
- swap, swapd
- over, 2over
  - 1 2 3 over → 1 2 3 2
- pick: (x y z → x y z x)
  - 1 2 3 pick → 1 2 3 1
- rot: (x y z → y z x)
  - 1 2 3 rot → 2 3 1

Conditional combinators

- ?: (b true false → true/false)
  - 24 42 < “good” “broken” ? print
- if: (A b true: (A → B) false: (A → B) → B)
  - 1 2 2dup < [+] [*] if
  - if := ? call
- when, unless: (b q →)
  - -5 dup 0 < [10 +] when .

Looping combinators

- while, until:
  - (A cond: (A → B b) body: (B → A) → B)
  - 0 [dup 42 <] [1 +] while
- do: (cond body → cond body)
  - Modifier.
  - Body is executed at least once.
  - [p] [q] do while

Dataflow combinators

- Preserving combinators.
  - dip, 2dip, 3dip, 4dip, keep, 2keep, 3keep
    - dip … invokes quot temporarily hiding the top of stack.
      - 2 3 4 [+] dip → 5 4
    - keep … invokes quote keeping the top.
      - 2 3 4 [+] keep → 2 7 4
- Cleave combinators.
  - bi, 2bi, 3bi, tri, 2tri, 3tri, cleave, 2cleave, 3cleave
  - { 3 6 8 7 4 2 } [sum] [length] bi /
  - { 3 6 8 7 4 2 } { [sum] [length] } cleave /

Dataflow combinators

- Spread combinators.
  - bi*, 2bi*, tri*, 2tri*, spread
  - 1 2 3 [1 -] [2 -] [3 -] tri*
  - 1 2 3 { [1 -] [2 -] [3 -] } spread
- Apply combinators.
  - Apply single quotation to multiple values
  - bi@, 2bi@, tri@, 2tri@
  - 1 2 [1 +] bi@

Sequence combinators

- each: (… seq quot: (… x → ...) → ...)
  - { 1 2 3 } [42 +] each
- map: (… seq quot: (… x → y) → neqseq)
  - { 1 2 3 } [42 +] map
- filter: (… seq quot: (… x → ?) → … subseq)
  - { 1 2 3 4 5 6 7 8 9 10 } [even?] filter
- reduce: (… seq id quot: (… prev curr → next) → … result)
  - { 1 2 3 4 5 6 } 1 [*] reduce

Other combinators

- times
  - 5 [“Juhuhu” print] times
- curry
  - 1 2 [+] curry curry → 1 [2 +] curry → [1 2 +]

Evaluation

- Stack-based evaluation
  - Words are functions from stack to stack.
  - Stack-effect declaration.
    - dup: $(b \rightarrow b\ b)$

2 3 * 4 5 * +

	()
2	(2)
3	(2, 3)
*	(6)
4	(6, 4)
5	(6, 4, 5)
*	(6, 20)
+	(26)

Evaluation

- Term-rewriting.

2 3 * 4 5 * +	2 3 * → 6
6 4 5 * +	4 5 * → 20
6 20 +	6 20 + → 26
26	

Languages

- Stack-based
  - Forth
  - Postscript
  - JVM – Java virtual machine
  - CLI – Common Language Infrastructure
  - RPL – ROM-based procedural language (HP)
  - BibTeX (bst files)
  - CPython

Languages

- Concatenative
  - Joy
    - Canonical concatenative language.
  - Cat
    - Static typing.
  - Factor
    - Probably most advanced.
  - Enchilada
    - Term-rewriting.
  - 5th, Raven, Onyx, Staapl, Lviv, Deque

Factor

- Word definition.
  - : name (stack effect declaration) def ;
  - : plus5 (x -- y) 5 + ;
  - : sq (x -- y) dup * ;
- Data types.
  - Integers, floats, complex, rationals.
  - Sequences: { 1 2 3 4 }
  - Objects.

Turing completeness

- Of course, but which primitives?
- The theory of concatenative combinators, Brian Kerby.
- Complete base:
  - [A] i → A
  - [B] [A] dip → A [B]
  - [B] [A] cons → [[B] A]
  - [A] dup → [A] [A]
  - [A] zap →

Turing completeness

- Lambdas.
  - A\ … pop the top and bind it to the word 'A'.
  - e.g. A\ B\ A [B] … dip

A\ B A [C A]
[B] dip **A\ A** [C **A**]
[B] dip dup A\ **A1\ A1** [C A]
[B] dip dup **A\ i** [C A]
[B] dip dup [i] dip **A\ [C A]**
[B] dip dup [i] dip [**A\ C** A] cons
[B] dip dup [i] dip [[C] dip **A\ A**] cons
[B] dip dup [I] dip [[C] dip i] cons

… A\ B → [B] dip A\
… A\ A → dup A\ A1\ A1
… A1\ A1 → i
… A\ i → [i] dip A\
… A\ [*] → [A\ *] cons
… A\ C → [C] dip \A
… A\ A → i