Building Interpreters by Transforming Stratified Monads

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

June 1994

Abstract

This paper shows how to construct programming language interpreters from a set of mix-and-match parts. By composing a sequence of semantic modules, we form an abstract data type (ADT) of computations, which is then used to build an ADT of language constructs. We represent the ADT of computations by a *stratified monad* and the modules by stratified monad *transformers*. These results extend previous work on monads and have applications to language extensibility, interpreter construction, and the study of semantic models.

Topics: interpreters, functional programming
Word count: 4200 (8 pages) without examples, figures, and appendices

1 Introduction

We describe Semantic Lego, an interpreter construction toolkit capable of building interpreters for a wide variety of languages from a set of reusable parts. By representing a modular theory of language design in computational terms, we raise the level of dialogue between the designer and the computer [AKHS88]. Language designers can use the toolkit to prototype new languages, to experiment with new semantics, and to build interpreters for use in other systems.

To see the lack of modularity in interpreters, consider adding stores to a language without them. Even though most language constructs leave the store unchanged, it is necessary to rewrite the interpreter completely. Tools such as attribute grammars [DJL88] and abstract semantic algebras [Mos92] were developed to solve this problem but cannot handle complex semantics. Using the toolkit, we simply add the stores module to the language specification.

To achieve these results, we reformulate Moggi's theory of modular denotational semantics [Mog89a, Mog91a]. We introduce *stratified monads*, sets of named semantic levels, related in pairs by monads. We define language constructs using the monad operators conceptually relevant to them, independent of other details of the semantics. We realize semantic modules, such as stores and continuations, as stratified monad *transformers*.

This paper is a significant advance over previous work that uses monads as a basis for modular language specification. Specifically,

- Moggi [Mog89a] presents a complex formalism that is often difficult to understand. We introduce a simpler and more flexible theory. By representing it in computational terms, we render it both more useful and more accessible.
- Wadler [Wad92] builds interpreters from two parts, a base and an extension. Our theory is more general, allowing us to compose languages from any number of semantic modules.
- Steele [Ste94] uses pseudomonads to build a limited but complex interpreter toolkit. Our work is simpler and handles a wider range of constructions.

The paper is organized as follows. Section 2 presents a series of examples, section 3 discusses how the toolkit works, and section 4 compares it to previous work. We assume an elementary understanding of denotational semantics and functional programming; for further background, see [Wad92]. We present all examples and code fragments in Scheme [CR91].

2 Examples

We discuss languages as abstract data types, show the toolkit in action, and use it to explore the interaction between nondeterminism and continuations.

2.1 Languages as ADTs

To see what the toolkit does, we need to specify what "generating an interpreter" means. Most authors define a function such as

```
eval : Exp * Env -> Val
```

where `Exp` is an ADT of expressions with constructors

```
var : Name -> Exp
lam : Name * Exp -> Exp
app : Exp * Exp -> Exp
...
```

This approach does not express the point of denotational semantics, which is that `eval` should map expressions to denotations. Thus, it is better to write

```
eval    : Exp -> Den
compute : Den -> Val
Den     = Env -> Val
```

where `compute` is used to execute programs. Furthermore, denotational semantics is compositional, which means that the denotations of expressions are functions of the denotations of their subexpressions. We can represent these functions directly as

```
%var : Name -> Den
%lam : Name * Den -> Den
%app : Den * Den -> Den
...
```

These operations form an abstract data type (ADT) of denotations [GTWW77]. We trivially define `eval` as

```
(eval (var v))      = (%var v)
(eval (lam v body)) = (%lam v (eval b))
(eval (app f a))    = (%app (eval f) (eval a))
...
```

A sample program execution now reads

```
(compute
  (eval (app (lam 'x (add (var 'x) (var 'x)))
             (num 5))))
⇒ 10
```

However, we could just as well write

```
(compute
  (%app (%lam 'x (%add (%var 'x) (%var 'x)))
        (%num 5)))
⇒ 10
```

and eliminate `eval` and its associated syntax constructors. In this way, we evaluate programs using the syntax of the language in which the ADT of denotations is embedded.

So, what we mean by "an interpreter" is an implementation of an ADT of denotations for a language. Along with the implementations of ADT operators, it is easy to generate an `eval` function, an ADT of expressions, and a user interface, but there is no *a priori* need. In algebraic terms, initiality of syntax yields a unique homomorphism to semantics [GTWW77].

In the rest of the paper, we usually refer to the ADT of denotations as an ADT of *computations*. This change is (roughly) in accord with the existing literature on monads.

2.2 An example

We construct an interpreter for a language with environments, call-by-value procedures, stores, continuations, nondeterminism, and errors. Figure 1 shows the complete language specification, the type of computations, and two example expressions. Types written by hand are in infix form as usual, but machine-generated types are in prefix, using `let` for abbreviation. For example,

```
(let A0 (* a sto) (-> A0 A0))
```

means `A * Sto -> A * Sto`.

To build an interpreter, we define an ADT of computations using `make-computations`. This procedure accepts a list of semantic modules, implemented as stratified monad transformers. It composes them and applies the result to the identity stratified monad. The operators of the resulting stratified monad form the desired ADT. `Make-computations` is defined as

```
(define (make-computations . transformers)
  ((apply compose transformers) (make-identity-sm)))
```

Next, we load several files of language constructs. These extract operators from the computations ADT and use them to define the language ADT. Constructs may be defined over any stratified monad that includes the appropriate semantic modules. For example, the `%amb` construct requires the `nondeterminism` module. In general, the same construct definition yields different semantics when applied to different stratified monads.

Finally, we illustrate the behavior of the interpreter by evaluating several expressions. Names of language constructs begin with percent signs to avoid conflict with Scheme.

A typical construct is `%let`, whose definition (from `cbv-environments`) is shown in Figure 2. We can only interpret `%let` over a stratified monad that defines the levels `top`, `envs`, and `env-values`. All stratified monads have levels `top` and `bottom`, which refer respectively to computations and values. Furthermore, all monads built using the `environments` transformer have `envs` and `env-results`, which are related by

```
envs = Env -> env-results
```

In words, `%let` reduces `c1` to the `env-value v1` using `bindV`, reduces `c2` to the `envs e2` using `bindE`, and then returns, using `unitE`, the `envs` that accepts an environment E and applies `e2` to E extended with `v1`.

Although Scheme procedures are usually opaque, MIT Scheme allows us to reify them as abstract syntax. To see the results that the toolkit produces, we apply a program simplifier that performs inlining and β and η reduction. The result of simplifying `%let` in the context of the specified ADT of computations, shown in Figure 3, is exactly what we would have written by hand. The whole point of the toolkit is that the source definition of `%let` did not mention stores or continuations, yet they were introduced properly and automatically.

2.3 Amb and call/cc

In this section, we use the toolkit to explore interactions between non-determinism and continuations. We use three different computation types but leave the definitions of all language constructs unchanged. For reference, Figure 4 gives the source definition of `%amb`. For each semantics, we show the modules forming the type of computations, the type itself, the simplified version of `%amb`, and the evaluation of an example program.

In the first semantics (Figure 5), the subexpressions of `%amb` are run with `list` as a continuation. The results are appended and returned. In the example, the `list` continuation is replaced by a continuation that adds one, hence the result `51`. This semantics actually results from using Scheme's `call/cc` with the `amb` derived from Filinski's monadic reflection operators [Fil94] over the list monad. Monadic reflection is not limited to this semantics; to obtain others, we can reflect over the monad

```
TA = (A -> List A) -> List A
```

and define our own `call/cc`.

In the second semantics (Figure 6), we replace `continuations` with `continuations2`. These modules differ only in their treatment of operators on continuation answers. The `continuations` transformer passes down an identity continuation, applies the operator to the results, and then applies `k` (in the appropriate way). `Continuations2` passes `k` down

```
;; ADT of computations

(define computations
  (make-computations
    environments stores continuations nondeterminism errors))

(set-computations! computations)

;; ADT of language constructs

(load "error-exceptions" "numbers" "booleans" "numeric-predicates" "amb"
      "cbv-static" "cbv-environments" "stores" "while" "cbv-callcc")

;; ADT implementation type

(get-type 'bottom 'top)

⇒ (-> env
       (-> sto
           (let A0 (* a sto)
             (let A1 (+ (list A0) errors)
               (-> (-> A0 A1) A1)))))

;; Sample expressions

(compute
 (%call (%lambda 'x (%+ (%var 'x) (%var 'x)))
        (%amb (%num 1) (%num 2))))

⇒ (2 4)    ; would be (2 3 3 4) in call-by-name

(compute
 (%begin
  (%store 'n (%amb (%num 4) (%num 5)))
  (%store 'r (%num 1))
  (%call/cc
   (%lambda 'exit
    (%while (%true)
      (%begin
        (%if (%zero? (%fetch 'n))
             (%call (%var 'exit) (%fetch 'r))
             (%skip))
        (%store 'r (%* (%fetch 'r) (%fetch 'n)))
        (%store 'n (%- (%fetch 'n) (%num 1)))))))))

⇒ (24 120)
```

Figure 1: Example specification and expressions

```
(define %let
  (let ((unitE (get-unit 'envs 'top))
        (bindE (get-bind 'envs 'top))
        (bindV (get-bind 'env-values 'top)))
    (lambda (name c1 c2)
      (bindV c1
        (lambda (v1)
          (bindE c2
            (lambda (e2)
              (unitE
               (lambda (env)
                 (e2 (env-extend env name v1)))))))))))
```

Figure 2: `%let` source definition

```
(lambda (name c1 c2)
  (lambda (env)
    (lambda (sto)
      (lambda (k)
        (((c1 env) sto)
         (lambda (a)     ; val * sto
           (((c2 (env-extend env name (left a))) (right a)) k)))))))
```

Figure 3: Result of simplifying `%let`

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

Figure 4: `%amb` source definition

```
;; ADT of computations

  (define computations
    (make-computations environments continuations nondeterminism))

;; ADT implementation type

  (-> env (let A0 (list val) (-> (-> val A0) A0)))

;; Simplified %amb

  (lambda (x y)
    (lambda (env)
      (lambda (k)
        (reduce append ()
                (map k (append ((x env) list) ((y env) list)))))))

;; Example

  (compute
   (%+ (%num 1)
       (%call/cc
        (%lambda 'k
         (%* (%num 10)
             (%amb (%num 3) (%call (%var 'k) (%num 4))))))))

  ⇒ (31 51)
```

Figure 5: `%amb` version 1

directly and applies the operator to the results. The evaluation of the example in this semantics is clear.

In the third semantics (Figure 7), we compose the `continuations` and `nondeterminism` modules in the opposite order. Here, continuations accept lists of values, rather than just values. `%amb` takes two lists, appends them, and continues with the result. In the example, invoking the captured continuation aborts this process and returns `4` directly. Hence, the expression has only one value in contrast to the other two semantics. Of the semantics presented here, this is the only one that Steele's system can generate [Ste94]. Incidentally, replacing `continuations` with `continuations2` leaves `%amb` unchanged.

3 Methods

This section explains how the toolkit works. We consider monads, transformation versus composition, stratified monads, and the definition of language constructs over stratified monads.

```
;; ADT of computations

  (define computations
    (make-computations environments continuations2 nondeterminism))

;; ADT implementation type

  (-> env (let A0 (list val) (-> (-> val A0) A0)))

;; Simplified %amb

  (lambda (x y)
    (lambda (env)
      (lambda (k)
        (append ((x env) k) ((y env) k)))))

;; Example

  (compute
   (%+ (%num 1)
       (%call/cc
        (%lambda 'k
         (%* (%num 10)
             (%amb (%num 3) (%call (%var 'k) (%num 4))))))))

  ⇒ (31 5)
```

Figure 6: %amb version 2

```
;; ADT of computations

  (define computations
    (make-computations environments nondeterminism continuations))

;; ADT implementation type

  (-> env (let A0 (list val) (-> (-> A0 A0) A0)))

;; Simplified %amb

  (lambda (x y)
    (lambda (env)
      (lambda (k)
        ((x env)
         (lambda (a)
           ((y env)
            (lambda (a0)
              (k (append a a0)))))))))

;; Example

  (compute
   (%+ (%num 1)
       (%call/cc
        (%lambda 'k
         (%* (%num 10)
             (%amb (%num 3) (%call (%var 'k) (%num 4))))))))

  ⇒ (5)
```

Figure 7: %amb version 3

3.1 Monads

We briefly review monads; for a longer introduction, see [Wad92]. For our purposes, a monad is type constructor `T` along with operators

```
unit : A -> TA
bind : TA * (A -> TB) -> TB
compute : TA * (A -> Rep) -> Rep
```

where `Rep` is a fixed type of observable representations. The notation `TA` means the type constructor `T` applied to the type `A`. In general, juxtaposition denotes application (rather than composition) and associates to the left. Hence `FTA`, which appears in the next section, is equivalent to the Scheme expression `((F T) A)`.

The intuition behind the use of monads in semantics, due to Moggi [Mog89b], is that `TA` is the type of computations over the values `A`. `Unit` lifts values to computations, `bind` lifts functions on values to functions on computations, and `compute` lifts representors of values to representors of computations. Monads capture a wide variety of "notions of computation" [Mog89b].

We can lift operators on values to operators on computations using `unit` and `bind`, and we can define new operators on computations directly. For example, Figure 8 defines the environment monad, lifts the operator `+v` (add values) to `+c` (add computations), and defines a new variable reference operator.

Monads are usually required to satisfy several identities, especially

```
(bind (unit a) f) = (f a)
```

which means that `bind` lifts functions *through* `unit`. Our constructions usually satisfy these identities, but we do not require them, since our purpose is to build interpreters, not to reason about them. We leave for future work the investigation of principles for reasoning about modularly constructed languages.

3.2 Monad transformers

Although type constructors compose, monads do not (at least not without help — see [JD93]). To see why (informally), we compose the type constructors of two different environment monads. Figure 9 shows the unique monad that this composition can support, but its `bind` operator cannot be formed by composition from the two environment monads. In fact, Jones and Duponcheel [JD93] prove rigorously that monads cannot in general compose.

However, monads do *transform*. Figure 10 shows the environment monad transformer, which takes a monad `T` into a monad `FT`. Applying two (different) environment monad transformers to the identity monad indeed yields the double environment monad. Other monad transformers are listed in Table 1. Notice that composition of transformers is not commutative. The use of monad transformers in semantics originates with Moggi [Mog89a, Mog91a].

3.3 Stratified monads

Suppose we build a monad of computations using several monad transformers. The monad allows access to the top and bottom levels (computations and values) but not to levels

```
;; Environment monad: TA = Env -> A

(define (unit v)
  (lambda (env) v))

(define (bind c f)
  (lambda (env) ((f (c env)) env)))

(define (compute c f)
  (f (c empty-env)))

;; Lift +v to +c

(define (+c x y)
  (bind x (lambda (vx)
            (bind y (lambda (vy)
                      (unit (+v vx vy)))))))

;; Variable reference operator

(define (%var name)
  (lambda (env) (env-lookup env name)))
```

Figure 8: Environment monad and example operators

```
;; TA = EnvX -> EnvY -> A

(define (unit v)
  (lambda (envX) (lambda (envY) v)))

(define (bind c f)
  (lambda (envX) (lambda (envY) (((f ((c envX) envY)) envX) envY))))

(define (compute c f)
  (f ((c empty-envX) empty-envY)))
```

Figure 9: Double environment monad

```
;; FTA = Env -> TA

(define (environment-monad-transformer T)
  (let ((unitT (monad-unit T))
        (bindT (monad-bind T))
        (computeT (monad-compute T)))

  (define (unit v)
    (lambda (env) (unitT v)))

  (define (bind c f)
    (lambda (env) (bindT (c env) (lambda (a) ((f a) env)))))

  (define (compute c f)
    (computeT (c empty-env) f))

  (make-monad unit bind compute)))
```

Figure 10: Environment monad transformer

Transformer	Action on types
State	`FTA = Sto -> T(A*Sto)`
Output	`FTA = T(A*Monoid)`
Exceptions	`FTA = T(A + X)`
Resumptions	`FTA = fix(X, T(A + X))`
Environments	`FTA = Env -> TA`
Continuations	`FTA = (A -> TA) -> TA`
Nondeterminism	`FTA = T(List A)`

Table 1: Monad transformers

between. Without access to intermediate levels, we cannot define many language constructs. A possible solution, due to Moggi [Mog89a], is to interleave the definition of operators with the application of transformers. We arrange for transformers to transform *operators* as well as monads [Esp94]. However, it is still difficult (or impossible) to define constructs that interact with multiple levels. For example, if we define `%call/cc` in the continuations module, it is difficult for it to interact with the environments module (to prepare arguments to procedures).

A better solution is to construct multiple monads, where each provides access to a different level. In fact, we require a monad relating each pair of levels. Using these, it is possible to define language constructs that involve several levels, simply by using the appropriate operators. Also, we can define language constructs *after* building the monads. Thus, we separate language specifications into a computations ADT (the monad operators) and a language ADT (the language constructs).

We now present a more formal definition of stratified monads, starting from several subparts. A *level* is a pair of a type constructor and a set of names for the level. As with monads, the type constructor builds a type of computations *at that level* from a type of values. The top half of Table 2 shows the levels for the language of Figure 1.

Names are identifiers that allow language constructs to refer to levels. Essentially, they are the wiring that connects the language ADT to the computations ADT. Levels may have multiple names because conceptually distinct levels may coincide. For instance, level 6 of Table 1 is known to store constructs as `stores` and to environment constructs as `env-results`. Of course, distinct levels must have distinct names.

Contrary to appearance, levels are *not* ordered and are referenced only by their names. Also, although it seems that a stratified monad can have only a single level of environments (or stores, etcetera), we can easily remove this restriction by parametrizing both semantic modules and language constructs by the names used to connect them.

A monad `T` *relates* levels `X` below and `Y` above if `YA = T(XA)`. Then

```
unit : XA -> YA
bind : YA * (XA -> YB) -> YB
```

rewrites to

```
unit : A' -> TA'
bind : TA' * (A' -> TB') -> TB'
```

for `A' = XA` and `B' = XB`. This formulation shows that `T` legitimately relates computations at levels `X` and `Y` over different base types `A` and `B`.

A *situated monad* is a triple of a monad and two levels related by the monad. A *stratified monad* is a set of situated monads and two distinguished levels with the names `bottom` and `top` (among other names, possibly). The operators `get-unit` and `get-bind` take two names and extract the respective operator from the monad relating the levels having those names. Table 2 shows the structure of the stratified monad defined in Figure 1. Notice that not all pairs of levels have monads defined for them.

3.4 Stratified monad transformers

Stratified monad transformers are similar to the ordinary monad transformers. Their actions are unrestricted, except that they must return a stratified monad. Table 3 shows how the

Level	Names	Constructor T(val) =
1	env-values, store-values, bottom	val
2	store-pairs, cont-values	(* val sto)
3	lists	(list (* val sto))
4	cont-answers, errors	(+ (list (* val sto)) errors)
5	store-results, conts	(let A0 (* val sto) (let A1 (+ (list A0) errors) (-> (-> A0 A1) A1)))
6	env-results, stores	(-> sto (let A0 (* val sto) (let A1 (+ (list A0) errors) (-> (-> A0 A1) A1))))
7	envs, top	(-> env (-> sto (let A0 (* val sto) (let A1 (+ (list A0) errors) (-> (-> A0 A1) A1)))))

Bot	Top	Monad TA =
1	1	a
2	2	a
3	3	a
2	3	(list a)
4	4	a
3	4	(+ a errors)
2	4	(+ (list a) errors)
5	5	a
4	5	(-> cont a)
3	5	(-> cont (+ a errors))
2	5	(-> cont (+ (list a) errors))
6	6	a
5	6	(-> sto a)
4	6	(-> sto (-> cont a))
3	6	(-> sto (-> cont (+ a errors)))
2	6	(-> sto (-> cont (+ (list a) errors)))
1	6	(-> sto (-> cont (+ (list (* a sto)) errors)))
7	7	a
6	7	(-> env a)
5	7	(-> env (-> sto a))
4	7	(-> env (-> sto (-> cont a)))
3	7	(-> env (-> sto (-> cont (+ a errors))))
2	7	(-> env (-> sto (-> cont (+ (list a) errors))))
1	7	(-> env (-> sto (-> cont (+ (list (* a sto)) errors))))

Table 2: Stratified monad for the language of Figure 1

Before		After	
Level	Names	Level	Names
`A`	`bottom, ...`	`A`	`bottom, env-values, ...`
`TA`	`top, ...`	`TA`	`env-results, ...`
`...`		`Env -> TA`	`top, envs`
`...`		`...`	

Table 3: Environment transformer action on levels

Type	Examples	Form
Bottom	`exceptions` `nondeterminism`	`FT = T ∘ U`
Top	`environments`	`FT = S ∘ T`
Around	`stores`	`FT = S ∘ T ∘ U`
Continuation	`continuations` `continuations2`	`FT = (A -> TA) -> TA`

Table 4: Stratified monad transformer types

environment transformer acts on the levels of a stratified monad. It adds a single new level `Env -> TA`, adds the name `env-results` to `TA`, and adds the name `env-values` to `A`. It calls the new level `envs` and shifts the name `top` from `TA` to `Env -> TA`. It makes no changes to other levels.

Stratified monad transformers typically act by adding monads, not by removing them. For example, the environment transformer transforms all monads relating to `top`, specifying that they now relate the same level to the new `top`. The transformer also adds an identity monad on the new `top`.

The environment transformer adds only relating monads, since if `T` relates a level `l` to `top`, then `FT` relates `l` to the new `top`. If `Top` is the type construction for `top` and `L` is the constructor for `l`, then we are saying that if `Top = TL`, then `F(Top) = F(TL)`. Furthermore, it is clear, though more difficult to establish, that all relating monads are added.

The current toolkit includes four classes of stratified monad transformers: top, bottom, around, and continuation. Table 4 shows the form of each of these, and they are described fully in appendix A. Table 5 lists the levels and names defined by each transformer.

3.5 Defining language constructs

In general, operators that act primarily at a single level, such as `%amb` (Figure 4) and `%let` (Figure 2), are easy to write using standard idioms. More complex operators, such as `%call/cc`, are best written by abstracting from their definitions in an example semantics. Using a sufficiently complex semantics ensures that conceptually distinct levels are not confused. Table 7 lists the available modules and the value types and language constructs they define. Leading percent signs are omitted from the names.

The toolkit includes four types of procedures. Table 6 shows the levels of their domains and codomains. We define all four types of procedures over the same `environments` monad

Transformer	Level	Name
`nondeterminism`	`List A`	`lists`
`exceptions`	`A + X`	`errors`

`environments`	`Env -> TA`
	`TA`
	`A`
		`envs`
		`env-results`
		`env-values`

`stores`	`Sto -> T(A*Sto)`
	`T(A*Sto)`
	`A*Sto`
	`A`
		`stores`
		`store-results`
		`store-pairs`
		`store-values`

`continuations`	`(A -> TA) -> TA`
	`TA`
	`A`
		`conts`
		`cont-results`
		`cont-values`

Table 5: Transformer levels and names

Procedure type	Domain	Codomain
`cbv-static`	`env-values`	`env-results`
`cbn-static`	`env-results`	`env-results`
`cbv-dynamic`	`env-values`	`envs`
`cbn-dynamic`	`env-results`	`envs`

Table 6: Procedure types

transformer by writing different versions of `%lambda` and `%call`. Since names in call-by-value and call-by-name environments denote different types of values, the environment transformer is actually polymorphic. Unfortunately, this polymorphism is not explicitly represented in Scheme.

4 Discussion

This section discusses typed versus untyped value domains, the use of Scheme, previous work, and future work.

4.1 Typed versus untyped values

In the monadic framework, computations are polymorphic over values. For example, we can type the language construct `%zero?` as

```
%zero? : C number -> C boolean
```

where `C` is the type of computations. Using an ADT instead of an `eval` function (see section 2.1), it is easy to reuse the types and values of the language in which the interpreter is written. Thus we can write `%zero?` as above without having to write

Module	Values	Constructs
amb		amb
booleans	booleans	true, false, not, if, boolean?
numbers	numbers	num, +, -, *, /
numeric-predicates		=?, zero?, number?
error-values	errors	error
error-exceptions		error
cbv-environments		let, letrec, var
cbn-environments		let, letrec, var
cbv-static	procedures	lambda, call
cbn-static	procedures	lambda, call
cbv-dynamic	procedures	lambda, call
cbn-dynamic	procedures	lambda, call
cbv-callcc		call/cc
cbn-callcc		call/cc
stores		fetch, store, begin, skip
while		while

Table 7: Modules and language constructs

```
eval-boolean : boolean-exp -> C boolean
eval-number  : number-exp  -> C number
...
```

Similarly, we could define `%call` as

```
%call : C P(a,b) * C a -> C b
```

where `P(a,b)` is the type of procedures from `a` to `b`. Using this technique, Wadler could have implemented the simply-typed lambda calculus in [Wad92], rather than the untyped, by reusing Haskell's type system. Such a treatment would adhere more closely to Moggi's original [Mog89b].

In the toolkit, we compute over a single untyped value domain, represented as an extensible disjoint union. This choice exposes the treatment of types in the semantic equations and abstracts from the existing Scheme types; however, it would be interesting to try various typed approaches.

Steele [Ste94] tries to extend the domain of values using monads, and I follow him in [Esp94]. However, as he points out, using the exceptions monad to build sums yields behavior such as

```
(compute (%+ (%num 3) (%true)))
⇒ true
```

which is not at all what we want. Rather than add a complex error system to the monad operators, it is easier and more natural to build a separate value domain that language construct modules can extend (see Figure 7).

4.2 Why Scheme?

Why choose an untyped language when types are at the core of these ideas? Indeed, it would be preferable to express explicitly more of the toolkit's structure, particularly when Scheme's implicit polymorphism hides non-trivial constructions. The toolkit represents types as values but does not take this approach far enough.

Because we need to treat monads and monad transformers as values, ML and Haskell's type systems are not sufficient. We require higher-order types as values, and languages with these features [Car89] are not in common use. The usual reason for restrictive type systems is the intractability of inference in complex systems, but surely there is a way to make a well-defined class of simple inferences within a complex type system.

4.3 Previous work

Through his work on the partial lambda calculus [Mog86], Eugenio Moggi realized that the categorical concept of monads was applicable to the problem of modular semantics. In [Mog89b, Mog91b], he shows how to divide an "applied" lambda calculus into a core (variables and environments) and an extension (other features), expressed as a monad, and presents many such extensions.

In the second half of [Mog89a], Moggi explains how to use monad transformers to form complex monads from parts. This crucial ability was missing from his earlier papers; however, it is difficult to understand Moggi's presentation, and few researchers realized that he had made significant progress toward modularizing denotational semantics.

After reexplaining and implementing Moggi's methods in [Esp94], I saw that they do not provide sufficient modularity to handle constructs involving multiple semantic levels, such as `%call/cc` or even `%+` (because it raises errors). Stratified monads solve this problem, and, by separating semantic extensions from the language constructs they enable, increase language design flexibility.

Philip Wadler [Wad92] popularized Moggi's ideas by presenting monadic interpreters written in Haskell. Their restriction to extension by a single monad was the primary motivation for the present work. Also, Wadler and David King [KW92] show how to combine continuations and lists with other monads. Despite Moggi's earlier formulation of monad transformers, they discuss "combining `M` and `L`" rather than "constructing `ML` from `M`". Our toolkit treats monad constructors in full generality and exhibits a complete system for building interpreters from multiple modules, not just two.

Guy Steele [Ste94] shows how to compose *pseudomonads*, a new construction. Although they compose, pseudomonads are both more complex and less general than monad transformers. In fact, pseudomonads are essentially monad transformers limited to right composition. That is, they can realize

```
FTA = T(List A)
FTA = T(A + X)
FTA = T(A * M)
```

but not

```
FTA = Env -> TA
FTA = Sto -> T(A*Sto)
```

Steele's claim that pseudomonads improve on monad transformers by providing a fixed composition operator would be true if they were equally powerful. Steele's main contribution is a complete implementation of a modular semantics, which was really inspiring.

Mark Jones and Luc Duponcheel [JD93] address the problem of composing monads. They find that if one of several auxiliary maps is defined relating the structures of two monads, they can be composed, in an order depending which map is used. They do not attempt to build interpreters, and we note that monad composition is strictly less powerful than monad transformation.

Peter Mosses [Mos92] describes *action semantics*, a reformulation of his theory of *abstract semantic algebras*. Mosses attempts to achieve modularity in semantics by defining an intermediate level, called *actions*, into which one can translate language constructs. Actions are low-level enough to be flexible, but high-level enough to hide the details of interactions between *facets* of a language.

Since Mosses has not yet been able to combine facets, he gives a *single* algebra of actions, with environments, stores, etcetera. This algebra corresponds to a *single* stratified monad generated by the toolkit. Mosses's theory is more modular than pure denotational semantics because constructs are isolated from each other, but, from our point of view, it is not modular at all. One can only define modularly the constructs that the algebra was designed to define. That is, one can define stores modularly because the fixed algebra of actions includes modular stores.

Recalling that Moggi cites Mosses's comments on the non-modularity of denotational semantics, we can more gently view the toolkit as a final step in Mosses's program. Stratified monads could be the modular facets Mosses was looking for.

4.4 Future work

Using the toolkit, it should be possible to build abstract interpretations, translations (such as CPS), and simple compilers from reusable parts. The methods used to build interpreters could also be adapted to build other semantically complex systems, such as communication protocol handlers.

Although the toolkit evolved through the correction of flaws in its design, one remains. In defining `%call/cc`, it is necessary to lift a map from `store-values -> stores` to `store-pairs -> store-results`. This complex-sounding lifting is accomplished by the trivial procedure

```
(lambda (f)
  (lambda (a*s)
    ((f (left a*s)) (right a*s))))
```

This procedure cannot be written using `unit` and `bind`, and it was necessary to introduce `iunit`, an inverse to `unit`. Since `unit` is usually injective, an inverse is not a problem, but a reformulation of monads around the idea of liftings could provide a framework in which to define functions such as the above.

As stratified monads are a model-theoretic construction, the biggest challenge posed by this work is to find their proof-theoretic counterpart. That is, we would like to develop modular calculi for reasoning about the languages that the toolkit can build. Moggi's computational lambda calculus [Mog89b, Mog91b] captures precisely the inferences valid for lambda calculus over an arbitrary monad. Can we go further? Moggi's syntactic approach [MC93] is certainly relevant but does not seem to address the problem directly.

5 Conclusion

We have built a language designer's workbench, a collection of mix-and-match modules that assemble to form programming language interpreters. These interpreters run example programs and can be simplified to yield semantic equations. The toolkit defines a language from a sequence of semantic modules and a set of language constructs. The semantic modules combine to form an abstract data type of computations which constructs augment with particular linguistic features. These contributions yield increased flexibility and modularity over previous systems based on monads.

6 Acknowledgements

I am grateful to Mary Ng, Bill Rozas, and Gerry Sussman for encouragement and inspiration, to Jonathan Rees for getting me into monads and category theory, to Eugenio Moggi for his work, to AT&T for financial support, and above all to Franklyn Turbak for friendship and assistance beyond the call of duty.

A Transformer actions

This appendix describes the action of bottom, top, around, and continuation transformers on stratified monads using a simple pattern language in which

- Rules send sets to sets.
- The first matching case is used.
- Ellipses indicate repetition.

In patterns, we write levels as `<{name, ...}, type-constructor>` and situated monads as `<bottom-level, top-level, monad>`. We write `new-bot` and `old-top` (etcetera) for the levels named `bottom` and `top`. After each rule, we describe its action in words.

A.1 Bottom transformers

A bottom transformer is specified by three names and a monad transformer. The names are labelled `b`, `m`, and `t` for bottom, middle, and top. The monad transformer `F` is of the form `FTA = T(UA)` for some type constructor `U` (which can be given monad structure by applying `F` to the identity monad). In the resulting stratified monad, `b` refers to the level `A`, `m` refers to `UA`, and `t` refers to `T(UA)`. The action on levels is:

```
{<{bottom, n1, ...}, Bot>,
 <{top, n2, ...}, Top>,
 <{n3, ...}, T>, ...}
⇒
{<{b, bottom}, Id>,
 <{m, n1, ...}, F(Bot)>,
 <{t, top, n2, ...}, F(Top)>,
 <{n3, ...}, FT>, ...}
```

- `F` is applied to all type constructors.
- A new bottom level is added with identity type constructor.
- The name `bottom` is removed from the old bottom level.
- The names `b`, `m`, `t` are added to the new bottom, old bottom, and top levels, respectively.

The action on situated monads is:

```
{<old-bot, l, T>, ...,
 <a, b, M>, ...}
⇒
{<new-bot, new-bot, Id>,
 <old-bot, l, T>, ...,
 <new-bot, l, FT>, ...,
 <a, b, M>, ...}
```

- Monads relating to bottom are transformed.
- An identity monad is added at the new bottom.

A.2 Top transformers

Top transformers are specified by three names `b`, `m`, `t` and a monad transformer `F`, which is of the form `FTA = S(TA)`. The name `b` refers to `A`, `m` refers to `TA`, and `t` refers to `S(TA)`. The action on levels is:

```
{<{bottom, n1, ...}, Bot>,
 <{top, n2, ...}, Top>,
 <{n3, ...}, T>, ...}
⇒
{<{b, bottom, n1, ...}, Bot>,
 <{m, n2, ...}, Top>,
 <{t, top}, F(Top)>,
 <{n3, ...}, T>, ...}
```

- Only the top type constructor is transformed.
- `top` is removed from the old top level.
- `t`, `m`, `b` are added to the new top, old top, and bottom.

The action on situated monads is:

```
{<l, old-top, T>, ...,
 <a, b, M>, ...}
⇒
{<new-top, new-top, Id>,
 <l, old-top, T>, ...,
 <l, new-top, FT>, ...,
 <a, b, M>, ...}
```

- Monads relating to top are transformed.
- An identity monad is added at the new top.

A.3 Around transformers

Around transformers are specified by four names `b`, `m1`, `m2`, `t` and three monad transformers `F`, `G`, `H`. Essentially, `F` is the around transformer, while `G` and `H` are the bottom and top transformers from which it is composed.

We require that `FTA = S(T(UA))`, `GTA = T(UA)`, and `HTA = S(TA)` for some `S` and `T`. Hence `F = G ∘ H = H ∘ G`. The name `b` refers to `A`, `m1` refers to `UA`, `m2` refers to `T(UA)`, and `t` refers to `S(T(UA))`. The action on levels is:

```
{<{bottom, n1, ...}, Bot>,
 <{top, n2, ...}, Top>,
 <{n3, ...}, T>, ...}
⇒
{<{bottom, b}, Id>,
 <{m1, n1, ...}, G(Bot)>,
 <{m2, n2, ...}, G(Top)>,
 <{top, t}, F(Top)>,
 <{n3, ...}, GT>, ...}
```

- All levels are transformed by `G`.
- The top level is also transformed by `F`.
- `top` and `bottom` are moved to the new top and bottom.
- `b`, `m1`, `m2`, `t` are added to the new bottom, old bottom, old top, and new top.

The action on situated monads is:

```
{<old-bot, old-top, T1>,
 <old-bot, l1, T2>, ...,
 <l2, old-top, T3>, ...,
 <a, b, M>, ...>}
⇒
{<new-bot, new-bot, Id>,
 <new-top, new-top, Id>,
 <new-bot, new-top, F(T1)>,
 <new-bot, old-top, G(T1)>,
 <old-bot, new-top, H(T1)>,
 <new-bot, l1, T2>, ...,
 <new-bot, l1, G(T2)>, ...,
 <l2, old-top, T3>, ...,
 <l2, new-top, H(T3)>, ...,
 <a, b, M>, ...}
```

- Monads relating to bottom, top, and both are transformed by `G`, `H`, and `F`.
- Identity monads are added at the new bottom and new top.

A.4 Continuation transformers

Continuation transformers are specified by three names `b`, `m`, `t` and two monad transformers `F` and `G`. `F` is always `FTA = (A -> TA) -> TA`, but `G` varies. Continuation transformers shift `T` into the answer type, so their action is rather unusual. Essentially, `F` handles continuations in the usual way, ignoring `T` since continuations are polymorphic in the type of answers, while `G` controls how operators on `T` are lifted to operators on `(A -> TA) -> TA`. `G` actually accepts two monads, rather than one, since its action on monads `M` relating to top depends on `T`, the monad relating bottom and top. The action on levels is identical to that of top transformers:

```
{<{bottom, n1, ...}, Bot>,
 <{top, n2, ...}, Top>,
 <{n3, ...}, T>, ...}
⇒
{<{b, bottom, n1, ...}, Bot>,
 <{m, n2, ...}, Top>,
 <{t, top}, F(Top)>,
 <{n3, ...}, T>, ...}
```

- Only the top type constructor is transformed.
- `top` is removed from the old top.
- `t`, `m`, `b` are added to the new top, old top, and bottom.

The action on situated monads is:

```
{<old-bot, old-top, T>,
 <l, old-top, M>, ...,
 <a, b, M>, ...}
⇒
{<new-top, new-top, Id>,
 <old-bot, old-top, T>,
 <old-bot, new-top, FT>,
 <old-bot, new-top, GTT>,
 <l, old-top, M>, ...,
 <l, new-top, GTM>, ...,
 <a, b, M>, ...}
```

- An identity monad is added at the new top.
- Monads relating to top are transformed by `G`.
- The monad relating bottom and top is transformed by `F`.

B Language construct definitions

This appendix lists some of the more interesting language construct definitions.

```
;;; Call-by-value call/cc

;; Proc : env-values -> env-results
;; Cont : cont-values -> cont-answers

(define %call/cc
  (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)))
    (lambda (exp)
      (mapC exp
        (lambda (cont)
          (lambda (k)
            (cont
             (lambda (cv)
               ((iunitK
                 (bindS (unitR cv)
                   (lambda (p)
                     (p (unitP
                         (lambda (v)
                           (mapK (unitE v)
                                 (lambda (cont)
                                   (lambda (k1) (cont k))))))))))
                k)))))))))
```

```
;;; Call-by-value static procedures

;;; Proc : env-values -> env-results

(define %var
  (let ((unitT (get-unit 'env-values 'env-results))
        (unitE (get-unit 'envs 'top)))
    (lambda (name)
      (unitE
       (lambda (env)
         (unitT (env-lookup env name)))))))

(define %lambda
  (let ((unitE (get-unit 'envs 'top))
        (bindE (get-bind 'envs 'top))
        (unitP (get-value-unit
                'procedures 'env-results)))
    (lambda (var body)
      (bindE body
        (lambda (body)
          (unitE
           (lambda (env)
             (unitP
              (lambda (arg)
                (body (env-extend
                       env var arg)))))))))))

(define %call
  (let ((bindP (get-value-bind 'procedures 'top))
        (bindV (get-bind 'env-values 'top))
        (unitR (get-unit 'env-results 'top)))
    (lambda (proc arg)
      (bindP proc
        (lambda (proc)
          (bindV arg
            (lambda (arg)
              (unitR (proc arg)))))))))
```

```
;;; Call-by-name dynamic procedures

;;; Proc : env-results -> envs

(define %var
  (let ((unitE (get-unit 'envs 'top)))
    (lambda (name)
      (unitE
       (lambda (env)
         (env-lookup env name))))))

(define %lambda
  (let ((bindE (get-bind 'envs 'top))
        (unitP (get-value-unit 'procedures 'top)))
    (lambda (var body)
      (bindE body
        (lambda (body)
          (unitP
           (lambda (arg)
             (lambda (env)
               (body (env-extend
                      env var arg))))))))))

(define %call
  (let ((bindP (get-value-bind 'procedures 'top))
        (bindR (get-bind 'env-results 'top))
        (unitE (get-unit 'envs 'top)))
    (lambda (proc arg)
      (bindP proc
        (lambda (proc)
          (bindR arg
            (lambda (arg)
              (unitE (proc arg)))))))))
```

C Stratified monad transformer definitions

This appendix lists the Scheme code for the stratified monad transformers used in the toolkit. The code for transforming inverse unit operators is omitted for clarity.

```
;;; Environments: FTA = Env -> TA

(define environments
  (make-top-transformer
   'env-values 'env-results 'envs
   (lambda (unit bind compute t)
     (make-monad
      (lambda (a)
        (lambda (env) (unit a)))

      (lambda (c f)
        (lambda (env)
          (bind (c env)
                (lambda (a)
                  ((f a) env)))))

      (lambda (c f)
        (compute (c empty-env) f))

      (lambda (a)
        `(-> env ,(t a)))))))


;;; Nondeterminism: FTA = T(List A)

(define nondeterminism
  (make-bottom-transformer
   #f 'lists #f
   (lambda (unit bind compute t)

     (define (amb x y)
       (bind x
         (lambda (x)
           (bind y
             (lambda (y)
               (unit (append x y)))))))

     (make-monad
      (lambda (a)
        (unit (list a)))

      (lambda (c f)
        (bind c
          (lambda (l)
            (reduce amb (unit '()) (map f l)))))

      (lambda (c f)
        (compute c (lambda (l) (map f l))))

      (lambda (a)
        (t `(list ,a)))))))
```

```
;;; Side-effects: FTA = Sto -> T(A*Sto)

(define side-effects
  (make-around-transformer
   'store-values 'store-pairs
   'store-results 'stores

   ;; G (bottom)
   (lambda (levels) '())

   ;; H (top)
   (level-transformer
    (lambda (unit bind compute t)
      (make-monad
       (lambda (a)
         (lambda (sto) (unit a)))

       (lambda (c f)
         (lambda (sto)
           (bind (c sto)
                 (lambda (a)
                   ((f a) sto)))))

       (lambda (c f)
         (compute (c empty-store) f))

       (letify
        (lambda (a)
          `(-> sto ,(t a)))))))

   ;; F (around)
   (level-transformer
    (lambda (unit bind compute t)
      (make-monad
       (lambda (a)
         (lambda (sto)
           (unit (pair a sto))))

       (lambda (c f)
         (lambda (sto)
           (bind (c sto)
             (lambda (as)
               ((f (left as)) (right as))))))

       (lambda (c f)
         (compute (c empty-store)
                  (compose f left)))

       (letify
        (lambda (a)
          `(-> sto ,(t `(* ,a sto))))))))))
```

```
;;; Errors: FTA = T(A + X)

(define errors
  (make-bottom-transformer
   #f 'errors #f
   (lambda (unit bind compute t)
     (make-monad
      (lambda (a) (unit (in-left a)))

      (lambda (c f)
        (bind c
         (sum-function
          f (lambda (x) (unit (in-right x))))))

      (lambda (c f)
        (compute c (sum-function
                    f identity-procedure)))

      (lambda (a)
        (t `(+ ,a ,name)))))))


;;; Continuations: FTA = (A -> TA) -> TA

(define continuation-monad-transformer
  (lambda (unit bind compute t)
    (make-monad
     (lambda (a)
       (lambda (k) (k a)))

     (lambda (c f)
       (lambda (k)
         (c (lambda (a) ((f a) k)))))

     (lambda (c f)
       (compute (c unit) f))

     (letify
      (lambda (a)
        ((letify
          (lambda (ta)
            `(-> (-> ,a ,ta) ,ta)))
         (t a)))))))
```

```
(define continuations
  (make-cont-transformer
   'cont-values 'cont-answers 'conts

   continuation-monad-transformer

   (lambda (unitT bindT computeT T)
     (lambda (unitM bindM computeM M)
       (make-monad
        (lambda (a)
          (lambda (k)
            (bindT (unitM a) k)))

        (lambda (c f)
          (lambda (k)
            (bindM (c unitT)
              (lambda (a)
                ((f a) k)))))

        (lambda (c f)
          (computeM (c unitT) f))

        (lambda (a)
          `(-> cont ,(m a))))))))

(define continuations2
  (make-cont-transformer
   'cont-values 'cont-answers 'conts

   continuation-monad-transformer

   (lambda (unitT bindT computeT T)
     (lambda (unitM bindM computeM M)
       (make-monad
        (lambda (a)
          (lambda (k)
            (unitM a)))

        (lambda (c f)
          (lambda (k)
            (bindM (c k)
              (lambda (a)
                ((f a) k)))))

        (lambda (c f)
          (computeM (c unitT) f))

        (lambda (a)
          `(-> cont ,(m a))))))))
```

References

[AKHS88] Hal Abelson, Jacob Katzenelson, Matthew Halfant, and Gerald Jay Sussman. The Lisp Experience. *Annual Review of Computer Science*, 3:167–195, 1988.

[Car89] Luca Cardelli. Typeful programming. Technical Report 45, DEC Systems Research Center, Palo Alto, CA, May 1989.

[CR91] Will Clinger and Jonathan Rees. Revised^4 Report on Scheme. *Lisp Pointers*, 4(3), 1991.

[DJL88] Pierre Deransart, Martin Jourdan, and Bernard Lorho. *Attribute Grammars*, volume 323 of *Lecture Notes in Computer Science*. Springer Verlag, 1988.

[Esp94] David Espinosa. Semantic Lego. FTP from martigny.ai.mit.edu : pub/dae, January 1994.

[Fil94] Andrzej Filinski. Representing monads. In *Proceedings of the 21st Annual ACM Symposium on Principles of Programming Languages*, Portland, OR, January 1994.

[GTWW77] J. A. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright. Initial algebra semantics and continuous algebras. *Journal of the ACM*, 24:68–95, 1977.

[JD93] Mark P. Jones and Luc Duponcheel. Composing monads. Technical Report YALEU / DCS / RR-1004, Yale University, December 1993.

[KW92] David King and Philip Wadler. Combining monads. In *Proceedings of the Fifth Annual Glasgow Workshop on Functional Programming*, Ayr, Scotland, 1992. Springer Workshops in Computer Science.

[MC93] Eugenio Moggi and Pietro Cenciarelli. A syntactic approach to modularity in denotational semantics. In *Category Theory and Computer Science*, Lecture Notes in Computer Science. Springer Verlag, 1993.

[Mog86] Eugenio Moggi. Categories of partial morphisms and the partial lambda calculus. In *Category Theory and Computer Programing*, volume 240 of *Lecture Notes in Computer Science*, Guildford, England, 1986. Springer Verlag.

[Mog89a] Eugenio Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Laboratory for Foundations of Computer Science, University of Edinburgh, Edinburgh, Scotland, June 1989. FTP from theory.doc.ic.ac.uk.

[Mog89b] Eugenio Moggi. Computational lambda calculus and monads. In *IEEE Symposium on Logic in Computer Science*, pages 14–23, Asilomar, CA, June 1989.

[Mog91a] Eugenio Moggi. A modular approach to denotational semantics. In *Category Theory and Computer Science*, volume 530 of *Lecture Notes in Computer Science*, pages 138–139. Springer Verlag, September 1991.

[Mog91b] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.

[Mos92] Peter D. Mosses. *Action Semantics*, volume 26 of *Tracts in Theoretical Computer Science*. Cambridge University Press, 1992.

[Ste94] Guy L. Steele Jr. Building interpreters by composing monads. In *Proceedings of the 21st Annual ACM Symposium on Principles of Programming Languages*, Portland, OR, January 1994.

[Wad92] Philip Wadler. The essence of functional programming. In *Proceedings of the 19th Annual ACM Symposium on Principles of Programming Languages*, pages 1–14, Albuquerque, NM, January 1992.