Constructing Language Processors with Algebra Combinators

Philip Weaver Garrin Kimmell Nicolas Frisby Perry Alexander

Information Technology and Telecommunication Center
Department of Electrical Engineering and Computer Science
The University of Kansas
2335 Irving Hill Road
Lawrence, KS 66045
{pweaver,kimmell,nfrisby,alex}@ittc.ku.edu

Abstract

Modular Monadic Semantics (MMS) is a well-known mechanism for structuring modular denotational semantic definitions for programming languages. The principal attraction of MMS is that families of language constructs can be independently specified and later combined in a mix-and-match fashion to create a complete language semantics. This has proved useful for constructing formal, yet executable, semantics when prototyping languages. In this work we demonstrate that MMS has an additional *software engineering* benefit. Rather than composing semantics for various language constructs, we can use MMS to compose various differing semantics for the same language constructs. We describe *algebra combinators*, the principal vehicle for achieving this reuse, along with a series of applications of the technique for common language processing tasks.

Categories and Subject Descriptors D.1.2 [*Programming Techniques*]: Automated Programming

General Terms Languages

Keywords Meta-Programming, Modular Monadic Semantics

1. Introduction

Modular monadic semantics (MMS) is a technique that has been used extensively to construct language semantics that allow modular composition of orthogonal language constructs. Families of language constructs are identified, and each family is assigned a semantic definition. These independent denotations are then combined into a complete denotation. The advantage to this approach is that it leads to considerable reuse and flexibility when developing new language denotations.

Existing MMS work has focused primarily on the modularization of denotations for a single language processing task, such as type checking, interpretation, or compilation. We identify this pattern of composition as combining *common semantics* across *varied constructs*. In this paper we propose that MMS holds an additional software engineering benefit that has heretofore remained unexploited: the combination of complete specifications for various language processing tasks into more sophisticated specifications. In contrast to the previous pattern of composition, this involves combining *varied semantics* across *common constructs*.

Unfortunately, much of the machinery used in MMS to simplify the first pattern of composition – for common semantics – complicates the second form of composition. There are two primary explanations for this complication. First, semantic definitions are written in non-recursive style, allowing easy composition of denotation functions. The framework utilizes a universal *catamorphism* combinator for recursive processing of abstract syntax trees (Meijer et al. 1991). However, using a catamorphism makes it more difficult to control the traversal in a context-dependent manner, such as choosing which of two denotations to apply at a given point. The second factor is the reliance on monads to encapsulate computational effects. When combining two denotations where the result of the first is used as an input to the second, it is necessary to maintain the monadic effects of the first when applying the second. It seems necessary to break the monadic encapsulation to accomplish this.

While monads appear to be the source of the difficulties in combining semantics, we demonstrate that they are actually the saving grace in resolving those limitations. The trick is to capture the logic for controlling traversal or maintaining effects as yet another computational effect, captured as a monad. By doing this, we define the notion of an *algebra combinator*. In our MMS framework, we use the term *semantic algebra* (or simply algebra) to describe denotation functions. An algebra combinator provides a standard pattern for combining component semantic algebras into a more sophisticated semantic algebra.

The notion of an algebra combinator is the key contribution of this paper. We identify and describe two specific algebra combinators:

- `switchAlg` that combines two semantic algebras with a common semantic domain into a single algebra in which context dictates which algebra to apply to a term.
- `sequenceAlg` that allows the result of one semantic algebra at each node to be used as an input to a second semantic algebra.

We demonstrate the utility of these two approaches with a collection of small examples typical of common language processing tasks.

The remainder of this paper is structured as follows: Section 2 provides an overview of monads and modular monadic semantics. Section 3 presents an example of using the framework to define both static (type checking) and dynamic (interpretation) semantics for a small collection of language semantics. Section 4 presents the key contribution of this paper, two *algebra combinators* for combining different semantics for the same language. In section 5 we demonstrate the use of these algebra combinators with two

GPCE'07, October 1–3, 2007, Salzburg, Austria.
Copyright © 2007 ACM 978-1-59593-855-8/07/0010...$5.00

examples. Section 6 describes related work, and we conclude and identify future work in section 7. Throughout the paper we present our examples in Haskell (Jones 2003). We assume the reader has a basic familiarity with Haskell or a similar statically-typed pure functional language.

2. Modular Monadic Semantics

The foundations of MMS are well represented in the literature. In this section we present an overview of the topic, however the overview is necessarily brief and intended only to give sufficient detail to support the development in later sections. For a more thorough review, we refer the reader to the extensive literature (Duponcheel 1995; Espinosa 1995; Harrison and Kamin 2000; Hutton 1998; Labra Gayo et al. 2001; Liang and Hudak 1996; Liang et al. 1995; Moggi 1990, 1991; Steele 1994; Wadler 1990)

Modular monadic semantic definitions consist of a collection of *semantic building blocks*. Each building block defines a mapping from a particular syntactic language construct to a semantic domain. Within the framework we use the term semantic algebra to refer to these building blocks. These individual semantic algebras can be combined into complete language denotations. Similarly to semantic algebras, these denotations map from a syntactic domain to a semantic domain. They contrast with the semantic building blocks in that they don't define new semantics, but rather are constructed by combining those building blocks in a regular manner. As a result of using the MMS framework to perform this composition, there is minimal effort in constructing new combinations of semantic algebras.

The machinery necessary to support this composition involves two main components: monads and syntactic functors. First, we use *monads* to structure the semantic domains. The monads serve to capture the computational effects of the semantic algebras. We treat these monads as abstract datatypes when defining the semantic algebras. This allows us to delay the construction of a specific monad until the algebras are combined; *monad transformers* make this construction relatively painless. The second component needed to perform the composition of semantic algebras are *syntactic functors* to allow the syntactic space to be specified independently of each other and the semantic denotation functions to be combined. This is accomplished using techniques drawn from generic programming (Jansson and Jeuring 1997).

2.1 Monads

Moggi (1990, 1991) first described the use of monads for structuring computational effects in programming language denotations. A monad is a concept from category theory and was initially used to give a formal account for computational side effects. Wadler (1993, 1990) detailed the usefulness of monads for programming effectful computation within a pure functional language such as Haskell. Using monads serves to ease the effort necessary to program effects while maintaining the referentially-transparent reasoning that is a prime benefit of pure languages. This has proved useful in capturing the semantics of a wide range of common operations in Haskell which have an effectful behavior (Jones 2002).

A formulation of a monad is captured by the following Haskell type class declaration:

```
class Monad m where
  return :: a → m a
  (≫=) :: m a → (a → m b) → m b
```

A monad is a type constructor, `m`, that encapsulates computational effects. A computation that has the type `m a` will yield a value of type `a` when executed. These computations are constructed with the functions `return`, that "lifts" a pure value into an monadic computation; and `>>=` (pronounced "bind"), that allows computations to be sequenced, with the effects threaded through the computations. Not represented in this declaration are three monad laws to which all monads must adhere.

The `Monad` type class represents the minimal support for monadic features. However, there is no way with this particular signature to actually introduce any computational effects. To facilitate this, monads include a collection of *non-proper morphisms* that provide access to the computational effects. The set of non-proper morphisms will vary depending upon the effects being modeled for that particular monad.

As an example, consider the `Reader` monad. This monad represents computations which require a context parameter, such as the typing context in a type checker or a lexical environment in an interpreter. The `Reader` monad has two non-proper morphisms. The first, `ask`, gives the computation access to the environment. The second, `local`, takes a function that modified the environment, and a `Reader` computation, then runs the computation in the modified environment. The `MonadReader` type class declares this interface.

```
class MonadReader e m where
  ask :: m e
  local :: (e → e) → m a → m a
```

The `Reader` declaration below implements this interface. The `Reader` type is parameterized over any environment type; this property allows the `Reader` monad to be used for a variety of different situations with different environment types.

```
newtype Reader e a = Reader { runReader :: (e → a) }

instance MonadReader (Reader e) where
    ask       = Reader id
    local f x = Reader (λe → runReader x (f e))
```

To execute a `Reader` computation, yielding a final value, apply `runReader` to the computation and an initial environment. This function will be used later to execute a type checker or interpreter.

The `Reader` monad is but one of a collection of commonly used monads. Others include a `State` monad capturing state-transformation behavior, an `Error` monad for propagating exceptions, and a `Writer` monad for logging-style write-only behavior. These are all available in the Haskell base libraries, and are employed in various situations. We focus on the `Reader` monad because it is key to the development in the following sections.

The semantic algebras for various language constructs may utilize different computational features. Although the type class for each monad provides an abstract interface for constructing those features, it still remains necessary to define a *specific* monad type constructor that represents all of those features. Constructing this specific type is non-trivial because monads do not, in general, compose (Jones and Duponcheel 1993). This implies that, given two monads `m` and `n`, the type constructor resulting from the composition of those two types is not necessarily a monad. Consequently, combining semantic algebras which utilize two different monads is a challenge that would seem to require defining a completely new monad for each different combination.

This problem can be avoided by using a *monad transformer* (Espinosa 1995; Liang et al. 1995). A monad transformer is a parameterized monad, which takes as a type argument another monad. A monad transformer `t`, parameterized over another monad `m`, has a function `lift :: m a →t m a` that lifts the features of the inner monad `m` into the outer monad transformer. This interface is captured by the `MonadTrans` type class shown below.

```
class MonadTrans t where
  lift :: m a → t m a
```

The Haskell libraries define transformer versions of the standard monads suitable for composition. Defining instances of the various monad interfaces requires a pairwise construction between each monad transformer and inner monad interface. While this could potentially lead to an explosion in code, we have found in practice that most tasks require just a small set of monadic features and the Haskell libraries – that provide the pairwise lifting definitions – are sufficient.

2.2 Modular Semantics

Monads and monad transformers provide the necessary support for combining the computational effects of component semantic algebras within the MMS framework. Combining semantic algebras requires some additional support. First, we must be able to construct lifted value spaces. For example, if one construct returns `Int`s and the other returns `Bool`s, then the combined value space includes both `Int`s and `Bool`s. Secondly, we must be able to combine term (or abstract syntax) spaces. For example, if one set of constructs includes arithmetic operations, such as `Add` and numerals, and a second set includes a branching `If` construct and `Bool` literals, then the combined syntactic domain will include all of these constructs.

2.2.1 Composite Value Domains

The `SubType` class is the interface used to construct combined domains. It defines a relation between a simple value of type `a` and a composite value represented by the type variable `b`.

```
class SubType a b where
  inj :: a → b
  prj :: b → Maybe a
```

The `inj` function injects simple values into the composite domain, and the `prj` value projects a simple value from the composite domain. The `prj` function returns a value of type `Maybe a`, indicating that the projection is a partial function, which will fail (and return the value `Nothing`) if the value represented in the composite value is not of type `a`. Otherwise, the `prj` function will yield a result `Just x`.

A composite domain can be constructed using the `Either` type, which creates a disjoint-union of two component types. Instances for `SubType` for the `Either` type constructor are also shown.

```
data Either a b = Left a | Right b

instance SubType a (Either a b) where
  inj x = Left x
  prj (Left x) = Just x
  prj (Right _) = Nothing

instance SubType a c ⇒ Subtype a (Either b c) where
  inj x = Right (inj x)
  prj (Left x) = Nothing
  prj (Right x) = prj x
```

When specifying a composite type, the `Either` type constructor is applied right-associatively. For example, a compound domain including `Int`s, `Bool`s, and `String`s would be represented as shown below.

```
type Domain = Either Int (Either Bool (Either String ()))
```

The first instance of `SubType` above corresponds with the situation where the basic type is in the left-most position of the sum, and the second instance corresponds with the the situation when the basic type is positioned in the right-hand side of the `Either`. These two instances serve as a polymorphic recursive definition of `SubType` over `Either`s. The `()` type at the right-most position in `Domain` serves to terminate the recursion.

Using the `SubType` class allows the type of the denotation functions to be generalized. For example, the type of a denotation `evalArith` that maps Arithmetic syntax to `Int`s is:

```
evalArith :: Arith → Int
```

The same function, written using the the `SubType` class, is shown below. Rather than a fixed `Int` codomain, the type has a constraint that the codomain can be any type `a`, as long as `Int` is a `SubType` of that type.

```
evalArith :: SubType Int a ⇒ Arith → a
```

2.2.2 Composite Syntax and Semantics

Using monads and the `SubType` class, we can mechanically construct the semantic domain of a semantic algebra. The problems of composing independent syntactic (AST) types and semantic functions remain.

Typically, an abstract syntax tree is represented as an algebraic data type, such as `Arith` and `IfExpr`.

```
data Arith = Add Arith Arith
           | Num Int

data IfExpr = If IfExpr IfExpr IfExpr
            | BLit Bool
```

An interpretation semantic algebra for these forms would have the following types:

```
evalArith :: SubType Int a ⇒ Arith → a
evalIfExpr :: Subtype Bool a ⇒ IfExpr → a
```

However, combining these two functions is not possible without redefining the `Arith` and `IfExpr` datatypes to create a single AST type which contains all of the constructs. This is necessary because both `Arith` and `IfExpr` are recursively-defined closed types, and there is no possibility for extension.

To allow for combining syntactic forms, the `Arith` and `IfExpr` types are defined as non-recursive type constructors. Each recursive occurrence of a type is replaced with the type parameter `x`.

```
data Arith x = Add x x
             | Num Int
data IfExpr x = If x x x
              | BLit Bool
```

Much as the `Either` type constructor creates composite value spaces, the `FSum` type constructor is used to create composite syntactic spaces from component syntax elements `f` and `g`. Similarly to the `SubType` class, a `SubFunctor` class allows injection into and projection out of a composite syntactic domain. We omit the the definitions of `SubFunctor` instances.

```
data FSum f g x = L (f x) | R (g x)
class SubFunctor f fsum where
  inj :: f x → fsum x
  prj :: fsum x → Maybe (f x)
```

`Arith` and `IfExpr` syntactic families are combined using `FSum`. Finally, a `Fix` type constructor is used to "tie the knot" and recover a recursive type.

```
data Fix f = In (f (Fix f))
type Composite  = FSum Arith IfExpr
type Lang = Fix Composite
```

Using `FSum` and `Fix` with non-recursive syntactic domains, and `SubType` for composite value domains, we can account for the types of combining algebras. The last remaining issue is to handle combining the semantic functions themselves.

Below are denotations for the `Arith` and `IfExpr` constructs, using the `SubType` interface to manage composition of domains.

```
evalArith :: SubType Int a ⇒ Arith → a
evalArith (Add x y) =
     case (evalArith x,evalArith y) of
       (Just xv, Just yv) → inj (xv+yv)
       _  → error ‘‘Projection error’’
evalArith (Num x) = inj x
evalIfExpr (If x y z) =
    case prj (evalIfExpr x) of
      Just v → if v
                 then (evalIfExpr y)
                 else (evalIfExpr z)
      Nothing → error ‘‘Projection error’’
evalIfExpr (BLit x) = inj x
```

Similarly to the original closed recursive definitions of the `Arith` and `IfExpr` types, these recursively defined functions are closed and cannot be directly combined. However, both denotations demonstrate a common pattern: the denotation function is applied to each of the subterms, and then a semantic function, such as `+` or `if`, is applied to these evaluated subterms.

Recognizing this pattern, the denotation function can be written in two phases. The first, which handles the recursive application of the denotation to the subterm, is handled generically. The second phase captures the specific meaning of the particular denotation.

A composite syntactic domain will be represented as the `Fix` of some component syntactic constructs. It follows that a composite denotation `eval` will be of type `Fix f →vspace`, where `f` is the `FSum` of the component syntactic constructs, and `vspace` is the composite value domain.

The first step, evaluating all of the subterms, is handled generically using the `Functor` type class.

```
class Functor f where
  fmap :: (a → b) → f a → f b
```

The `fmap` function lifts an input function into the type `f`. Applying `fmap eval` to a term of type `f (Fix f)` yields a term of type `f vspace`. After this step, the specific semantic function, of type `f vspace →vspace` is applied, yielding a value of type `vspace`.

Such functions with the type `f a →a` are called *semantic algebras*. The type constructor `f` is a syntactic functor, and the type `a` is the carrier of the algebra.

Term spaces are combined with `FSum`, making it necessary to define a function that lifts two algebras, `f a →a` and `g a →a`, into a single algebra `(FSum f g) a →a`. This lifting operator `sumAlg` can be defined universally.

```
sumAlg :: (f a → a) → (g a → a) → FSum f g a → a
sumAlg falg galg (L x) = falg x
sumAlg falg galg (R x) = galg x
```

In an extension to the standard MMS framework, we define a higher-order `Algebra` type constructor which contains semantic algebras. Values of type `Algebra f a` are constructed with `mkAlg` and de-constructed using `apply`. The `Algebra` type simplifies the manipulation of semantic algebras as first-class objects, for example incremental update of semantic algebras.

```
newtype Algebra f a = Algebra { apply :: f a → a }
mkAlg :: (Functor f) ⇒ (f a → a) → Algebra f a
mkAlg f = Algebra f
```

The `cata` (catamorphism) function combines all of the steps together. The function is parameterized over an `Algebra`, phi, and a term `In x`. The function first uses `fmap` to map itself recursively across the subterms of the term `x`, then applies `phi` to the resulting term using `apply`.

```
cata :: (Functor f) ⇒ Algebra f carrier → Fix f → carrier
cata phi (In x) = apply phi (fmap (cata phi) x)
```

This completes the MMS framework. Value spaces are combined using `SubType`, computation effects are combined using monad transformers, syntactic functors are combined using `FSum`, and semantic algebras are combined using `sumAlg`. The key is to write the semantics non-recursively, then use the `Fix` type constructor for syntax and the `cata` combinator to complete the recursion.

2.3 InterpreterLib

The overview of modular monadic semantics presented above is a general description of the framework, as has been described extensively in existing literature. We have realized the MMS framework in a library that we call *InterpreterLib*. The library provides a domain-specific language embedded in Haskell that supports common tasks involved in the definition of language processing systems. This library is derivative of the Language Prototyping System (LPS) described by Labra Gayo et al. (2001). InterpreterLib uses the Haskell type class system extensively to simplify chores such as injecting and projecting values into and out of lifted value domains, and similarly for projecting and injecting syntactic functors to and from lifted syntactic domains.

In InterpreterLib, algebras are explicitly passed to the traversal combinators. This facilitates one of the principal goals of InterpreterLib, which is to allow reuse and combination of multiple different semantic algebras for a single language construct. Algebras are first-class objects, and InterpreterLib provides a wide variety of operators to support the combination and modification of algebras.

3. Example Semantics

Constructing a complete semantics using the modular monadic approach involves defining: i) a set of functors representing syntax; ii) a set of semantic algebras; iii) a representation for a combined value space; and iv) a monad transformer providing the necessary computational features. In this section, we demonstrate this process to construct a type checker and interpreter using InterpreterLib.

3.1 Abstract Syntax

In InterpreterLib, we define abstract syntax in a simple term specification language. An application called AlgebraCompiler (AlgC) processes this specification and generates Haskell datatype definitions and a significant amount of boilerplate, such as `Functor` instances, needed to integrate the types into the InterpreterLib framework. InterpreterLib relies heavily on generic programming techniques, allowing the automation of this boilerplate generation.

Figure 1 gives example specifications for arithmetic, booleans (including if expressions), products, and the simply typed lambda calculus. Recursive subterms are indicated by asterisks. These “holes” may be filled with anything: i) other terms, when constructing ASTs; ii) values, during evaluation; or iii) types during type checking.

```
signature Arith
    = Int Int
    | Add * *
```

(a) Arithmetic Terms

```
signature Booleans
    = Tru
    | Fls
    | If * * *
```

(b) Boolean Terms

```
signature Products
    = Pair * *
    | PrjL *
    | PrjR *
```

(c) Products Terms

```
signature TLC ty
    = Var String
    | Lambda (String, ty) *
    | App * *
```

(d) Lambda-Calculus Terms

Figure 1. Syntactic Declarations

InterpreterLib defines an infix type constructor, `:$:`, which sums two functors together to yield a single functor. It is synonymous with `FSum`. This type constructor is used to sum together all elements in our syntactic domain into a single term space `LangS`. The `FUnit` type serves a purpose similar to that of the `()` type for the `Either` of the `SubType` class; it terminates recursion. We apply `Fix` to this and supply a type space `Ty` to get a specific syntactic domain `Lang`.

```
type LangS ty = ( Arith :$: Booleans :$:
                  Products :$: TLC ty :$: FUnit )
type Lang = Fix (LangS Ty)
```

AlgC processes the language signatures and generates a function for each constructor that can be used to generate the corresponding term. The name of each function is the constructor name prepended with `mk`. The following examples demonstrate how to construct terms using these functions.

```
term :: (SubFunctor Booleans sum, SubFunctor Arith sum,
         SubFunctor Products sum) ⇒ Fix sum
term = mkIf (mkPrjR (mkPair mkTru mkFls))
            (mkInt 1) (mkInt 2)
```

This term could be of type `Lang` or any other fixed point sum of various syntactic functors, as the SubFunctor constraint indicates. In section 5.1, `LangS` is extended to include `Quote` and `UnQuote` constructs. The `mk*` functions generated above can be reused in that language without modification.

3.2 Types and Values

There are four types of values in our target language: integers, booleans, pairs, and functions. The `Ty` datatype defines a value domain for the type checking algebra. To simplify the presentation, this type is defined recursively rather than using an `Either` type.

```
data Ty = TyInt
        | TyBool
        | Ty :*: Ty
        | Ty :→ Ty
```

Defining the semantic domain for interpretation for these types is more complex. For the non-recursive types we simply use Haskell's `Int` and `Bool` types. We define the other two types non-recursively, just as we defined data structures for abstract syntax in section 2.2.2. Instances of `Functor` must also be defined for these, but for brevity we do not include them here.

```
newtype Prod x = Prod { unProd :: (x,x) }
newtype Exp m x = Exp (x → m x)
```

We now combine our four individual semantic domains into a single value space.

```
type ValueS ty m = Int :+: Bool :+: Exp m :$: Sum :$. Prod
type Value ty m = Fix (ValueS ty m)
```

Notice that `:$:` expects its parameters to be type constructors, but `Int` and `Bool` simple types. InterpreterLib provides `:+:`, which is identical to `:$:`, except that it turns its left-hand argument into a constant functor. The `:$.` type constructor is identical to `:$:`, except that it automatically adds `FUnit` to the end of the sum. Instances of `SubType` are predefined for such a datatype.

3.3 Environment Utilities

We define the following environment for use by both type checking and interpretation algebras, as well as functions for access and modifying environments. An environment of `vs` is a list of `(String,v)` pairs. The function `lookupEnv` takes in a `String` variable name and yields a computation of of that variable's value (or type). The function `withEnv` executes a computation with a new environment.

```
type Env v = [(String, v)]

lookupEnv :: MonadReader (Env v) m ⇒ String → m v
withEnv :: MonadReader (Env v) m ⇒ (Env v) → m v → m v
```

3.4 Type Checking

Figure 2 contains the type checking algebras for the language. Following the pattern described in section 2.2.2, these are single-level definitions. Each algebra expects that the catamorphism has replaced the subterm in each syntactic functor with an `Reader` monad computation that yields the type of that subterm.

The algebras from `Arith`, `Booleans`, and `Products` don't use the effects of the `Reader` monad. They simply force each monadic computation in the subterms, then check to see if the returned types are correct. These algebras use the `fail` function, which is defined for all monads to represent an exception. The `algTyTLC` for the lambda calculus uses the `Reader` monad to propagate a type context within the body of the lambda.

A type checker algebra for the entire language is constructed by summing together the type checking algebras for each construct in the language, and applying `cata` to the result. The infix operator `@+@` is synonymous with `sumAlg`.

```
toAlg = algTyArith @+@ algTyBooleans @+@
        algTyProducts @+@ algTyTLC @+@ funitAlg

typeOf = cata toAlg
```

Finally, we use `runReader` to execute the type checker, providing it with an initial, empty environment.

```
runTypeOf :: Lang → Ty
runTypeOf t = runReader (typeOf t) []
```

3.5 Interpretation

The individual evaluation algebras for each construct in the language are shown in figure 3. The structure of these denotations resembles that of the type checking algebras, but denote to dynamic values rather than static types.

The `+` operator is overloaded, so we must ascribe it to a specific type, allowing Haskell to use the correct `SubType` instance to project `x` and `y` to integers and inject the result back into the value space.

An interpretation algebra for the entire language is constructed by summing together the algebras for each family of language constructs the same way we sum together terms to make the term space. Applying `cata` to the interpretation algebra converts the algebra into a function that takes in a term in the language and pushes interpretation into all the subterms.

```
evalAlg = algArith @+@ algBooleans @+@
          algProducts @+@ algTLC @+@ funitAlg

eval = cata evalAlg
```

The result of `eval` is monadic, so we need to apply `runReader` to it. Our first attempt is the following:

```
runEval t = runReader (eval t) []
```

But what is the type of this function? `Value` is parameterized over a monad, `Env` is parameterized over the value space and `MonadReader` is parameterized over the environment. This leads to the following recursive type in Haskell:

```
runEval :: Lang →
  Value Ty (Reader (Env (Value Ty (Reader (Env ... )))))
```

We solve this problem using monadbuilder, a preprocessor included in InterpreterLib. We define a monad, `EnvMonad` for monadbuilder as follows:

```
algTyArith = mkAlg phi where
    phi (Int _) = return TyInt

    phi (Add x y) = do
      tyx ← x
      when (tyx /= TyInt) (fail "Type␣error␣in␣Add")
      tyy ← y
      when (tyy /= TyInt) (fail "Type␣error␣in␣Add")
      return TyInt
```

(a) Arithmetic Type Checking

```
algTyBooleans = mkAlg phi where
    phi Tru = return TyBool
    phi Fls = return TyBool
    phi (If cond tru fls) = do
      ty ← cond
      when (ty /= TyBool) (fail "Type␣error␣in␣If")
      ty1 ← tru
      ty2 ← fls
      when (ty1 /= ty2) (fail "Type␣error␣in␣If")
      return ty1
```

(b) Boolean Type Checking

```
algTyProducts = mkAlg phi where
    phi (Pair x y) = do
      xty ← x
      yty ← y
      return (xty :*: yty)

    phi t@(PrjL x) = do
      tyx ← x
      case tyx of
        (tyL :*: _) → return tyL
        _ → fail ("Type␣error␣in␣PrjL")
```

(c) Products Type Checking

```
algTyTLC = mkAlg phi where
   phi (Var x) = lookupEnv x
   phi (Lambda (x, ty) body) = do
      env ← get
      bty ← withEnv ((x,ty):env) body
      return (ty :→ bty)
   phi (App f x) = do
     tyf ← f
     tyx ← x
     case tyf of
       (ty1 :→ ty2) | tyx == ty1 → return ty2
       _ → fail ("Type␣error␣in␣App")
```

(d) Lambda-Calculus Type Checking

Figure 2. Type Checking Algebras

```
algArith :: (SubType Int v, Monad m) ⇒
             Algebra Arith (m v)
algArith = mkAlg phi where
    phi (Int x) = returnInj x
    phi (Add x y) = do
      xv ← x
      yv ← y
      returnInj (x + y :: Int)
```

(a) Arithmetic Interpretation

```
algBooleans :: (SubType Bool v, Monad m) ⇒
               Algebra Booleans (m v)
algBooleans = mkAlg phi where
    phi Tru = returnInj True
    phi Fls = returnInj False
    phi (If cond tru fls) = do
                  c ← checkTypeM cond
                  if c then tru else fls
```

(b) Boolean Interpretation

```
algProducts ::
  (Monad m, SubType (Prod v) v) ⇒
  Algebra Products (m v)
algProducts = mkAlg phi where
    phi (Pair x y) = do
       xv ← x
       yv ← y
       returnInj (Prod (xv,yv))

    phi (PrjL x) = do
      (Prod xv) ← checkTypeM
      return (fst xv)
```

(c) Products Interpretation

```
algTLC :: (MonadReader (Env v) m,
           SubType (Exp m v) v) ⇒
           Algebra (TLC ty) (m v)
algTLC = mkAlg phi where
    phi (Var x) = lookupEnv x
    phi (Lambda (x, _) body) = do
      env ← getEnv
      returnInj
        (Exp (λv → withEnv ((x, v):env) body))
    phi (App f x) = do
      Exp f' ← checkTypeM f
      x' ← x
      f' x'
```

(d) Lambda-Calculus Interpretation

Figure 3. Interpretation Algebras

```
monad EnvMonad ty = Reader (Env (Value ty (EnvMonad ty)))
```

The definition of EnvMonad, including a runEnvMonad function, can be derived mechanically in InterpreterLib. Fortunately, monadbuilder generates the necessary definitions from this simple one-line definition. We can now define runEval using this EnvMonad:

```
runEval :: Lang → Value Ty EnvMonad
runEval t = runEnvMonad (eval t) []
```

4. Algebra Combinators

The MMS framework described in section 2 simplifies the combination of semantic algebras defined for a common language processing task for a varied collection of language constructs. The examples in section 3 showed how to do this for a collection of type checking and interpretation algebras. An orthogonal form of algebra composition combines two semantic algebras defined over the same set of constructs but representing different language processing tasks. The standard MMS framework provides little support for this form of composition, but can be extended as shown below for two specific use cases.

4.1 Switch Algebra

InterpreterLib uses a catamorphism combinator to combine semantic algebras. As a consequence of using this combinator, the programmer specifying the algebra relinquishes control of the traversal of the AST. This becomes an obstacle when the algebra needs to control the traversal and apply a different denotation based on some contextual information. The switchAlg algebra combinator allows algebras to control a catamorphic term traversal.

A switch algebra allows the programmer to select between algebras based on a local contextual switch, as well as manipulate that context. The following is an example of an algebra that, based on the switch, chooses whether to apply alg0, alg1 or alg2.

```
data Switch = Zero | One | Two

choiceAlg :: (Functor f) ⇒ Switch → Algebra f a
choiceAlg sw = mkAlg phi where
   phi t = case sw of
      Zero → apply alg0 t
      One → apply alg1 t
      Two → apply alg2 t
```

We can then build an algebra from choiceAlg using a function defined in InterpreterLib called switchAlg. We use a monad called MonadSwitchAlgebra to manage the local switch. It is identical to MonadReader, except that it can be used alongside MonadReader, which we will do later. The type signature for switchAlg is shown below; the details of its implementation are elided.

```
switchAlg :: (MonadSwitchAlgebra sw m, Functor f) ⇒
             (sw → Algebra f (m a)) → Algebra f (m a)

alg :: (Functor f, MonadSwitchAlgebra Switch m) ⇒
       Algebra f (m a)
alg = switchAlg choiceAlg
```

This algebra, alg, is the top-level algebra for the language. For this algebra to be interesting, the algebras alg0, alg1 and alg2 should modify the switch. InterpreterLib provides a localSwitch function, which takes in a function that modifies the switch and evaluates a computation with the modified switch.

```
localSwitch :: (MonadSwitchAlgebra r m) ⇒
               (r → r) → m a → m a
```

Applying cata to alg yields an interpreter. To run the interpreter we must use runSwitchAlgebra and provide it with an initial switch. This is identical to runReader of MonadReader.

```
eval :: (Functor f, MonadSwitchAlgebra Switch m) ⇒
        Fix f → m a
eval = cata alg

runEval t = runSwitchAlgebra (eval t) Zero
```

When eval is applied to a term, the catamorphism uses fmap, as defined in instances of Functor, to push alg into all recursive subterms of that term. But each application of alg translates into an application of either alg0, alg1 or alg2, depending on the local context of the MonadSwitchAlgebra.

4.2 Sequence Algebra

The sequenceAlg algebra combinator allows for one algebra to access the intermediate results of another. Just as switchAlg used a monadic feature (the switch environment) to recover some control of the traversal from the catamorphism, the sequence combinator uses monadic features to obtain the persistence of intermediate results even though the catamorphism would otherwise not make those results available. This monadic features of the combinator are again isomorphic to other monads, but the details are too involved to merit a discussion here. However, the behavior of the combinator is quite intuitive.

A sequence algebra allows the programmer to use the results of one algebra in the definition of another. For example, if an algebra for compilation were to rely upon type information to make optimization decisions, then the the type algebra and the compilation algebra could be sequenced so that the type information is available to the compilation algebra.

Within InterpreterLib, the notion that one algebra is dependent upon another is captured as an *indexed algebra*. Thus, the sequence algebra combinator combines one algebra with another algebra indexed by the carrier of the first algebra.

```
newtype IdxAlgebra f a b =
  IdxAlgebra { idxApply :: a → f a → Algebra f b }

seqMAlg :: MonadSequenceAlgebra f a b m ⇒
  Algebra f (m a) → IdxAlgebra f a b → Algebra f (m b)
```

A crucial aspect of the sequence algebra combinator is that it respects the encapsulation provided by the monadic carrier of the first algebra. Note that the monad of the first algebra's carrier is still present in the carrier of the result algebra. If the carrier of the first algebra were not monadic, then the sequence algebra combinator would be rather trivial to write, but the type of traversals the algebra could represent would be severely limited. The sequence algebra combinator makes the results of the monadic computation available to the second algebra without violating the encapsulation of effects. Indeed, it does so by introducing another monadic layer, as indicated by MonadSequenceAlgebra. Since features of this monad are only used to achieve the sequencing of the algebras and not meant to provide an interface to be used by the programmer, the only necessary interface is the run function runSequenceAlgebraT.

5. Applications

The switchAlg and sequenceAlg algebra combinators provide a generic means of combining distinct algebras over the the same term space. We present an example of using switchAlg to define a generic semantics for metaprogramming, and an example using seqAlg to do type-directed synthesis.

5.1 Metaprogramming

As an application of the switchAlg, we can define a generic semantics for quasiquote-style metaprogramming. To add metaprogramming features to the language, we extend the language with quasiquote syntax and add syntactic terms to the value space. The

```
signature Quotes
   = Quote *
   | UnQuote *

class Reify val syn where reify :: val → syn

type ML lang = Fix (Quotes :$: lang)

newtype DynValue a = DV { unDV :: a }

instance Reify (DynValue l) l where
  reify (DV x) = x
```

Figure 4. Types and instances supporting metaprogramming

original algebras are combined with algebras to handle quasiquote semantics, yielding a complete interpreter for the language.

We define a generic framework for adding metaprogramming capabilities to any arbitrary language written in InterpreterLib. We provide the following to facilitate this task.

- Quote and UnQuote syntax.
- A `Reify` class for converting values into syntax.
- An algebra, `reifyAlg`, used to evaluate the inside of a quoted term by simply injecting the term into the value space. This algebra is extremely modular, as it will work for any term space and any value space by using instances of `Reify`.
- A new value, `DynValue`, for adding syntax to any value space, as well as an instance of `Reify` for `DynValue`.
- Utilities for building a switch algebra, including a monad called `MonadSwitchAlgebra` that manages the switch. In this case, the the switch is an integer representing the quote level.
- The `addMeta` function, which is parameterized over en evaluation algebra for the original language and yields an algebra for the extended metaprogramming language.
- Semantics for `Quote` and `UnQuote`, embedded in the `addMeta` function

The following rules describe how to evaluate a term in the metaprogramming language given i) the current quote level; ii) an evaluation algebra, `evalAlg`, for our original language without quote syntax; and iii) the `reifyAlg` function defined above.

- Quote Level = 0
  - `Quote t` - Evaluate `t` with incremented quote level
  - `UnQuote t` - Error
  - `t` - Evaluate `t` using `evalAlg`
- Quote Level = 1
  - `Quote t` - Evaluate `t` with incremented quote level, and put the quote back on
  - `UnQuote t` - Evaluate `t` with a decremented quote level
  - `t` - Evaluate `t` using `reifyAlg`
- Quote Level > 1
  - `Quote t` - Evaluate `t` with incremented quote level, and put the quote back on
  - `UnQuote t` - Evaluate `t` with a decremented quote level, and put the unquote back on
  - `t` - Evaluate `t` using `reifyAlg`

These rules are realized by the following `addMeta` function that takes in an evaluation algebra for the original language, `evalAlg`, and yields an algebra for the metaprogramming language.

```
addMeta :: forall lang m vspace ∘ (SwitchAlgebra Int m,
  Traversable lang, SubType (DynValue (ML lang)) vspace,
  Reify vspace (ML lang), CompositeFunctor lang)
  ⇒ Algebra lang (m vspace) →
     Algebra (Quotes :$: lang) (m vspace)

addMeta evalAlg = switchAlg phiSwitch where
  phiSwitch 0 = (mkAlg phiQuoteZero) @+@ evalAlg
  phiSwitch 1 = (mkAlg phiQuoteOne) @+@ reifyAlg
  phiSwitch n = (mkAlg phiQuotePos) @+@ reifyAlg

  phiQuoteZero (Quote t) = localSwitch succ t
  phiQuoteZero (UnQuote t) = fail "Top-level␣UnQuote"

  phiQuoteOne (Quote t) = do
    DV (t' :: ML lang) ← checkTypeM (localSwitch succ t)
    returnInj (DV  (mkQuote t'))
  phiQuoteOne (UnQuote t) = localSwitch pred t

  phiQuotePos (Quote t) = do
    DV (t' :: ML lang) ← checkTypeM (localSwitch succ t)
    returnInj (DV  (mkQuote t'))
  phiQuotePos (UnQuote t) = do
    DV (t' :: ML lang) ← checkTypeM (localSwitch pred t)
    returnInj (DV (mkUnQuote t'))

  reifyAlg = mkAlg phi where
    phi t = do
      v ← unwrapMonad (traverse (WrapMonad ∘ f) t)
      returnInj (DV (inn (FSum (Right v)) :: ML lang))
    f x = x ≫= return ∘ reify
```

The `reifyAlg` algebra converts any syntactic term into a semantic value by injecting it into the value space. The function works by traversing all subterms of `t`, applying `reify` to each one, and then injecting the entire result into the value space as a `DynValue`. All instances of `Traversable` are automatically generated by AlgC.

This technique maximizes code reuse, requiring that the user only define an instance of `Reify` for each element in the value space and trivially extend the syntactic and semantic domains.

5.1.1 Reify Class

Each instance of `Reify` defines how to turn certain semantic values back into syntactic terms. For example, the following instance defines reification from integers in the value space to an `Arith` term in the term space. Like most instances of `Reify`, this one is simple. To turn an integer into syntax and inject it into the term space, we simply apply `mkInt` to it. This instance will work for any term space that includes `Arith` and any value space that includes `Int`.

```
instance SubFunctor Arith sum
    ⇒ Reify Int (Fix sum) where
  reify = mkInt


instance (SubFunctor Products sum, Reify a (Fix sum))
    ⇒ Reify (Prod a) (Fix sum) where
  reify (Prod (x,y)) = mkPair (reify x) (reify y)
```

Other instances of `Reify` are just as easy to write, except for `Exp`. We omit the instance of `Reify` for `Exp` because it over-complicates our example, requiring a `State` monad for the generation of new variable names. However, this is a solved problem (Sheard 1997), which can be incorporated into our examples.

5.1.2 Putting it Together

As mentioned earlier, extending the value space with syntax is trivial. Notice that this value space is parameterized over the term space, `lang`, so it can be used with any syntactic domain.

```
type MetaValS lang ty m = DynValue lang :+: ValueS ty m
type MetaVal lang ty m = Fix (MetaValS lang ty m)
```

We use monadbuilder to build `EnvSwitchMonad` in terms of `Reader` and `SwitchAlgebra` from the following definition.

```
monad EnvSwitchMonad lang ty =
  Reader (Env (MetaVal lang ty (EnvSwitchMonad lang ty)))
  ∘ SwitchAlgebra Int
```

Now we define our specific term space and evaluation algebra for our metaprogramming language, followed by a "run" function to evaluate terms with an initial environment and initial quote level.

```
type MetaLang = ML (LangS Ty)

metaEval = cata (addMeta evalAlg)

runMetaEval :: MetaLang →
  MetaVal MetaLang Ty (EnvSwitchMonad MetaLang Ty)
runMetaEval t = runEnvSwitchMonad (metaEval t) [] 0
```

5.2 Type-directed synthesis

The metaprogramming semantics presented above uses a `switchAlg` to combine multiple algebras for the same term and value space. Only one of these algebras will be applied to each term, depending on the quote level.

In this example we demonstrate the pattern of composition exhibited by the `seqAlg`, where an algebra and an indexed algebra are combined and *both* will be applied at each term with the result of the first algebra passed to the second. The motivation for this example is a compiler which synthesizes VHDL netlists from a simple functional language. The compilation target uses very low-level representations of data bitvectors, and many of operations require precise bitvector indexing. The information necessary to do this indexing can be extracted from the types of the expression.

We focus on the `Products` construct in this example. The size of a bitvector representing a product will vary depending on the size of the component values. We treat a product of two bitvectors as the concatenation of those component bitvectors. In the following code, the `splice` and `sliceLeft` functions calculate the appropriate indices based on the types.

The types can be manually propagated through the compiler by returning tuples of types and netlists, rather than just netlists.

```
netlist (Pair x y) = do
   (xty,xnl) ← checkTypeM x
   (yty,ynl) ← checkTypeM y
   znl ← splice (xty :+: yty) xnl ynl
   returnInj (xty :*: yty,znl)

netlist (PrjL p) = do
   (pty@(xty :+: _), pnl) ← checkTypeM p
   znl ← sliceLeft xty pty pnl
   returnInj (xty,znl)
```

The problem with this approach is that it requires the programmer to re-implement the type rules from figure 2(c) within the compiler. For the isolated `netlist` algebra, this requires minimal extra effort. However, for a more complex language, with a large number of different syntactic constructs, the duplication is significant.

This problem can be avoided through the use of the `seqAlg` combinator. Rather than pass around tuples of values and netlists, `seqAlg` handles this automatically. The compilation function is an indexed algebra, as shown below. The first argument is the result of the type checker algebra for that term. The second argument is the situated results of the type checker algebra for the subterms.

```
netlist zty (Pair xty yty) = mkAlg phi
  where phi (Pair x y) = do
          xnl ← checkTypeM x
          ynl ← checkTypeM y
          znl ← splice zty xnl ynl
          returnInj znl

netlist xty (PrjL pty) = mkAlg phi
  where phi (PrjL p) =  do
           pnl ← checkTypeM p
           znl ← sliceLeft xty pty pnl
           returnInj znl
```

The type checking algebra from figure 2(c) and the netlist algebra are combined using `seqAlg` into a single algebra making it trivial to combine multiple algebras sequentially.

```
synthProd :: (Monad t,SubType NetList v,
     SequenceAlgebra Products Ty (t v) m) ⇒
     Algebra Products (m (t v))
synthProd = seqAlg algTyProducts (mkIdxAlg netlist)
```

6. Related Work

InterpreterLib is based on two main bodies of research, modular monadic semantics and generic programming.

MMS Moggi (1990, 1991) described the use of monads to structure computational effects in language denotations. Steele (1994) used monads to construct executable interpreters, and Espinosa (1995) and Liang et al. (1995) used monad transformers to simplify the construction of a monad with the necessary computational features. Liang and Hudak (1996) and Harrison and Kamin (2000) applied the framework to the construction and verification of compilers.

Duponcheel (1995) and Hutton (1998) used a non-recursive form to represent abstract syntax, allowing syntactic domains to be compositional. Labra Gayo et al. (2001) defined the LPS library for writing MMS specifications and defined semantics for a wide variety of constructs using the library.

Generic Programming Lämmel and Visser (2002) use combinators for representing common traversal patterns. This work has been applied in the development of the Strafunski (Lämmel and Visser 2003) Haskell library, in addition to Stratego (Visser 2001), a domain-specific language for program transformations. Many of the combinators introduced in this work have also been implemented in InterpreterLib. Most importantly, they define a `letTU` combinator which performs a similar function to the `seqAlg` combinator introduced in this paper. A major difference between these two is that `letTU` appears to address non-monadic computations, while `seqAlg` is expressly designed to handle the difficulties involved in preserving monadic effects across a sequence of algebra applications. The `seqAlg` is a monadic form of the *zygomorphism* combinator (Malcolm 1990).

PolyP (Jansson and Jeuring 1997) uses non-recursive functor representations of datatypes to define a variety of generic programs. PolyP uses a preprocessor to generate instances of Haskell type classes for data structures; this is similar to the use of AlgC to generate instances in InterpreterLib. An alternative approach published in a series of "Scrap Your Boilerplate" papers utilizes a dynamic cast operator to embed generic programming combinators within Haskell, eliminating the need for non-recursive representations (Lammel and Jones 2004; Lämmel and Peyton Jones 2003).

7. Conclusions and Future Work

Intuitive algebra combinators, like sequence and switch, facilitate reuse of semantics algebras without compromising the benefits of modular monadic semantics. Previous work establishes the effectiveness of traditional MMS in defining individual interpreters in a modular fashion. In this work we have shown how algebra combinators extend this approach to defining compositions of different interpreters to accomplish traditional language processing tasks such as type checking and synthesis as well as extending a language to support meta-programming.

The InterpreterLib implementation of MMS and algebra combinators is actively used in the development of analysis tools for the Rosetta (Alexander 2006) system-level specification language. The heterogeneous nature of system-level design mandates the integration of multiple, concurrent design domains when making design decisions. InterpreterLib provides exactly this capability allowing definition and composition of interpreters over heterogeneous Rosetta domains.

Current plans to extend the library include expanding the collection of algebra combinators to capture further patterns of traversal. Also, the library uses some advanced features of Haskell's type class system to enforce genericity; oftentimes maintaining the balance between generality and programmability is a delicate task. We plan to continue researching ways to ease this burden.

References

Perry Alexander. *System-Level Design with Rosetta*. Morgan Kaufmann Publishers, Inc., 2006.

Luc Duponcheel. Using catamorphisms, subtypes and monad transformers for writing modular functional interpreters., 1995.

David Espinosa. *Semantic Lego*. PhD thesis, Columbia University, 1995.

William L. Harrison and Samuel N. Kamin. Metacomputation-based compiler architecture. In *Mathematics of Program Construction*, pages 213–229, 2000.

Graham Hutton. Fold and unfold for program semantics. In *Proceedings 3rd ACM SIGPLAN Int. Conf. on Functional Programming, ICFP'98, Baltimore, MD, USA, 26–29 Sept. 1998*, volume 34, pages 280–288. ACM Press, New York, 1998.

Patrik Jansson and Johan Jeuring. PolyP—A polytypic programming language extension. In *Conf. Record 24th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL'97, Paris, France, 15–17 Jan 1997*, pages 470–482. ACM Press, New York, 1997.

Mark P. Jones and Luc Duponcheel. Composing monads. Research report YALEU/DCS/RR-1004, Yale University, Yale University, New Haven, Connecticut, Dec 1993.

Simon Peyton Jones, editor. *Haskell 98 Language and Libraries – The Revised Report*. Cambridge University Press, Cambridge, England, 2003.

Simon Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in haskell. Tutorial at Marktoberdorf Summer School, 2000., March 2002.

Jose E. Labra Gayo, M. C. Luengo Díez, J. M. Cueva Lovelle, and A. Cernuda del Río. LPS: A language prototyping system using modular monadic semantics. In M. van der Brand and D. Parigot, editors, *Proceedings 1st Workshop on Language Descriptions, Tools and Applications, LDTA'01, Genova, Italy, 7 Apr 2001*, volume 44(2). Elsevier, Amsterdam, 2001.

Ralf Lammel and Simon Peyton Jones. Scrap more boilerplate: reflection, zips, and generalised casts. *SIGPLAN Not.*, 39(9): 244–255, 2004. ISSN 0362-1340.

Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. *ACM SIGPLAN Notices*, 38(3):26–37, 2003. Proc. of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003).

Ralf Lämmel and Joost Visser. Design Patterns for Functional Strategic Programming. In *Proc. of Third ACM SIGPLAN Workshop on Rule-Based Programming RULE'02*, Pittsburgh, USA, 5 2002. ACM Press. 14 pages.

Ralf Lämmel and Joost Visser. A strafunski application letter. In Verónica Dahl and Philip Wadler, editors, *PADL*, volume 2562 of *Lecture Notes in Computer Science*, pages 357–375. Springer, 2003. ISBN 3-540-00389-4.

Sheng Liang and Paul Hudak. Modular denotational semantics for compiler construction. In *Programming Languages and Systems – ESOP'96, Proc. 6th European Symposium on Programming, Linköping*, volume 1058, pages 219–234. Springer-Verlag, 1996.

Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In ACM, editor, *Conference record of POPL '95, 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: San Francisco, California, January 22–25, 1995*, pages 333–343, New York, NY, USA, 1995. ACM Press. ISBN 0-89791-692-1.

Grant Malcolm. *Algebraic Data Types and Program Transformation*. PhD thesis, Groningen University, 1990.

Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In J. Hughes, editor, *Proceedings 5th ACM Conf. on Functional Programming Languages and Computer Architecture, FPCA'91, Cambridge, MA, USA, 26–30 Aug 1991*, volume 523, pages 124–144. Springer-Verlag, Berlin, 1991.

Eugenio Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Dept. of Computer Science, Edinburgh Univ., 1990.

Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

Tim Sheard. A type-directed, on-line partial evaluator for a polymorphic language. In *Proceedings of the ACM-SIGPLAN Symposium on Partial Evaluation and semantic based program manipulations PEPM'97, Amsterdam*, pages 22–35. ACM, 1997.

Guy L. Steele. Building interpreters by composing monads. In ACM, editor, *Conference record of POPL '94, 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: Portland, Oregon, January 17–21, 1994*, pages 472–492, New York, NY, USA, 1994. ACM Press. ISBN 0-89791-636-0.

Eelco Visser. Stratego: A language for program transformation based on rewriting strategies. *Lecture Notes in Computer Science*, 2051:357+, 2001.

P. Wadler. Monads for functional programming. In M. Broy, editor, *Program Design Calculi: Proceedings of the 1992 Marktoberdorf International Summer School*. Springer-Verlag, 1993.

P. L. Wadler. Comprehending monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming, Nice*, pages 61–78, New York, NY, 1990. ACM.