Semantic Lego

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

January 1994

Abstract

We exhibit SEMANTIC LEGO, a modular system for constructing programming language implementations from denotational-style building blocks. We present languages as abstract data types, show that ADTs have advantages over interpreters, and incrementally construct semantics for them using type transformations and liftings. We derive specific type transformations (third order) and liftings (monads) from more fundamental considerations and show that monads transform rather than compose. SEMANTIC LEGO extends Moggi, Wadler, and Steele's earlier work and has applications to language extensibility, the construction of interpreters and compilers, and the study of semantic models.

1 Introduction

Our goal is to construct implementations for a wide variety of programming languages from a small set of reusable parts. Beginning from Eugenio Moggi's work [Mog89a], we present languages as abstract data types and incrementally construct ADT implementations from semantic modules. We work in the area of "computational semantics" in that we demonstrate SEMANTIC LEGO, a prototype implementation in Scheme, rather than construct domain-theoretic models.

To incrementally construct an ADT, we begin with a base type and set of operators. We repeatedly transform the type, lift the operators to the new type, and add new operators. To build domains of complexity sufficient to model programming languages, we use third order polymorphic types and their associated lifting operators, but the basic paradigm remains the same.

Although these methods were developed in the context of Moggi, Wadler, and Steele's work on monads, we try to depart from monads as much as possible in order to place them in a broader context. The reader does not need to understand monads to read this paper.

With the right building blocks, SEMANTIC LEGO has the potential to construct an enormous range of models for an equally large range of languages. Soon it should be possible to construct from parts not only interpreters with features such as non-determinism, state, and parallelism, but also translators such as CPS conversion, environment conversion, and code generation, and semantics for program debugging and monitoring. The main contributions of this paper are:

- to propose that ADTs are more appropriate than interpreters for embedding one language in another,
- to show in practice how to construct complex semantic ADTs from reusable parts,
- to show, in an appropriate context, the necessity of third order types for building semantic models,
- to present monads as an instance of the more general concept of lifting, and
- to show, in an appropriate context, the necessity of monads for extensible lambda calculus.

2 Language as ADT

This section presents programming languages as abstract data types, questions the need for interpreters, and exhibits monolithic and modular implementations. The idea of language as ADT pertains mainly to computer implementation, since it is already well-known in mathematical semantics.

2.1 Signatures

We embed a *child* programming language in a *parent* as an abstract data type. To be convenient, the parent should have adequate support for ADTs as well as first-class functions, sums, and products.

`while`	:	$exp \times exp$	$\rightarrow$	exp
`c-if`	:	$exp \times exp \times exp$	$\rightarrow$	exp
`seq`	:	$exp \times \ldots \times exp$	$\rightarrow$	exp
`store`	:	$symbol \times exp$	$\rightarrow$	exp
`fetch`	:	$symbol$	$\rightarrow$	exp
`num`	:	$number$	$\rightarrow$	exp
`c+,c-,c*,c/`	:	$exp \times exp$	$\rightarrow$	exp
`true`	:	exp		
`false`	:	exp		
`c-not`	:	exp	$\rightarrow$	exp
`c-zero?`	:	exp	$\rightarrow$	exp
`show`	:	exp	$\rightarrow$	$value$

Figure 1: SIPL ADT signature

We consider a simple imperative programming language SIPL, specified by the ADT signature shown in figure 1, which introduces the single abstract type *exp* of value-returning, store-transforming expressions. Since we use Scheme as a parent language, we prefix some operators with `c` or `c-` (for *child*) to avoid name conflicts. The types *symbol*, *number*, and *value* are those of the parent language. With a more complex signature, we could describe a strongly-typed language with different abstract types for boolean expressions, numeric expressions, and statements.

Using an appropriate implementation of the ADT, the following Scheme expression, which is also a SIPL program, evaluates to `120`. The syntax can be improved, as will be seen in section 2.3.2.

```
(show
 (seq
  (store 'n (num 5))
  (store 'r (num 1))
  (while
   (c-not (c-zero? (fetch 'n)))
   (seq
    (store 'r (c* (fetch 'r) (fetch 'n)))
    (store 'n (c- (fetch 'n) (num 1)))))
  (fetch 'r))).
```

Papers and textbooks often exhibit interpreters rather than ADTs. Interpreters allow variation in syntax but lack a clear abstraction barrier between syntax and semantics. ADTs explicitly allow multiple implementations of the same signature to provide different semantics. For example, a semantics could translate programs to assembly language instead of than evaluate them.

The above program runs without an interpreter, showing that new semantics does not require new syntax. In essence, the child reuses the parent's syntax, showing that "metalinguistic abstraction" is not fundamentally different from procedural and data abstraction, contrary to some presentations [AKHS88]. Since the interactivity of the child is also inherited, the ADT formulation suitable for both interpreted and compiled parent languages.

On the other hand, compositionality, that the meaning of the whole is determined by the meaning of the parts, is the essential glue that binds parent and child. If either is non-compositional, more complex techniques are required.

2.2 Monolithic Implementations

Using standard techniques, we construct a model for SIPL:

$$\begin{aligned} Exp &= Sto(Val) \rightarrow Val \times Sto(Val) \\ Val &= boolean + number, \end{aligned}$$

where *Sto(A)* is an ADT for stores. Figure 2 shows a Scheme implementation of SIPL over this domain. By relying on Scheme's implicit sum of types, this implementation is shorter but less clear than had sums been represented explicitly.

2.3 Modular Implementations

We use SEMANTIC LEGO to construct, from modular components, models for both SIPL and a simple functional language. Neither exercises the system's potential, and I expect more impressive examples by the time final copy is due.

2.3.1 Imperative language

First, we construct a type of imperative computations and obtain its signature:

```
(define imperative-computations
  (construct-type
   booleans
   numbers
   side-effects))

(type-signature imperative-computations)
;Value: (let A0 (+ num (+ bool 0))
          (-> (sto A0) (* (sto A0) A0))).
```

`Construct-type` composes a sequence of modules, each of which forms part of the resulting model. The order of module composition is important. We can also obtain a list of operators:

```
(type-operators imperative-computations)
;Value: (show boolean-unit boolean-bind + - * /
         number-unit number-bind fetch store seq)
```

These operators implement SIPL, whether directly (like `fetch`) or indirectly (like `number-unit`). The `booleans` module provides `boolean-unit` and `boolean-bind`, the

```
;;; SIPL ADT implementation

;; Exp = Sto(Val) -> Val*Sto(Val)
;; Val = boolean + number

(define (show exp)
  (car (exp empty-store)))

(define ((c-if e1 e2 e3) s)
  (let ((vs (e1 s)))
    (if (car vs)
        (e2 (cdr vs))
        (e3 (cdr vs)))))

(define ((while e1 e2) s)
  (let ((vs (e1 s)))
    (if (car vs)
        ((seq e2 (while e1 e2))
         (cdr vs))
        (true (cdr vs)))))

(define (seq . exps)
  (reduce seq2 true exps))

(define ((seq2 e1 e2) s)
  (e2 (cdr (e1 s))))

(define (true s) (cons #t s))
(define (false s) (cons #f s))

(define ((num n) s)
  (cons n s))

(define ((store name exp) s)
  (let ((vs (exp s)))
    (true (store-store (cdr vs) name (car vs)))))

(define ((fetch name) s)
  (cons (store-fetch s name) s))

(define (((make-unary-op op) exp) s)
  (let ((vs (exp s)))
    (cons (op (car vs)) (cdr vs))))

(define c-not (make-unary-op not))
(define c-zero? (make-unary-op zero?))

(define (((make-binary-op op) e1 e2) s)
  (let ((vs1 (e1 s)))
    (let ((vs2 (e2 (cdr vs1))))
      (cons (op (car vs1) (car vs2))
            (cdr vs2)))))

(define c+ (make-binary-op +))
(define c- (make-binary-op -))
(define c* (make-binary-op *))
(define c/ (make-binary-op /))
```

Figure 2: Monolithic SIPL implementation

`numbers` module provides `number-unit`, `number-bind`, `+`, `-`, `*`, and `/`, and the `side-effects` module provides `seq`, `fetch`, and `store`. The base type `0` provides `show`. After extracting the operators, we can use them to define the rest of the SIPL ADT:

```
(define (get-op name)
  (get-operator name imperative-computations))

(define boolean-unit (get-op 'boolean-unit))
(define number-bind  (get-op 'number-bind))

(define (is-zero? n)
  ((number-bind
    (lambda (n)
      (boolean-unit (zero? n))))
   n))
...
```

2.3.2 Functional language

For a second example, we build a call-by-value functional language with a nondeterminism operator *amb*:

```
(define functional-computations
  (construct-type
   numbers
   booleans
   cbv-procedure-environments
   lists))

(type-signature functional-computations)
;Value: (let A0 (+ (+ (+ 0 num) bool) proc)
          (-> (env A0) (list A0)))
```

After defining extra features, using only the operations provided by the ADT, we can execute programs such as:

```
(show
 (let ((n (var 'n))
       (fact (var 'fact)))
   (c-letrec
    'fact
    (c-lambda 'n
      (c-if (c-zero? n)
            (num 1)
            (c* n (call fact (c- n (num 1))))))
    (call fact (num 5)))))
;Value: (120)

(show
 (let ((x (var 'x)))
   (call (c-lambda 'x (c+ x x))
         (amb (num 1) (num 2)))))
;Value: (2 4)
```

The use of `let` to abbreviate variable reference shows that semantic abstraction in the parent language allows syntactic abstraction in the child. By writing procedures to generate child code, we obtain a simple but powerful macro facility. If the parent language itself has extensible syntax, we can improve child syntax further.

2.4 Applications

SEMANTIC LEGO now includes modules for booleans, numbers, exceptions, side-effects, and several types of environment / procedure combinations, but this selection hardly reflects its potential. By providing modules that manipulate expressions rather than procedures, I expect to build CPS transforms, environment conversions, and other semantic-based translations. By mixing procedural and symbolic representations, it should be possible to propagate environments and state through translations. In general, the space of realizable semantics should run from parallelism and nondeterminism through abstract interpretation, compilation, monitoring, and debugging.

Syntactic modules may be powerful enough to construct expression representations of ADT implementations. If not, we can form them by lifting the concepts discussed in section 3 to the syntactic domain. Implementations could then be transformed to increase efficiency. Syntactic treatments of monads and monad constructors are given in [Wad92b] and [MC93] respectively.

3 Modular Models

To build the models shown, `construct-type` begins with the base type `0` and the operator `show`. Building blocks such as `booleans` and `side-effects` each contain a type transformation, a set of lifting operators, and a set of language operators that are used to:

1. Transform the old type to form a new one,
2. Lift the old operators onto the new type, and
3. Add new operators.

For example, a `numbers` module might transform the type A into $A + number$, lift operators to act as usual on A but as the identity on numbers, and add operations $+$ and $*$ that act as usual on numbers but as the identity on A. Raising errors, rather than acting as the identity, requires a connection between the `numbers` and `errors` modules, as discussed in section 3.5. By combining several modules, we can form a useful semantic ADT.

3.1 Types

This section shows that second order types, such as $T(A) = A + Num$, are insufficient to build semantic domains. Fortunately, the same three step paradigm applies to third order types, which are sufficient. Suppose we want to form, from modular components, a model for a language with stores and call-by-value procedures:

$$Env(A) \to Sto(A) \to A \times Sto(A),$$

where A is the type of values. As a first try, we compose these type constructors:

$$\begin{array}{lcl} E(A) & = & Env(A) \to A \\ S(A) & = & Sto(A) \to A \times Sto(A). \end{array}$$

Unfortunately, we obtain

$$\begin{array}{lcl} E(S(A)) & = & Env(A_1) \to A_1 \text{ where} \\ A_1 & = & Sto(A) \to A \times Sto(A), \end{array}$$

not what we wanted, since environments denote A_1 instead of A. The problem is that the two constructions are not sufficiently "intertwined". A solution is to pass from type constructors to type constructor constructors, which we abbreviate as TCs and TCCs. Instead of E and S above, we write

$$\begin{array}{lcl} E(T)A & = & Env(A) \to TA \\ S(T)A & = & Sto(A) \to T(A \times Sto(A)). \end{array}$$

where T is a TC. Now, as desired, we have

$$E(S(I))A \quad = \quad Env(A) \to Sto(A) \to A \times Sto(A),$$

where I is the identity TC. We can recover the undesirable model as $E(I)S(I)A$. Other useful TCCs are:

- Lists: $F(T)A = T(List\ A)$
- Sums: $F(T)A = T(A + X)$
- Resumptions: $F(T)A = \mu X.\ T(A + X)$
- Monoids: $F(T)A = T(A \times M)$
- Continuations: $F(T)A = (A \to T0) \to T0$
- CBN environments: $F(T)A = Env(TA) \to TA$,

where μ is a fixed point operator on types. It is easy to define more, but section 5 discusses constraints on their structure.

At any TCC F in a composition of several, the argument T is determined by the TCCs applied before F (as usual), while A is determined by those applied after (possibly unexpected). By changing the order of composition, we achieve different useful results. For example, composing lists and side-effects in both orders, we obtain:

$$\begin{array}{lcl} S(L(I))A & = & Sto(A) \to List(A \times Sto(A)) \\ L(S(I))A & = & Sto(List\ A) \to List\ A \times Sto(List\ A). \end{array}$$

Current polymorphically typed languages such as ML and Haskell permit only second order types, which we call TCs. Since TCCs are third order, they cannot be implemented directly in these languages.

3.2 Building Blocks

We construct ADTs from types, TC modules, and TCC modules. TC modules transform types, while TCC modules transform TC modules. Types, which represent ADTs, consist of a signature and set of operators. TC and TCC modules consist of a type constructor (a TC or TCC), a set of associated lifting methods, and a set of associated operators.

To apply a TC module to a type, we lift the type's operations through the TC and instantiate the TC's operations on the type. For example, suppose we apply *TA = List A* to *Number*, yielding *List Number*. We instantiate *append* on *List A* to act on *List Number*. Instantiation is just type application. We lift *+* on *Number* to act on *List Number*. The next section describes lifting.

Similarly, to apply a TCC module to a TC module, we lift the TC's operations through the TCC and instantiate the TCC's operations on the TC. As before, instantiation is type application, and we discuss lifting in section 3.4. TCC application differs from TC application because we also lift lifting operators, from lifting through *T* to lifting through *F(T)*, as discussed in section 3.4.2.

3.3 Lifting Through TCs

This section explains what lifting means and shows how to do it. First, we lift binary addition through the lists TC. Addition has type $Num \times Num \rightarrow Num$. Lifted addition has type $List\ Num \times List\ Num \rightarrow List\ Num$. It is computed by forming a cartesian product of the argument lists and mapping addition across the tuples. For example, the lifted sum of *(1, 10)* and *(3, 4, 5)* is *(4, 5, 6, 13, 14, 15)*.

Second, we lift a function $f : A \rightarrow TB$ through $TA = Env \rightarrow A$. The result has signature $TA \rightarrow TB$, that is, $(Env \rightarrow A) \rightarrow (Env \rightarrow B)$ and is written `(λ (ta) (λ (env) (f (ta env))))`.

Once constrained by several properties, liftings are usually unique. First, they must respect a function $unit : A \rightarrow TA$ that lifts values. For lists, *unit* is `(λ (v) (list v))`. For environments, *unit* is `(λ (v) (λ (env) v))`. Second, liftings must respect identities and composition of functions. If $f : B \rightarrow C$ and $g : A \rightarrow B$, we write:

$$\begin{array}{lcl} lift(id) & = & id \\ lift(f \circ g) & = & lift(f) \circ lift(g) \\ lift(f)(unit(a)) & = & unit(f(a)). \end{array}$$

3.3.1 Product Signatures

Since they are common and simple, we focus on lifting signatures of the form

$$(A \mid X) \times \ldots \times (A \mid X) \rightarrow A$$

where X is any parameter type and A is either A, TA, or $F(T)A$ depending on whether the signature describes an operator on types, TCs, or TCCs. This restriction is severe; for instance, these signatures are not products:

- $(A \rightarrow A) \rightarrow A$
- $TA \rightarrow A$
- $TA \rightarrow F(T)A$

The simplest lifting function $map : (A \rightarrow B) \rightarrow (TA \rightarrow TB)$ cannot lift even $A \times A \rightarrow A$. However, when accompanied by a *tensorial strength* $ts : A \times TB \rightarrow T(A \times B)$, *map* can lift all product signatures.

3.3.2 Monads

This section defines monads and explains how they are used. A monad is a triple of a TC *T*, a function $unit : A \rightarrow TA$ for lifting values, and a function $bind : (A \rightarrow TB) \rightarrow (TA \rightarrow TB)$ for lifting functions of signature $A \rightarrow TB$ up to $TA \rightarrow TB$. These functions obey identities similar to those above. In other words, a monad is a lifting operator for the signature $A \rightarrow TB$.

We adopt monads as a common language for lifting. Each TC provides a monad that signatures use to lift functions of their type. This technique succeeds for signatures other than $A \rightarrow TB$ because *bind* can lift products by currying. For example, to lift $f : A \times A \rightarrow A$, we write:

```
(define ((lift f) ta1 ta2)
  ((bind
    (lambda (a1)
      ((bind (lambda (a2) (f a1 a2)))
       ta2)))
   ta1)).
```

Section 3.6 discusses the real purpose of *bind*. That it can lift products is a pleasant side-effect; *map* and *ts* are the weaker operators and therefore more appropriate.

3.4 Lifting Through TCCs

When applying a TCC to a TC, we lift the TC's operators through the TCC. For example, when applying $F(T)A = Env(A) \rightarrow TA$ to $TA = List\ A$, we lift *append* on *List A* to act on $Env(A) \rightarrow List\ A$.

Lifting through TCCs is similar to lifting through TCs. We consider first product signatures, then monads. The final copy of this paper will also describe the properties obeyed by TCC liftings.

3.4.1 Lifting Product Signatures

We lift TC operators by changing T to $F(T)$. For example $TA \times TA \to TA$ becomes $F(T)A \times F(T)A \to F(T)A$. As before, lifting depends on F and on the operator signature, but there is a crucial simplification. Since product signatures involve only TA, and not A, lifting can ignore changes made by right composition. For example, an operator on TA is also an operator on $T(List\ A)$. Left composition requires only slightly more work. If $F(T)A = X \to TA$, an operator on TA lifts through F just as operator on A lifts through $TA = X \to A$.

3.4.2 Lifting Monads

In addition to the operators defined on it, a TC T has methods for lifting functions from A to TA, specifically the monad operators *unit* and *bind*. When applying F to T, the result $F(T)$ must also have *unit* and *bind*. Thus TCCs must describe how to lift monads as well as operators. However, lifting a monad through the TCC F is only slightly harder than than defining it for the TC $F(I)$. For example, *bind* for $F(I)A = Sto(A) \to A \times Sto(A)$ is written:

```
;; f : A -> TB
;; ta : Sto(A) -> A*Sto(A)

(define (((bind f) ta) sto)
  (let ((a*sto (ta sto)))
    (let ((a (car a*sto))
          (s (cdr a*sto)))
      ((f a) s)))).
```

Lifting *bind* through $F(T)A = Sto(A) \to T(A \times Sto(A))$ is only slightly harder:

```
;; f : A -> F(T)B
;; fta : Sto(A) -> T(A*Sto(A))
;; bind : (A -> TB) -> (TA -> TB)

(define (((lifted-bind f) fta) sto)
  ((bind
    (lambda (a*sto)
      (let ((a (car a*sto))
            (s (cdr a*sto)))
        ((f a) s))))
   (fta sto))).
```

Section 5 discusses which lifting operators it is *possible* to lift through which TCCs. Because of stratification, it is not necessary for lifting operators to lift themselves.

3.5 Limitations

This section discusses locality, the primary limitation of type constructors. At present, interacting language features must be defined by a single TCC. Examples include environments and procedures and errors and numbers (for division by zero). A promising solution is to export only lifting operators, such as `number-unit` and `number-bind` (as seen earlier), and later define language features using them. This idea allows access to multiple modules while preserving abstraction by interacting only through exported interfaces.

On the other hand, TCCs are not entirely local. For instance, the side-effects TCC $F(T)A = Sto(A) \to T(A \times Sto(A))$ modifies T by both right and left composition in one step. As noted by Jones and Duponcheel [JD93], this construction yields a different monad than composition with two separate TCs.

In general, module interfaces should be as flexible as possible, since they need not implement language ADTs directly. This organization leaves an outlet for parametrization in the style of metaobject protocols [Kic92, KBdR91].

3.6 Why Monads?

This section describes when we need monads in place of the weaker *map* and *ts* operators described in section 3.3.1. Monads are connected with lambda calculus, so we examine the call-by-value environment TCC, $F(T)A = Env(A) \to TA$. This definition fails to account for procedures, so we have more precisely:

$$\begin{array}{lcl} F(T)A & = & Env(Val) \to TVal \\ Val & = & A + Proc \\ Proc & = & Val \to TVal. \end{array}$$

Choosing $TVal$ as the procedure return type allows use of operators defined by T in procedure bodies. Taking $TA = List\ A$ for example, if we wrote Val instead of $TVal$, we could not use *amb* in procedure bodies.

Calling a procedure implies returning $TVal$ from "inside" T. All formulations of monads include an operator for this purpose; its most explicit form is $join : TTA \to TA$. Monads are useful only if a TCC applies T inside T, as in environments or resumptions. Simpler TCCs, such as lists or exceptions, do not require them. Unfortunately, if one TCC's operators require monads, others applied before must propagate them.

The structure of "lambda calculus over a monad" arises entirely from the environment TCC. Monads are required because procedural abstraction over language features implies nested use of T.

4 Previous Work

This section relates Semantic Lego to previous work and addresses the question of why it took so long for the rest of us to realize what Moggi knew all along.

4.1 Moggi

In [Mog89a, Mog89b, Mog91b], Eugenio Moggi realized that monads could be used to structure denotational models. He showed that lambda calculus could be parametrized by an arbitrary monad to yield widely varying semantics. His course notes [Mog89a], written while he was visiting Stanford, also present much additional material, including *monad constructors*, called TCCs here. The present paper transfers Moggi's ideas from mathematics to computation, simplifying and motivating them.

For several reasons, the functional programming community has not realized that monad constructors are sufficient to build modular models. First, having understood Moggi's original concept of "monads as a notion of computation", researchers began to search for a way to *combine* monads rather than *transform* them. Composition is simpler than transformation, and type constructors already compose, so monads could be expected to as well. Since monads are so clearly appropriate to lambda calculus, it was difficult to imagine how to obtain the same functionality without using them directly. That a higher-level structure, a monad constructor, could express the same semantic content was difficult to see.

Second, Moggi's notes use non-elementary category theory and, on first reading, appear impractical and difficult. The notes are also not widely available, and the only published paper referring to monad constructors is the two-page summary [Mog91a]. His more recent paper [MC93] presents a syntactic treatment in hope of "hiding the category-theoretic concepts and sophistication of the original approach in suitable metalanguages". It describes what is essentially monad constructor translation, but, despite Moggi's good intentions, is no easier than the original.

We may also wonder how Moggi realized that transformation is more tractable than composition. In category-theoretic terms, monads are more likely to be objects than arrows, since they do not have an obvious composition. Once viewed as objects, they transform via endofunctors on the category. So category theory leads, as is often the case, to a suitable design.

4.2 Wadler

In [Wad92a], Philip Wadler popularized Moggi's ideas by presenting monadic interpreters written in Haskell. Later, in [WPJ93, Wad92b], he demonstrated the utility of monads for structuring programs rather than semantics. Monads factor loosely-coupled programs and thereby hide imperative features such as state and I/O, solving the old problem of passing information through a functional program without introducing additional arguments.

In [KW92], Wadler and David King showed how to combine the lists monad L with any monad M to form a monad ML. Despite Moggi's earlier formulation of monad constructors, they described their work as "combining M and L" rather than "constructing ML from M", and even in discussing monad constructors did not realize the significance of the idea.

4.3 Steele

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

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

but not

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

Steele's claim that pseudomonads improve on monad constructors by providing a fixed composition operator would be true if they were equally powerful, but they are not.

Steele's main contribution is a complete implementation of a modular semantics, which was really inspiring. Also, by using the "self argument" technique of object-oriented programming, he shifts focus from pairs of types to *towers* of types. After reading his paper, I had planned to define *stratified monads*, towers of *unit* and *bind* operators that could transformed and thus allow left composition. This approach resembles pseudomonads but is still not as simple or powerful as monad constructors.

4.4 Jones and Duponcheel

In [JD93], Mark Jones and Luc Duponcheel seriously address the problem of composing monads. They find that if one of several auxiliary maps is defined relating the structures of two monads, they can be composed, in an order depending which map is used. They find that some monads compose naturally to the left and some to the right. They come as close to inventing monad constructors as one could without actually arriving, and their work provides useful information about the structure of semantic models.

5 Future Work

This section tries to suggest some interesting questions for future work.

First, liftings cannot be defined for all combinations of TCs and signatures. For example, *map : (A → B) → (TA → TB)* cannot be defined for *TA = A → X*. There is a syntactic criterion (covariance) that determines whether *map* can be defined for a given TC built from sums, products, and functions. Does a syntactic criterion exist for *bind* or *join*?

Second, we observe that *bind* can be lifted through *F(T)A = X → TA* but not through *F(T)A = T(X → A)*. In general, some extensions work by right composition and some by left. Are these useful distinctions? See sections 4.3 and 4.4 for further remarks.

Third, through the Curry-Howard isomorphism, the existence of operators is equivalent to the provability of their types in constructive logic. Jonathan Rees recently used a natural deduction theorem prover to derive monads for a given TC [Ree]. Inversely, what are the internal logics of the Kleisli categories of various monads? Can a theorem prover be parametrized by a monad?

Fourth, in [Mog89a], Moggi discusses lifting natural transformations in the category of (strong) monads. Can this formulation be generalized to higher types or simplified to depend less strongly on monads? Also, are sums and products of monads useful?

6 Acknowledgements

I am grateful to Mary Ng, Franklyn Turbak, Jonathan Rees, and Bill Rozas for encouragement and inspiration, to Eugenio Moggi for his work, and to AT&T for financial support.

References

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

[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.

[Kic92] Gregor Kiczales. Toward a new model of abstraction in software engineering. In *Proceedings of the International Workshop on Reflection and Meta-Level Architecture*, pages 1–11, Tama-City, Tokyo, November 1992.

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

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

[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 : theory/papers/Moggi.

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

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

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

[Ree] Jonathan Rees. Personal communication.

[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.

[Wad92a] 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.

[Wad92b] Philip Wadler. Monads for functional programming. In Manfred Broy, editor, *Program Design Calculi*, Proceedings of the Marktoberdorf Summer School, 1992.

[WPJ93] Philip Wadler and Simon Peyton-Jones. Imperative functional programming. In *Proceedings of the 20th Annual ACM Symposium on Principles of Programming Languages*, Charleston, SC, January 1993.