Semantic Lego

David A. Espinosa

Submitted in partial fulfillment of the
requirements for the degree
of Doctor of Philosophy
in the Graduate School of Arts and Sciences.

COLUMBIA UNIVERSITY
1995

ABSTRACT

Semantic Lego

David A. Espinosa

Denotational semantics [Sch86] is a powerful framework for describing programming languages; however, its descriptions lack modularity: conceptually independent language features influence each others' semantics. We address this problem by presenting a theory of modular denotational semantics.

Following Mosses [Mos92], we divide a semantics into two parts, a *computation* ADT and a *language* ADT (abstract data type). The computation ADT represents the basic semantic structure of the language. The language ADT represents the actual language constructs, as described by a grammar. We define the language ADT *using* the computation ADT; in fact, language constructs are polymorphic over many different computation ADTs.

Following Moggi [Mog89a], we build the computation ADT from composable parts, using *monads* and *monad transformers*. These techniques allow us to build many different computation ADTs, and, since our language constructs are polymorphic, many different language semantics.

We automate these ideas in Semantic Lego (SL), a modular language construction set written in Scheme. SL generates interpreters automatically from composable parts and is a useful tool for programming language design.

Contents

List of Figures

List of Tables

Acknowledgements

Mary Ng, my fiancée, has been waiting patiently for this thesis for several years. I could have done it without her, but it would have been much worse, and it was already bad. Mary is easily the happiest result of grad school.

Joanne Espinosa, my mother, has been great for 28 years. Thanks, mom.

Gerald J. Sussman, who I have known for a decade now, has always been an inspiration. His faith in his students never fails, and talking with him makes you believe, if only for a moment, that you can do anything. On the more material side, Jerry let me hang out at his lab for the last year (or more).

Sal Stolfo, my advisor at Columbia, has been an extremely tolerant observer of my graduate school career. In part, I picked Sal as an advisor because he was a nice guy; remarkably, he still is.

My defense committee, Gail Kaiser, Ken Ross, and Mukesh Dalal, helped me get out of Columbia in one piece.

Albert Greenberg rescued me from unemployment for several summers at AT&T. He initially hired me because "it takes less paper work to hire Ph.D. scholars". We had fun hacking parallel Fourier transforms and solving models of communication networks. The connection between that and monads seems certain, but, for the present time, obscure.

An AT&T Ph.D. scholarship supported me for five years, and they didn't even make me beg too hard for a fifth year. Unfortunately, all they gave me was money (Albert notwithstanding).

Thanks to Phil Chan, Mauricio Hernandez, Sushil Da Silva, Paul Michelman, and Bulent Yener for keeping me company at Columbia. Similarly for Michael Blair, Kleanthes Koniaris, Natalya Cohen, Raj Surati, and all the other fourth-floor people at MIT.

Thanks also to my musical friends, Joseph Briggs, Kerstin Kup, Brian and Karen Neal, Lois Winter, and Johelen Carleton.

Albert Meyer has been great fun on many occasions. It's splendid to talk to someone who knows semantics inside and out, along with most of its history. You just can't get that from papers!

Eugenio Moggi deserves my thanks for his work, *sine qua non.* Albert objected to Moggi's inclusion here since our relationship is scientific rather than personal (especially since I've never met him). Albert, as a logician, splits any hair he can find.

Jonathan Rees introduced me to monads and category theory. I hope we'll be able to work more together later. Now he's off chasing bugs in England.

Bill Rozas helped me out on many, many occasions and was always happy to discuss semantics or architecture. Bill is incredibly generous and made me feel that we were equals, even when we weren't. I envy him this quality.

Carl Gunter has been a great source of advice and assistance. When I first met him at LFP in 1992, he was a soft-spoken man who, after a heated semantics argument died down, would say, "Actually, the *real* answer is ...". His explanations and his book [Gun92] are crystal clear.

Charles Leiserson provided convincing evidence to attend MIT as an undergrad by being the most interesting person at Brown when I visited there. He did a great job teaching me algorithms, but I figure that field's too easy anyway. Charles has an astounding ability to formalize just about anything.

Franklyn Turbak and I have had enormous fun hacking interpreters and languages for the last two years. Until meeting Lyn, I was convinced that formal semantics was a non-sensical hodgepodge of Greek letters intended to confuse the reader into thinking the field had actual content. My present views you'll have to ascertain by reading this thesis.

Chapter 1

Introduction

Denotational semantics is a powerful framework for defining programming languages. Using it, we can describe languages concisely and unambiguously and build interpreters that execute actual programs. It is not a difficult theory to understand, especially considering its power.

Unfortunately, it is hard to read and write denotational descriptions, primarily because they lack modularity. Each language construct interacts with all the semantic building blocks that form the language's foundation. For example, if we model assignment using a store, then *every* language construct must interact with the store, not just assignment. The complexity of this interaction makes denotational descriptions overly intricate.

This thesis presents a modular style of writing denotational descriptions, which we automate as SEMANTIC LEGO[1](SL), a Scheme program that builds interpreters from component parts. In essence, SL is a language for describing languages. This work makes several important contributions:

- We reintroduce the idea of programming languages as abstract data types. Interpreters written in this style are shorter and clearer than usual.

- We restate Moggi's theory of *lifting* in simple terms, making it accessible to a

[1] Lego is a registered trademark of Interlego AG.

wider audience.

- We describe a new theory of *stratification* that is simpler than lifting yet more powerful. This theory extends Mosses's work on semantic algebras, adding structure and modularity.

- We show two styles of writing modular interpreters, based respectively on lifting and stratification.

- We present SEMANTIC LEGO, a modular language construction set based on stratification, and give several examples.

This work has several important consequences:

- We can understand, discuss, and teach languages better by decomposing them into parts. For example, the resumptions model of parallelism appears complex until we see it as a combination of several simple features.

- We can experiment with new languages. SL handles the bookkeeping associated with denotational descriptions, leaving the designer free to consider higher-level issues. SL's underlying theory can also help suggest new language constructs.

The following story illustrates SL's power. Three teaching assistants for the MIT graduate programming languages course needed to describe the semantics of a sophisticated control construct (`shift`) in the presence of state. Although they understood control and state independently, they were unable to find a suitable interaction between these features before distributing the problem set.

Using SL, I generated two solutions in under a minute. In fact, SL formed complete interpreters, not just the single required construct. It was also easy to add errors generation and handling, another semantic complication. Since I was confident in SL's ability to produce correct semantics, I did not even examine the definitions to see if they were well-formed.

The thesis is organized as follows. Chapter 3 discusses lifting; chapter 4 discusses stratification. Chapter 5 compares these approaches and reviews previous work. Appendix A covers issues tangentially related to the thesis.

We assume an elementary understanding of denotational semantics and functional programming; for further background, see [Wad92]. All examples and code fragments are in Scheme [CR91].

The rest of this chapter presents languages as abstract data types, demonstrates that the usual style of writing interpreters isn't modular, shows two styles of writing modular interpreters, and exercises SL with a series of examples.

1.1 Denotational Semantics

In this section, we discuss denotational semantics, the theory on which SL is based. Excellent references are [Gun92, Sch86]. In general, semantics is the study of meaning. Our goal is to determine, for example, that `x = x+1` means "add one to `x`".

Programming language semantics is actually a branch of mathematical logic; the main difference is that logics are for *reasoning*, while programming languages are for *computation*. Semantics (whether of propositions or programs) is split into two interacting parts. Proof theory, the more syntactic part, lets us reason and compute. Model theory, the more semantic part, lets us describe what we are actually reasoning and computing about.

1.1.1 Domains / Types

Our first step is to describe the raw material for building models. This elaborate subject is called *domain theory*; however, most of it is irrelevant to our interest in modeling language features. I usually call domains *types*, although the concepts are not quite the same.

We begin with base types such as *Num*, *Bool*, and *String*. We glue these together using the type constructors $\rightarrow$ (function space), $\times$ (cartesian product), and $+$ (disjoint union). $A \to B$ is the set of functions from A to B. Since $\to$ associates to the right, $A \to B \to C$ means $A \to (B \to C)$. $A \times B$ is the set of all pairs (a, b). $A + B$ is the set of all $(0, a)$ and $(1, b)$. That is, elements of $A + B$ are either A's or B's, except that we can always tell which is which by checking the tag. Thus, $A + A$ is different from A.

Two types A and B are *isomorphic* (set-theoretically) if there exist functions $f : A \to B$ and $g : B \to A$ such that $f \circ g = g \circ f = id$. Writing $\cong$ for isomorphism, we have

$$
\begin{aligned}
A + B &\cong B + A \\
A \times B &\cong B \times A
\end{aligned}
$$

$$
\begin{aligned}
(A + B) + C &\cong A + (B + C) \\
(A \times B) \times C &\cong A \times (B \times C)
\end{aligned}
$$

$$
\begin{aligned}
A \times (B + C) &\cong A \times B + A \times C \\
A \to (B \to C) &\cong A \times B \to C \\
(A \to C) \times (B \to C) &\cong (A + B) \to C
\end{aligned}
$$

Thus, $+$, $\times$, and $\to$ resemble addition, multiplication, and exponentiation. Because of this structure, types built using these constructors are called *algebraic* types. Also, all one-element sets are isomorphic (and similarly for other cardinalities). If we distinguish a set of each cardinality, we can also write identities such as $0 \times A \cong 0$ and $1 + 1 \cong 2$.

1.1.2 Logics

A *logic* is given by a *language*, a class of *models*, a set of *inference rules*, and a *meaning* or *denotation* function for each model. The meaning function maps terms in the language to elements of the model and interprets the language as being *about* the model. The inference rules are used for reasoning or computation. Let's take a simple arithmetic language:

$$\begin{array}{lcl} Exp & = & Num \\ & & \mid Exp + Exp \\ & & \mid Exp \times Exp \\ & & (Exp) \\ Num & = & Dig \mid Dig\,Num \\ Dig & = & 0 \ldots 9 \end{array}$$

An example expression is $3 \times (10 + 1)$. For a model, we have the usual natural numbers. For inference rules, we have the usual identity, associative, commutative, and distributive laws. The meaning function sends expressions to actual numbers. For example, it sends $3 \times (10 + 1)$ to 33. The inference rules must be *sound*; that is, they must preserve meanings. For example, if we apply the distributive law to $3 \times (10+1)$ to get $3 \times 10 + 3 \times 1$, the meaning of this totally different expression must also be 33. Note that we specify only a single model, while our definition allows a class of models; see [Mes89] for a more thorough treatment of logics.

Interestingly, we have two ways of evaluating expressions: inside the language, using the inference rules, and outside the language, using the meaning function. If we use the meaning function, we need a way to compute in the language in which the meaning function is described (the *metalanguage*). Sometimes this means is available; for example, if we implement an evaluator for the above language in C, we can use C's arithmetic. Sometimes it is not; in performing arithmetic by hand, we use the inference rules to justify the syntactic manipulations of the usual addition and multiplication algorithms. In this thesis, we build interpreters using meaning functions and focus on models rather than inference rules.

Most meaning function definitions are *compositional*: the meaning of an expression depends only on the meaning of its subexpressions, rather than their exact syntax. For example, in the arithmetic language, the meaning of $a + b$ is the sum of the meanings of a and b.

1.1.3 Environments

The semantics of more complex programming languages is simply an elaboration of the above example. In the rest of this section, we discuss the basic techniques. For example, suppose we add variables and a binding construct `let` to the language. Now we can write (shifting to a Scheme-like notation):

```
(let ((a 1) (b 2))
  (+ a (* b 3)))
```

The meaning of the overall program could still be a number, but what about the meaning of an expression such as `a`? We take the meaning of an expression to be a function from an *environment* to a number, where the *environment* assigns values to variables. We can view the environment as a function from variables to values or as a list of variable / value pairs; it doesn't matter which.

Thus, meanings (or denotations), instead of being numbers, are elements of

$$Den = Env \rightarrow Num$$

The primary inference rule for variables is *substitution*. It must be carefully defined, and there are several variations, but, in general, we can infer that

```
(let ((a 1) (b 2))
  (+ a (* b 3)))
```

has the same meaning as `(+ 1 (* 2 3))`. Although it seems simple, it is already not easy to prove the soundness of this inference rule with respect to the environment model.

1.1.4 Stores

The next complication arises in modeling state. Suppose we have a language with assignment statements and sequencing. We can write programs such as

```
a = 1
b = 2
a = a + b
return a
```

The meanings of statements are functions from stores to stores, where a *store* is just another name for an environment. That is, a statement accepts a set of variable bindings and returns a new set. We write this as

$$Den = Sto \rightarrow Sto$$

For example, the meaning of `a = a + 1` sends a store binding `a` to 3 to a store binding `a` to 4. To be accurate, we need separate meaning functions for programs, statements, and expressions. We have

$$\begin{array}{lcl} Den_{Program} & = & Num \\ Den_{Statement} & = & Sto \rightarrow Sto \\ Den_{Expression} & = & Sto \rightarrow Num \end{array}$$

We also need to handle unbound variables somehow; there are several viable approaches. We could return an observable error value (that we could test), an unobservable error value (that would propagate through all operators unchanged), or we could abort the computation immediately.

1.1.5 Continuations

One way to model an `abort` operator is via *continuations*. The continuation represents the rest of the computation, a map from the current store to the final store. A statement accepts both a store and a continuation. Most statements form a new store (as usual) and apply the continuation to the result, thus continuing the computation. To abort, statements return the store directly, ignoring the continuation. Thus, a model for statements is

$$\begin{array}{lcl} Den & = & Sto \rightarrow Cont \rightarrow Sto \\ Cont & = & Sto \rightarrow Sto \end{array}$$

In a language with operators for manipulating continuations, such as `catch` and `throw`, we can build non-local control structures for backtracking or coroutines. This example shows that semantic concepts can sometimes suggest new and useful language features.

1.1.6 The importance of types

Quite often, once we specify the language and the model, there are few ways to map terms to denotations. Once we know that

$$\mathit{Den} = \mathit{Env} \rightarrow \mathit{Num}$$

(and that environments map variables to numbers), it is clear how to define variable reference. Thus, an experienced semanticist needs only the language, the model, and some general instructions about how the language should behave in order to reconstruct its exact semantics. The model alone conveys most of the interesting information, which is why types are good tools for describing semantics.

Why do models determine language construct semantics so completely? With respect to the values they manipulate, language constructs are *general* and *uniform*. For example, function call works for all types, and in the same way. This notion, formalized by Reynolds, is called *parametric polymorphism*.

In the above semantics, it is clear that the only sensible meaning for a variable is to look it up in the environment. How else could we get a number? We would have to make one up, which would not be general. A good reference for this material is [Wad89], which describes Reynolds' result that polymorphic functions obey identities derivable solely from their types. For example, a function `f` : $\mathit{List}(A) \rightarrow \mathit{List}(A)$ must obey

```
(f (map g l)) = (map g (f l))
```

for any `l` : $\mathit{List}(A)$ and `g` : $A \rightarrow B$.

Since models tightly constrain language semantics, we always begin by finding a suitable model. Furthermore, it is not surprising that a modular theory of interpreters begins with a modular theory of models.

1.2 Languages as ADTs

We begin with a simple interpreter in the style of Abelson and Sussman [ASS85] and reduce it to its essence, eliminating issues of syntax as much as possible. This approach shows that "metalinguistic abstraction" is no different from ordinary abstraction. In other words, it is not necessary to "go outside" the implementation language in order to program in the target language. It also provides a streamlined style of writing interpreters that shortens them and makes the separation of syntax and semantics more apparent.

A simple interpreter for a purely functional language appears in figures 1.1 – 1.3. A typical use of it is

```
(compute '((lambda x (* x x)) 9))
⇒ 81
```

This interpreter parses concrete syntax in the form of lists. That is, it recognizes the subset of lists that are valid programs. Parsing has little to do with semantics, so we pass to an abstract syntax, using the constructors shown in figure 1.4. Now the same program reads as

```
(compute (%call (%lambda 'x (%* (%var 'x) (%var 'x))) (%num 9)))
⇒ 81
```

which is more explicit (but less readable).

These constructors, along with the procedure `compute`, describe the interpreter as an abstract data type (ADT), whose signature is shown in figure 1.5. Of course, the signature only partially specifies the behavior of the interpreter. The easiest way to describe its behavior more completely is to provide a "model" implementation, such as the one in figures 1.1 – 1.4.

Now that we have specified the interpreter's interface, we can ask whether there is a simpler implementation. In fact, figure 1.6 shows that there is. In this figure, we use the Scheme syntax for defining curried functions, so that

```
(define ((f a) b) ...)
```

```
(define (eval exp env)
  (cond ((number? exp) (eval-number exp env))
        ((variable? exp) (eval-variable exp env))
        ((lambda? exp) (eval-lambda exp env))
        ((if? exp) (eval-if exp env))
        ((+? exp) (eval-+ exp env))
        ((*? exp) (eval-* exp env))
        (else (eval-call exp env))))

(define (compute exp)
  (eval exp (empty-env)))

(define (eval-number exp env)
  exp)

(define (eval-variable exp env)
  (env-lookup exp env))

(define (eval-lambda exp env)
  (lambda (val)
    (eval (lambda-body exp)
          (extend-env env (lambda-variable exp) val))))

(define (eval-call exp env)
  ((eval (call-operator exp) env)
   (eval (call-operand exp) env)))

(define (eval-if exp env)
  (if (eval (if-condition exp) env)
      (eval (if-consequent exp) env)
      (eval (if-alternative exp) env)))

(define (eval-+ exp env)
  (+ (eval (op-arg1 exp) env)
     (eval (op-arg2 exp) env)))
```

Figure 1.1: Interpreter

```
(define (empty-env) '())

(define (env-lookup var env)
  (let ((entry (assq var env)))
    (if entry
        (error "Unbound variable: " var)
        (right entry))))

(define (env-extend var val env)
  (pair (pair var val) env))
```

Figure 1.2: Environment ADT

expands into

```
(define f (lambda (a) (lambda (b) ...)))
```

Notice that the syntactic constructors and selectors have entirely disappeared. The new implementation is shorter, yet it preserves the semantic content of the original. We call figure 1.6 a *denotational* implementation of the language ADT because we represent expressions by their denotations rather than their syntax. We call an equation such as

$$Den = Env \to Val$$

the *basic semantics* of the language. We write Den in place of Exp to reflect the change in point of view, but the ADT semantics remains the same.

Despite these advantages, few authors write interpreters in this style, perhaps because most languages encourage programmers to use concrete (rather than abstract) data types. For example, Scheme emphasizes lists, while ML and Haskell emphasize algebraic data types (sums and products). Programmers in languages with better support for ADTs might arrive more easily at this style, although first-class functions are also necessary.

The denotational style shows explicitly that the semantics is *compositional*, which is to say that the meaning of an expression is composed from the *meanings* of its imme-

```
(define variable? symbol?)

(define (lambda? exp)
  (eq? 'lambda (first exp)))

(define lambda-variable second)
(define lambda-body third)

(define call-operator first)
(define call-operand second)

(define (if? exp)
  (eq? 'if (first exp)))

(define if-condition second)
(define if-consequent third)
(define if-alternative fourth)

(define (+? exp)
  (eq? '+ (first exp)))

(define (*? exp)
  (eq? '* (first exp)))

(define op-arg1 second)
(define op-arg2 third)
```

Figure 1.3: Expression predicates and selectors

```
(define (%num x) x)
(define (%var name) name)
(define (%lambda name exp) (list 'lambda var exp))
(define (%call e1 e2) (list e1 e2))
(define (%if e1 e2 e3) (list 'if e1 e2 e3))
(define (%+ e1 e2) (list '+ e1 e2))
(define (%* e1 e2) (list '* e1 e2))
```

Figure 1.4: Expression constructors

`compute`	: Exp	$\rightarrow Val$
`%num`	: Val	$\rightarrow Exp$
`%var`	: $Name$	$\rightarrow Exp$
`%lambda`	: $Name \times Exp$	$\rightarrow Exp$
`%call`	: $Exp \times Exp$	$\rightarrow Exp$
`%if`	: $Exp \times Exp \times Exp$	$\rightarrow Exp$
`%+`	: $Exp \times Exp$	$\rightarrow Exp$
`%*`	: $Exp \times Exp$	$\rightarrow Exp$

Figure 1.5: Interpreter interface

```
;;  Den = Env → Val
;;  Proc = Val → Val

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

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

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

(define ((%call d1 d2) env)
  ((d1 env) (d2 env)))

(define ((%if d1 d2 d3) env)
  (if (d1 env) (d2 env) (d3 env)))

(define ((%+ d1 d2) env)
  (+ (d1 env) (d2 env)))

(define ((%* d1 d2) env)
  (* (d1 env) (d2 env)))
```

Figure 1.6: Denotational implementation

```
(define (D exp)
  (cond ((number? exp) (%num exp))
        ((variable? exp) (%var exp))
        ((lambda? exp)
         (%lambda (lambda-variable exp)
            (D (lambda-body exp))))
        ((if? exp)
         (%if (D (if-condition exp))
              (D (if-consequent exp))
              (D (if-alternative exp))))
        ((+? exp)
         (%+ (D (op-arg1 exp))
             (D (op-arg2 exp))))
        ((*? exp)
         (%* (D (op-arg1 exp))
             (D (op-arg2 exp))))
        (else
         (%call (D (call-operator exp))
                (D (call-operand exp))))))
```

Figure 1.7: Map from syntax to semantics

diate subexpressions. The original implementation does not preclude the possibility that the meaning of an expression could depend on the *syntax* of its subexpressions.

Figures 1.1 – 1.4 and figure 1.6 implement the same interface (figure 1.5) using very different base types. The former uses expressions, while the latter uses denotations. As shown in figure 1.7, we can define a map from expressions to denotations, that is, from syntax to semantics. For example, `(* 2 3)` goes to `(%* (%num 2) (%num 3))`.

The denotational approach to interpreters originates with [GTWW77]. This paper shows that the expression implementation is *initial* in the category of implementations of an ADT interface (see section 2.1.4). A consequence is that all syntaxes are isomorphic, and hence, from a mathematical point of view, syntax doesn't matter.

The presentation of languages as ADTs shows that, contrary to [ASS85] or even [Wad92], there is no real difference between "metalinguistic" abstraction and data abstraction. New syntax (even abstract syntax) is not necessary for new languages.

In essence, every ADT forms a new language, and vice-versa. Of course, we cannot have language without syntax; in fact, we reuse Scheme's syntax. For example, the expression

```
(%+ (%num 1) (%num 2))
```

has meaning in Scheme (directly) and in the interpreted language (using `compute`). With an extensible parser, we could make the interpreted language more readable. Finally, observe that the denotational style can be used in languages other than Scheme. With a little more work, we could implement the language ADT in C, and the result would be equally usable.

1.3 Monolithic interpreters

In this section, we examine the usual monolithic style of writing interpreters and show that it is non-modular. *Monolithic* means that a program is not textually divided into modules. *Non-modular* means that a local conceptual change requires a global textual change. Hence, monolithic is a syntactic property, while non-modular is a semantic property.

To see an example of non-modularity, we extend the language presented above to include stores. We add three new operations:

$$\begin{array}{lll} \texttt{\%begin} & : \mathit{Exp} \times \mathit{Exp} & \rightarrow \mathit{Exp} \\ \texttt{\%fetch} & : \mathit{Loc} & \rightarrow \mathit{Exp} \\ \texttt{\%store} & : \mathit{Loc} \times \mathit{Exp} & \rightarrow \mathit{Exp} \end{array}$$

The intuitive meanings of these operations are that `%begin` threads a store through two expressions in sequence, `%fetch` reads a value from the store, and `%store` writes a value into the store. We could define `%begin` using `%let` but, since they are operations in the same ADT, I prefer to give them equal status.

Figures 1.8 and 1.9 show a monolithic denotational implementation of the extended language. The store ADT, shown in figure 1.10, is almost identical to the environment ADT. The intuitive meaning of the base semantics

```
;; Den = Env → Sto → Val × Sto
;; Proc = Val → Sto → Val × Sto

(define (((%num n) env) sto)
  (pair n sto))

(define (((%var name) env) sto)
  (pair (env-lookup name env) sto))

(define (((%lambda name den) env) sto)
  (pair (lambda (val) (den (env-extend env name val)))
        sto))

(define (((%call d1 d2) env) sto)
  (with-pair ((d1 env) sto)
    (lambda (v1 s1)
      (with-pair ((d2 env) s1)
        (lambda (v2 s2)
          ((v1 v2) s2))))))

(define (((%if d1 d2 d3) env) sto)
  (with-pair ((d1 env) sto)
    (lambda (v1 s1)
      (if v1
          ((d2 env) s1)
          ((d3 env) s1))))))
```

Figure 1.8: Monolithic interpreter, part 1

$$Den = Env \to Sto \to Val \times Sto$$

is that an expression is interpreted relative to an environment and a store. For example, to evaluate `(%fetch 'a)`, we need to know what is stored in location `a`. In addition to returning a value, a denotation also returns an updated store.

Even though we have added only three new language constructs, the implementations of the other constructs change drastically. For instance, although numbers have nothing to do with stores, we are forced to write

```
(define ((((make-op op) d1 d2) env) sto)
  (with-pair ((d1 env) sto)
    (lambda (v1 s1)
      (with-pair ((d2 env) s1)
        (lambda (v2 s2)
          (pair (op v1 v2) s2))))))

(define %+ (make-op +))
(define %* (make-op *))

(define (((%begin d1 d2) env) sto)
  ((d2 env) (right ((d1 env) sto))))

(define (((%fetch loc) env) sto)
  (pair (store-fetch loc sto) sto))

(define (((%store loc den) env) sto)
  (with-pair ((den env) sto)
    (lambda (val sto)
      (pair 'unit (store-store loc val sto)))))

(define (with-pair p k)
  (k (left p) (right p)))
```

Figure 1.9: Monolithic interpreter, part 2

```
(define (empty-store) '())

(define (store-fetch loc sto)
  (let ((entry (assq loc sto)))
    (if entry
        (error "Empty location: " loc)
        (right entry))))

(define (store-store loc val sto)
  (pair (pair loc val) sto))
```

Figure 1.10: Store ADT

```
(define (((%num n) env) sto)
  (pair n sto))
```

in place of

```
(define ((%num n) env)
  n)
```

Thus we have an instance of non-modularity: a conceptually local change requires a textually global change.

1.4 Modular interpreters

Modular programs have several advantages over monolithic programs:

- They are easier to understand.
- They are easier to reason about.
- They are easier to extend and modify.

In this section, we describe two modular interpreters. We begin by examining the interpreter constructed in the last section. The basic semantics is

$$Den = Env \to Sto \to Val \times Sto$$

In this type, we distinguish three distinct "levels":

$$\begin{aligned} E &= Env \to Sto \to Val \times Sto \\ S &= Sto \to Val \times Sto \\ V &= Val \end{aligned}$$

Modularity is possible because most language constructs operate primarily at a single level. For example, `%var` operates on environments, `%+` operates on values, and `%store` operates on stores.

There are two methods for building a modular interpreter, which we describe by analogy with building a house. In both methods, we build a floor at a time, starting from the bottom. However, in the first, we move in our belongings (rugs, furniture, china, paintings, books) after we finish each floor. In the second, we wait until the house is complete before moving in. It's not surprising that the second method works better.

In the first method, we start with the value level and constructs such as `%+`. Then we add the stores level and more constructs such as `%fetch`. We also lift the value constructs up to the stores level (the interesting part). Then we add the environments level and constructs such as `%call`. We also lift the value and store constructs up to the environments level.

In the second method, we define operators for lifting values and functions between all pairs of levels. For example, `unitVE` lifts from values to environments. We can define these operators in stages, but it is easier to define them all at once. Then we use them to define each language construct in a single step, without lifting it through several levels.

In both interpreters, we use monads to relate pairs of levels. A *monad* is a triple $(T, \texttt{unit}, \texttt{bind})$ of a type constructor and two polymorphic operators

$$\begin{aligned}\texttt{unit} &: A \rightarrow T(A) \\ \texttt{bind} &: T(B) \times (A \rightarrow T(B)) \rightarrow T(B)\end{aligned}$$

The operators are required to obey several identities, as discussed in section 2.2. Two types A and B are *related* by a monad $(T, \texttt{unit}, \texttt{bind})$ if $B = T(A)$. `Unit` lifts values from A to B, and `bind` lifts functions of type $A \rightarrow B$ to functions of type $B \rightarrow B$. We can use `bind` to lift functions of other types also.

Let's examine what lifting means. Suppose that we have a function `square` : $Num \rightarrow Num$ and we want to define a function `square-list` : $List(Num) \rightarrow List(Num)$ that squares each number in a list. Given the list monad

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

(define (unit a)
  (list a))

(define (bind tb f)
  (flatten (map f tb)))
```

we can define `square-list` as

```
(define (square-list l)
  (bind l (lambda (n) (unit (square n)))))
```

We can perform this lifting using the standard Scheme `map` function, but monads can lift functions that `map` (and its generalizations) cannot, as shown in section 2.2.3. We present a formal definition of lifting in section 3.1.

1.4.1 Lifting interpreter

This section presents a modular interpreter built using lifting. Since stratification is simpler and more powerful, it may be better to skip to the next section on first reading.

The interpreter is shown in figures 1.11 – 1.14. The first figure shows a set of lifting operators. These accept a monad relating levels A and B and a function defined on A. They return a function defined on B. Functions may accept parameter types (written X), that are untouched by the lifting process. The parameters always come before the actual arguments. The operator `lift-pN-aM` lifts a function of N parameters and M arguments. For example, `lift-p1-a2` takes functions

$$\mathtt{f} : X \times A \times A \rightarrow A$$

to lifted functions

$$\mathtt{f}' : X \times B \times B \rightarrow B$$

The lifting operators assume that all functions return values of type A.

```
(define ((lift-p1-a0 unit bind op) p1)
  (unit (op p1)))

(define ((lift-p0-a1 unit bind op) d1)
  (bind d1
    (lambda (v1)
      (unit (op v1)))))

(define ((lift-p0-a2 unit bind op) d1 d2)
  (bind d1
    (lambda (v1)
      (bind d2
        (lambda (v2)
          (unit (op v1 v2)))))))

(define ((lift-p1-a1 unit bind op) p1 d1)
  (bind d1
    (lambda (v1)
      (unit (op p1 v1)))))

(define ((lift-if unit bind op) d1 d2 d3)
  (bind d1
    (lambda (v1)
      (op v1 d2 d3))))
```

Figure 1.11: Lifting operators

The second figure shows the values level and the constructs defined on it. The third figure uses the lifting operators and the stores monad to lift these operators to the level of stores. It also defines several new operators. The fourth figure does the same for environments. Using appropriate laws for reasoning about Scheme programs (essentially call-by-value lambda calculus), we can show that the final constructs are operationally equivalent to the monolithic definitions of figure 1.8.

Although the code for this interpreter is somewhat long, it is fairly modular. For example, the definitions of

- `%num`, `%+`, and `%*` do not involve environments or stores,

```
;;; V = Val

(define computeV id)

(define %numV id)
(define %+V +)
(define %*V *)

(define (%ifV d1 d2 d3)
  (if d1 d2 d3))
```

Figure 1.12: Value level

- `%fetch` and `%store` do not involve environments, and
- `%var`, `%lambda`, and `%call` do not involve stores.

We obtain modularity by lifting operators in a canonical way using `unit` and `bind`. Canonical means that operators with identical types have identical liftings. An exception is `%if`, which needs special treatment. Even in this case, the lifting of `%if` is uniform for all levels.

A more serious lack of modularity occurs when defining `%var`, `%lambda`, and `%call`. Here we use `unitS` and `bindS`, which were intended solely for the stores level. Also, we assume that environments contain values from the values level. Since the environment constructs interact with multiple levels, we say that they are *non-local.*

1.4.2 Stratified interpreter

The second interpreter is much simpler than the first. We define all language constructs using five operators that relate levels in pairs:

```
;;; S = Sto → V × Sto

;; Store monad

(define (unitS v)
  (lambda (sto)
    (pair v sto)))

(define (bindS s f)
  (lambda (sto)
    (let ((v*sto (s sto)))
      (let ((v (left v*sto))
            (sto (right v*sto)))
        ((f v) sto)))))

;; Lifted operators

(define (computeS den)
  (computeV (left (den (empty-store)))))

(define %numS   (lift-p1-a0 unitS bindS %numV))
(define %+S     (lift-p0-a2 unitS bindS %+V))
(define %*S     (lift-p0-a2 unitS bindS %*V))

(define %ifS (lift-if unitS bindS %ifV))

;; New operators

(define ((%fetchS loc) sto)
  (pair (store-fetch loc sto) sto))

(define ((%storeS loc den) sto)
  (let ((v*s (den sto)))
    (let ((v (left v*s))
          (s (right v*s)))
      (pair 'unit
            (store-store loc v s)))))

(define ((%beginS d1 d2) sto)
  (d2 (right (d1 sto))))
```

Figure 1.13: Store level

```
;;; E = Env → S
;;; Proc = V → S

;; Environment monad

(define (unitE s)
  (lambda (env) s))

(define (bindE e f)
  (lambda (env)
    ((f (e env)) env)))

;; Lifted operators

(define (compute den)
  (computeS (den (empty-env))))

(define %num   (lift-p1-a0 unitE bindE %numS))
(define %+     (lift-p0-a2 unitE bindE %+S))
(define %*     (lift-p0-a2 unitE bindE %*S))
(define %if    (lift-if unitE bindE %ifS))
(define %fetch (lift-p1-a0 unitE bindE %fetchS))
(define %store (lift-p1-a1 unitE bindE %storeS))
(define %begin (lift-p0-a2 unitE bindE %beginS))

;; New operators

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

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

(define ((%call d1 d2) env)
  (bindS (d1 env)
    (lambda (v1)
      (bindS (d2 env)
        (lambda (v2) (v1 v2))))))
```

Figure 1.14: Environment level

$$
\begin{aligned}
\texttt{unitSE} &: S \to E \\
\texttt{unitVS} &: V \to S \\
\texttt{unitVE} &: V \to E \\
\texttt{bindSE} &: E \times (S \to E) \to E \\
\texttt{bindVE} &: E \times (V \to E) \to E
\end{aligned}
$$

We have left out `bindVS` since we don't need it. These operators form an abstract data type of *computations*, from which we can build the usual language ADT. We could alternatively call it an ADT of *denotations*, but Moggi's work on monads sets a precedent for "computations", although we have altered his meaning somewhat.

Peter Mosses was the first author to describe an ADT abstracting the basic semantics of a language [Mos92]. What is new here is stratification, which has several advantages:

- We can define non-local language constructs more naturally.
- We can understand computations and language constructs via the structure that stratification provides.
- We can build stratified computation ADTs automatically from component modules.

We return to this approach in chapter 4.

Figure 1.15 shows the computation ADT for this semantics, and figures 1.16 and 1.17 show the language ADT built from it. Once again, this interpreter is observationally equivalent to the original monolithic interpreter. It is also somewhat non-modular; specifically, all constructs assume

- There are no levels above E.
- Level S is immediately below E.
- Level V is immediately below S.

Section 4.3 solves these modularity problems in the context of automatically generated interpreters by giving each level several names.

```
;;  E = Env → S
;;  S = Sto → V × Sto
;;  V = Val

(define ((unitSE s) env)
  s)

(define ((unitVS v) sto)
  (pair v sto))

(define (((unitVE v) env) sto)
  (pair v sto))

(define ((bindSE t f) env)
  ((f (t env)) env))

(define (((bindVE t f) env) sto)
  (let ((p ((t env) sto)))
    (let ((v (left p))
          (s (right p)))
      (((f v) env) s))))
```

Figure 1.15: Level-negotiating operators

1.5 Examples

The examples in this section show Semantic Lego's input / output behavior; the next two chapters explain the mechanisms behind it. We consider

- A full-featured, Scheme-like language,
- Three interactions between nondeterminism and continuations,
- Lamping's unified system of parametrization, and
- A parallel language modeled using resumptions.

```
;;  E = Env → S
;;  S = Sto → V × Sto
;;  V = Val
;;  Proc = V → S

(define (%num v)
  (unitVE v))

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

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

(define (%call d1 d2)
  (bindVE d1
    (lambda (v1)
      (bindVE d2
        (lambda (v2)
          (unitSE (v1 v2)))))))

(define (%if d1 d2 d3)
  (bindVE d1
    (lambda (v1)
      (if v1 d2 d3))))
```

Figure 1.16: Modular interpreter, part 1

1.5.1 A Scheme-like language

We construct an interpreter for a language with environments, call-by-value procedures, stores, continuations, nondeterminism, and errors. Figure 1.18 shows the complete language specification, the basic semantics, and two example expressions. SL automatically generates descriptions of the basic semantics in prefix form.

We build an interpreter in two steps. In essence, SL automates the manual methods used to build the stratified interpreter just shown. First, we define a computation

```
(define ((make-op op) d1 d2)
  (bindVE d1
    (lambda (v1)
      (bindVE d2
        (lambda (v2)
          (unitVE (op v1 v2)))))))

(define %+ (make-op +))
(define %* (make-op *))

(define (%begin d1 d2)
  (bindVE d1
    (lambda (v1)
      d2)))

(define (%fetch loc)
  (unitSE
    (lambda (sto)
      (pair (store-fetch loc sto) sto))))

(define (%store loc den)
  (bindVE den
    (lambda (val)
      (unitSE
        (lambda (sto)
          (pair 'unit (store-store loc val sto)))))))
```

Figure 1.17: Modular interpreter, part 2

ADT using `make-computations`, which accepts a list of semantic modules. The resulting ADT is just a collection of appropriately named `unit` and `bind` operators.

Second, we load several files of language constructs. These define the language ADT using operators extracted from the computation ADT. These definitions are similar to those of the last section. Constructs may be defined over any computation ADT 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 defined over different computation ADTs.

```
;; Computation ADT

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

;; Language ADT

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

;; Basic semantics

(show-computations)
⇒ (-> Env
       (-> Sto
           (let A0 (* Val Sto)
             (let A1 (+ (List A0) Err)
               (-> (-> 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))
           (%unit))
      (%store 'r (%* (%fetch 'r) (%fetch 'n)))
      (%store 'n (%- (%fetch 'n) (%num 1)))))))))
⇒ (24 120)
```

Figure 1.18: 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 1.19: `%let` source definition

A typical construct is `%let`, whose source definition (from the `environments` file) is shown in figure 1.19. We have not yet described enough of SL to explain this definition in detail, but its form should be clear. Appendix B.2 shows the definition of each construct presently available in SL.

Although Scheme procedures are usually opaque, MIT Scheme allows us to reify them as abstract syntax. We then apply a program simplifier that performs inlining and β and η reduction. By inlining the operators of the computation ADT and simplifying, we automatically generate denotational-style definitions of language constructs.

The result of simplifying `%let` in the context of the specified computation ADT, shown in figure 1.20, is exactly what we would have written by hand. The whole point of SL is that the source definition of `%let` did not mention stores or continuations, yet they were introduced properly and automatically.

1.5.2 Nondeterminism and continuations

In this section, we use SL to explore the interaction between nondeterminism and continuations. We use three different computation ADTs but leave the definitions

```
(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 1.20: `%let` definition simplified

```
(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 1.21: `%amb` source definition

of all language constructs unchanged. For reference, figure 1.21 gives the source definition of `%amb`. For each semantics, we show the modules forming the computation ADT, the basic semantics, the simplified version of `%amb`, and the evaluation of an example program.

In the first semantics (figure 1.22), the subexpressions of `%amb` are evaluated 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`.

In the second semantics (figure 1.23), 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 the original continuation in the appropriate way. `Continuations2` passes the original continuation down directly and applies the operator to the results. The evaluation of the example in this semantics is clear.

In the third semantics (figure 1.24), 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.

1.5.3 Unified system of parametrization

In this section, we use SL to realize John Lamping's "Unified System of Parametrization" [Lam88]. Lamping describes a semantics in which expressions are parametrized over variables that (recursively) denote expressions. This recursion models a substitution in which substituted terms can contain variables. The language also includes call-by-name static environments; hence the basic semantics is

$$Den = Env \rightarrow EEnv \rightarrow Val$$

where both Env and $EEnv$ contain $EEnv \rightarrow Val$. Figure 1.25 shows the SL language specification and the semantics of `%evar` and `%elet`, which are used to form expressions. The line marked `***` is especially interesting. Figure 1.26 shows several examples.

1.5.4 Resumptions

Resumptions are a denotational model of interruptable execution sequences. The basic structure of a resumption semantics is

$$Den = \mathit{fix}(X)\ T(Val + X)$$

```
;; Computation ADT

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

;; Basic semantics

(-> Env (let A0 (List Ans) (-> (-> 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 1.22: `%amb` version 1

where T is a type constructor describing other features present in the language. This construction means that a computation either *terminates*, producing a value, or *pauses*, producing a computation with which to continue. The typical use of resumptions is to interleave several computations by executing one until it pauses, then executing the next, etcetera.

The standard parallel semantics, described in [Sch86] has

$$T(A) = Sto \to List(A \times Sto)$$

```
;; Computation ADT

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

;; Basic semantics

(-> Env (let A0 (List Ans) (-> (-> 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 1.23: `%amb` version 2

so that computations accept and return stores and can fork into multiple computations. Thus the complete type of denotations is

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

Figure 1.27 shows the SL specification for this language, along with several examples. An expression evaluates to a list of values, one for each possible order of execution. The definition of `%par` appears in appendix B.2, figure B.27; we could show its expansion, but it is not especially enlightening.

```
;; Computation ADT

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

;; Basic semantics

(-> Env (let A0 (List Ans) (-> (-> (List Val) 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 1.24: %amb version 3

```
;; Computation and language ADTs

(define computations
  (make-computations cbn-environments exp-environments))

(load "error-values" "numbers" "booleans" "numeric-predicates"
      "environments" "exp-environments")

;; Simplified %evar  and %elet

(lambda (name)
  (lambda (env)
    (lambda (eenv)
      (if (env-lookup eenv name)
          ((right (env-lookup eenv name)) eenv)   ; ***
          (in 'errors (unbound-error name))))))

(lambda (name c1 c2)
  (lambda (env)
    (lambda (eenv)
      ((c2 env) (env-extend eenv name (c1 env))))))
```

Figure 1.25: Unified system of parametrization

```
(compute
 (%let 'f (%* (%evar 'x) (%evar 'x))
    (%+ (%elet 'x (%num 3) (%var 'f))
        (%elet 'x (%num 4) (%var 'f)))))
⇒ 25

(compute
 (%let 'g (%+ (%evar 'a) (%evar 'a))
   (%let 'f (%elet 'a (%* (%evar 'x) (%evar 'x))
               (%var 'g))
     (%elet 'x (%num 3) (%var 'f)))))
⇒ 18
```

Figure 1.26: Unified parametrization examples

```
;; Computation and language ADTs

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

(load "error-values" "numbers" "booleans" "begin" "while"
   "products" "numeric-predicates" "amb" "stores" "resumptions")

;; Examples

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

(compute
 (%seq
  (%store 'x (%unit))
  (%par
   (%store 'x (%pair (%num 3) (%fetch 'x)))
   (%store 'x (%pair (%num 2) (%fetch 'x)))
   (%store 'x (%pair (%num 1) (%fetch 'x))))
  (%fetch 'x)))
⇒ ((pair 3 (pair 2 (pair 1 unit)))
    (pair 2 (pair 3 (pair 1 unit)))
    (pair 3 (pair 1 (pair 2 unit)))
    (pair 1 (pair 3 (pair 2 unit)))
    (pair 2 (pair 1 (pair 3 unit)))
    (pair 1 (pair 2 (pair 3 unit))))

(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)))
⇒ (2 3 4 5 6 7 7 1)
```

Figure 1.27: Parallel language using resumptions

Chapter 2

Monads

In this chapter, we first present some basic category theory, then discuss monads, maps between monads, monad composition, and monad transformation. This may sound like "everything you always wanted to know about monads", but in reality it barely scratches the surface. See [BW85, Mog89a] for more information.

Monads may be a "hot topic" in the 1990's functional programming community, but the real "monad explosion" occurred in the category theory / algebraic topology community during the 1960's, when they were first invented. Although I consider myself a fairly good "monad hacker" by 1990's standards, I have to admit that I'm not even on the 1960's chart. Even so, I find hard to hear computer scientists ask, "Monads – aren't those about state?". That's like asking, "Algebra – isn't that about $1 + 1 = 2$?".

2.1 Basic category theory

In this section, we define the basic notions of category theory, discuss the relationship between category theory and functional programming, and mention some references.

2.1.1 Categories

Categories abstract the composition of typed functions. A *category* is a set of *objects* (these are the types), a set of *arrows* (these are the functions), and a composition operator on the arrows. Each arrow points from one object (its domain) to another (its codomain). If $f : A \to B$ and $g : B \to C$ are two arrows then $g \circ f : A \to C$ is their composition. There is a distinguished identity arrow from each object to itself. Composition must be associative with the identity arrows as left and right identities.

The basic category that we use depends on whether we are doing semantics or functional programming. In semantics, we use a suitable domain theory (see [Gun92]). In functional programming, we use the types and functions of our language, in this case, Scheme [CR91]. Scheme has no explicit types, so we have to imagine them ourselves.

In a category, composition is primary, rather than application. In functional programming, it's the reverse, unless we program in a combinator language. This change in point of view causes few problems in practice; we use whichever is most convenient.

2.1.2 Functors

In category theory, whenever we define a class of objects, we also define the appropriate maps between them, thus making them into a category. For this reason, we now consider maps between categories.

A *function* T between categories $\mathcal{C}$ and $\mathcal{D}$ is a map from $\mathcal{C}$'s objects to $\mathcal{D}$'s objects. An *endofunction* is a function from a category to itself. In our case, an endofunction is a type constructor; it builds one type from another. For example, $T(A) = \mathit{List}(A)$ builds lists of any type we like. The other type constructors we use are function space ($\to$), products ($\times$), and sums ($+$).

Functions are insufficient as maps between categories because they have no action on arrows. We define a *functor* $T : \mathcal{C} \to \mathcal{D}$ to be a function, also called T, along with an function mapT from $\mathcal{C}$'s arrows to $\mathcal{D}$'s arrows such that

```
;; mapT :  (A → B) → (T(A) → T(B))

(mapT id)         = id
(mapT (oC g f)) = (oD (mapT g) (mapT f))
```

An *endofunctor* is a functor from a category to itself, so that we need only one composition operator. For example, the ordinary map function on lists makes $T(A) = List(A)$ into an endofunctor. Functors are the appropriate class of maps between categories, since they respect identities and composition structure. Other functors are the pairing functor

```
;; T(A) = A × A

(define ((map f) ta)
  (pair (f (left ta)) (f (right ta))))
```

and the environment functor, which parametrizes a type by an environment:

```
;; T(A) = Env → A

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

2.1.3 Natural transformations

A *natural transformation* from a functor S to a functor T is a polymorphic function

$$\texttt{sigma} : S(A) \to T(A)$$

such that

```
(o sigma (mapS f)) = (o (mapT f) sigma) :  S(A) → T(B)
```

for all $\texttt{f} : A \to B$. It is easy to remember this law as "sigma commutes with map". Examples include

$$
\begin{array}{ll}
\texttt{reverse} & : List(A) \rightarrow List(A) \\
\texttt{flatten} & : List(List(A)) \rightarrow List(A) \\
\texttt{list} & : A \rightarrow List(A) \\
\texttt{left} & : A \times A \rightarrow A \\
\texttt{diag} & : A \rightarrow A \times A
\end{array}
$$

where `list` is natural from Id to $List$, `left` is natural from pairing to Id, and `diag` is natural from Id to pairing.

In categorical terms, a natural transformation is a map from objects to arrows. Given an object A, we obtain an arrow `sigmaA`: $S(A) \rightarrow T(A)$. In other words, we obtain a family of functions, indexed by type. The naturality condition above structures our choice of arrows; we cannot pick arbitrarily. This yields *parametric* polymorphism rather than *ad-hoc* polymorphism. For further information, see [Wad89].

2.1.4 Initiality

An object in a category is *initial* if there is a unique arrow from it to each object of the category. A object is *terminal* is there is a unique arrow *to* it from each object. Initial and terminal objects are unique up to *isomorphism* if they exist. Two objects A, B of a category are *isomorphic* if there exist arrows $f : A \rightarrow B$ and $g : B \rightarrow A$ such that $g \circ f = Id_B$ and $f \circ g = Id_A$.

For example, in the category of sets and total functions, the empty set is initial and any one-element set is terminal. Notice that there are many one-element sets, all of which are isomorphic. Initiality will not see much use in this thesis, although it is perhaps *the* fundamental concept of category theory.

2.1.5 Duality

Given a category C, we can form its *dual* $\mathcal{C}^{op}$ by reversing the direction of each arrow and the order of composition. Needless to say, this operation is quite difficult in ordinary functional programming. If an object is initial in C, it is terminal in $\mathcal{C}^{op}$, and vice-versa. Hence we say that initial and terminal are *dual* concepts. Other well-known dual concepts are products / sums and injective / surjective. In general, we can form the dual of any concept formulated solely in category-theoretic terms.

In his brilliant master's thesis [Fil89], Filinski shows that values and continuations are dual. Although it is difficult to develop an intuition for his language, his thesis contains many surprising insights.

2.1.6 Category theory and functional programming

It is important to remember that, at least for the moment, mathematics and programming are two different activities. The main problem is that current languages provide no automated support for representing and verifying properties of programs.

In this thesis, we embed category theory within functional programming in a particular way: objects are types, and arrows are functions. Other embeddings are possible; for example, see [RB90], which represents objects as values. Their approach is less straightforward, but more flexible.

Our choice of embedding has several problems:

- Current languages have weak, non-existent, or implicit type systems (see section A.1). In category theory, we can form categories with any kind of objects at all.

- It may not be easy to represent categorical composition as functional composition. Also, we cannot represent uncomputable compositions.

The clearest and most comprehensive treatment of category theory and functional programming in this embedding is [Spi93]; hopefully, Spivey will soon publish these handwritten notes in electronic or book form. A much-abbreviated version appears as [Spi89].

2.1.7 References

General references on category theory for computer science are [Pie91] and [BW90]. The latter contains many examples and applications and is easy despite its length.

Category theory is not terribly hard to learn, because its rich descriptive content encourages the reader to acquire concepts one at a time, relating each to already-understood notions from other fields.

Category theory may be considered part of abstract algebra. MacLane and Birkhoff's larger book [MB88] is a wonderful introduction to algebra, not only because it presents category theory near the end, but because it applies category-theoretic insight throughout.

2.2 Monads

In this section, we present two formulations of monads and discuss the intuitions behind them. Monads are functors with additional structure, in the same way that functors are functions with additional structure.

2.2.1 First formulation

A monad is a triple[1] (T, `unit`, `join`) of an endofunctor and two natural transformations

$$\begin{aligned}\texttt{unit} &: A \to T(A)\\ \texttt{join} &: T(T(A)) \to T(A)\end{aligned}$$

`Unit` is natural from the identity to T and maps values into T. For example, `unit` for the list monad is `list`. `Unit` is not required to be injective, although it actually is in most applications. `Join` is natural from $T \circ T$ to T and flattens multiple T's into a single T. `Join` for the list monad is `flatten`.

`Unit` and `join` for the environment monad $T(A) = Env \to A$ are

```
(define ((unit a) env)
  a)

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

[1] Monads are also called *triples*.

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	$\mathit{fix}(X)\ (A + X)$

Table 2.1: Monad type constructors

`Unit` and `join` must satisfy the additional properties

```
(o join unit)       = id              :  T(A) → T(A)
(o join (map unit)) = id              :  T(A) → T(A)
(o join (map join)) = (o join join)   :  T(T(T(A))) → T(A)
```

This formulation presents monads as modified monoids [Mac71] (whence the name), where `unit` is the identity and `join` is the monoid operator. The above laws are left and right identity and associativity.

Table 2.1 shows the type constructors for some common monads used in semantics. We describe their `unit` and `join` operators in the next section (via the second formulation).

2.2.2 Second formulation

We can also describe a monad as a triple $(T, \texttt{unit}, \texttt{bind})$ where T is an *endofunction*, `unit` is a family of arrows that is not necessarily required to be natural, and `bind` is a map between sets of arrows:

$$\texttt{unit} : A \to T(A)$$
$$\texttt{bind} : (A \to T(B)) \to (T(A) \to T(B))$$

`Unit` functions exactly as before. Just as `map` takes functions from $A \to B$ into $T(A) \to T(B)$, `bind` takes functions of the more general form $A \to T(B)$ into $T(A) \to T(B)$. `Unit` and `bind` obey several properties:

```
;; f :  A → T(B)
;; g :  B → T(C)

(bind unit)               = id                       :  T(A) → T(A)
(o (bind f) unit)         = f                        :  A → T(B)
(o (bind g) (bind f)) = (bind (o (bind g) f)) :  T(A) → T(C)
```

To show the equivalence of the two monad formulations (`bind` versus `map` and `join`), we can write

```
(define ((map f) ta) (bind ta (o unit f)))

(define (join tta)   (bind tta id))

(define (bind ta f)  (join ((map f) ta)))
```

Proving the two sets of laws equivalent is easy.

The second formulation is easier to understand if we rephrase it in terms of the *Kleisli composition* `oT` on the space of functions $A \rightarrow T(B)$:

```
;; oT :  (B → T(C)) × (A → T(B)) → (A → T(C))

(define ((oT g f) a)
  (bind (f a) g))
```

Then the laws become

```
(oT unit f)      = f
(oT f unit)      = f
(oT h (oT g f)) = (oT (oT h g) f)
```

In other words, `oT` is associative and has `unit` as left and right identity. Thus, we can form a *Kleisli category* whose objects are types and whose arrows are functions of the form $A \rightarrow T(B)$, with `oT` as composition. Like all systems of functional combinators, Kleisli composition is useful for stating and deriving laws but is unwieldy for writing programs.

Figures 2.1 and 2.2 define `unit` and `bind` for the type constructors shown in table 2.1. These figures use an applicative (rather than compositional) version of `bind` obtained by uncurrying and reversing arguments:

$$\texttt{bind} : T(A) \times (A \rightarrow T(B)) \rightarrow T(B)$$

```
;; Identity: T(A) = A

(define (unit a)
  a)

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

;; Lists: T(A) = List(A)

(define (unit a)
  (list a))

(define (bind ta f)
  (reduce append '() (map f ta)))

;; Environments: T(A) = Env → A

(define (unit a)
  (lambda (env) a))

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

;; Stores: T(A) = Sto → A × Sto

(define (unit a)
  (lambda (sto) (pair a sto)))

(define (bind ta f)
  (lambda (sto)
    (let ((a*s (ta sto)))
      (let ((a (left a*s))
            (s (right a*s)))
        ((f a) s)))))
```

Figure 2.1: Example monads, part 1

```
;; Exceptions: T(A) = A + X

(define (unit a)
  (in-left a))

(define (bind ta f)
  (sum-case ta
    (lambda (a) (f a))
    (lambda (x) (in-right x))))

;; Monoids: T(A) = A × M

(define (unit a)
  (pair a monoid-unit))

(define (bind ta f)
  (let ((a1 (left ta))
        (m1 (right ta)))
    (let ((a*m (f a1)))
      (let ((a2 (left a*m))
            (m2 (right a*m)))
        (pair a2 (monoid-product m1 m2))))))

;; Continuations: T(A) = (A → Ans) → Ans

(define (unit a)
  (lambda (k) (k a)))

(define (bind ta f)
  (lambda (k) (ta (lambda (a) ((f a) k)))))

;; Resumptions: T(A) = fix(X)(A + X)

(define (unit a)
  (in-left a))

(define (bind ta f)
  (sum-case ta
    (lambda (a) (f a))
    (lambda (ta) (bind ta f))))
```

Figure 2.2: Example monads, part 2

2.2.3 Interpretations

In this section, we give several interpretations of monads. Moggi's model, which we discuss last, is the one most relevant to this thesis.

Monads resemble monoids

MacLane [Mac71] describes monads as a variation on monoids, with `unit` as the identity and `join` as the monoid product. A *monoid* product (on a type constructor) has type

$$* : T \times T \rightarrow T$$

For example, consider `append` on $List(A)$. A *monad* product has type

$$\texttt{join} : T \circ T \rightarrow T$$

For example, consider `flatten`. That these notions have a common generalization is rather odd.

Monads model substitution

Suppose $T(A)$ is a type of arithmetic expressions over a set of variables A:

$$T(A) = A \mid T(A) + T(A) \mid T(A) * T(A)$$

Then `unit` transforms a variable into an expression, and `bind` performs substitution. A substitution is a map $A \rightarrow T(B)$ that gives an expression $T(B)$ over B for each variable of A. `Bind` takes an expression over A and a substitution and returns an expression over B.

`Join` also performs substitution by flattening "expressions over expressions". `Bind` accomplishes everything that `join` does, but we never have to see more than one application of T.

Monads model lifting

We can view `bind` as a generalization of `map`. The naturality of `unit` means that

```
(unit (f a)) = ((map f) (unit a))
```

In other words, it doesn't matter whether we apply `f` before `unit` or `(map f)` after. We say that `(map f)` is a *lifting* of `f` through `unit`. We give a very general definition of lifting in chapter 3.

Just as `map` lifting functions of the form $A \to B$, we can say that `bind` lifts functions of the more general form $A \to T(B)$, and the first two monad laws ensure a property similar to the above. `Bind` can also lift functions on products. For example, we can lift `+` to act on lists of numbers using the list monad:

```
(define (list+ l1 l2)
  (bind l1
    (lambda (n1)
      (bind l2
        (lambda (n2)
          (unit (+ n1 n2)))))))
```

This definition is not possible using `map` alone. However, with a map

$$\texttt{product} : T(A) \times T(B) \to T(A \times B)$$

we can write

```
(define (list+ l1 l2)
  (map (product l1 l2)
    (lambda (n1*n2)
      (+ (left n1*n2) (right n1*n2)))))
```

A purely categorical model of lambda calculus over a monad (see [Mog89b]) actually requires a "tensorial strength" similar to `product`, even when using `bind`. Thus, some of the power of `bind` (when compared to `map`) comes from Scheme, rather than from pure category theory.

Monad	A computation exists when it
Lists	Produces a single value
Lifting	Terminates
Environments	Is independent of environment
Stores	Leaves the store unchanged
Exceptions	Causes no exceptions
Output	Doesn't output anything
Continuations	Invokes its continuation exactly once
Resumptions	Terminates in one step

Table 2.2: Meaning of existence

Monads model computation

Moggi's insight was that $T(A)$ represents a *computation* of a value of type A. For example, a nondeterministic computation produces not just a single value but a set of possible values. `Unit` lifts a value to a computation that produces that value (and does nothing else). `Join` flattens computations of computations into single computations. `Bind` composes functions from values to computations.

Moggi also made the following definition: a computation *is a value* or *exists* if it is in the image of `unit`, that is, if it equals (`unit` v) for some value v. Thus, we can say that a nondeterministic computation exists if it produces only a single value. Table 2.2 shows the meaning of existence for other monads.

If `unit` is a way into a monad, we also need a way out. We cannot have a map from computations to values because we might want to see more than one value or perhaps know what final store a computation produced. Thus, we augment all our monads with a map

$$\texttt{compute} : T(A) \times (A \to \mathit{Rep}) \to \mathit{Rep}$$

where *Rep* is a universal representation type designed for users to read. In many languages, this type would be *String*, but in Scheme we use lists, numbers, etcetera. An alternative is

$$\texttt{compute} : T(Rep) \to Rep$$

in which we `map` a function of type $A \to Rep$ across $T(A)$, then apply `compute`. Although this approach is direct, it conflicts with the computational analogy, since $T(Rep)$ is a computation of a representation rather than a computation of a value.

We do not discuss `compute` in the rest of the thesis, but section B.1 shows its definition for each of the monad transformers we use.

Why monads?

Moggi's intuition of "values and computations" does not quite explain why we need `join` in addition to `map` and `unit`. In fact, we need it to abstract over computations using functions.

In a language with a nondeterminism operator `amb`, modeled by the semantics

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

consider the program

```
(define (f n)
  (g (amb n (+ n 1))))

(define (g n)
  (amb n (* n 2)))
```

The function `g`, although it accepts values, must return computations, since it uses `amb`. The function `f` must apply `g` to a computation rather than a value. Thus, we need a function

$$\texttt{apply} : (Val \to Comp) \times Comp \to Comp$$

This function is essentially `bind`.

2.3 Monad morphisms

In keeping with the "categorical imperative" that arrows between objects are as important as the objects themselves, we define arrows between monads. Thus, we form a category of monads and monad morphisms.

Since Kleisli categories were helpful in developing the monad laws, we suppose that an arrow between monads S and T is a functor K between their Kleisli categories. K acts as the identity on objects. On arrows, we have

$$\texttt{mapK} : (A \to S(B)) \to (A \to T(B))$$

satisfying the functorial properties

```
;; f :  A → S(B)
;; g :  B → S(C)

(mapK idS)        = idT
(mapK (oS g f)) = (oT (mapK g) (mapK f))
```

We can reformulate this definition in terms of `unit`, `bind`, and a natural transformation $K : S(A) \to T(A)$, in which case

```
(K (unitS a))     = (unitT a)
(K (bindS sa f)) = (bindT (K sa) (o K f))
```

An example of an arrow between monads is `reverse` from the list monad to itself:

```
(reverse (list a))           = (list a)
(reverse (append-map f l)) = (append-map (o reverse f) (reverse l))
```

An example of a natural transformation from the list monad to itself that is *not* an arrow between monads is `(lambda (l) '())`, which fails the first law.

2.4 Monads don't compose

Given the variety of semantic features that monads offer, it seems that we should have no trouble building all sorts of languages. Unfortunately, monads don't compose.

```
;; S(A) = EnvS → A
;; T(A) = EnvT → A
;; ST(A) = EnvS → EnvT → A

(define ((joinS ssa) envS)
  ((ssa envS) envS))

(define ((joinT tta) envT)
  ((tta envT) envT))

(define (((joinST ststa) envS) envT)
  ((((ststa envS) envT) envS) envT))
```

Figure 2.3: Monads don't compose

This may seem odd, since functors *do* compose. `Unit` also composes, but `join` and `bind` do not.

As an example, let's try to compose two environment monads S and T. Figure 2.3 shows `join` for the individual monads and for their composition. It is clear by inspection that the latter cannot be defined from the former, even using `unit` and `map`.

Jones and Duponcheel [JD93] give a rigorous proof based on the propositions as types analogy, showing that the type of `joinST` is not provable in implicational logic from the types of the other operators. However, if we think of monads as generalized monoids, that is, as acting on sequences of S's and T's, we realize that `unit` introduces an S or T, `join` flattens SS or TT, and `map` allows us to act anywhere in a sequence. Is is clear that we cannot reduce $STST$ to ST using these operators, since they never decrease the number of S/T boundaries.

2.5 Monads do compose

In the last section, we showed that, given two monads S and T, there is no way to form a monad on ST using only the operators of S and T. There are three equivalent ways around this difficulty, via *distributive laws*, *liftings*, and *compatibility*.

A *distributive law* is a map

$$\texttt{swap} : TS \to ST$$

that distributes T over S and obeys several side conditions. Clearly, this map allows us to reduce arbitrary sequences of S's and T's to a single pair.

A *lifting* of S over T is a monad on the Eilenberg-Moore category of T-algebras, which we will not discuss. It may also be possible to lift S onto T's Kleisli category, although the construction would be less direct. It may also be possible to present monad transformers (section 2.6) as liftings of this form.

Finally, we can describe conditions under which ST is *compatible* with S and T. There are two formulations of these conditions. The first, due to Barr [BW85] (page 315), is

$ST = S \circ T$
```
map = mapS o mapT

(C1) unitST = unitS o unitT = mapS(unitT) o unitS

(C2) joinST o mapST(unitS) = mapS(joinT)
(C3) joinST o mapS(unitT)  = joinS

(C4) joinS o mapS(joinS)          = joinST o joinS
(C5) joinST o mapST(mapS(joinT)) = mapS(joinT) o joinST
```

Drawn as diagrams, these laws are just triangles and squares, which are sufficiently described by their types:

$$\begin{array}{ll}
(\mathrm{C1}) : Id & \to ST \\
(\mathrm{C2}) : STT & \to ST \\
(\mathrm{C3}) : SST & \to ST \\
(\mathrm{C4}) : SSTST & \to ST \\
(\mathrm{C5}) : STSTT & \to ST
\end{array}$$

The second formulation, due to Beck [Bec69], requires that the two maps

$$\begin{array}{ll}
\texttt{unitS} & : T \to ST \\
\texttt{mapS(unitT)} & : S \to ST
\end{array}$$

be monad morphisms (see section 2.3) and that the *middle unitary law* holds:

`joinST o mapS(unitT o unitS) = id :` $ST \to ST$

Distributive laws and liftings were discovered by Beck [Bec69]. Both Beck [Bec69] and Barr [BW85] prove that liftings, distributive laws, and compatibility are equivalent. Why Barr used a different formulation of compatibility is unclear[2]. Barr studied distributive laws at the same time as Beck (see Beck's paper); perhaps the two sets were derived independently. It remains to verify that the conditions are indeed equivalent.

Jones and Duponcheel's paper [JD93] is entirely about distributive laws but misses the earlier references. This omission is unsurprising since the relevant section in [BW85] occurs late in the book and is titled "distributive laws" rather than "monad composition".

Using compatibility, we can develop monad composition as a relation rather than a function. The composition of two monads is thus the *set* of all monads compatible with them. The compatibility laws imply that composing a monad T with the identity yields exactly T and no other monads. I have not been able to show associativity – further conditions may be necessary.

If associativity holds, we can form a *relational category* whose arrows are monads. The notion of a relational category is obvious but appears not have been studied before (perhaps it lacks sufficient structure to be interesting). A request for references to a large mailing list of category theorists yielded few replies. One was directly relevant: Martin Wirsing (Munich) mentioned that an (unnamed) student of his is studying the idea in relation to nondeterminism and that his/her thesis is expected during the summer of 1995.

Needless to say, the compatibility laws don't yield a method for composing monads. We cannot even verify computationally that three monads are compatible, since we cannot check equality of functions. For the same reason, we cannot check that one function is a lifting of another (section 3.1). Nevertheless, the compatibility laws do constrain possible compositions and allow us to reason about them.

[2] Barr graciously replied to my questions but doesn't recall why his conditions are different.

2.6 Monad transformers

Since monad composition fails to be constructive, we try monad transformation; categorically speaking, if monads aren't arrows, let them be objects. That is, we build monads from other monads. After motivating the basic concepts, we formalize our constructions.

2.6.1 Motivation

For example, the environment monad transformer, shown in figure 2.4, adds an environment to any monad. We write the environment transformer as

$$F(T)(A) = \mathit{Env} \rightarrow T(A)$$

which indicates that it accepts a monad T and returns a new monad $F(T)$. The action of $F(T)$ on a type A is as shown above.

Applying monad transformers for EnvS and EnvT to the identity monad yields precisely the monad for both environments, with `joinST` as shown in figure 2.3. Thus, we immediately see an example of a construction we could not previously make.

The transformation of `join` is fairly complex. It accepts an argument

$$\texttt{ftfta} : \mathit{Env} \rightarrow T(\mathit{Env} \rightarrow T(A))$$

and forms a value of type $T(T(A))$ in order to use `joinT`. Thus it reduces $\mathit{Env} \rightarrow T(A)$ to $T(A)$ *inside* $T(\mathit{Env} \rightarrow T(A))$ using `mapT`.

Other monad transformers are listed in table 2.3, one for each monad in tables 2.1 and 2.2. More precisely, applying the X monad transformer to the identity monad yields the X monad, where X is environments, stores, etcetera. Appendix B.1 shows the definition of each transformer. As usual, composition of transformers is not commutative, and we shall make creative use of this fact later.

To illustrate the need for monad transformers further, we model a language with nondeterminism and state using the type of denotations

$$\mathit{Den}(A) = \mathit{Sto} \rightarrow \mathit{List}(A \times \mathit{Sto})$$

```
;;  F(T)(A) = Env → T(A)

(define (environment-transformer m)
  (let ((unitT (monad-unit m))
        (mapT  (monad-map m))
        (joinT (monad-join m)))

      (define (unit a)
        (lambda (env) (unitT a)))

      (define ((map f) fta)
        (lambda (env)
          ((mapT f) (fta env))))

      (define (join ftfta)
        (lambda (env)
          (joinT
           ((mapT (lambda (fta) (fta env)))
            (ftfta env)))))

      (make-monad unit map join)))
```

Figure 2.4: Environment monad transformer

Transformer	Action on types $F(T)(A) =$
Identity	$T(A)$
Nondeterminism	$T(List(A))$
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	$fix(X)\ T(A + X)$

Table 2.3: Monad transformers

`Unit` for this type is

```
(define (unit a)
  (lambda (sto) (list (pair a sto))))
```

but it is clear that it cannot be defined from

```
(define (unitS a)
  (lambda (sto) (pair a sto)))

(define (unitL a)
  (list a))
```

In fact, there is no way to build $Den(A)$ by composition. The only monad with type constructor $T(A) = A \times Sto$ has

```
(define (unitT a)
  (pair a (empty-store)))
```

which is useless. On the other hand, we can easily construct this type by composing the store and nondeterminism transformers.

2.6.2 Formalization

Since we have already constructed a category of monads, we have several choices for defining monad transformers. They could be functions (on objects), functors (with an action `map` on arrows), premonads (functors with `unit`), or even monads. We develop these ideas in sequence and show that monads on the category of monads are less complex than they sound. Formally, we define a *monad transformer* to be a premonad on the category of monads, since there is at least one case of a useful monad transformer (stores) that is not quite a monad in the higher category.

A monad transformer's action on objects is to create a monad $F(T)$ from a monad T. Its action on arrows is to send an arrow `K` $: S \to T$ between monads to an arrow `(mapF K)` $: F(S) \to F(T)$ such that the functorial properties are satisfied. For example, the action of the list monad transformer on arrows is

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

(define ((mapF K) fta)
  (map K fta))
```

Its action on objects (monads) is more complex and is given in appendix B.1. For each monad transformer F, we require a natural transformation

$$\texttt{unitF} : T(A) \to F(T)(A)$$

from the identity to F. `UnitF` allows us to lift values from $T(A)$ in $F(T)A$.

To lift functions from T to $F(T)$ we can either use `mapF`, or, if possible, define a map

$$\texttt{bindF} : F(T)(A) \times (T(A) \to F(T)(B)) \to F(T)(B)$$

obeying the usual laws. `BindF` makes F into a monad on the category of monads.

Before we faint from lack of air at these dizzying heights of abstraction, let's take the environment monad transformer as an example. First we consider its action on objects (monads):

```
;; F(T)(A) = Env → T(A)

;; unitFT :  A → F(T)(A)
;; bindFT :  F(T)(A) × (A → F(T)(B)) → F(T)(B)

(define (unitFT a)
  (lambda (env) (unitT a)))

(define (bindFT fta f)
  (lambda (env)
    (bindT (fta env)
      (lambda (a)
        ((f a) env)))))
```

Next we consider its action on arrows (between monads):

```
;; F(T)(A) = Env → T(A)

;; mapF : (S(A) → T(A)) → (F(S)(A) → F(T)(A))

(define (((mapF K) fsa) env)
  (K (fsa env)))
```

And finally, we consider `unitF` and `bindF`:

```
;; F(T)(A) = Env → T(A)

;; unitF : T(A) → F(T)(A)
;; bindF : F(T)(A) × (T(A) → F(T)(B)) → F(T)(B)

(define (unitF ta)
  (lambda (env) ta))

(define (bindF fta f)
  (lambda (env)
    ((f (fta env)) env)))
```

This last definition is in fact identical to that of the usual environment monad (see figure 2.1). `UnitF` and `bindF` are thus much simpler than `unitFT` and `bindFT`. As we pass to higher levels of abstraction, the definitions become simpler but their types become more complex. Essentially, the more polymorphic a function is, the less it knows about its arguments, so the less it can do (see section 1.1.6). As before, `mapF` is unnecessary if we have `unitF` and `bindF`.

2.6.3 Classes of monad transformers

Suppose F transforms type constructors. It may not be possible to extend F's action to monads. For example, we can extend

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

but not

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

We do not give a rigorous proof, but let's try it and see what happens:

Type	Form
Bottom	$F(T) = T \circ U$
Top	$F(T) = S \circ T$
Around	$F(T) = S \circ T \circ U$

Table 2.4: Monad transformer classification

```
(define (bindFT fta f)
  (bindT fta
    (lambda (env->a)
      ...)))

(define (bindFT fta f)
  (unitT
    (lambda (env)
      (bindT fta          ; ***
        (lambda (env->a)
          (env->a env))))))
```

The first attempt falls flat. The second appears more promising, but is badly typed at the starred line. Similar arguments indicate that we can define

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

but not

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

These observations naturally lead to a classification of monad transformers as *top*, *bottom*, or *around*, as shown in table 2.4. Continuations and resumptions do not fit into this classification, but table 2.5 classifies the other transformers we have discussed. Lifting appears twice since there are two different lifting transformers.

Note that S and T in bottom and top transformers have monadic structure, since we can apply the transformer to an identity monad. In around transformers, $S \circ T$ has monadic structure. Although we have only one good example each of top and around transformers, the classification will prove useful later, in section 4.2.

The nondeterminism monad transformer should actually use sets, rather than lists. In fact, "monads" created by list transformer don't obey the associative law

Name	Type $F(T)(A) =$	Classification
Nondeterminism	$T(List A)$	Bottom
Exceptions	$T(A+X)$	Bottom
Monoids	$T(A \times M)$	Bottom
Lifting1	$T(1 \rightarrow A)$	Bottom
Lifting2	$1 \rightarrow T(A)$	Top
Environments	$Env \rightarrow T(A)$	Top
Stores	$Sto \rightarrow T(A \times Sto)$	Around

Table 2.5: Classification examples

unless the original monad is commutative (see [JD93]); however, in our interpreters, we represent sets as lists and ask the reader to collapse distinctions of order and multiplicity.

2.6.4 Composition of monad transformers

We can use transformers to build quite complex types. For example, consider a language with environments, stores, continuations, nondeterminism, and exceptions. We compose the transformers as

```
(compose
 environments
 stores
 continuations
 nondeterminism
 exceptions))
```

obtaining

$$
\begin{aligned}
F(T)(A) = Env \rightarrow \\
\quad Sto \rightarrow \\
\quad\quad (A \times Sto \rightarrow List(Ans + Err)) \rightarrow \\
\quad\quad\quad List(Ans + Err))
\end{aligned}
$$

Chapter 3

Lifting

This chapter shows how to use monad transformers to build interpreters via the important notion of *lifting*. The first section presents a general definition of lifting, and the second describes several methods of building interpreters.

3.1 Lifting

In this section, we formalize lifting and show how monads can lift operations with simple signatures.

3.1.1 Formal lifting

We define a language of types $t(S)$ parametrized by a functor S:

$$\begin{array}{lll} t(S) & = A & \text{(constants)} \\ & \mid V & \text{(variables)} \\ & \mid t \times t & \text{(pairs)} \\ & \mid t \to t & \text{(functions)} \\ & \mid S(t) & \text{(functors)} \end{array}$$

The form of this definition is from [LHJ95]; a slightly more complex version appears in [Mog89a]. It is also nearly identical to the fundamental definition in Reynolds' work on parametricity [Wad89].

For any particular S, $t(S)$ is still polymorphic, since we allow type variables. Given two functors S, S' and a natural transformation `sigma` : $S \to S'$, a *lifting* of type t through `sigma` is a map

$$\texttt{L} : t(S) \to t(S')$$

such that

```
(L a)            = a                     (constants)
(L v)            = v                     (variables)
(L (pair x y)) = (pair (L x) (L y))    (pairs)
((L f) (L x))   = (L (f x))             (functions)
(L s)            = (sigma (mapS L s))    (functors)
```

Notice that

```
(sigma (mapS L s)) = (mapS' L (sigma s))
```

since `sigma` is natural. This definition specifies lifing as a relation, not a function. In fact, given t and `sigma` there may be many liftings, one, or none. For example, suppose we fix the following signature, functor, and function:

$$\begin{array}{lcl} t(S) & = & S(A) \to A \\ S(A) & = & A \\ \texttt{(mapS f a)} & = & \text{(f a)} \\ \texttt{id} & : & t(S) \end{array}$$

Then if we specify a functor S' and a natural transformation `sigma` from S to S', we can enumerate the liftings of `id` along `sigma`. First, we try

```
;; S'(A) = A × A

(define ((mapS' f) p)
  (pair (f (left p)) (f (right p))))

(define (sigma a) (pair a a))
```

Then there are two liftings of `id`: `left` and `right`. The constraint on liftings `f` is

```
(f (pair a a)) = a
```

which both `left` and `right` satisfy. Another choice of S' and `sigma` is

```
;; S'(A) = List(A)

(define (mapS' f l) (map f l))

(define (sigma a) (list a))
```

Liftings in this case have type $List(A) \to A$. But there are no functions of this type, since we wouldn't know where to send the empty list. Vacuously, all of them meet the lifting constraint

```
(f (list a)) = a
```

Given a monad T, we lift functions from Id to T through `unit`. Both cases above are examples of this form. Given a monad transformer F, we lift functions from T to $F(T)$ through `unitF`. For example, consider `append` defined on the list monad:

$T(A) = List(A)$
`append` : $T(A) \times T(A) \to T(A)$

If we lift `append` through the environment monad transformer

```
;; F(T)(A) = Env → T(A)

(define (unitF ta)
  (lambda (env) ta))
```

we obtain

```
;; lifted-append : F(T)(A) × F(T)(A) → F(T)(A)

(define (lifted-append fta1 fta2)
  (lambda (env)
    (append (fta1 env) (fta2 env))))
```

There are additional naturality conditions on these liftings that we do not discuss (see [Mog89a]).

3.1.2 Monads and lifting

By now, it should be obvious that monads can define liftings. For example, let's lift a binary operator `f` $: A \times B \to C$ up to `F` along `unit` for a monad T. We write

```
(define (F ta tb)
  (bind ta
    (lambda (a)
      (bind tb
        (lambda (b)
          (unit (f a b)))))))
```

Using the monad laws, we can show that `F` is a lifting of `f` according to the previous section. We need

```
(F (unit a) (unit b)) = (unit (f a b))
```

Using substitution and the first monad law twice, we have

```
(F (unit a) (unit b))

  = (bind (unit a)
      (lambda (a)
        (bind (unit b)
          (lambda (b)
            (unit (f a b))))))

  = (bind (unit b)
      (lambda (b)
        (unit (f a b))))

  = (unit (f a b))
```

It would be interesting to determine the set of signatures liftable using monads. Some useful operators, such as `%callcc`, are not apparently liftable.

3.2 Pragmatics

Consider a composition of monad transformers applied to a monad:

$$(F_1 \circ \cdots \circ F_n)(T)$$

From the bottom up, we form a sequence of monads $F_n(T), F_{n-1}(F_n(T)), \ldots$. From the top down, we form a sequence of monad transformers $F_1, F_1 \circ F_2, \ldots$. Naturally, we can combine these approaches by splitting the sequence of transformers at some point, forming the left half top-down and the right half bottom-up, then combining the two halves by application.

3.2.1 Bottom-up

In the bottom-up approach, we begin with a basic monad, usually the identity, and apply monad transformers to it. As we apply a transformer we

- Lift existing operators through the the transformer, and
- Add new operators to the resulting transformed monad.

Thus, we obtain a new monad with not only the existing operators, but several new ones as well. In this case, we lift along the `unitF` operator of the monad transformer. Although we are technically working with monads on the category of monads, Scheme's implicit polymorphism allows us to use ordinary monads for lifting. For example, we can lift an operator from $T(A)$ to $Env \to T(A)$ using the ordinary environment monad. Even easier, an operator on $T(A)$ is *already* an operator on $T(List(A))$, with no lifting needed. These short cuts would not be possible in the polymorphic lambda calculus, where we would have to keep track of types explicitly.

It appears that we always lift operators through monad transformers, rather than monads, but this is not quite the case. To define operators on monad transformers, we must often lift values and functions through the monads they transform. For example, in order to define call-by-value variable reference on $Env \to T(A)$, we must lift values using `unitT`:

```
;; %var :  Name → Env → T(A)
;; Env    = Name → A

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

Similarly, to define `%amb` on $T(List(A))$, we lift `append` through T. Modulo these considerations, building a system based on lifting is straightforward.

3.2.2 Top-down

The top-down approach yields a system of extensible interpreters generalizing Wadler [Wad92]. In a sequence $F_1, F_1 \circ F_2, \ldots$, we view F_1 as an interpreter parametrized by a monad; for example, Wadler's basic interpreter is $F(T)(A) = Env \to T(A)$. However, instead of supplying a monad, we supply a monad *transformer* to obtain another interpreter. In other words, given a parametrized interpreter I and a monad transformer F, we form another parametrized interpreter $I \circ F$. Of course, we must also take care to lift operators properly. Steele searched for this approach in [Ste94] but missed passing to higher-order types.

Chapter 4

Stratification

In this chapter, we formally define stratified monads and their transformers and describe how SL actually works.

4.1 Stratified monads

Using compatible monads (section 2.5), we can formalize the notion of "levels related by monads" discussed in chapters 1 and 3.

A *level* is simply a type constructor (an endofunction on the category). A monad T *relates* L_1 to L_2 if $L_2 = T \circ L_1$. We can form categories whose objects are *levels* and whose arrows are monads by defining composition any way we like, subject to the category laws *and the compatibility of composites.*

For example, let's form a category of levels and monads for the semantics $Den(A) = Env \to Sto \to A \times Sto$. We have levels

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

These are related by monads

$$\begin{aligned}
T_{34}(A) &= Env \rightarrow A \\
T_{23}(A) &= Sto \rightarrow A \\
T_{24}(A) &= Env \rightarrow Sto \rightarrow A \\
T_{13}(A) &= Sto \rightarrow A \times Sto \\
T_{14}(A) &= Env \rightarrow Sto \rightarrow A \times Sto
\end{aligned}$$

where T_{34} and T_{23} are environment monads, T_{13} is the store monad, T_{24} is a "double environment" monad, and T_{14} is an "environment / store" monad. We also have an identity monad from each level to itself (not shown). Notice that there is no monad from L_1 to L_2. Composition is given by

$$\begin{aligned}
T_{34} \circ T_{13} &= T_{14} \\
T_{34} \circ T_{23} &= T_{24}
\end{aligned}$$

both of which satisfy the compatibility laws.

A *stratified monad* is a category of levels and monads satisfying several additional properties that do not follow solely from the category structure. We require

- All diagrams commute.
- There are distinguished levels *Bot* and *Top*.
- *Bot* is the identity type constructor.
- *Bot* and *Top* are related by a monad T. We also call the entire stratified monad T.
- *Bot* must be minimal, and *Top* must be maximal, meaning that no monad can relate any L to *Bot* or *Top* to any L.

The requirement that all diagrams commute means there is at most monad relating any two levels (there may be none), since two parallel arrows form a diagram. Also, by forgetting the structure of the arrows, we obtain a partial order in which $L_1 \sqsubseteq L_2$ if and only if there is a monad relating L_1 to L_2.

In necessary, we can drop the "all diagrams commute" condition; however, most semantics obey it, since there aren't usually multiple ways to relate levels. Indeed, we could call a language "non-uniform" if it requires multiple monads between levels to define its constructs. This restriction makes the implementation of SL easier since we don't have to specify *which* monad we want.

We do not require *Bot* and *Top* to be initial and terminal (stronger conditions that minimal and maximal), since some semantics include levels unrelated to *Bot* and *Top*; for instance, in the previous example, *Bot* fails to be initial since there is no monad T_{12}. In many semantics, however, they are actually initial and terminal.

A level L is a monad (rather than just an endofunction) if there is a monad relating *Bot* to it; hence, *Bot* is initial if and only if all levels are monads. Thus, in the previous example, L_2 isn't a monad. I haven't found any real examples where *Top* fails to be terminal, but we weaken this condition for symmetry.

4.2 Stratified monad transformers

According to the "categorical imperative", we should now form a category of stratified monads; however, in this application, we do not need this structure.

Thus, a *stratified monad transformer* is an endofunction on the *set* of stratified monads. We can verify these directly for each transformer that it respects the stratified monad structure.

In practice, we build stratified monad transformers by "lifting" ordinary monad transformers to act on stratified monads. Here the lifting is along the map that extracts the monad T relating *Bot* to *Top* from the stratified monad T (recall our naming convention). The action of a stratified monad transformer is not hard to guess from its action on levels. Let's do an example. By applying stratified monad transformers, we build the semantics

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

We begin with an identity stratified monad, which has a single level and a single monad:

$$L_1(A) = A$$

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

We apply the nondeterminism stratified monad transformer $F(T)(A) = T(List(A))$. Omitting the identity monads at each level, we obtain

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

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

Remember that each T_{ij} is an entire *monad*, not just a type constructor. Now we apply the store stratified monad transformer $F(T)(A) = Sto \to T(A \times Sto)$, obtaining

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

$$T_{34}(A) = Sto \to A$$

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

Finally, we apply the environment stratified monad transformer $F(T)(A) = Env \to T(A)$:

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

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

In the next few sections, we elaborate the action of the stratified monad transformers built from the classes of monad transformers discussed in section 2.6.3.

4.2.1 Top transformers

Top transformers have the form $F(T) = S \circ T$. F acts on the levels of a stratified monad by adding a new top level $S \circ Top$. F acts on the monads by applying F to all monads relating to *Top*. We add the new monads without deleting the originals.

We can verify this action in the example above for the environment transformer, which has $S(A) = Env \to A$. We formed each of the monads involving environments by transforming a monad relating some level to *Top*. Conversely, all such monads were transformed.

4.2.2 Bottom transformers

Bottom transformers have the form $F(T) = T \circ U$. F acts on levels by composing each level with U and adding a new identity at the bottom. F acts on monads by applying F to all monads relating to *Bot*. As before, we add the new monads without deleting the originals.

We can verify this action in the example above for the nondeterminism transformer, which has $U(A) = List(A)$. We formed each of the monads involving lists by transforming a monad relating some level to *Bot*. Conversely, all such monads were transformed.

4.2.3 Around transformers

Around transformers are somewhat more complex than bottom and top transformers. In order to construct all the possible monads relating different levels, we require *three* ordinary monad transformers, not just one. If the around transformer is

$$F_A(T) = S \circ T \circ U$$

we also require

$$\begin{aligned} F_B(T) &= S \circ T \\ F_T(T) &= T \circ U \end{aligned}$$

The action on levels is to add a new top level $S \circ Top$, compose each level with U below, and add a new identity as *Bot*. The action on monads is to transform the monad

T using F_A, to transform all monads relating to Bot using F_B, and to transform all monads relating to Top using F_T. Note that we also transform T using F_B and F_T. As before, we add the new monads to the result, leaving the old ones in place.

We obtain the store monad transformer by taking

$$\begin{aligned} Ar(T)A &= Sto \to T(A \times Sto) \\ Top(T)A &= Sto \to T(A) \end{aligned}$$

We do not use a Bot transformer, since it would have to be

$$Bot(T)A = T(A \times Sto)$$

and we have seen that this choice does not make sense.

4.2.4 Continuation transformers

The continuation transformer is

$$F(T)(A) = (A \to T(Ans)) \to T(Ans)$$

where Ans is a fixed domain of answers. F acts on levels as $F(L)(A) = L(Ans)$. That is, once T is applied to answers, it ignores whatever else we apply it to. We add a single new level $L(A) = (A \to T(Ans)) \to T(Ans)$, where T is the old top level.

F acts on monads as follows. It transforms T using the continuation transformer, yielding a monad relating Bot to the new Top. It also transforms monads M relating to Top via a special "answer transformer" F_{Ans}, yielding monads relating M to the new Top. These monads allow us to access the levels of the answer type $T(Ans)$.

There are two choices for F_{Ans}, shown in figures 4.1 and 4.2. If we define `amb` in the semantics

$$Den(A) = (A \to List(Ans)) \to List(Ans)$$

using each of these choices, we obtain

```
(define ((unit1 a) k)
  (bindT (unitM a) k))

(define ((bind1 c f) k)
  (bindM (c unitT)
    (lambda (a)
      ((f a) k))))
```

Figure 4.1: First answer transformer

```
(define ((unit2 a) k)
  (unitM a))

(define ((bind2 c f) k)
  (bindM (c k)
    (lambda (a)
      ((f a) k))))
```

Figure 4.2: Second answer transformer

```
(define ((amb1 d1 d2) k)
  (reduce append ()
    (map k (append (d1 list) (d2 list)))))

(define ((amb2 d1 d2) k)
  (append (d1 k) (d2 k)))
```

as discussed in section 1.5.2. Both definitions are reasonable.

4.3 Computation ADTs

A computation ADT *is* a stratified monad, except that we associate a set of names with each level. Since there is at most a single monad relating any pair of levels, monads are uniquely identified by the levels they relate.

We use *sets* of names because a single level can play multiple roles in a semantics. For example, in

$$Den(A) = Env \to List(A)$$

the level $List(A)$ is called both *Lists* and *Env-Results*.

Each stratified monad transformer adds several new names. For example, the environment transformer adds the following pairs of names and levels:

$$\begin{array}{lll} \mathit{Envs} & \Rightarrow & Env \to T(A) \\ \mathit{Env\text{-}Results} & \Rightarrow & T(A) \\ \mathit{Env\text{-}Values} & \Rightarrow & A \end{array}$$

Table 4.1 shows the names added by each transformer. In some cases, we reuse the same stratified monad transformer, changing only the names that it adds. For example, we build both the Stores and Batch I/O modules using the store transformer. We can build a semantics using multiple instances of the same transformer (environments, for example) by assigning different names to the instances.

Table 4.2 shows the names and levels associated with the following language definition:

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

Using the information contained in table 4.1, we can define language constructs over stratified monads. Construct definitions assume the existence of various levels and monads. For example, `%amb` assumes the existence of the level *Lists* and a monad relating *Lists* to *Top*. The definition of `%amb` is

Module	Names	Levels
Environments	*Envs* *Env-Results* *Env-Values*	$Env \to T(A)$ $T(A)$ A
Stores	*Stores* *Store-Results* *Store-Pairs* *Store-Values*	$Sto \to T(A \times Sto)$ $T(A \times Sto)$ $A \times Sto$ A
Batch I/O	*IO* *IO-Results* *IO-Pairs* *IO-Values*	$IO \to T(A \times IO)$ $T(A \times IO)$ $A \times IO$ A
Lifting 1	*Lifts*	$1 \to T(A)$
Lifting 2	*Lifts*	$1 \to A$
Errors	*Errors*	$A + Err$
Output	*Output*	$A \times Out$
Nondeterminism	*Lists*	$List(A)$
Continuations	*Conts* *Cont-Values* *Cont-Answers* *Answers*	$(A \to T(Ans)) \to T(Ans)$ A $T(Ans)$ Ans
Resumptions	*Res-Top* *Res-Bottom*	$fix(X)\ T(A+X)$ $A + fix(X)\ T(A+X)$

Table 4.1: Names associated with each transformer

```
(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)))))))))
```

The stratified monad operators are not quite sufficient to form a truly abstract data type of computations. We need to know precisely the additional information contained in table 4.1. For example, *Lists* are lists of some type, *Envs* are functions from environments to *Env-Results*, etcetera. Thus, if we desire to build true abstract data

Names	Level $L(A) =$
Envs, Top	$Env \rightarrow$ $\quad Sto \rightarrow$ $\quad\quad let\ A1 = List(Ans \times Sto) + Err\ in$ $\quad\quad\quad (A \times Sto \rightarrow A1) \rightarrow A1$
Env-Results, Stores	$Sto \rightarrow$ $\quad let\ A1 = List(Ans \times Sto) + Err\ in$ $\quad\quad (A \times Sto \rightarrow A1) \rightarrow A1$
Store-Results, Conts	$let\ A1 = List(Ans \times Sto) + Err\ in$ $\quad (A \times Sto \rightarrow A1) \rightarrow A1$
Cont-Answers, Errors	$List(Ans \times Sto) + Err$
Lists	$List(Ans \times Sto)$
Answers	$Ans \times Sto$
Store-Pairs, Cont-Values	$A \times Sto$
Env-Values, Store-Values, Bottom	A

Table 4.2: Levels and names for a complex language

types, we have to represent all this information as part of the interface.

The types *Sto* and *Env* are parameters to the semantics that can be specified quite late. That is, after forming a computation ADT, we can decide what type variables should denote. Of course, the language constructs must implement this choice, and not all choices make sense.

4.4 Language ADTs

In general, operators that act primarily at a single level, such as `%amb` (figure 1.21) and `%let` (figure 1.19), 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 4.4 lists the available modules and the values and language constructs they define. We omit leading percent signs from the names.

SL includes four types of procedures. Table 4.3 shows the levels of their domains and codomains. We define all four types of procedures over the same `environments`

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 4.3: Procedure types

monad transformer by writing different versions of `%lambda` and `%call`. Clearly, we can define other types of procedures as well.

Module	Values	Constructs
Amb		`amb`
Batch I/O		`read, write, end-of-input?`
Begin	unit	`begin, unit`
Booleans	booleans	`true, false, not, if, boolean?`
CallCC		`callcc`
Dynamic procedures	procedures	`lambda, call, procedure?`
Environments		`var, let`
Error exceptions		`error`
Error values	errors	`error`
Exp environments		`evar, elet`
Fix		`fix, rec, letrec`
Numbers	numbers	`num, +, -, *, /`
Numeric predicates		`=?, zero?, number?`
Output		`write`
Procedures	procedures	`lambda, call, procedure?`
Products	pairs	`pair, left, right, pair?`
Resumptions		`pause, seq, par`
Shift		`shift, reset`
Stores		`fetch, store`
Sums	sums	`case, in-left, in-right, sum?`
While		`while`

Table 4.4: Modules and language constructs

Chapter 5

Conclusion

This chapter compares lifting and stratification, describes the limitations of these ideas, relates this work to previous research, and suggests directions for further exploration.

5.1 Lifting versus stratification

The main problem with lifting is that it ties language constructs too tightly to the monad transformers on which they are defined. For example, if we define variable reference on $Env \to T(A)$, there is no easy way to raise an unbound variable error. We could define an ad-hoc set of lower-level operators to circumvent this problem (see [LHJ95]); however, stratification shows how to define a *principled* set of lower-level operators.

Phrased differently, constructs cannot interact with multiple semantic levels. For example, `+` interacts with values and errors, raising an error for non-numeric arguments, and `callcc` interacts with continuations, environments, and values. Table 5.1 shows the levels referenced by the more complex language constructs.

Lifting also interleaves the creation of new operators with the lifting of old ones. Not only are these actions conceptually separate, but when we use a transformer, we might not want all of the operators that come with it. Again, stratification provides

Construct	Modules referenced
`%end-of-input?`	IO, booleans
`%read, %write`	IO, numbers
`%call/cc, %shift`	continuations, environments, procedures
`%lambda, %call`	environments, procedures
`%var`	environments, errors
`%fix, %rec`	lifting, environments, procedures
`%letrec`	lifting, environments
`%/`	numbers, errors
`%case`	sums, environments, procedures

Table 5.1: Non-local language constructs

a solution by separating the language constructs from the base semantics. Still, the lifting approach remains viable for building abstract data types whose operators are more local.

5.2 Limitations

SL has several *intended* limitations:

- Since it was designed to build denotational models, SL does not address issues of type and syntax, which occupy large parts of most language specifications.
- We could extend SL to perform almost any compositional program analysis; however, it provides no help in defining "extra semantic" mechanisms such as unification or constraint set solution. That is, although an SL-constructed analysis could derive a set of type constraints from a typed program, it would not solve them.
- Although SL can build the basic semantics for a real-world language like C, by the time we add all the specific details of C's language constructs, we would hardly claim to have built C from reusable parts.

SL also has several unintended limitations, but their descriptions are rather technical. First, store transformers do not compose modularly with each other. For example, suppose we define a language that includes both stores and batch I/O (two different parametrizations of the store transformer). By composition, we obtain

$$Sto \to (IO \to (Val \times Sto) \times IO)$$

Consider the following construct (`%read` would do as well):

```
(define (((%end-of-input?) sto) io)
  (pair (pair (in 'booleans (null? io)) sto) io))
```

To define it directly (see appendix B.2, figure B.18), we need a monad relating Val to $Val \times Sto$, an impossibility.

There are two unsatisfactory solutions to this problem. The first is to compose the transformers in the opposite order and hope that the store operators don't require a monad from Val to $Val \times IO$ (indeed, they don't).

The second solution is, instead of lifting Val to $Val \times Sto$, to return *Val in place of* $Val \times Sto$, as though we could put any type at all there. Then we adjust the result appropriately at the end (see figure 5.1). Hudak et al. adopt this solution ubiquitously [LHJ95], as described in section 5.3. The real problem is that monads are not flexible enough to handle store transformers. We require a more sophisticated lifting operator, although its form is not yet clear.

The second unintended problem is similar to the first. When defining `%callcc` in the usual Scheme semantics, we need to uncurry a function from

$$f : Val \to Sto \to Val \times Sto$$

to

$$f' : Val \times Sto \to Val \times Sto$$

Surprisingly enough, the stratified monad operators on stores cannot perform this transformation. To be clear, we should say that our goal is not explicitly to uncurry f but to change its type from

```
(define %end-of-input?
  (let ((unitT (get-unit 'io-pairs 'io-results))
        (unitS (get-unit 'io 'top))
        (unitB (get-value-unit 'booleans 'top))
        (bindV (get-bind 'io-values 'top)))
    (lambda ()
      (bindV
       (unitS
        (lambda (io)
          (unitT (pair (null? (batch-input io))
                       io))))
       (lambda (b)
         (unitB b))))))
```

Figure 5.1: Odd definition of `%end-of-input?`

$$f : \textit{Store-Values} \rightarrow \textit{Stores}$$

to

$$f' : \textit{Store-Pairs} \rightarrow \textit{Store-Results}$$

(see section 4.3).

To circumvent this problem, we add a left inverse to `unit` to each monad, called `iunit`. Then the required lifting can be defined (see the `tilt` function in the definition of `%callcc`, appendix B.2). Since we require `unit` to be injective, a left inverse always exists; however, it is less apparent that the argument to `iunit` in the `%callcc` is always in its domain. In this case, if we could prove using the monad laws that its argument is always `unit` of something, we could probably eliminate `iunit`.

`iunit` is clearly a hack; once again, the monad operators are simply not powerful enough to perform all manipulations of the store transformer that occur in standard construct definitions.

5.3 Related work

Spivey [Spi90] used monads to abstract over exception handling but did not connect these ideas with extensibility.

Moggi [Mog89b, Mog91] split an "applied" lambda calculus into a core (variables and environments) and an extension (other features), expressed as a monad. He presented many extensions and derived a "computational lambda calculus" for reasoning about programs.

Moggi also showed that monad transformers can build complex monads from parts [Mog89a]. This crucial facility was hitherto missing; however, his presentation is difficult, and few researchers realized that he had made substantial progress.

Rewriting Moggi's methods [Esp94], I saw that they did not easily handle constructs involving multiple semantic levels, such as `%call/cc` or even `%+` (because it raises errors on non-numbers). Stratified monads solve this problem, increasing modularity by inserting an abstraction barrier between computation ADT and language ADT.

Wadler [Wad92] popularized Moggi's ideas by presenting monadic interpreters written in Haskell. The interpreters' limitation to extension by a single monad motivated this thesis. Also, Wadler and King showed how to combine continuations and lists with other monads [KW92]. Despite Moggi's earlier formulation of monad transformers, they discussed "combining M and L" rather than "constructing ML from M". SL treats monad constructors in general and exhibits a complete system for building interpreters from multiple modules, not just two.

Steele [Ste94] showed 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 *bottom* monad transformers. That is, they can realize

$$
\begin{aligned}
F(T)(A) &= T(\mathit{List}\,A) \\
F(T)(A) &= T(A+X) \\
F(T)(A) &= T(A\times M)
\end{aligned}
$$

but not

$$
\begin{aligned}
F(T)(A) &= \mathit{Env} \to T(A) \\
F(T)(A) &= \mathit{Sto} \to T(A\times \mathit{Sto})
\end{aligned}
$$

Steele's claim that pseudomonads improve on monad transformers by providing a fixed composition operator fails to hold since they are not equally powerful; however, Steele's complete implementation of a modular semantics was inspiring, and the stratified approach described here is based on his *tower* of pseudomonads.

Jones and Duponcheel [JD93] addressed the problem of composing monads. They showed rigorously that monads do not compose, but that if one of several auxiliary maps is defined relating the structures of two monads, they *can* be composed. They found that some monads compose naturally to the left and some to the right (corresponding to our *bottom* and *top*). Although composition is strictly weaker than transformation, they came as close as one could to discovering monad transformers, and their work provides useful information about the structure of semantic models; however, they did not attempt to build interpreters.

Mosses [Mos92] showed how an *abstract semantic algebra*, which we call a computation ADT, could modularize a semantics. By choosing algebra operators low-level enough to be flexible, yet high-level enough to hide irrelevant details, we can make a semantics much easier to understand.

Mosses gave a *single* ADT with operators for environments, stores, and continuations. Using this ADT, we can define other semantic features, but only in an unnatural and non-modular way. With SL, we can build ADTs custom tailored to the languages we define. In essence, SL is the final step in Mosses's program, the ability to combine algebras.

Filinski [Fil94] showed how to compute over an arbitrary monad in a language with composable continuations. His construction is a direct use of the continuation monad transformer

$$F(T)(A) = (A \to T(Ans)) \to T(Ans)$$

Most of his paper presents a rather technical proof that computing over $F(T)$ yields the same results as computing over T, as long as our constructs don't use F. It is possible that his proof could be simplified using the general properties of monad transformers.

Although composable continuations yield extensibility, we could alternatively add reflection operators to a language directly. Thus, we would need nothing but lambda calculus and primitives to be "monadically complete"; however, this point is irrelevant, since we probably want to stores and continuations to be primitive for efficiency. Extensibility through continuations costs nothing (beyond the continuations) if we don't use it. This point is not obvious; see Filinski's paper.

Cartwright and Felleisen [CF94] postulate that a computation is either a value or an *effect* and include a resource administrator to manage effects. Their semantics already includes environments, stores, and continuations, the latter two of which are hidden using a monad (`bind` is called `handle`).

Their semantics employs object-oriented techniques such as extensible products (for stores), extensible sums (for values), and "self" arguments (for interpreter composition). These techniques recall my earlier thesis proposals

[Esp93a, Esp93b], although there was no direct connection.

In general, it not surprising that Felleisen and Cartwright's system can be extended with stores and continuations, since they are already included, in the guise of a resource administrator. The intuitive value of this abstraction remains to be seen. We *can* extend the store; however, stores are already extensible in most languages, since we can create new locations on demand.

Liang, Hudak, and Jones [LHJ95] recently published a paper improving on my earlier work [Esp94] and on Moggi's work as well.

Their first improvement was to lift `callcc` using operators designed specifically for it. Both Moggi and I lifted all operators of a given type in a similar way (parametrically, rather than ad-hoc). Ad-hoc liftings are non-modular, since they require a new lifting method for each operator and monad transformer. Using stratified monads, we can define a single reusable `callcc` (see figure B.20). Although its definition is complex, it is independent of new transformers.

Their second improvement was to express monad transformers in a typed language (Gofer, an extension of Haskell). This work shows the power of Jones's constructor classes, since monad transformers cannot be expressed in pure Haskell.

The main problem with Liang, Hudak, and Jones's approach is the treatment of non-local language constructs (those involving multiple levels). They shift all interaction to the top or bottom of the single monad that defines the base semantics. That is, in a semantics

$$Den = Env \rightarrow List(Val)$$

procedures have types

$$\begin{aligned} CBVProc &= Val \rightarrow Den \\ CBNProc &= Den \rightarrow Den \end{aligned}$$

Although these appear reasonable at first glance, procedure arguments and results unnecessarily include environments. That is, without access to intermediate levels in the semantics, their types lose precision.

Similarly, to define variable reference, they form $Den(Env)$, denotations that return environments. In effect, this approach desugars a reference to x into

```
(env-lookup 'x (the-environment))
```

Explaining variable reference using first-class environments is inappropriate, since environments would not normally appear in a language's value domain.

Of course, after simplification, the actual constructs reduce to the definitions we would expect, so perhaps the ends justify the means. In general, since their approach is based on lifting, it is much weaker than stratification, as discussed in section 5.1.

5.4 Future work

Implementation As mentioned in section A.1, we could rewrite SL in Quest [Car89] to verify proper use of higher-order types.

By forming domains from expressions instead of functions, we could build abstract interpretations, translations (such as CPS), and simple compilers.

We could apply the methods developed here to other semantically complex domains, such as communication protocols. If we don't need functional abstraction, simpler lifting operators than monads should suffice (see section 2.2.3).

Extensibility We could generalize call-by-value and call-by-name to abstract over other semantic levels. For instance, abstracting over the top level of denotations yields most of the functionality due to macros. Also, we could define a language capable of abstracting over *each* semantic level.

We could define a uniform family of reification and reflection operators (see [Fil94]), one for each semantic level. These would generalize constructs such as `the-environment`, `call/cc`, and exception handling.

Models Are stratified monads derivable from more categorical considerations? Since the monad laws follow nicely from the Kleisli formulation, can we define a Kleisli category for several monads at once? This construction would presumably be a product of the individual Kleisli categories, indexed by monad. The key idea is that compositions associate.

Having presented two approaches to building interpreters, lifting and stratification, can we show their equivalence, at least in the cases that lifting can handle?

Logics Can we develop modular calculi for reasoning about the languages we construct? Moggi's computational lambda calculus [Mog91] captures precisely the inferences valid for lambda calculus over an arbitrary monad. Moggi's syntactic approach [MC93] is relevant but does not seem to address the problem directly.

Calculi can probably be derived via both lifting and stratification. Given a set of laws on T, can we lift them to $F(T)$? Or, can we use the stratified monad laws to derive laws for a completed semantics? Abramsky's work [Abr91] on deriving program logics from domain equations is applicable to domains built using type constructors; however, his methods seem to derive calculi that are still very low-level.

5.5 Conclusion

Four simple imperatives summarize this thesis:

- Think with types, both abstract and concrete.
- Compute with denotations, not expressions.

- Split a complex interpreter into a computation ADT and a language ADT.
- Structure the computation ADT using monads and monad transformers.

The first two of these are most important, since types let us think in a simple yet structured way, and denotations let us implement interpreters easily and directly. These two ideas make an otherwise difficult field accessible to anyone who understands functional programming.

Bibliography

[Abr91] Samson Abramsky. Domain theory in logical form. *Annals of Pure and Applied Logic*, 51:1–77, 1991.

[ASS85] Harold Abelson, Gerald J. Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, MA, 1985.

[Bec69] Jon Beck. Distributive laws. In *Seminar on Triples and Categorical Homology Theory*, volume 80 of *Lecture Notes in Mathematics*, pages 119–140. Springer Verlag, 1969.

[BW85] Michael Barr and Charles Wells. *Toposes, Triples, and Theories.* Springer Verlag, New York, 1985.

[BW90] Michael Barr and Charles Wells. *Category Theory for Computing Science.* Prentice-Hall, 1990.

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

[CF94] Robert Cartwright and Matthias Felleisen. Extensible denotational language specifications. In *Theoretical Aspects of Computer Software*, Sendai, Japan, April 1994.

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

[Esp93a] David Espinosa. Language extensibility via first-class interpreters and constructive modules. See http://www.cs.columbia.edu, April 1993.

[Esp93b] David Espinosa. Language features for extensible programs. See http://www.cs.columbia.edu, October 1993.

[Esp94] David Espinosa. Semantic Lego. See http://www.cs.columbia.edu, January 1994.

[Fil89] Andrzej Filinski. Declarative continuations and categorical duality. Master's thesis, University of Copenhagen, August 1989. See http://www.cs.cmu.edu:8001.

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

[GM94] Carl Gunter and John Mitchell, editors. *Theoretical Aspects of Object-Oriented Programming.* MIT Press, Cambridge, MA, 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.

[Gun92] Carl Gunter. *Semantics of Programming Languages.* MIT Press, Cambridge, MA, 1992.

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

[KBdR91] Gregor Kiczales, Daniel G. Bobrow, and Jim des Rivieres. *The Art of the Metaobject Protocol.* MIT Press, Cambridge, MA, 1991.

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

[Lam88] John Lamping. A unified system of parametrization for programming languages. In *Conference Record of the 1988 ACM Symposium on Lisp and Functional Programming*, pages 316–326, Snowbird, Utah, July 1988.

[LHJ95] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Proceedings of the 22nd Annual ACM Symposium on Principles of Programming Languages*, San Francisco, CA, January 1995.

[Mac71] Saunders MacLane. *Category theory for the Working Mathematician.* Springer Verlag, New York, 1971.

[MB88] Saunders MacLane and Garrett Birkhoff. *Algebra.* Chelsea, New York, 3rd edition, 1988.

[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.

[Mes89] José Meseguer. General logics. In H. D. Ebbinghaus, editor, *Logic Colloquium 87*, pages 275–329. North Holland, 1989.

[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.

[Mog91] 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.

[Pie91] Benjamin C. Pierce. *Basic Category Theory for Computer Scientists*. MIT Press, Cambridge, MA, 1991.

[RB90] David Rydeheard and Rod Burstall. *Computational Category Theory*. Prentice-Hall, New York, 1990.

[Sch86] David A. Schmidt. *Denotational Semantics*. Allyn and Bacon, New York, 1986.

[Spi89] Michael Spivey. A categorical approach to the theory of lists. In *Mathematics of Program Construction*, volume 375 of *Lecture Notes in Computer Science*, pages 399–408. Springer Verlag, 1989.

[Spi90] Michael Spivey. A functional theory of exceptions. *Science of Computer Programming*, 14(1):25–42, June 1990.

[Spi93] Michael Spivey. Category theory and functional programming. Technical Report PRG TR 7-93, Oxford University, June 1993.

[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.

[Wad89] Philip Wadler. Theorems for free. In *Functional Programming Languages and Computer Architecture*, London, England, September 1989.

[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.

Appendix A

Miscellanea

This appendix discusses issues tangent to the thesis proper at various points.

A.1 Why Scheme?

The real reason for writing this thesis in Scheme[1] is that I'm used to it. Although Scheme has many problems (notably the lack of modules and abstract data types), it is still pretty fun to program in. But, for this thesis, Scheme has several disadvantages:

- It fails to express the typed structure of the mathematics.
- Implicit polymorphism does not distinguish between a polymorphic function and its instantiations at various types; hence, a whole level of structure is lost.
- Since types are not mechanically verified, our understanding of them could be incorrect.

On the other hand, Scheme's advantage is that it does not limit what we can express. For example, in the usual Hindley-Milner type system, polymorphic values

[1]When I failed to explain an example sufficiently in an earlier thesis proposal, one reader exclaimed, "You can't write your thesis in Scheme!"

are not first-class, since we cannot instantiate them at different types. F_2 has first-class polymorphism but cannot treat type constructors as types, so that monads cannot be first-class.

We can probably type SL in F_3, which includes type constructors; however, it is not clear how to type the levels of a stratified monad. For instance, we would need types *Envs* and *Env-Results* such that

$$\mathit{Envs} = \mathit{Env} \rightarrow \mathit{Env\text{-}Results}$$

(see section 4.3). It would be a good exercise to rewrite SL in Cardelli's Quest language, which includes higher-order polymorphism as well as several other advanced typing ideas [Car89]. Also, Liang, Hudak, and Jones [LHJ95] have implemented monad transformers in Gofer, a version of Haskell with an enhanced type system (essentially Hindley-Milner extended to higher types).

In general, the programming languages community is realizing that current type systems are inadequate; however, we should go further and question the entire basis for typed languages. In general, types are a form of specification. When we say that an expression has a type, we really mean that its evaluation meets a specification. This point of view leads naturally to more expressive types. Although inference and verification are intractable for these systems, we should recall that for years, proof verifiers have automated a well-defined class of simple inferences within complex logics.

To strengthen the case for more expressive types, we cite two examples from the monad literature of problems with limited type systems. In [Ste94], Steele writes, "...the principal practical motivation for [a program simplifier] was to transform the code into a form acceptable to the Haskell type-checker". In other words, he had to treat programs at the syntactic level in order to bypass the type system. In [Fil94], Filinski circumvents ML's type system using a universal type. He also states in a different context, "Peyton-Jones and Wadler probe the relationship between monads and CPS further, and Wadler analyzes composable continuations from a monadic perspective, but in both cases the restriction to Hindley-Milner typeability obscures the deeper connections."

A.2 Typed versus untyped values

Typed languages have multiple value domains, one for each type, and determine expression types statically. Untyped languages have a single value domain that is a sum of several others, and determine types dynamically. Untyped languages also make fewer type distinctions than typed languages, particularly with respect to procedures. For instance, Scheme does not distinguish procedures returning numbers from procedures returning pairs.

Monads are typed, as are all category-theoretic concepts, and denotations are polymorphic over values. For example, we can type the language construct `%zero?` as

`%zero?` : $Den(Num) \rightarrow Den(Bool)$

However, in SL, we compute over a single untyped value domain, represented as an extensible sum. Thus `%zero?` has type

`%zero?` : $Den \rightarrow Den$

where we write Den instead of $Den(Val)$. Tagging values with their types exposes the treatment of types in the semantic equations and abstracts from the existing Scheme types.

Most type systems are not powerful enough to type typed interpreters. For example, the interpreters in [Wad92] are untyped even though they are written in Haskell, which is typed. The problem is that we would like to write

`(%var 'x)` : Num

when `x` is a number; however, the type of `(%var 'x)` depends on the context in which the expression occurs. In general, the type of an expression needs to include the names and types of its free variables, just as in the usual sequent-based typing rules.

On a slightly different subject, 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 extensible sums yields behavior such as

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

which shows that monads are not the right tool for building extensible sums.

A.3 Extensible sums and products

Although extensible sums and products play little part in this thesis, they are useful for building extensible systems (see [Esp93b]), and I believe they capture the essential aspects of object-oriented programming. They also demonstrate how category theory can aid language design. We consider types

$$\begin{aligned} S &= S_1 + S_2 + \cdots \\ P &= P_1 \times P_2 \times \cdots \end{aligned}$$

that can be extended either statically or dynamically. It doesn't matter which; these considerations are orthogonal to the basic idea. To S and P there correspond extensible functions

$$s : S \to B$$
$$p : A \to P$$

Note that s and p have opposite types. As we extend S or P, we also extend s or p. As an example, suppose that s computes various vehicles' maximum speeds. Then

$$s : \mathit{Vehicle} \to \mathit{Number}$$

where *Vehicle* is an extensible sum. Extensible sums are simply generic functions in the sense of CLOS [KBdR91], while extensible products are not commonly used. The types of s and p come directly from the category-theoretic definitions of sum and product. For other theoretical treatments of object-oriented programming, see [GM94].

Appendix B

Code

This appendix lists the Scheme code for the monad transformers and language constructs presently supported by SL.

B.1 Monad transformer definitions

The code for transforming types and inverse unit operators is omitted for clarity.

```
;;; Environments: F(T)(A) = Env -> T(A)

(define (env-trans t)
  (with-monad t
   (lambda (unit bind compute)
     (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))

      ))))
```

Figure B.1: Environment transformer

```
;;; Exceptions: F(T)(A) = T(A + X)

(define (exception-trans t)
  (with-monad t
   (lambda (unit bind compute)
     (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 compute-x)))

      ))))
```

Figure B.2: Exception transformer

```
;;; Continuations: F(T)(A) = (A -> T(Ans)) -> T(Ans)

(define (continuation-trans t)
  (with-monad t
   (lambda (unit bind compute)
     (make-monad

      (lambda (a)
        (lambda (k) (k a)))

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

      (lambda (c f)
        (compute (c (compose1 unit value->answer)) f))

      ))))
```

Figure B.3: Continuation transformer

```
;;; Stores: F(T)(A) = Sto -> T(A * Sto)

(define (store-trans t)
  (with-monad t
   (lambda (unit bind compute)
     (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 (initial-store))
          (lambda (a*s)
            (compute-store (f (left a*s)) (right a*s)))))

      ))))
```

Figure B.4: Store transformer

```
;;; Lifting 1: F(T)(A) = 1 -> T(A)

(define (lift1-trans t)
  (with-monad t
   (lambda (unit bind compute)
     (make-monad

      (lambda (a)
        (lambda () (unit a)))

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

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

      ))))
```

Figure B.5: First lifting transformer

```
;;; Lifting 2: F(T)(A) = T(1 -> A)

(define (lift2-trans t)
  (with-monad t
   (lambda (unit bind compute)
     (make-monad

      (lambda (a)
        (unit (lambda () a)))

      (lambda (c f)
        (bind c (lambda (l) (f (l)))))

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

      ))))
```

Figure B.6: Second lifting transformer

```
;;; Lists: F(T)(A) = T(List(A))

(define (list-trans t)
  (with-monad t
   (lambda (unit bind compute)

      (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))))

       ))))
```

Figure B.7: List transformer

```
;;; Monoids: F(T)(A) = T(A * M)

(define (monoid-trans t)
  (with-monad t
   (lambda (unit bind compute)
     (make-monad

      (lambda (a) (unit (pair a (monoid-unit))))

      (lambda (c f)
        (bind c
          (lambda (a*m)
            (let ((c2 (f (left a*m))))
              (bind c2
                (lambda (a*m2)
                  (unit
                   (pair (left a*m2)
                         (monoid-product
                          (right a*m) (right a*m2))))))))))

      (lambda (c f)
        (compute
         c (lambda (a*m)
             (compute-m (f (left a*m)) (right a*m)))))

      ))))
```

Figure B.8: Monoid transformer

```
;;; Resumptions: F(T)(A) = fix(X) T(A + X)

(define (resumption-trans t)
  (with-monad t
   (lambda (unit bind compute)
     (make-monad

      (lambda (a) (unit (in-left a)))

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

      (lambda (c f)
        (compute
         (let loop ((c c))
           (bind c
             (sum-function
              (compose1 unit f)
              loop)))
         id))

      ))))
```

Figure B.9: Resumption transformer

B.2 Language construct definitions

```
;;; 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)))))))))
```

Figure B.10: Amb

```
;;; Reset

;; [[ (reset E) ]] k = k (E \i.i)

(define %reset
  (let ((mapC (get-map 'conts 'top))
        (unitC (get-unit 'cont-values 'cont-answers))
        (bindC (get-bind 'cont-values 'cont-answers)))
    (lambda (exp)
      (mapC exp
        (lambda (cont)
          (lambda (k)
            (bindC (cont unitC) k)))))))
```

Figure B.11: Reset

```
;;; Stores

(define %fetch
  (let ((unitT (get-unit 'store-pairs 'store-results))
        (unitS (get-unit 'stores 'top)))
    (lambda (loc)
      (unitS
       (lambda (sto)
         (unitT (pair (store-fetch sto loc) sto)))))))

(define %store
  (let ((unitS (get-unit 'stores 'top))
        (unitT (get-unit 'store-pairs 'store-results))
        (bindV (get-bind 'store-values 'top)))
    (lambda (loc val)
      (bindV val
        (lambda (val)
          (unitS
           (lambda (sto)
             (unitT (pair val (store-store sto loc val))))))))))
```

Figure B.12: Stores

```
;;;; Output

(define %write
  (let ((unitV (get-value-unit 'unit 'top))
        (mapO (get-map 'output 'top)))
    (lambda (message)
      (mapO (unitV 'unit)
        (lambda (a*m)
          (let ((a (left a*m))
                (m (right a*m)))
            (pair a (cons message m))))))))
```

Figure B.13: Output

```
;;;; While

(define %while
  (let ((bindB (get-value-bind 'booleans 'top)))
    (lambda (c1 c2)
      (letrec
          ((loop
            (bindB c1
              (lambda (b)
                (if b
                    (%begin2 c2 loop)
                    (%unit))))))
        loop))))
```

Figure B.14: While

```
;;; Begin

(define-show 'unit (lambda (b) 'unit))

(define %unit (make-op0 'unit 'unit))

(define %unit? (make-type-predicate 'unit))

(define %begin2
  (let ((bindV (get-bind 'bottom 'top)))
    (lambda (c1 c2)
      (bindV c1
        (lambda (v1) c2)))))

(define (%begin . s)
  (reduce %begin2 (%unit) s))
```

Figure B.15: Begin

```
;;;; Error exceptions

(define (raise-error top)
  (let ((unit (get-unit 'errors top)))
    (lambda (msg) (unit (in-right msg)))))

(define %error (raise-error 'top))
```

Figure B.16: Error exceptions

```
;;;; Error values

(define-show 'errors
  identity-procedure)

(define-predicate 'errors
  (lambda (x)
    (and (pair? x)
         (eq? (car x) 'error))))

(define (raise-error top)
  (get-value-unit 'errors top))

(define %error (raise-error 'top))

(define %error? (make-type-predicate 'errors))
```

Figure B.17: Error values

```
;;; Batch I/O

;; IO     = Input * Output
;; Input  = List Number
;; Output = List Number

(define %end-of-input?
  (let ((unitT (get-unit 'io-pairs 'io-results))
        (unitS (get-unit 'io 'top))
        (unitB (get-value-unit 'booleans 'io-values)))
    (lambda ()
      (unitS
       (lambda (io)
         (unitT (pair (unitB (null? (batch-input io)))
                      io)))))))

(define %read
  (let ((unitT (get-unit 'io-pairs 'io-results))
        (unitS (get-unit 'io 'top))
        (unitN (get-value-unit 'numbers 'io-values)))
    (lambda ()
      (unitS
       (lambda (io)
         (unitT (pair (unitN (car (batch-input io)))
                      (make-batch (cdr (batch-input io))
                                  (batch-output io))))))))))

(define %write
  (let ((unitS (get-unit 'io 'top))
        (unitT (get-unit 'io-pairs 'io-results))
        (bindN (get-value-bind 'numbers 'top))
        (unitU (get-value-unit 'unit 'io-values)))
    (lambda (val)
      (bindN val
        (lambda (val)
          (unitS
           (lambda (io)
             (unitT
              (pair (unitU 'unit)
                    (make-batch
                     (batch-input io)
                     (cons val (batch-output io))))))))))))
```

Figure B.18: Batch I/O

```
;;; Booleans

(define-show 'booleans
  (lambda (b) (if b 'true 'false)))

(define-predicate 'booleans
  boolean?)

(define %boolean? (make-type-predicate 'booleans))

(define %true (make-op0 'booleans #t))
(define %false (make-op0 'booleans #f))

(define (b-or x y)  (or x y))
(define (b-and x y) (and x y))

(define %not (make-op1 'booleans 'booleans not))
(define %or  (make-op2 'booleans 'booleans b-or))
(define %and (make-op2 'booleans 'booleans b-and))

(define %if
  (let ((bind (get-value-bind 'booleans 'top)))
    (lambda (p c a)
      (bind p
        (lambda (p)
          (if p c a))))))
```

Figure B.19: Booleans

```
;;; 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)))

    ;; tilt : cont-value * (procedures -> env-results) -> conts

    (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)))))))))
```

Figure B.20: Call with current continuation

```
;;; Dynamically scoped procedures

(define-show 'procedures
  (lambda (p) '<procedure>))

(define-predicate 'procedures
  procedure?)

;; Proc : env-values -> envs

(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))
        (bindV (get-bind 'env-values 'top))
        (unitE (get-unit 'envs 'top)))
    (lambda (proc arg)
      (bindP proc
        (lambda (proc)
          (bindV arg
            (lambda (arg)
              (unitE (proc arg))))))))))
```

Figure B.21: Dynamically scoped procedures

```
;;; Environments

;; Env = Id -> env-values

(define %var
  (let ((unitT (get-unit 'env-values 'env-results))
        (unitE (get-unit 'envs 'top))
        (error (raise-error 'env-results)))
    (lambda (id)
      (unitE
       (lambda (env)
         (let ((binding (env-lookup env id)))
           (if binding
               (unitT (right binding))
               (error (unbound-error id)))))))))

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

Figure B.22: Environments

```
;;; Fixed points

(define %fix
  (let ((bindP (get-value-bind 'procedures 'top))
        (unitL (get-unit 'lifts 'top))
        (bindV (get-bind 'env-values 'lifts))
        (unitR (get-unit 'env-results 'lifts)))
    (lambda (p)
      (bindP p
        (lambda (p)
          (unitL
           (fix (lambda (l)
                  (bindV l
                    (lambda (v)
                      (unitR (p v))))))))))))

(define %rec
  (let ((unitE (get-unit 'envs 'top))
        (bindE (get-bind 'envs 'top))
        (bindV (get-bind 'env-values 'lifts))
        (unitL (get-unit 'lifts 'env-results))
        (unitR (get-unit 'env-results 'lifts)))
    (lambda (name c)
      (bindE c
        (lambda (c)
          (unitE
           (lambda (env)
             (unitL
              (fix (lambda (l)
                     (bindV l
                       (lambda (v)
                         (unitR (c (env-extend env name v)))
                         ))))))))))))
```

Figure B.23: Fixed points

```
;;; Letrec

(define %letrec
  (let ((unitE (get-unit 'envs 'top))
        (bindE (get-bind 'envs 'top))
        (bindV (get-bind 'env-values 'lifts))
        (unitL (get-unit 'lifts 'env-results))
        (unitR (get-unit 'env-results 'lifts)))
    (lambda (name c1 c2)
      (bindE c1
        (lambda (c1)
          (bindE c2
            (lambda (c2)
              (unitE
               (lambda (env)
                 (unitL
                  (bindV
                   (fix
                    (lambda (l)
                      (bindV l
                        (lambda (v)
                          (unitR (c1 (env-extend env name v)))
                        ))))
                   (lambda (v)
                     (unitR (c2 (env-extend env name v)))
                   ))))))))))))
```

Figure B.24: Letrec using fixed points

```
;;; Numbers

(define-show 'numbers identity-procedure)
(define-predicate 'numbers number?)

(define %num (get-value-unit 'numbers 'top))

(define %+ (make-op2 'numbers 'numbers +))
(define %- (make-op2 'numbers 'numbers -))
(define %* (make-op2 'numbers 'numbers *))

(define %1+ (make-op1 'numbers 'numbers 1+))
(define %-1+ (make-op1 'numbers 'numbers -1+))

(define %/
  (let ((unit (get-value-unit 'numbers 'top))
        (bind (get-value-bind 'numbers 'top))
        (error (raise-error 'top)))
    (lambda (a1 a2)
      (bind a1
        (lambda (a1)
          (bind a2
            (lambda (a2)
              (if (zero? a2)
                  (error (divide-by-zero-error))
                  (unit (/ a1 a2))))))))))

;;; Numeric predicates

(define %zero? (make-op1 'numbers 'booleans zero?))

(define %= (make-op2 'numbers 'booleans =))
(define %< (make-op2 'numbers 'booleans <))
(define %> (make-op2 'numbers 'booleans >))
(define %<= (make-op2 'numbers 'booleans <=))
(define %>= (make-op2 'numbers 'booleans >=))

(define %number? (make-type-predicate 'numbers))
```

Figure B.25: Numbers

```
;;; Statically scoped procedures

(define-show 'procedures
  (lambda (p) '<procedure>))

(define-predicate 'procedures
  procedure?)

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

(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)))))))))

(define %procedure? (make-type-predicate 'procedures))
```

Figure B.26: Statically scoped procedures

```
;;; Resumptions

;; F(T)(A) = rec(X) T(A + X)
;; res-top = T(A + X), res-bottom = A + X

(define %pause
  (let ((bindR (get-bind 'res-top 'top))
        (unitT (get-unit 'res-bottom 'top)))
    (lambda (c)
      (bindR c
        (lambda (t)
          (unitT (in-right t)))))))

(define (%seq2 c1 c2)
  (%begin2 c1 (%pause c2)))

(define (%seq . s)
  (reduce %seq2 (%pause (%unit)) s))

(define (%par2 c1 c2)
  (%amb (%then c2 c1)
        (%then c1 c2)))

(define (%par . s)
  (reduce %par2 (%pause (%unit)) s))

(define %then
  (let ((bindR (get-bind 'res-bottom 'top))
        (unitT (get-unit 'res-top 'top)))
    (lambda (c1 c2)
      (bindR c1
        (sum-function
         (lambda (a) (%pause c2))
         (lambda (t) (%pause (%par2 (unitT t) c2))))))))
```

Figure B.27: Resumptions

```
;;; Products

(define-show 'pairs
  (lambda (p)
    `(pair ,(show-value (left p))
           ,(show-value (right p)))))

(define-predicate 'pairs pair?)

(define %pair? (make-type-predicate 'pairs))

(define (make-pair-op op)
  (let ((unit (get-unit 'bottom 'top))
        (bind (get-value-bind 'pairs 'top)))
    (lambda (a)
      (bind a
        (lambda (a)
          (unit (op a)))))))

(define %left (make-pair-op left))
(define %right (make-pair-op right))

(define %pair
  (let ((unit (get-value-unit 'pairs 'top))
        (bind (get-bind 'bottom 'top)))
    (lambda (c1 c2)
      (bind c1
        (lambda (v1)
          (bind c2
            (lambda (v2)
              (unit (pair v1 v2))))))))))
```

Figure B.28: Products

```
;;; Sums

(define-show 'sums
  (lambda (p)
    (sum-case p
      (lambda (x) `(in-left ,(show-value x)))
      (lambda (x) `(in-right ,(show-value x))))))

(define-predicate 'sums sum?)

(define %sum? (make-type-predicate 'sums))

(define (make-sum-op op)
  (let ((unit (get-value-unit 'sums 'top))
        (bind (get-bind 'bottom 'top)))
    (lambda (a)
      (bind a
        (lambda (a)
          (unit (op a)))))))

(define %in-left (make-sum-op in-left))
(define %in-right (make-sum-op in-right))

(define %case
  (let ((bindS (get-value-bind 'sums 'top))
        (bindP (get-value-bind 'procedures 'top))
        (unitV (get-unit 'bottom 'env-values))
        (unitR (get-unit 'env-results 'top)))
    (lambda (x f g)
      (bindS x
        (sum-function
         (lambda (x)
           (bindP f (lambda (f) (unitR (f (unitV x))))))
         (lambda (x)
           (bindP g (lambda (g) (unitR (g (unitV x))))))))))))
```

Figure B.29: Sums

```
;;; Shift

;; [[ (shift p) ]] k = (p (\v. \k1. (k1 (k v))) \i.i)

(define %shift
  (let ((mapC (get-map 'conts 'top))
        (mapK (get-map 'conts 'env-results))
        (iunitK (get-iunit 'conts 'env-results))
        (unitC (get-unit 'cont-values 'cont-answers))
        (bindC (get-bind 'cont-values 'cont-answers))
        (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)))

    ;; tilt : cont-value * (procedures -> env-results) -> conts

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

    (define (cont-compose k1 k2)
      (lambda (cv)
        (bindC (k2 cv) k1)))

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

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

            (cont
             (lambda (cv)
               ((tilt cv
                      (lambda (p)
                        (p (unitP shift-proc))))
                unitC)))))))))
```

Figure B.30: Shift