Shen - Programming Language

Aditya Siram

May 2, 2013

Cash & Candy

Setting variables

```
(set *candy* [snickers hersheys twix])
(set *currency* [quarter dime nickel dollar])
```

Getting variables

```
(value *candy*)
  => [snickers hersheys twix]
```

Mutating variables

```
(set *candy* (append [payday] (value *candy*)))
(value *candy*)
  => [payday snickers hersheys twix]
```

Pricing & Denominations

- Notice type signatures

Denominations

```
(define faceValue
  {currency --> number}
  quarter -> 25
  dime    -> 10
  nickel  -> 5
  dollar  -> 100)
```

Candy cost

```
(define candy-cost
  {candy --> number}
  snickers -> 100
  twix     -> 125
  hersheys -> 75
  payday   -> 95)
```

Simple typing

Typing Cash & Candy

```
(datatype items
  if (element? X (value *currency*)) \\ premise
  ___________________________________
  X : currency;                      \\ conclusion

  if (element? X (value *candy*))
  _______________________________
  X : candy;)
```

- Notice commenting syntax
- Shen was developed on Windows

Parsing

$a^{n>0}$

```
(defcc <as>
  a <as>;
  a;)
(compile (function <as>) [a a a])
  => [a a a]
(compile (function <as>) [a a b])
  => parse error
```

Parsing

$a^{n>0}b^{m>0}$

```
(defcc <bs>
  b <bs>;
  b;)

(defcc <asbs>
  <as> <bs>;)

(compile (function <asbs>) [a a a])
  => parse error
(compile (function <as>) [a a b])
  => [a a b]
```

Parsing

Vending Machine Grammar

```
(defcc <instruction>
  list <vending-machine-state>;)

(defcc <vending-machine-state>
  candy;
  money;)
```

Try it!

```
> list money
> list candy
```

Parsing

Vending Machine Grammar

```
(defcc <instruction>
  add <inputs>;)

(defcc <inputs> <currencies> := [[currency|[<currencies>]]];)

(defcc <currencies>
    <currency> <currencies>; <currency>;)

(defcc <currency> C := [C]
   where (element? C (value *currency*));)
```

Try it!

```
> add quarter dollar nickel
```

Parsing

Vending Machine Grammar

```
(defcc <instruction> ...;)
(defcc <sudo>
   sudo;
   := [user];)
(defcc <instructions>
  <sudo> <instruction> := (append <sudo> [<instruction>]);)
```

Parsing to an AST

```
(compile (function <instructions>) [add quarter dollar])
 => [user [add [currency [quarter dollar]]]]
(compile (function <instructions>) [sudo add quarter dollar])
 => [sudo [add [currency [quarter dollar]]]]
```

Internal Representation

Machine state

```
(@p [(@p snickers 2)
     (@p twix 20)
       ..]
    [(@p dollar 3)
     (@p quarter 10)
       ...])
```

Types

```
(synonyms state       (candyStore * coinStore)
          candyStore (list (candy * number))
          coinStore  (list (currency * number)))
```

Adding coins

Add instruction

```
> add quarter dollar
```

Add coin routine

```
(define add-coins
  { coinStore --> (list currency) --> coinStore }
  CoinStore []     -> CoinStore
  CoinStore Coins -> (add-coins
                         (add-coin CoinStore (head Coins))
                         (tail Coins)))
(define add-coin
  { coinStore --> currency --> coinStore}
  CoinStore Coin -> (with-key CoinStore Coin (+ 1)))
```

Adding Coins

Updating A Lookup Table

```
(define with-key
  { (list (A * B)) --> A --> (B --> B) --> (list (A * B)) }
  [(@p K V) | KVs] K F -> (append [(@p K (F V))]
                                  (with-key KVs K F))
  [KV | KVs] K F       -> (append [KV]
                                  (with-key KVs K F))
  [] K F               -> [])
```

Typing commands

Sample untyped commands from ‘defcc‘

```
> [sudo [add [currency [quarter dollar]]]]
> [user [list money]]
```

Typing a command

```
(datatype command-line

  ------------------
  [sudo X] : command-line;


  ------------------------
  [user X] : command-line;)
```

Processing a command

Command processor

```
(define process-request
  { state --> command-line --> state -->
    (string * state * state)}
  ...
  VM [sudo [list money]] US -> (@p (show-coins VM) VM US)
  VM [user [list money]] US -> (@p (show-coins US) VM US)
  ...
  )
```

Processing a command

Processing currencies

```
(define process-request
  { state --> command-line --> state -->
    (string * state * state)}
    VM [user [add [currency Currencies]]] US
              > (@p "Success."
                     VM
                    (@p (fst US)
                        (add-coins (snd US) Currencies)))
  )
```

Typing currency commands

Currency command type

```
(datatype currency

  ________________________
  [currency X] : blah;

  [currency X] : blah;
  ________________________
  X : (list currency);)
```

Generating types

Generating currency command type

```
(defmacro connector-type-macro
  [connect-type Name TypeA X TypeB] ->
    (let Connector (gensym connector-)
     [datatype Name

      _________________
      TypeA : Connector;

      TypeA : Connector;
       _________________
       X : (eval TypeB);]));
```

Generating types

Example Usage

```
(connect-type currency [currency X] X [list currency])
```

Type Generated

```
(datatype currency
  ____________________________
  [currency X] : connector-3047;

  [currency X] : connector-3047;
  _____________________________
   X : (list currency);)
```

Type Required

```
(datatype currency
  ___________________
  [currency X] : blah;

  [currency X] : blah;
  ___________________
   X : (list currency);)
```

Load Order

Currency Macro loading

```
(tc -)                                  \\ turn off typechecking
(defmacro connector-type-macro ...)     \\ declare macro
(tc +)                                  \\ turn on typechecking
(connect-type [currency X] ...)         \\ generate type
(define process-request ...)            \\ use type
```

Concurrency Layer

- Each connection gets a thread
- Each thread has a state

  `(candyStore * coinStore)`
- Each connection is stored globally in:

  `*connectionStore*`
- Synchronized via SBCL's mailbox

Concurrency Layer

Global Mailbox For Concurrency

```
(set *mailbox* (((protect READ-FROM-STRING)
                 "SB-CONCURRENCY::MAKE-MAILBOX")))
```

Sending/Receiving a message

```
(define send-message
  Mailbox Message -> (((protect READ-FROM-STRING)
                       "SB-CONCURRENCY::SEND-MESSAGE")
                           Mailbox Message))
(define receive-message
  Mailbox -> (((protect READ-FROM-STRING)
               "SB-CONCURRENCY::RECEIVE-MESSAGE")
                  Mailbox))
```

CL Interop

In Common Lisp

```
(package-name:function arg1 arg2 ...)
```

Same thing in Shen

```
(((protect READ-FROM-STRING) "PACKAGE-NAME::FUNCTION")
     arg1 arg2 ..)
```

- Couldn't figure out the macro

Network Layer

Wrapping socket listener

```
(define socket-listen
  Host Port -> (((protect READ-FROM-STRING)
                    "USOCKET::SOCKET-LISTEN")
                Host
                Port))
```

Calling from Shen

```
(define open-socket
  Host Port -> (let Sock (socket-listen Host Port)
                 (do (add-to *connectionStore* Sock)
                     Sock)))
```

Threading

Wrapping thread maker

```
(define make-thread
  Function -> (((protect READ-FROM-STRING)
          "SB-THREAD::MAKE-THREAD") Function))
```

Creating a thread

```
(make-thread
  (freeze (do (... request-response loop ...)
              (... close connection ....    )))
```