Semantic Lego

David Espinosa
Department of Computer Science
Columbia University
New York, NY 10027
espinosa@cs.columbia.edu

March 31, 1995

Semantic Lego

SL builds interpreters from parts.

- Based on denotational semantics
- Covers all of Schmidt (1986)
- Implemented in Scheme

Why? To study

- Languages
- Semantics
- Modularity
- Extensibility

Contributions

- Re-present programming languages as ADTs.
- Re-present Moggi's theory of lifting.
- Develop two styles of hand-written modular interpreters.
- Develop theory of stratification (extends Mosses).
- Implement theory of lifting.
- Implement theory of stratification.

Outline

- * Usual interpreters aren't modular
- Modular interpreters by hand
- Modular interpreters by machine
- Technical aspects
- Conclusion

Languages as ADTs

$$
\begin{array}{lcll}
\texttt{compute} & : & \mathit{Exp} & \to \mathit{Ans} \\
\\
\texttt{\%num} & : & \mathit{Num} & \to \mathit{Exp} \\
\texttt{\%+} & : & \mathit{Exp} \times \mathit{Exp} & \to \mathit{Exp} \\
\\
\texttt{\%abort} & : & \mathit{Exp} & \to \mathit{Exp} \\
\\
\texttt{\%var} & : & \mathit{Name} & \to \mathit{Exp} \\
\texttt{\%lambda} & : & \mathit{Name} \times \mathit{Exp} & \to \mathit{Exp} \\
\texttt{\%call} & : & \mathit{Exp} \times \mathit{Exp} & \to \mathit{Exp}
\end{array}
$$

```
(compute
  (%call (%lambda 'x (%+ (%var 'x) (%var 'x)))
         (%abort (%num 5))))
=> 5
```

An implementation

```
;; Den  = Env -> Cont -> Ans
;; Proc = Val -> Cont -> Ans
;; Cont = Val -> Ans

(define (compute den)
  ((den (empty-env)) val->ans))

(define (((%num n) env) k)
  (k n))

(define (((%+ d1 d2) env) k)
  ((d1 env)
   (lambda (v1)
     ((d2 env)
      (lambda (v2) (k (+ v1 v2)))))))

(define (((%var name) env) k)
  (k (env-lookup env name)))

(define (((%lambda name den) env) k)
  (k (lambda (val) (den (env-extend env name val)))))

(define (((%call d1 d2) env) k)
  ((d1 env)
   (lambda (v1)
    ((d2 env)
     (lambda (v2)
       ((v1 v2) k))))))

(define (((%abort den) env) k)
  ((den env) val->ans))
```

Problems

Not modular!

- Each construct involves all modules.
- Hard to understand and reason about.
- Definitions are too specialized.

Outline

- Usual interpreters aren't modular
- * Modular interpreters by hand
- Modular interpreters by machine
- Technical aspects
- Conclusion

Computation ADTs

Split language definitions into two parts:

Computation ADT defines the “basic semantics”,
a type of denotations and operators on it.

Language ADT defines the actual constructs, as before.

Reference: Mosses

Stratification

Computation ADT is built from:

- a set of levels
- a set of lifting operators relating levels

Levels

Type constructors:

$$
\begin{aligned}
E(A) &= Env \to (A \to Ans) \to Ans \\
C(A) &= (A \to Ans) \to Ans \\
V(A) &= A
\end{aligned}
$$

Lifting operators 1

Monad = (T, `unit`, `bind`)

Type constructor T

Families of maps:

`unit` : $A \to T(A)$
`bind` : $(A \to T(B)) \to (T(A) \to T(B))$

```
bind(unit)          = id
bind(f) o unit      = f
bind(g) o bind(f) = bind(bind(g) o f))
```

Lifting operators 2

Monad = (T, `unit`, `join`)

Endofunctor T:

`map` : $(A \to B) \to (T(A) \to T(B))$

Natural transformations:

`unit` : $A \to T(A)$
`join` : $T(T(A)) \to T(A)$

```
join o unit       = id
join o map(unit) = id
join o map(join) = join o join
```

Monad examples 1

```
;; T(A) = List(A)

(define (unit a)
  (list a))

(define ((map f) ta)
  (list-map f ta))

(define ((bind f) ta)
  (append-map f ta))

(define (join tta)
  (reduce append '() tta))
```

Monad examples 2

```
;; T(A) = Env -> A

(define ((unit a) env)
  a)

(define (((map f) ta) env)
  (f (ta env)))

(define (((bind f) ta) env)
  ((f (ta env)) env))

(define ((join tta) env)
  ((tta env) env))
```

Monads relate levels

A monad T *relates* L_1 to L_2 if

$$L_2 = T \circ L_1$$

- V is related to C by the continuation monad.
- C is related to E by the environment monad.
- V is related to E by a combined monad.

Example monads

Monad	Action $T(A) =$
Identity	A
Lists	$List(A)$
Lifting	$1 \to A$
Environments	$Env \to A$
Stores	$Sto \to A \times Sto$
Exceptions	$A + X$
Monoids	$A \times M$
Continuations	$(A \to Ans) \to Ans$
Resumptions	$\mathit{fix}(X)\ (A + X)$

Monads in use

```
(define (%+ d1 d2)
  (bindVE d1
    (lambda (n1)
      (bindVE d2
        (lambda (n2)
          (unitVE (+ n1 n2)))))))
```

Computation ADT

```
;; E(A) = Env -> C(A)
;; C(A) = (A -> Ans) -> Ans
;; V(A) = A

;; unitAB : A -> B
;; bindAB : B * (A -> B) -> B

(define ((unitVC v) k)
  (k v))

(define ((unitCE c) env)
  c)

(define (unitVE v)
  (unitCE (unitVC v)))

(define ((bindVC c f) k)
  (c (lambda (v) ((f v) k))))

(define ((bindCE e f) env)
  ((f (e env)) env))

(define ((bindVE e f) env)
  (bindVC (e env)
    (lambda (v) ((f v) env))))
```

Language ADT

```
;; Proc = V -> C

(define (compute den)
  ((den (empty-env)) val->ans))

(define (%num n) (unitVE n))

(define (%+ d1 d2)
  (bindVE d1
    (lambda (n1)
      (bindVE d2
        (lambda (n2)
          (unitVE (+ n1 n2)))))))

(define ((%var name) env)
  (unitVC (env-lookup env name)))

(define ((%lambda var den) env)
  (unitVC (lambda (val) (den (env-extend env var val)))))

(define (%call d1 d2)
  (bindVE d1
    (lambda (p)
      (bindVE d2 (lambda (a) (unitCE (p a)))))))

(define (%abort den)
  (bindCE den
    (lambda (c)
      (unitCE (lambda (k) (c val->ans))))))
```

Outline

- Usual interpreters aren't modular
- Modular interpreters by hand
- * Modular interpreters by machine
- Technical aspects
- Conclusion

Kitchen sink language

`compute`	: Exp	$\to \mathit{Ans}$
`%unit`	: $\mathit{1}$	$\to \mathit{Exp}$
`%true`	: $\mathit{1}$	$\to \mathit{Exp}$
`%false`	: $\mathit{1}$	$\to \mathit{Exp}$
`%num`	: Num	$\to \mathit{Exp}$
`%+`, `%*`	: $\mathit{Exp} \times \mathit{Exp}$	$\to \mathit{Exp}$
`%=?`, `%>?`	: $\mathit{Exp} \times \mathit{Exp}$	$\to \mathit{Exp}$
`%or`, `%and`	: $\mathit{Exp} \times \mathit{Exp}$	$\to \mathit{Exp}$
`%not`	: Exp	$\to \mathit{Exp}$
`%number?`	: Exp	$\to \mathit{Exp}$
`%boolean?`	: Exp	$\to \mathit{Exp}$
`%callcc`	: Exp	$\to \mathit{Exp}$
`%amb`	: $\mathit{Exp} \times \mathit{Exp}$	$\to \mathit{Exp}$
`%fail`	: $\mathit{1}$	$\to \mathit{Exp}$
`%while`	: $\mathit{Exp} \times \mathit{Exp}$	$\to \mathit{Exp}$
`%if`	: $\mathit{Exp} \times \mathit{Exp} \times \mathit{Exp}$	$\to \mathit{Exp}$
`%begin`	: $\mathit{Exp} \times \mathit{Exp}$	$\to \mathit{Exp}$
`%store`	: $\mathit{Loc} \times \mathit{Exp}$	$\to \mathit{Exp}$
`%fetch`	: Loc	$\to \mathit{Exp}$
`%var`	: Name	$\to \mathit{Exp}$
`%lambda`	: $\mathit{Name} \times \mathit{Exp}$	$\to \mathit{Exp}$
`%let`	: $\mathit{Name} \times \mathit{Exp} \times \mathit{Exp}$	$\to \mathit{Exp}$
`%fix`	: Exp	$\to \mathit{Exp}$
`%rec`	: Exp	$\to \mathit{Exp}$
`%letrec`	: $\mathit{Name} \times \mathit{Exp} \times \mathit{Exp}$	$\to \mathit{Exp}$
`%write`	: Exp	$\to \mathit{Exp}$
`%error`	: String	$\to \mathit{Exp}$
`%pair`	: $\mathit{Exp} \times \mathit{Exp}$	$\to \mathit{Exp}$
`%left`	: Exp	$\to \mathit{Exp}$
`%right`	: Exp	$\to \mathit{Exp}$

Kitchen sink example program

```
(compute
 (%begin
  (%store 'n (%amb (%num 4) (%num 5)))
  (%store 'r (%num 1))
  (%callcc
   (%lambda 'exit
    (%letrec 'loop
      (%lambda 'u
        (%begin
          (%if (%zero? (%fetch 'n))
               (%call (%var 'exit) (%fetch 'r))
               (%unit))
          (%write (%pair (%fetch 'n) (%fetch 'r)))
          (%store 'r (%* (%fetch 'r) (%fetch 'n)))
          (%store 'n (%- (%fetch 'n) (%num 1)))
          (%call (%var 'loop) (%unit))))
      (%call (%var 'loop) (%unit)))))))

(value: (24 120)
 output: ((pair 4 1) (pair 3 4) (pair 2 12) (pair 1 24)
          (pair 5 1) (pair 4 5) (pair 3 20)
          (pair 2 60) (pair 1 120)))
```

Language specifications

Input:

- A list of (predefined) semantic modules

Output:

- An interpreter

Interpreter is used to

- Execute programs
- Produce semantics via program simplifier

Kitchen sink specification

```
(define computations
  (construct-type
   cbn-environments
   lifting1
   stores
   continuations1
   lists
   output
   errors))

(show-computations)
(-> Env
   (Lift (-> Sto
             (Let A0 (+ (* (List Ans) Out) Err)
               (-> (-> (* Val Sto) A0) A0)))))

(load "error-exceptions" "numbers" "booleans" "products"
      "procedures" "environments" "begin" "stores" "while"
      "callcc" "amb" "output" "fix")
```

Generic %amb

```
(define %amb
  (let ((unit (get-unit 'lists 'top))
        (bind (get-bind 'lists 'top)))
    (lambda (x y)
      (bind x
        (lambda (x)
          (bind y
            (lambda (y)
              (unit (append x y)))))))))
```

Generic %callcc

```
(define %callcc
  (let ((mapC (get-map 'conts 'top))
        (mapK (get-map 'conts 'env-results))
        (iunitK (get-iunit 'conts 'env-results))
        (unitE (get-unit 'env-values 'env-results))
        (unitP (get-value-unit 'procedures 'env-values))
        (unitR (get-unit 'cont-values 'env-results))
        (bindS (get-value-bind 'procedures 'env-results)))

    (define (tilt cv f)
      (iunitK (bindS (unitR cv) f)))

    (lambda (exp)
      (mapC exp
        (lambda (cont)
          (lambda (k)

            (define (callcc-proc v)
              (mapK (unitE v)
                    (lambda (cont)
                      (lambda (k1) (cont k)))))

            (cont
             (lambda (cv)
               ((tilt cv
                      (lambda (p)
                        (p (unitP callcc-proc))))
                k)))))))))
```

Simplified constructs

$$Den = Env \to (Val \to List(Ans)) \to List(Ans)$$

```
(define (%amb x y)
  (lambda (env)
    (lambda (k)
      (reduce append ()
        (map k (append ((x env) list)
                       ((y env) list)))))))

(define (%callcc den)
  (lambda (env)
    (lambda (k)
      (define (callcc-proc v)
        (lambda (k1) (k v)))
      ((den env)
       (lambda (cv)
         (if (is? 'procedures cv)
             (((value cv)
               (in 'procedures callcc-proc)) k)
             (k (in 'errors
                    (type-error
                     'procedures (type cv))))))))))
```

Resumption example

$$Den = \mathit{fix}(X)\ Sto \to List((Val + X) \times Sto)$$

```
(define computations
  (make-computations resumptions stores lists))

(compute
 (%par (%num 1) (%num 2) (%num 3)))

(3 3 2 2 1 1)

(compute
 (%seq
  (%store 'x (%num 1))
  (%store 'go (%true))
  (%par
   (%store 'go (%false))
   (%while (%and (%fetch 'go)
                 (%< (%fetch 'x) (%num 7)))
     (%pause (%store 'x (%1+ (%fetch 'x))))))
  (%fetch 'x)))

(1 2 3 4 5 6 7 7)
```

Outline

- Usual interpreters aren't modular
- Modular interpreters by hand
- Modular interpreters by machine
- * Technical aspects
- Conclusion

Monads don't compose

Suppose we have:

$$\begin{aligned}\texttt{unitS} : Id &\to S\\ \texttt{unitT} : Id &\to T\\ \texttt{joinS} : SS &\to S\\ \texttt{joinT} : TT &\to T\end{aligned}$$

We want:

$$\texttt{joinST} : STST \to ST$$

Can't define it!

Monad transformers

Transformer	Action $F(T)(A) =$
Identity	$T(A)$
Lists	$T(List(A))$
Lifting 1	$1 \to T(A)$
Lifting 2	$T(1 \to A)$
Environments	$Env \to T(A)$
Stores	$Sto \to T(A \times Sto)$
Exceptions	$T(A + X)$
Monoids	$T(A \times M)$
Continuations	$(A \to T(Ans)) \to T(Ans)$
Resumptions	$\mathit{fix}(X)\ T(A + X)$

Monad transformer types

Type	Form	Examples
Top	$F(T) = S \circ T$	Environments
Bottom	$F(T) = T \circ U$	Lists, Monoids
Around	$F(T) = S \circ T \circ U$	Stores

Stratified example 1

Let's build:

$$Den = Env \to Sto \to List(Val \times Sto)$$

Start with:

$$L_1(A) \; = \; A$$

$$T_{11}(A) \; = \; A$$

Stratified example 2

Apply $F(T)(A) = T(List(A))$:

$$\begin{aligned} L_2(A) &= List(A) \\ L_1(A) &= A \end{aligned}$$

$$T_{12}(A) = List(A)$$

Stratified example 3

Apply $F(T)(A) = Sto \to T(A \times Sto)$:

$$\begin{aligned}
L_4(A) &= Sto \to List(Sto \times A) \\
L_3(A) &= List(Sto \times A) \\
L_2(A) &= Sto \times A \\
L_1(A) &= A
\end{aligned}$$

$$\begin{aligned}
T_{34}(A) &= Sto \to A \\
T_{23}(A) &= List(A) \\
T_{24}(A) &= Sto \to List(A) \\
T_{14}(A) &= Sto \to List(Sto \times A)
\end{aligned}$$

Stratified example 4

Apply $F(T)(A) = Env \to T(A)$:

$$
\begin{aligned}
L_5(A) &= Env \to Sto \to List(Sto \times A) \\
L_4(A) &= Sto \to List(Sto \times A) \\
L_3(A) &= List(Sto \times A) \\
L_2(A) &= Sto \times A \\
L_1(A) &= A
\end{aligned}
$$

$$
\begin{aligned}
T_{45}(A) &= Env \to A \\
T_{34}(A) &= Sto \to A \\
T_{23}(A) &= List(A) \\
T_{35}(A) &= Env \to Sto \to A \\
T_{24}(A) &= Sto \to List(A) \\
T_{25}(A) &= Env \to Sto \to List(A) \\
T_{14}(A) &= Sto \to List(Sto \times A) \\
T_{15}(A) &= Env \to Sto \to List(Sto \times A)
\end{aligned}
$$

Compatibility laws

```
ST  = S o T
map = mapS o mapT

unitST = unitS o unitT
       = mapS(unitT) o unitS : Id -> ST

joinST o mapST(unitS) = mapS(joinT) : STT -> ST
joinST o mapS(unitT)  = joinS       : SST -> ST

joinS o mapS(joinS)
      = joinST o joinS       : SSTST -> ST

joinST o mapST(mapS(joinT))
      = mapS(joinT) o joinST : STSTT -> ST
```

References: Beck, Barr

Stratified monads

Categories in which:

- Objects = levels (type constructors)
- Arrows = monads T with $L_2 = T \circ L_1$
- Composition respects compatibility.
- All diagrams commute.
- Distinguished levels *Top* and *Bot* that are maximal and minimal.
- Distinguished monad T from *Bot* to *Top*.

Outline

- Usual interpreters aren't modular
- Modular interpreters by hand
- Modular interpreters by machine
- Technical aspects
- * Conclusion

Limitations

SL builds basic denotational models.

- Doesn't do compilation or abstract interpretation.
- Doesn't help with static aspects (types, syntax).
- Doesn't help with non-semantic aspects (unification, constraint solving).

Future work

- Extend to do compilation and abstract interpretation.
- Develop modular systems for reasoning about programs.

Previous work 1

ADTs in semantics:

- Goguen, Thatcher, Wagner, and Wright, "Initial algebra semantics and continuous algebras", 1977.
- Mosses, "Action Semantics", 1983 – 92.

Monadic interpreters:

- Moggi, "Computational lambda calculus and monads", 1989.
- Moggi, "Notions of computation and monads", 1991.
- Wadler, "The essence of functional programming", 1992.

Composing monads:

- Beck, "Distributive laws", 1969.
- Barr and Wells, "Triples, Toposes, and Theories", 1985.
- King and Wadler, "Combining monads", 1992.
- Jones and Duponcheel, "Composing monads", 1993.

Previous work 2

Compound interpreters:

- Moggi, "An abstract view of programming languages", 1989.
- Moggi, "A modular approach to denotational semantics", 1991.
- Moggi and Cenciarelli, "A syntactic approach to modularity in denotational semantics", 1993.
- Cartwright and Felleisen, "Extensible Denotational Language Specifications", 1994.
- Steele, "Building interpreters by composing monads", 1994.
- Espinosa, "Semantic Lego", 1994.
- Espinosa, "Stratified Monads", 1994.
- Liang, Hudak, and Jones, "Monad transformers and modular interpreters", 1995.
- Espinosa, Ph.D. thesis, 1995.