Constructing Language Processors with Algebra Combinators

Nicolas Frisby\textsuperscript{a} Garrin Kimmell\textsuperscript{a} Philip Weaver\textsuperscript{b} Perry Alexander\textsuperscript{a}

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

\textsuperscript{b}Signalicorp
3034 NE Tillamook St
Portland, OR 97212
pweaver@signalicorp.com

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. In addition to composing semantics for various language constructs, we can use MMS to compose various differing semantics for the same language constructs. This capability allows us to compose and reuse orthogonal language tasks such as typechecking and compilation. 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.

1 Introduction

Modular monadic semantics (MMS) is a technique 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 increased 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, evaluation, or compilation. We identify this pattern of composition as combining common semantics across varied constructs. In this article 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. 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 traversal in a context-dependent manner, such as choosing between two denotations to apply at a given point. Secondly, MMS uses monads to structure computational effects for orthogonal constructs, but when combining two denotations where the result of the first is used as an input to the second, the monadic effects of the first must be maintained 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 also the solution to those limitations. The key idea is to capture the logic for controlling traversal or maintaining effects as yet another monadic computational effect. 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 key contribution of this article is the extension of the MMS framework with algebra combinators. Algebra combinators, as we define in this paper, allow the composition of multiple semantics for the same construct. We identify and describe two specific algebra combinators, which demonstrate two different types of semantic combination:

- **switchAlgebra** that combines two semantic algebras with a common semantic domain into a single algebra where context dictates which algebra to apply to a term.
• **sequenceAlgebra** 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 combinators with a collection of small examples typical of common language processing tasks.

Throughout the article we present our examples in Haskell [Peyton Jones, 2003]. In some instances, we concisely deviate from idiomatic Haskell to improve the clarity or accessibility of the exposition. The implementation makes extensive use of Haskell’s type class system. Although we have attempted to structure this article so that general concepts of algebra combinators accessible to a reader with familiarity with Haskell or a similar statically-typed pure functional language, the details of the implementation often require a deeper understanding of the type class system. The full source to the code presented in this article, as well as the underlying InterpreterLib framework, is available online.¹

This article is an extended version of an earlier paper [Weaver et al., 2007]. The principal extension is a description of the implementation of the two algebra combinators in Section 6. In addition, Section 5 includes an example of using the sequenceAlgebra combinator to define a generic paramorphism combinator. Finally, Section 3.5 describes the implementation of a recursively-defined monad as the fixed-point of a type constructor.

## 2 Modular Monadic Semantics

Modular monadic semantic definitions consist of a collection of **semantic building blocks**. Each building block defines a semantic algebra mapping a particular syntactic form to a semantic domain. Individual semantic algebras can be combined into composite semantic algebras. Composite algebras contrast with the semantic building blocks in that they do not define new semantics, but rather are constructed by combining existing semantic building blocks in a regular manner. As a result of using the MMS framework, there is minimal effort in constructing new combinations of semantic algebras for new combinators of syntactic forms.

The machinery necessary to support this composition involves two main components: monads and syntactic functors. First, **monads** structure the semantic domains. Monads serve to capture the computational effects of the semantic algebras. We treat monads as abstract data types when defining the semantic algebras. This allows delaying the construction of a specific monad until the

algebras are combined; *monad transformers* make this construction straightforward. Second, *syntactic functors* allow the syntax and semantics of the language constructs to be independently specified. Using techniques from generic programming [Jansson and Jeuring, 1997], syntactic functors for orthogonal constructs can be combined into a complete syntactic signature for a language, and likewise semantic algebras for independent constructs can be combined into a complete language denotation.

The foundations of MMS are well represented in the literature. We present an overview of the topic, however the overview is 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, Gayo et al., 2001, Harrison and Kamin, 2000, Hutton, 1998, Liang and Hudak, 1996, Liang et al., 1995, Moggi, 1990, 1991, Steele, 1994, Wadler, 1990].

2.1 Monads

The monad is a concept from category theory and was initially used to give a formal semantics to computational side effects in programming language denotations by Moggi [1990, 1991]. Wadler [1995, 1990] detailed the usefulness of monads for programming effectful computation within a pure functional language such as Haskell. Using monads simplifies the programming of effects while maintaining the referentially-transparent reasoning that is a benefit of pure languages. This has proved useful in capturing the semantics of a wide range of common operations in Haskell having effectful behavior [Peyton Jones, 2002].

Monads are formulated in Haskell as a Haskell type class.

```haskell
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 of 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 a monadic computation; and `>>=` (pronounced “bind”), that allows sequencing multiple computations with effects threaded through the computations. The `>>=` function satisfies an associativity law, \(^2\) while `return` is the right- and left-unit of `>>=`.

\(^2\) The Kleisli composition operator `f >>= g ≡ λ x → f x >>= g` must be associative.
The `liftM` function below is part of the standard Haskell monad libraries. It witnesses the fact that every monad is also a functor. We give the definition here because it qualifies as Haskell jargon and will appear throughout the article. We also use `liftM2` for lifting pure functions with two arguments.

\[
\text{liftM} :: (a \rightarrow b) \rightarrow m a \rightarrow m b
\]

\[
\text{liftM} f m = m >>= \lambda a \rightarrow \text{return} (f a)
\]

The `Monad` type class provides the minimal support for monadic features. However, this signature cannot introduce computational effects. To accomplish this, any given monad must include a collection of non-proper morphisms that provide access to computational effects. The set of non-proper morphisms will vary from monad to monad depending upon the effects being modeled.

As an example, consider the `Reader` monad. This monad represents computations that require a context parameter, such as the typing context in a type checker or the environment of bound variables in an evaluator. Structuring these computations monadically alleviates the need to manually thread the context through each computation. The `Reader` monad has two non-proper morphisms. The first, `ask`, gives the computation access to the environment. The second, `local`, takes a modification function and uses it to adjust the environment of a `Reader` computation. The `MonadReader` type class declares this interface, with the type constructor parameter `m` representing the monad and the type parameter `r` representing the context parameter. The functional dependency [Jones, 2000] \( m \rightarrow r \) instructs the type-checker that any monad carries exactly one type of environment.

\[
\text{class Monad m} \Rightarrow \text{MonadReader r m} | m \rightarrow r \text{ where}
\]

\[
\text{ask :: m r}
\]

\[
\text{local :: (r \rightarrow r) \rightarrow m a \rightarrow m a}
\]

The `Reader` type implements this interface. It is parametrized over any environment type; this supports using the `Reader` monad for a variety of different environment types.

\[
\text{newtype Reader r a} = \text{Reader} (r \rightarrow a)
\]

\[
\text{runReader :: Reader r a \rightarrow r \rightarrow a}
\]

\[
\text{runReader} (\text{Reader} m) = m
\]

\[
\text{instance Monad (Reader e) where}
\]

\[
\text{return x = Reader} \; (\lambda r \rightarrow x)
\]

\[
\text{Reader m >>= f = Reader} \; (\lambda r \rightarrow f \; (m \; r) \; r)
\]

3 The `Monad` class defined in the Haskell98 prelude includes a method `fail :: String \rightarrow m a`. We omit this function in the class definition above, instead using an explicit monadic effect to model failure.
instance MonadReader (Reader r) where
  ask = Reader id
  local f m = Reader (λr → runReader m (f r))

To execute a Reader computation to yield a value, apply runReader to the
computation and an initial environment. The runReader function will be used
later to execute type checkers and evaluators.

The Reader monad is one in a collection of commonly used monads. Oth-
ers include a State monad capturing state-transformation behavior, an Error
monad for propagating and catching exceptions, and a Writer monad for
logging-style, write-only behavior. Each is available in the Haskell base li-
braries as a type class for the non-proper morphisms, a type constructor sup-
porting the interface, and a run function.

Semantic algebras for various language constructs utilize different computa-
tional features. Although the type class for each monad provides an abstract
interface for using those features, it remains necessary to define a specific
monad type constructor that represents the interface. Constructing this spe-
cific type is non-trivial because given two monads m and n, the type con-
structor resulting from the composition of those two types is not necessarily
a monad [Jones and Duponcheel, 1993]. Consequently, combining semantic
algebras that utilize two different monads is a challenge that would seem to
require defining a completely new monad for each possible combination.

The problem of monad composition can be avoided by using monad transfor-
mers [Espinosa, 1995, Liang et al., 1995]. A monad transformer is a parametrized
monad that takes a second monad as a type argument. A monad transformer t,
parametrized over another monad m, has a function lift :: m a → t m a that
lifts a computation in the inner monad m into the outer monad transformer.
This interface is captured by the MonadTrans type class shown below.

class Monad m ⇒ MonadTrans t where
  lift :: m a → t m a

The Haskell libraries define transformer versions of the standard monads suit-
able for composition. Defining instances of the various monad interfaces re-
quires a pairwise construction between each monad transformer and inner
monad interface. While this could potentially lead to quadratic growth in the
number of required instance declarations, in practice most tasks require a
small set of monadic features provided by the Haskell libraries.
2.2 Modular Semantics

Monads and monad transformers provide the necessary support for combining the computational effects used by component semantic algebras within the MMS framework. Combining semantic algebras requires some additional support. First, we must be able to combine syntactic forms. For example, if one set of constructs includes arithmetic operations, such as Add and numerals, and another set includes a branching If construct and boolean literals, then the combined syntax will include all of these constructs. Second, we must be able to combine value spaces. For example, if one construct returns Ints and another returns Bools, then the combined value space includes both Ints and Bools.

2.2.1 Composite Syntax and Semantics

In Haskell, an abstract syntax tree is usually represented as a recursive algebraic data type. Consider ArithRec and BoolsRec:

```haskell
data ArithRec = Add ArithRec ArithRec | Num Int

data BoolsRec = If BoolsRec BoolsRec BoolsRec | Tru | Fls
```

The standard evaluators for these term spaces are:

```haskell
evalArithRec :: ArithRec → Int
evalArithRec (Add x y) = evalArithRec x + evalArithRec y
evalArithRec (Num i) = i

evalBoolsRec :: BoolsRec → Bool
evalBoolsRec (If cond tru fls) =
  if evalBoolsRec cond
    then evalBoolsRec tru
    else evalBoolsRec fls
evalBoolsRec Tru = True
nevalBoolsRec Fls = False
```

Combining these two functions is not possible without redefining the ArithRec and BoolsRec data types to create a single AST type that contains all constructs. This is necessary because both ArithRec and BoolsRec are recursively-defined closed types, with no possibility for extension. A second complication is that the value spaces for the two functions, Int and Bool, differ. The solution to the first problem, combining syntactic forms, will be reused as the solution to the second problem.
To allow for combining syntactic forms, the \textit{Arith} and \textit{Bools} syntactic functors represent the syntactic signature instead of the term space. Each recursive occurrence of the term type is replaced with the new type parameter \( x \).

\begin{verbatim}
data Arith x = Add x x | Num Int

data Bools x = If x x x | Tru | Fls
\end{verbatim}

We combine the syntactic functors with a functor sum operator, \textit{FSum}, and an empty sentinel functor, \textit{FVoid}.

\begin{verbatim}
data FSum f g x = L (f x) | R (g x)
data FVoid x  -- requires the EmptyDataDecl language extension
\end{verbatim}

Syntactic functors are combined to form composite signatures by summing the syntactic functors in a right-associative manner and using \textit{FVoid} to represent the empty sum. For example, the syntactic functor \textit{Composite} represents the signature of the combined example language.

\begin{verbatim}
type Composite = FSum Arith (FSum Bools FVoid)
\end{verbatim}

The \textit{SubFunctor} class allows injection into, and projection out of, a composite syntactic signature. It is a specialization to functors of the \textit{SubType} class used by Liang et al. [1995]. The instances for \textit{FSum} are shown below; there is no instance for \textit{FVoid}. The first instance corresponds to when the syntactic functor of interest, \( f \), is in the left-most position of the sum. The second instance corresponds to when \( f \) instead occurs in the rest of the sum. In this case, the instance lifts the \textit{SubFunctor} relation through the functor sum. These instances require the overlapping instances extension to GHC, but its usage is safe because there are no functional dependencies associated with the class.

\begin{verbatim}
class SubFunctor f g where
  injF :: f x \rightarrow g x
  prjF :: g x \rightarrow Maybe (f x)

instance SubFunctor f (FSum f g) where
  injF a = L a
  prjF (L a) = Just a
  prjF (R _) = Nothing

instance SubFunctor f h \Rightarrow SubFunctor f (FSum g h) where
  injF a = R (injF a)
  prjF (L a) = Nothing
  prjF (R a) = prjF a
\end{verbatim}

The last generic definition for achieving compositional syntax is the \textit{Fix} type constructor. It recovers an inductive type for the abstract syntax tree from
the syntactic signature by introducing recursion. The constructor \texttt{In} and destructor \texttt{out} witness the isomorphism between the \texttt{f (Fix f)} and \texttt{Fix f} types that results from the fixed point operator.

\begin{verbatim}
newtype Fix f = In (f (Fix f))
out :: Fix f \rightarrow f (Fix f)
out (In x) = x
\end{verbatim}

The \texttt{Language} type defines the term space by applying the \texttt{Fix} operator to the \texttt{Composite} functor, which represents the full set of constructs for the language.

\begin{verbatim}
type Language = Fix Composite
\end{verbatim}

\subsection*{2.2.2 Composite Value Spaces}

The technique for combining syntactic forms also works for combining differing value spaces, such as those of the \texttt{evalArithRec} and \texttt{evalBoolsRec} denotation functions. The first uses \texttt{Int} for its value space whereas the second uses \texttt{Bool}. The \texttt{SubFunctor} class can be used to relax such restrictions on the value space. However, the \texttt{SubFunctor} class operates on type constructors, not ground types such as \texttt{Int} and \texttt{Bool}. This mismatch is resolved using the \texttt{Const} type constructor, which promotes a ground type to a trivial functor.

\begin{verbatim}
newtype Const a x = Const a
getConst (Const x) = x
\end{verbatim}

The type constructors are no longer necessarily functors, but may be any type constructor (of kind $\ast \rightarrow \ast$). If they are functors, the library of Interpreter-Lib functor traversal combinators can be used to define functions over the resulting data type. Without the \texttt{fmap} morphism, the data type's interface is limited to injections and projections, which correspond to constructors and case expressions. These operations are enough for most semantics' use of types and values.

Using the \texttt{injF} and \texttt{prjF} functions, the \texttt{evalArithRec} and \texttt{evalBoolsRec} functions can be rewritten with a more general type. Note that the functions assume that projection will succeed or else the Haskell \texttt{error} function is invoked. In Section 3, a monadic effect will be used to model this failure.

\begin{verbatim}
evalArithSub :: SubFunctor (Const Int) v \Rightarrow ArithRec \rightarrow Fix v
evalArithSub (Add x y) =
  case (prjF (out (evalArithSub x)), prjF (out (evalArithSub y)) of
    (Just (Const a), Just (Const b)) \rightarrow In (injF (Const (a+b)))
    _ \rightarrow error "Projection_error."
  evalArithSub (Num x) = In (injF (Const x))
\end{verbatim}
evalBoolsSub :: SubFunctor (Const Bool) v ⇒ BoolsRec → Fix v
evalBoolsSub (If p t e) =
    case prjF (out (evalBoolsSub p)) of
    Just (Const b) → if b then evalBoolsSub t
                          else evalBoolsSub e
    Nothing       → error "Projection_error."
evalBoolsSub Tru = In (injF (Const True))
evalBoolsSub Fls = In (injF (Const False))

Using the SubFunctor class allows the value space for a semantics to be specified as a SubFunctor class constraint rather than a concrete type. When multiple semantics are combined, the value space for the composite semantics must satisfy the SubFunctor constraints from all component semantics. The cost of this flexibility is the additional overhead of the injection and projection of values into the combined value space.

2.2.3 Composite Semantics

The FSum and Fix type constructors allow the combination of syntax and value spaces. The final remaining issue is to combine the semantics themselves. This utilizes the composite syntax and value space technique as well as some additional machinery.

Like the original closed recursive definitions of the ArithRec and BoolsRec types, the recursively defined evalArithSub and evalBoolsSub functions are closed and cannot be directly combined. Both denotations follow the same pattern: the denotation function is recursively applied to each of the sub-terms, and then a semantic function, such as + or if, is applied to the evaluated sub-terms. Recognizing this pattern, the denotation function is written in two phases. The first handles the recursive application of the denotation to the sub-term and is implemented generically. The second captures the specific meaning of the particular denotation.

The first step, evaluating all sub-terms, is handled 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 functor f. There are fmap instances for FSum, FVoid, and Const.
instance (Functor f, Functor g) ⇒ Functor (FSum f g) where
fmap f (L x) = L (fmap f x)
fmap f (R x) = R (fmap f x)

instance Functor FVoid where
fmap _ _ = ⊥  -- safe because FVoid is an empty type

instance Functor (Const a) where
fmap f (Const a) = Const a

While the previous instances are part of the MMS framework, we also need instances of Functor for each syntactic functor. See Appendix A for a discussion of generating boilerplate.

instance Functor Arith where
fmap f (Add a b) = Add (f a) (f b)
fmap f (Num a) = Num a

instance Functor Bools where
fmap f (If p t e) = If (f p) (f t) (f e)
fmap f Tru = Tru
fmap f Fls = Fls

Functions of type type Algebra f a = f a → a, are semantic algebras mapping structures to associated values. The type constructor f is a syntactic functor, and the type a is the carrier of the algebra. In MMS, the carrier is typically the value space or a monadic computation yielding the value space.

The evalArith and evalBools functions are the semantic algebras for evaluation of the Arith and Bools syntactic functors. They define the semantics in such a way that they can be combined.

evalArith :: SubFunctor (Const Int) v ⇒ Algebra Arith (Fix v)
evalArith (Add a b) =
  case (prjF (out a), prjF (out b)) of
    (Just (Const x), Just (Const y)) → In (injF (Const (x + y)))
    _ → error "Can’t project"
evalArith (Num x) = In (injF (Const x))

evalBools :: SubFunctor (Const Bool) v ⇒ Algebra Bools (Fix v)
evalBools (If p t e) =
  case prjF (out p) of
    Just (Const b) → if b then t else e
    Nothing → error "Can’t project"
evalBools Tru = In (injF (Const True))
evalBools Fls = In (injF (Const False))
The \texttt{sumAlg} operator combines algebras just as \texttt{FSum} combines syntactic functors. This operator is defined universally for all syntactic functors. The \texttt{voidAlg} algebra serves a similar purpose to \texttt{FVoid}; it terminates a series of algebras. In fact, \texttt{voidAlg} will never be applied, as \texttt{FVoid} has no constructors. Because of this, the definition for \texttt{voidAlg} is safely left as undefined.

\begin{verbatim}
sumAlg :: Algebra f a → Algebra g a → Algebra (FSum f g) a
sumAlg fphi gphi (L x) = fphi x
sumAlg fphi gphi (R x) = gphi x

voidAlg :: Algebra FVoid a
voidAlg = ⊥
\end{verbatim}

Using \texttt{sumAlg} and \texttt{voidAlg}, \texttt{evalArith} and \texttt{evalBools} can be combined into a single semantic algebra over the composite syntactic signature.

\begin{verbatim}
evalComposite ::
    (SubFunctor (Const Bool) v, SubFunctor (Const Int) v)
⇒ Algebra Composite (Fix v)
evalComposite = evalArith 'sumAlg' evalBools 'sumAlg' voidAlg
\end{verbatim}

The catamorphism function, \texttt{cata}, extends a semantic algebra to a denotation function [Meijer et al., 1991]. It introduces recursion to a semantic algebra just as \texttt{Fix} introduces recursion to a syntactic functor. The function is parametrized over an algebra \texttt{phi} and a term \texttt{t}. The function first uses \texttt{fmap} to map itself recursively across the sub-terms of the term \texttt{t}, which have been exposed by \texttt{out}. The result is a functor value with the sub-terms replaced by corresponding carrier values. The recursion operator then applies the semantic algebra to reduce the structured value to a single carrier value.

\begin{verbatim}
cata :: Functor f ⇒ Algebra f carrier → Fix f → carrier
cata phi t = phi (fmap (cata phi) (out t))
\end{verbatim}

The \texttt{evaluate} denotation function applies the \texttt{evalComposite} semantic algebra using \texttt{cata}, reducing an AST data structure of type \texttt{Language} to a single value.

\begin{verbatim}
evaluate ::
    (SubFunctor (Const Bool) v, SubFunctor (Const Int) v)
⇒ Language → v
evaluate t = cata evalComposite t
\end{verbatim}

This completes the modular semantics framework. In summary, syntactic functors and value functors are combined using \texttt{FSum}, \texttt{FVoid}, and \texttt{SubFunctor}; computation effects are combined using monad transformers; and semantic algebras are combined using \texttt{sumAlg}. The key is defining the syntax and semantics without recursion, and then using the \texttt{Fix} type constructor for syntax and
values and the \texttt{cata} combinator for algebras to introduce recursion once extensibility is no longer required. The use of monads for modularity will be demonstrated in the next section.

3 InterpreterLib

We have implemented the MMS framework in a library that we call \textit{InterpreterLib}. The library provides a domain-specific language embedded in Haskell that supports common tasks involved in the definition of language processing systems. InterpreterLib extends MMS with algebra combinators, which allow further reuse of semantic algebras.

In the ideal usage scenario, an InterpreterLib user selects syntactic functors and associated semantic algebras from a library of reusable components in order to construct the language they desire. Syntactic functors are combined with \texttt{FSum} and \texttt{FVoid} to define the syntactic signature and then \texttt{Fix} is applied to define term space. Composite algebras are similarly combined with \texttt{sumAlg}, \texttt{voidAlg} and then \texttt{cata} is applied to define the denotation function. \texttt{SubFunctor} and monadic interface constraints found in the type context of the denotation function guide the creation of the concrete monad from the library of monad transformers and the value space from type-constructors, \texttt{FSum}, \texttt{FVoid}, and \texttt{Fix}. The final mapping from the language to the value space is the composition of the denotation function and the monad’s run function.

To demonstrate the approach, we construct a type-checker and an evaluator for a language with arithmetic, booleans, products, and the typed lambda calculus. The data types representing these syntactic functors are shown in Figure 1. The syntactic functors for arithmetic and booleans are the same as Section 2, but the semantic algebras are more sophisticated and the new syntactic functors require a monadic semantics.

3.1 Syntactic Functors

InterpreterLib’s more advanced features require syntactic functors to exhibit certain properties, implemented as type classes. The \texttt{Functor} class is the most basic property and allows extending a semantic algebra to a denotation function via \texttt{cata}. The algebra combinators require instances of both \texttt{FunctorZip}, a monoidal property, and \texttt{FunctorContext}, a data type differentiation operator [Abott et al., 2005]. Both of these classes are discussed in detail in Section 6.

Beyond instances of the functor type classes, it is useful to define convenience
functions, called smart constructors, for creating terms. Each constructor of
the syntactic functor results in a smart constructor, named by prepending \texttt{mk}
on to the constructor name. The smart constructor \texttt{mkIf}, for example, con-
structs an \texttt{If} term from three other terms, just as would the \texttt{If} constructor
in a conventional Haskell algebraic data type.

\texttt{mkIf} :: \texttt{SubFunctor Bools f} ⇒ \texttt{Fix f} → \texttt{Fix f} → \texttt{Fix f}
\texttt{mkIf} \texttt{arg0} \texttt{arg1} \texttt{arg2} = (\texttt{In} \circ \texttt{injF}) (\texttt{If} \texttt{arg0} \texttt{arg1} \texttt{arg2})

Smart constructors encapsulate the application of \texttt{SubFunctor} injection and
the \texttt{In} constructor for the fixed point. The following example demonstrates
construction of terms using smart constructors.

\texttt{term} :: (\texttt{SubFunctor Bools f}, \texttt{SubFunctor Arith f},
\texttt{SubFunctor Products f}) ⇒ \texttt{Fix f}
\texttt{term} = \texttt{mkIf} (\texttt{mkPrjR} (\texttt{mkPair} \texttt{mkFls} \texttt{mkTru})) (\texttt{mkNum} 1) (\texttt{mkNum} 2)

The \texttt{term} declaration preserves the polymorphism of smart constructors. It can
serve as a term in any language that includes the necessary syntactic functors
in the composite signature functor, as the \texttt{SubFunctor} constraints indicate.

The full language includes all the syntactic functors defined in Figure 1. We
use the functor sum operator \texttt{FSum} to extend the base syntactic functors into
a syntactic functor for the full signature, \texttt{Composite}. We apply \texttt{Fix} to define
the term space \texttt{Language}.

\texttt{type Composite ty} = \texttt{FSum Products (FSum (TLC ty) Section2.Composite)}
\texttt{type Language ty} = \texttt{Fix (Composite ty)}

With a specific term space like \texttt{Language}, we can show the result of ascribing a
monomorphic type to \texttt{term}. Ignoring the parameter of the \texttt{TLC} construct, \texttt{term}
has the following structure as an element of the Language () term space:

\[
\begin{align*}
\text{In} & \ (R \ (R \ (L \ (\text{If} \\
& \quad \ (\text{In} \ (L \ (\text{PrjR} \\
& \quad \quad \ (\text{In} \ (L \ (\text{Pair} \\
& \quad \quad \quad \ (\text{In} \ (R \ (R \ (L \ \text{Tru})))) \\
& \quad \quad \quad \quad \ (\text{In} \ (R \ (R \ (L \ \text{Fls})))))))))) \ )))) \\
& \quad \ (\text{In} \ (R \ (R \ (L \ (\text{Num} \ 1)))))) \\
& \quad \ (\text{In} \ (R \ (R \ (L \ (\text{Num} \ 2)))))) 
\end{align*}
\]

Since the smart constructors are polymorphic with respect to the syntactic functor, constructors and any term built with them can be reused without modification in any other language that includes necessary functors. In Section 5.1, for example, Composite is extended to include Quote and UnQuote constructs; term can also be used in that extended language.

### 3.2 Types and Values

We define the type-constructors in the Types module that represent the types of the target language, the value space of the type-checking denotation. We import this module qualified, thus we are free to reuse the names Int and Bool. Each occurrence of these names in other modules will be qualified.

```haskell
module Types where

import InterpreterLib
import Prelude hiding (Int, Bool)
```

There are four types: integers, booleans, products, and functions.

```haskell
data Int x = Int deriving (Eq)

mkInt :: SubFunctor Int f ⇒ Fix f
mkInt = (In ◦ injF) Int

data Bool x = Bool deriving (Eq)

mkBool :: SubFunctor Bool f ⇒ Fix f
mkBool = (In ◦ injF) Bool

data Function x = Function x x deriving (Eq)

mkFunction :: SubFunctor Function f ⇒ Fix f → Fix f → Fix f
mkFunction x y = (In ◦ injF) (Function x y)

data Prod x = Prod x x deriving (Eq)

mkProd :: SubFunctor Prod f ⇒ Fix f → Fix f → Fix f
mkProd x y = (In ◦ injF) (Prod x y)
```
These type constructors are indeed functors. However, we intentionally do not define \texttt{Functor} instances, since we are using the type constructors for extensibility of the data structure for types in the language. We do not need to utilize functor properties, since the type data structure is not manipulated beyond introduction and case expressions.

Defining the components of the value space for the evaluation denotation is similar to the type space components. For the atomic values we use the \texttt{Int} and \texttt{Bool} types from the Haskell \texttt{Prelude} module, lifted with the \texttt{Const} functor; hence those types are not shown here. For products and for functions, however, we define new type constructors, since each involves recursion.

```
module Values where

import InterpreterLib

data Prod x = Prod x x
mkProd :: SubFunctor Prod f ⇒ Fix f → Fix f → Fix f
mkProd arg0 arg1 = (In ◦ injF) (Prod arg0 arg1)

newtype Exp m x = Exp (x → m x)
mkExp :: SubFunctor (Exp m) f ⇒ (Fix f → m (Fix f)) → Fix f
mkExp f = (In ◦ injF) (Exp f)
```

The \texttt{Exp} data type is parametrized over the monad capturing the effectful computations of evaluation semantics. Again, we declare no functor instance for \texttt{Prod} since it is unnecessary for the example. The type constructor for functions, \texttt{Exp}, exemplifies a type constructor that is not a functor: there is no valid \texttt{Functor} instance for \texttt{Exp}, yet we can still use it with the \texttt{SubFunctor} methods for extensibility.

### 3.3 Type Checking

Figure 2 lists four small modules defining the type checking algebras for our example. Following the pattern described in Section 2.2.1, these are single-level definitions: we write the algebras as if each sub-term in the syntactic functor has been replaced with the monadic computation that yields the type of that sub-term.

Algebras for the type-checker and for the evaluator rely on two small modules for managing the environment and exceptions, \texttt{Env} and \texttt{Error}. These modules are implemented in Appendix C.

The algebras for \texttt{Arith}, \texttt{Bools}, and \texttt{Products} do not use the effects of the
MonadReader monad interface. They execute the monadic computation for each sub-term and check to see if the returned types are correct. If not, the algebras use the MonadError interface to raise an exception. The algebra for the typed lambda calculus uses the MonadReader interface to propagate a type context within the body of the lambda, and Error.raise when a type error is discovered. The reliance on various monadic interfaces and the use of type forms can be discovered for each algebra by inspecting its type context.

For this case study, error reporting is impoverished. However, a more sophisticated error reporting can be integrated into system as an independent algebra. Moreover, information such as source positions can be integrated as a separate syntactic functor, which the error reporting algebra can use to generate user-friendly error messages.

To simplify the use of the SubFunctor interface for tagged values, these algebras use a checkTagM function. The checkTagM function takes a monadic computation, executes the computation, and projects the resulting value out of the tagged value space. If the projection fails by returning Nothing, a monadic failure is raised.

\[
\text{checkTagM} :: \\
(MonadError e m, SubFunctor f v) \Rightarrow m (\text{Fix } f) \rightarrow m (f (\text{Fix } v))
\]

\[
\text{checkTagM } m = \text{do} \\
v \leftarrow m \\
\text{case prjF (out } v \text{) of} \\
\quad \text{Just val } \rightarrow \text{return val} \\
\quad \text{Nothing } \rightarrow \text{Error.raise}
\]

A type checker algebra for the entire language is constructed by combining the type checking algebras for each construct in the language. The sumAlg function is used to sum the algebras in the same order that FSum sums the syntactic functors. The resulting algebra specifies the type semantics of Language.

\[
\phi :: \\
(\text{Error } e, \text{MonadError } e m, \\
\text{MonadReader } (\text{Env } (\text{Fix } v)) m, \\
\text{SubFunctor } \text{Types.Int } v, \\
\text{SubFunctor } \text{Types.Bool } v, \\
\text{SubFunctor } \text{Types.Prod } v, \\
\text{SubFunctor } \text{Types.Function } v, \\
\text{Eq } (v (\text{Fix } v)) ) \\
\Rightarrow \text{Algebra } (\text{Composite } (\text{Fix } v)) (m (\text{Fix } v))
\]

\[
\phi = \text{TypeOf.Arith.phi } \text{'}\text{sumAlg'} \\
\quad (\text{TypeOf.Bools.phi } \text{'}\text{sumAlg'} \\
\quad (\text{TypeOf.Products.phi } \text{'}\text{sumAlg'} \\
\quad (\text{TypeOf.TLC.phi } \text{'}\text{sumAlg'} \\
\quad \text{voidAlg } ))))
\]
Fig. 2. Modules for type checking

Note that the combined algebra \( \phi \) is still polymorphic with respect to the monad and the type space, as are the component algebras. Any monad and type constructor that satisfy the type constraints collected via the \texttt{sumAlg} combinator from the component algebras are suitable. In order to apply the denotation to a \texttt{Language} term, we must first construct such a concrete monad and type space.

A stack of monad transformers can be determined by the stack’s run function, so we specify that instead of declaring a type synonym.

\[
\text{run :: } \texttt{ErrorT e (ReaderT [r] Identity)} \ a \rightarrow \texttt{Either e a}
\]

\[
\text{run x = runIdentity (runReaderT (runErrorT x) [])}
\]
This function for running a monadic computation specifies that the monad is built from an error monad transformer stacked on top of a reader monad transformer stacked on top of the identity monad. The stack is built in the opposite order that the transformer’s run functions are applied. The ascribed type shown is the inferred type.

The concrete type space is constructed from the component type constructors introduced in Section 3.2. In general, the SubFunctor constraints from an algebra’s type context directly guide the construction of a suitable sum.

```haskell
type TypeSum = FSum Type.Int
               ( FSum Type.Bool
               ( FSum Type.Function
                 ( FSum Type.Prod
                   FVoid )))
type Type = Fix TypeSum
```

The `typeOf` function applies `cata` to the semantic algebra to yield the denotation function. We execute the resulting monadic computation with the `run` function to yield either an error message or the type of the term. The type-checker is complete.

```haskell
typeOf :: Language Type → Either String Type
typeOf = run ∘ cata phi
```

The `TypeOf` module exposes three definitions: i) the algebra `phi` for the composite language; ii) the run function `run` for the concrete monad; and iii) the denotation function `typeOf`. We include the algebra and the run function for extensibility and reuse concerns. Once the catamorphism is introduced to form the denotation function, the extensibility is lost. In contrast, the algebra itself can still be extended, and contains the essence of the denotation function. The run function will be composed to form a larger run function when the algebra is combined using the sequence algebra combinator, introduced in Section 4.

### 3.4 Evaluation

Figure 3 lists four modules defining evaluation algebras for the example language. In this subsection, we construct an evaluation algebra for the entire language from these component algebras and the necessary monad and value space for the denotation function. The development follows the same steps as the type-checker.

The evaluation semantics specified by the algebras are standard, but the Arith and Products algebras deserve discussion as they demonstrate InterpreterLib’s
Fig. 3. Evaluation Algebras

emphasis on algebras as first-class values. The algebra for arithmetic is simple: there are no computational effects and only one relevant value type, integers. To integrate the algebra into the evaluation function, it must match the form of other algebras that assume a monad and a tagged value space. Before encapsulating this semantic notion in a unary algebra operator, we briefly discuss a useful monadic operation.

The Traversable type class [McBride and Paterson, 2008] provides the qualified polymorphism over the syntactic functor $f$. Throughout this paper, we use the Traversable.sequence method to convert a functor structure filled with monadic computations into a single monadic computation returning a functor value; it witnesses the distributive law of any monad over any Traversable functor. This method specializes the Traversable interface for use with monads instead of the more general applicative functors [McBride and Paterson, 2008] it targets. The Haskell Prelude defines a sequence operator specialized to the list functor [], so we will consistently use the qualified name Traversable.sequence.
Traversable.sequence ::
(Monad m, Traversable f) ⇒ f (m a) → m (f a)

We use the Traversable.sequence operator to define the liftSubAlg algebra operator for promoting a single-valued algebra into one that injects and projects into a composite value space. Note in the type how the algebra’s carrier changes.

liftSubAlg ::
( Error e, MonadError e m,
  SubFunctor (Const a) v,
  Traversable f )
⇒ Algebra f a → Algebra f (m (Fix v))
liftSubAlg phi x = do
  f_v ← Traversable.sequence x
  let unConst = liftM getConst ◦ prjF ◦ out
  case Traversable.sequence (fmap unConst f_v) of
    Just f_a → return ((In ◦ injF ◦ Const) (phi f_a))
    Nothing → Error.throwStr "liftSubAlg␣error"

This operator converts a simple algebra over atomic values into a algebra suitable for a tagged value space. A monadic layer is added to the carrier so the algebra can raise an exception when it is applied to the wrong type of value. The new algebra uses the Traversable property of the syntactic functor twice. First, it executes each of the monadic computations for sub-terms. Then it attempts to project the resulting values to the value type used by the original algebra. This second use of Traversable taking place in the Maybe monad means the entire projection process will fail if any of the sub-terms are of the wrong type, e.g. not integers for Evaluate.Arith.phi. In that case, a monadic exception is raised since the original algebra is not applicable. Otherwise, the original algebra is applied to the functor value filled with integer sub-terms and the result is injected back into the tagged value space with In ◦ injF ◦ Const. This pattern is applicable to any simple algebra over atomic values, so it is useful to capture it as an independent abstraction.

We also define the Products evaluation algebra using an encapsulated pattern. The basic semantics is not concerned with the computational side-effects in the sub-terms, but it can introduce an exception. The core of the definition uses pure inputs and generates monadic outputs. The Traversable property of the syntactic functor is used to convert this half monadic algebra into a fully monadic algebra. The Traversable.sequence operator encapsulates the monadic form of the sub-terms from the semantic algebra.

An even more precise specification of the Products semantics could separate the exception-free Pair case from the cases for projections, but this would
require the constructors to be defined in separate syntactic functors. This suggests another way to separate syntactic forms into syntactic functors: one functor for introduction forms and another for elimination forms. The most precise separation gathers conceptually related syntactic forms whose semantic algebras have the same type. We group them by concept in this paper for simplicity.

The algebras for Bools and the typed lambda calculus do not exhibit such abstractions because the details of the semantic algebras need the expressiveness of monadic sub-terms and a tagged value space. Both specify the order of evaluation, thus the monadic interface cannot be factored out. Both also involve one concrete type (the boolean guard and the function) and the entire tagged value space (the branches and the domain and range), thus the SubFunctor interface also cannot be factored out.

As with type-checking, the component algebras are combined to form an algebra for the full signature. We elide the type of phi; although it is large, it is simply the combination of the types in Figure 3.

\[
\begin{align*}
\phi &= \text{Evaluate.Arith.phi} \; \text{'sumAlg'} \\
&\quad ( \text{Evaluate.Bools.phi} \; \text{'sumAlg'} \\
&\quad \quad ( \text{Evaluate.Products.phi} \; \text{'sumAlg'} \\
&\quad \quad \quad ( \text{Evaluate.TLC.phi} \; \text{'sumAlg'} \\
&\quad \quad \quad \quad \text{voidAlg} )))
\end{align*}
\]

We again need a concrete value space and it is directly derived from the SubFunctor constraints. We now use the type constructors for values introduced in Subsection 3.2.

\[
\begin{align*}
type \; \text{ValueSum} \; m &= \text{FSum} \; (\text{Const \; Int}) \\
&\quad ( \text{FSum} \; (\text{Const \; Bool}) \\
&\quad \quad ( \text{FSum} \; (\text{Values.Exp} \; m) \\
&\quad \quad \quad ( \text{FSum} \; \text{Values.Prod} \\
&\quad \quad \quad \quad \text{FVoid} )))
\end{align*}
\]

\[
\begin{align*}
type \; \text{Value} \; m &= \text{Fix} \; (\text{ValueSum} \; m)
\end{align*}
\]

As indicated by the parameter to Value, we also need a concrete monad. The construction of the evaluation monad is more involved than for type-checking because of its necessarily recursive type definition. The monad carries an environment, which holds the tagged value space. However, the Exp m component of the value space also involves the monad. This recursion can be identified in the SubFunctor and MonadReader constraints within the type context for Evaluate.TLC.phi in Figure 3.

The recursion gives rise to an infinite type for the monad. Since we intend to build the monad from monad transformers and none of the transformers
encapsulate recursion, the type cannot be expressed as a finite term. Assuming
the evaluation monad is named $M \circ v$ and parametrized over the value space (just
as the value space is parametrized over the monad), consider the type of the
evaluation function.

```haskell
-- invalid, infinite type
evaluate :: Language Type \rightarrow Either String (Value (M (Value (M ...
```

We resolve this issue with the monad in the next subsection, including the
definition of the $runM$ function. The rest of the evaluation code is as simple as
for the type-checker.

```haskell
run t = runM t []
```

```haskell
evaluate :: Language Type \rightarrow Either String (Value M)
evaluate = run \circ cata phi
```

Again, $phi$, $run$, and $evaluate$ are exposed by the $Evaluate$ module for exten-
sibility and reuse concerns.

3.5 Recursively-defined Monads

The value space and the monad used for evaluation are mutually recursive.
This is a consequence of the the algebra being polymorphic over the envi-
ronment monadic effect, using a $MonadReader$ class constraint. The mutual
recursion can be eliminated by fixing a concrete type for the environment
effects – e.g. $ReaderT (Env (Fix F)) m$, but this solution reduces the modu-
larity of the algebra, because it forces a structure on the monad used for the
semantics.

InterpreterLib provides a standard construction for dealing with situations
where an algebra’s monad and value types are mutually recursive, based on the
definition of a monad as a fixed point of a higher-order functor. The basic idea
is the same as the extensible data type. Just as the $Fix$ type constructor finitely
represents an inductive ground type of kind $\ast$, the $HoFix$ type-constructor
constructor finitely represents an inductive type constructor of kind $\ast \rightarrow \ast$.

```haskell
-- HoFix :: ((\ast \rightarrow \ast) \rightarrow (\ast \rightarrow \ast)) \rightarrow (\ast \rightarrow \ast)
newtype HoFix hof a = HoFix (hof (HoFix hof) a)
```

```haskell
unHoFix :: HoFix hof a \rightarrow hof (HoFix hof) a
unHoFix (HoFix x) = x
```

We can now construct the evaluation monad in valid Haskell.
type MF' self = ReaderT (Env (Value self)) (ErrorT String Identity)

newtype MF self a = MF (MF' self a)
unMF :: MF self a → MF' self a
unMF (MF m) = m

type M = HoFix MF

The MF' type synonym represents the essence of the monad declaration. Since type synonyms cannot be partially applied and they cannot introduce recursion, we use the newtype MF as a thin wrapper around MF' that allows us to introduce the recursion with HoFix.

We could introduce the recursion directly within the definition of MF, but we must declare instances for each of the numerous monadic interfaces specifying how the MF newtype derives those properties from MF'. This would be a significant amount of boilerplate, and so we avoid it. Minimizing the code necessary to inherit the monadic type class membership is the major advantage of defining monads with HoFix. To do so, we recognize the isomorphism introduced by the HoFix fixed point with a general-purpose class for capturing isomorphisms of functors, FunctorIso.

class FunctorIso f g | f → g where
  fOut :: f a → g a
  fInn :: g a → f a

Using this isomorphism, we declare one instance for each monadic type class specifying how a monad constructed with HoFix derives the methods from the underlying type synonym. The functional dependency is necessary for the type-checker to associate the underlying type synonym with the HoFix monad. The key is that there need only be one instance for each type class; the FunctorIso type class acts as a modularity boundary. For example, a HoFix monad is a member of the Monad type class if it is isomorphic to a monad.

instance ( Monad m, FunctorIso (HoFix hof) m ) ⇒ Monad (HoFix hof) where
  return = fInn ∘ return
  m >>= f = fInn ((fOut m) >>= (fOut ∘ f))

InterpreterLib introduces such HoFix instances for all of the standard monad interfaces. The definition of each method is rote since it is based on the isomorphism: the fOut and fInn witnesses are applied to inputs and outputs. It is easy for the user to introduce new HoFix instances for any custom monadic interfaces.

We declare the FunctorIso instance for M in order for it to inherit all of the
monadic capabilities of the MF' type synonym. We also declare the run function. The monad M is now a member of all the same type classes as MF': including Functor, Monad, MonadReader, and MonadError.

\[
\text{instance } \text{FunctorIso } M \text{ } (\text{MF'} M) \text{ where }
\begin{align*}
\text{fOut} &= \text{unMF} \circ \text{unHoFix} \\
\text{fInn} &= \text{HoFix} \circ \text{MF}
\end{align*}
\]

\begin{align*}
\text{runM} &: : M x \rightarrow \text{Env } (\text{Value } M) \rightarrow \text{Either } \text{String } x \\
\text{runM } m \text{ initR} &= \text{runIdentity } (\text{runErrorT } (\text{runReaderT } (\text{fOut } m) \text{ initR}))
\end{align*}

Although defining a monad via the HoFix construction is entirely mechanical, it requires that the user write a few trivial declarations and an instance declaration for FunctorIso. This boilerplate is minimal compared to the many monadic type class instance declarations that would be required of the user without the HoFix technique. Instead, these instances are inherited automatically via the FunctorIso instance and the library of instance of the monadic type classes for HoFix. The use of HoFix and FunctorIso to inherit instances supplants an extension of GHC called generalized newtype deriving. GHC (as of version 6.8.2), is unable to automatically derive instances for recursively-defined monads.

4 Algebra Combinators

The MMS framework simplifies combining semantic algebras for a single language processing task defined over a varied collection of language constructs. Examples in Section 3 demonstrated this for a collection of type checking and evaluation algebras. We now move to an orthogonal form of algebra composition combining two semantic algebras defined over the same set of constructs, representing different language processing tasks. The standard MMS framework provides little support for this form of composition, but can be extended as we demonstrate with two specific use cases.

4.1 Switch Algebra

InterpreterLib uses a catamorphism combinator as a control structure to capture the recursive application of semantic algebras. As a consequence of using this combinator, the programmer relinquishes control of the AST traversal. This becomes an obstacle when the algebra needs to apply a different denotation based on contextual information. The switchAlgebra 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 `choiceAlg` function is an example of an algebra for some functor F and carrier A that, based on a switch value, applies one of `alg0`, `alg1`, or `alg2`.

\[
alg0, \ alg1, \ alg2 :: (\text{Monad} \ m, \ldots) \Rightarrow \text{Algebra} \ F \ (m \ A)
\]

```
data Switch = Zero | One | Two
choiceAlg :: (\text{Monad} \ m, \ldots) \Rightarrow \text{Switch} \rightarrow \text{Algebra} \ F \ (m \ A)
choiceAlg Zero = alg0
choiceAlg One = alg1
choiceAlg Two = alg2
```

We can build an algebra from `choiceAlg` using a function defined in InterpreterLib called `switchAlgebra`. We use a type class called `MonadSwitch` to manage the switch. The details of its implementation are described in Section 6.1, but its behavior is similar to that of the `Reader` monad.

```
switchAlgebra ::
  (\text{MonadSwitch} \ sw \ m, \text{Functor} \ f) \Rightarrow (sw \rightarrow \text{Algebra} \ f \ (m \ a)) \rightarrow \text{Algebra} \ f \ (m \ a)
localSwitch :: \text{MonadSwitch} \ sw \ m \Rightarrow (sw \rightarrow sw) \rightarrow m \ a \rightarrow m \ a
runSwitchT :: \text{Monad} \ m \Rightarrow (\text{SwitchT} \ sw \ m) \ a \rightarrow sw \rightarrow m \ a
alg :: (\text{MonadSwitch} \ Switch \ m, \ldots) \Rightarrow \text{Algebra} \ F \ (m \ A)
alg = \text{switchAlgebra} \ choiceAlg
```

The algebra `alg` is built from the constituent algebras, `alg0`, `alg1`, and `alg2`. To select the algebra applied at any given term, the component algebras modify the switch feature of `m`. This is accomplished using the `localSwitch` morphism that takes an update function and evaluates a computation within the updated switch. It is assumed that the component algebras `alg0`, `alg1`, and `alg2` use `localSwitch` to invoke one another.

As with previous algebras, applying `cata` to `alg` yields a complete denotation. This denotation is monadic, with `MonadSwitch` features. Therefore, InterpreterLib includes a `SwitchT` monad transformer, defined in Section 6.1. The `SwitchT` layer is executed using `runSwitchT`, here provided an initial switch value of `Zero`. We assume that the rest of the monadic constraints are satisfied by a hypothetical monad `M` and its `runM` function.

```
runM :: M \ a \rightarrow a
evaluate :: \text{Fix} \ F \rightarrow A
```
evaluate \( t = \text{run} (\text{cata alg } t) \)
where \( \text{run } m = \text{runM} (\text{runSwitchT } m \text{ Zero}) \)

When `evaluate` is applied to a term, each application of `alg` resulting from the catamorphism equates to an application of one of `alg0`, `alg1`, or `alg2`, depending on the local context of the `MonadSwitch`.

### 4.2 Sequence Algebra

The `sequenceAlgebra` algebra combinator allows one algebra to access the intermediate results of another. For example, if an algebra for compilation were to rely upon type information to make optimization decisions, then the type algebra and the compilation algebra could be sequenced so that the type information would be available to the compilation algebra.

Just as `switchAlgebra` uses a monadic feature to recover some control of the traversal from the catamorphism, the sequence combinator uses monadic features to make intermediate results persistent even though the catamorphism would not make those results available by default. The semantics of the `sequenceAlgebra` combinator are quite intuitive. The implementation is less straightforward and discussed in detail in Section 6.2.

The sequence algebra combinator combines one algebra with another algebra indexed by the carrier of the first algebra.

\[
\text{sequenceAlgebra} :: \quad \text{MonadSequence } f \ a \ b \ m \Rightarrow \text{Algebra } f \ (m \ a) \\
\quad \quad \quad \quad \quad \rightarrow (a \rightarrow f \ a \rightarrow \text{Algebra } f \ b) \\
\quad \quad \quad \quad \quad \rightarrow \text{Algebra } f \ (m \ (a,b))
\]

\[
\text{runSequenceT} :: \text{Monad } m \Rightarrow (\text{SequenceT } f \ a \ b \ m) \ (a, b) \rightarrow m \ b
\]

If the carrier of the first algebra were not monadic, then the sequence algebra combinator would be trivial to write, but the traversals that the algebra could represent would be severely limited. Because the indexed algebra’s carrier `b` is wrapped-up in the first algebra’s monad `m` within the carrier of the resulting algebra, any effects introduced by the indexed algebra – if `b` is monadic – cannot affect the first algebra’s traversal - the combinator does not interleave the effects of the two traversals. The sequence algebra combinator makes the results of the monadic computation available to the second algebra without violating the encapsulation of effects or duplicating them. Indeed, it does so by introducing another monadic layer, as indicated by `MonadSequence`. The details of that monad and its run function are given in Section 6.2. Since the features of this monad transformer are used only to achieve the sequencing
of the algebras and are not meant to provide an interface to be used by the programmer, the only necessary interface is the run function.

5 Applications

The switchAlgebra and sequenceAlgebra algebra combinators provide generic mechanisms for combining distinct algebras over the same term space. We present three examples of using these combinators. First, we use switchAlgebra to define a generic semantics for meta-programming. Then we demonstrate the use of sequenceAlgebra, combined with a generic “identity” algebra, to construct a paramorphism recursion operator. Finally, we utilize sequenceAlgebra for type-directed synthesis, reusing the type-checking algebra from Section 3.3.

5.1 Example: Meta-programming

As an application of the switchAlgebra, we have defined a generic semantics for quasiquote-style meta-programming [Bawden, 1999]. To add meta-programming features to the language, we first extend the language with quasiquote syntax and add syntactic terms to the value space. The original algebras are combined with algebras to handle quasiquote semantics, yielding a complete evaluator for the language.

Below are three expressions written using meta-programming using Scheme syntax. To the left of the => is a term in the meta-programming language, and to the right is the resulting value after evaluation with the meta-programming evaluator. The backtick increases the quote level, while the comma decreases the quote level. The quote level is initially zero, and an expression is only evaluated when the quote level is zero. In the first example, there are no quasiquotes, so the entire expression evaluates. In the second example, the entire expression is preceded by a quasiquote, so it is not evaluated, and the result is syntax. The third example begins with a quasiquote, increasing the quote level to 1 and turning off evaluation; the unquote comma preceding the (+ 2 3) sub-term decrements the quote level, and so that term is evaluated to 5 and then included as syntax into the resulting term (+ 1 5).

(+ 1 (+ 2 3)) => 6
'(+ 1 (+ 2 3)) => (+ 1 (+ 2 3))
'(+ 1 ,(+ 2 3)) => (+ 1 5)

We define a generic framework for adding meta-programming capabilities to
data Quotes a = Quote a | UnQuote a

type MetaLanguage f = Fix (FSum Quotes f)

class Reify val syn where reify :: val → syn

newtype DynValue a = DV a

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

Fig. 4. Types and instances supporting meta-programming

any arbitrary language written in InterpreterLib. We introduce the following
definitions to facilitate this task, several of which are defined in Figure 4.

• The Quotes syntactic functor. The smart constructor functions mkQuote and
  mkUnQuote are derived from this definition.

• A Reify class for converting values into syntax. In this example, we focus
  only on reifying ground values.

• An algebra, reifyAlg, used to evaluate the inside of a quoted term by simply
  injecting the term into the value space. This algebra is modular over any
  term space and any value space that is an instance of Reify.

• A new value, DynValue, for adding syntax to any value space, as well as
  an instance of Reify for DynValue. The DynValue allows syntactic values to
  be easily integrated into the SubFunctor framework by providing a distin-
guished type for syntactic values. The smart constructor function mkDV is
  derived from this definition.

• Utilities for building a switch algebra. The MonadSwitch switch type is an
  integer, representing the quote level.

These components are combined in the addMeta function, shown in Figure 5. The
addMeta function is parametrized over an evaluation algebra for the origi-
 nal language and yields an algebra for the extended meta-programming lan-
guage. The semantics for Quote and UnQuote syntax, are embedded in the
addMeta function.

The following rules describe how to evaluate a term in the meta-programming
language: i) the current quote level; ii) an evaluation algebra, evalAlg, for the
original language without quote syntax; and iii) the reifyAlg function defined
in Figure 5.

Case 1 Quote Level = 0
  • Quote t - Evaluate t with incremented quote level
  • UnQuote t - Error
  • t - Evaluate t using evalAlg
Case 2 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 \texttt{reifyAlg}

Case 3 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 \texttt{reifyAlg}

These rules are realized by the \texttt{addMeta} function, shown in Figure 5, that takes an evaluation algebra for the original language, \texttt{evalAlg}, and yields an algebra for the meta-programming language.

To allow the appropriate \texttt{SubFunctor} instance to be selected by the Haskell typechecker, \texttt{addMeta} uses scoped type variables to explicitly introduce the type variable \( f \) to the original term type, and adds type ascriptions when projecting from the combined value space. The explicit \texttt{forall} brings the type variables into scope.

The \texttt{reifyAlg} algebra converts any syntactic term into a semantic value by injecting it into the value space. In principle, \texttt{reifyAlg} traverses \( t \), applying \texttt{reify} to each sub-term, and then injects the entire result into the value space as a \texttt{DynValue}.

The definition of \texttt{reifyAlg} in Figure 5 appears more complicated due to its use of the \texttt{Traversable} class. Recall that \texttt{reifyAlg} has a local binding \texttt{phi} which takes a parameter \( x \) of type \( f \ (m \ v) \). The function value \texttt{liftM reify} applies \texttt{reify} to the result of its argument computation without removing the monadic layer; the result type is \( m \ v \), with the additional property that the value will be a \texttt{DynValue}.

A language that uses the \texttt{addMeta} function to add meta-programming semantics must provide instances for the \texttt{Traversable} class. The \texttt{addMeta} function allows a user to augment an existing evaluation semantics to add quasiquote semantics. This technique maximizes code reuse, only requiring of the user an instance of \texttt{Reify} for each element in the value space and trivial extension of the syntactic signature and semantic domain.

5.1.1 Reify Class

Each instance of \texttt{Reify} defines how to convert semantic values back into syntactic terms. For example, the following instance defines reification from integers
addMeta :: forall f m v .
( Error e, MonadError e m,
  MonadSwitch Int m,
  SubFunctor (DynValue (MetaLanguage f)) v,
  Traversable f,
  Reify (v (Fix v)) (MetaLanguage f) )
⇒ Algebra f (m (Fix v)) → Algebra (FSum Quotes f) (m (Fix v))

addMeta evalAlg = switchAlgebra phiSwitch where
  phiSwitch 0 = phiQuoteZero 'sumAlg' evalAlg
  phiSwitch 1 = phiQuoteOne 'sumAlg' reifyAlg
  phiSwitch n = phiQuotePos 'sumAlg' reifyAlg

-- Case 1
phiQuoteZero (Quote t) = localSwitch succ t
phiQuoteZero (UnQuote t) = Error.throwStr "Top-level UnQuote"

-- Case 2
phiQuoteOne (Quote t) = do
  DV (t' :: MetaLanguage f) ← checkTagM (localSwitch succ t)
  return (mkDV (mkQuote t'))
phiQuoteOne (UnQuote t) = localSwitch pred t

-- Case 3
phiQuotePos (Quote t) = do
  DV (t' :: MetaLanguage f) ← checkTagM (localSwitch succ t)
  return (mkDV (mkQuote t'))
phiQuotePos (UnQuote t) = do
  DV (t' :: MetaLanguage f) ← checkTagM (localSwitch pred t)
  return (mkDV (mkUnQuote t'))

reifyAlg :: forall f m v .
( Monad m, Traversable f,
  Reify (v (Fix v)) (Meta f),
  SubFunctor (DynValue (Meta f)) v )
⇒ Algebra f (m (Fix v))
reifyAlg t = do
  v ← Traversable.mapM (liftM reify) t
  return (mkDV (inn (R v)) :: MetaLanguage f)

Fig. 5. addMeta definition

in the value space to an Arith term in the term space.

instance SubFunctor Arith f ⇒ Reify Int (Fix f) where
  reify = mkNum

31
Like most instances of \texttt{Reify}, this one is simple. We simply apply \texttt{mkInt} to turn an integer into syntax and inject it into the term space. This instance will work for any term and value spaces including \texttt{Arith} and \texttt{Int}, respectively. Similarly, we can define \texttt{Reify} instances for recursive types, such as \texttt{Value.Prod}.

\begin{verbatim}
instance ( SubFunctor Products f, Reify a (Fix f) ) ⇒ Reify (Prod a) (Fix f) where reify (Prod (x,y)) = mkPair (reify x) (reify y)
\end{verbatim}

The instance of \texttt{Reify} for \texttt{Exp} is more challenging than for the remainder of the types, as the \texttt{Exp} type includes a monadic computation. Sheard [1997] demonstrates a mechanism for performing this reification in the context of defining a type-directed partial evaluator.

\subsection{A Complete Semantics}

As mentioned earlier, extending the value space with syntax is trivial. This value space is parametrized over a term space constructed as a sum of functors \( f \), so it can be used with any syntactic signature.

\begin{verbatim}
type MetaValueSum f m = FSum (Const (DynValue f)) (ValueSum m) type MetaValue f m = Fix (MetaValueSum f m)
\end{verbatim}

The \texttt{MetaM} monad, which supports the monadic features necessary for the meta-programming semantics, is specified using the \texttt{HoFix} method. It is the composition of the \texttt{ReaderT}, \texttt{ErrorT}, and \texttt{SwitchT} monad transformers.

\begin{verbatim}
type MF' f self = ReaderT (Env (MetaValue f self)) (ErrorT String (SwitchT Int Identity))
newtype MF f self a = MF (MF' f self a) unMF :: MF f self a → MF' f self a unMF (MF m) = m
type MetaM f = HoFix (MF f)
instance FunctorIso (MetaM f) (MF' f (MetaM f)) where fOut = unMF ∘ unHoFix fInn = HoFix ∘ MF
runMetaM :: MetaM f a → Env (MetaValue f (MetaM f)) → Int → Either String a runMetaM x r sw = runIdentity (runSwitchT (runErrorT (runReaderT (fOut x) r)) sw)
\end{verbatim}
The `Language` type synonym defines the term space for the meta language, and `alg` defines the evaluation algebra for the meta-programming language. Finally, the `runMetaM` function is used to define the `metaEvaluate` function for evaluating terms in an initial environment and at an initial quote level.

```haskell
type Language = MetaLanguage (Eval.Composite Type)

alg x = addMeta Eval.alg x

metaEvaluate :: Language →
    Either String (MetaValue Language (MetaM Language))
metaEvaluate t = runMetaM (cata alg t) [] 0
```

### 5.2 Example: Paramorphism

Semantic algebras over a functor `F` and with a carrier `a` have the form `F a → a`. Using the catamorphism combinator, all of the sub-terms of a term of type `F` have already been evaluated to the carrier type `a` when the algebra is applied. Algebras are single-level operators, and there is no way to examine the original AST representation of a term.

Some semantic algebras may be dependent not only on the carrier-type values of the sub-terms which the catamorphism provides, but also the original representation of the term. These “algebras” have the type `Fix F → Algebra F a`; the algebra is parametrized over the original AST form. The catamorphism operator cannot be used to apply these forms of parametrized algebras, thus we use the `paramorphism` combinator [Meertens, 1992].

The paramorphism combinator requires the use of an “identity” algebra, which straightforwardly reconstructs a term from its structured sub-terms. Because the `idAlg` algebra is pure (not requiring any monadic effects), we lift it into a monadic carrier using the `liftMAlg` algebra operator. Using the universally-defined `sequenceAlgebra` and `idAlg`, a paramorphism combinator can be easily derived. The type context for `para` includes the restrictions on the functor, which will be discussed in later sections.

```haskell
idAlg :: Algebra f (Fix f)
idAlg = In

liftMAlg ::
    (Monad m, Traversable f) ⇒ Algebra f a → Algebra f (m a)
liftMAlg alg = liftM alg ∘ Traversable.sequence

paraAlg ::
    MonadSequence f (Fix f) b m
```
\[(\text{Fix } f \rightarrow \text{Algebra } f \, b) \rightarrow \text{Algebra } f \, (\text{m } (\text{Fix } f, b))\]

\[
\text{paraAlg idxAlg} = \\
(\text{liftMAlg idAlg}) \, \text{`sequenceAlgebra`} \, (\lambda \text{term terms} \rightarrow \text{idAlg} \, \text{term})
\]

\[
\text{para ::} \\
(\text{FunctorContext } f, \text{FunctorZip } f, \text{Traversable } f) \\
\Rightarrow (\text{Fix } f \rightarrow \text{Algebra } f \, a) \rightarrow \text{Fix } f \rightarrow a \\
\text{para pphi t} = \text{runIdentity} \, (\text{runSequenceT} \, (\text{cata (paraAlg pphi) t}))
\]

5.3 Example: Type-directed synthesis

The paramorphism combinator uses \text{sequenceAlgebra} to compose the non-monadic \text{idAlg} with an indexed algebra that has a monadic carrier. The \text{sequenceAlgebra} combinator is more expressive than is necessary for defining paramorphism, as it allows the sequencing of any two algebras that both have monadic carriers. This capability is demonstrated in the development of a compiler that synthesizes VHDL [IEEE] netlists from a functional language. The compilation target uses a low-level representation of data as bit-vectors, and many operations require precise bit-vector indexing. The information necessary to do this indexing is most naturally extracted from the types of the expression.

We focus on the \text{Products} construct in this example. The size of a bit-vector representing a product will vary depending on the size of the component values. We treat a product of two bit-vectors as the concatenation of those component bit-vectors. In the following code, the \text{splice} and \text{sliceLeft} functions calculate appropriate bit-vector indices based on types.

The types can be manually propagated through the compiler by returning tuples of types and netlists, rather than just netlists.
return (Types.mkProd xty yty, znl)

netlist (PrjL p) = do
  (pty, pnl) ← p
  case prjF (out pty) of
    Just (Prod xty _) → do
      znl ← sliceLeft xty pty pnl
      return (xty, znl)
    Nothing → Error.raise

The problem with this approach is that it requires the programmer to re-
implement the type rules from Figure 1(c) within the netlist 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. Additionally, as the type checking semantics
becomes more complex the duplication of code becomes greater and offers
more potential for introducing errors.

This problem can be avoided through the use of the sequenceAlgebra com-
binator. Rather than explicitly pass around tuples of values and netlists,
sequenceAlgebra 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 results of the type
checker algebra for the sub-terms.

netlist ::
  ...
  ⇒ Type → Products Type → Algebra Products (m Type)
netlist zty _ (Pair x y) = do
  xnl ← x
  ynl ← y
  znl ← splice zty xnl ynl
  return znl

netlist xty (PrjL pty) (PrjL p) = do
  pnl ← p
  znl ← sliceLeft xty pty pnl
  return znl

The type checking algebra from Figure 1(c) and the netlist algebra are com-
bined using sequenceAlgebra into a single algebra. This combination algebra
could be sequenced with yet other algebras allowing for arbitrarily long algebra
sequences. As an example of the usefulness of the sequenceAlgebra combina-
tor, we have also used the combinator to integrate a type-inference semantics
with netlist compilation semantics.
6 Implementing the Algebra Combinators

The switchAlgebra and sequenceAlgebra combinators are implemented by adding features to the monadic carriers of the algebras they operate on. This process is simplified due to the use of the monad class interfaces—rather than a specific monad formulation—within the specification of component algebras. Moreover, the fixed-point-of-a-functor representation of syntactic terms allows combinators to use an abstract structural interface for term representation. This capability is critical to implementation of the sequenceAlgebra combinator.

Both combinators add a layer of monadic features to the carrier of the algebra. The switchAlgebra combinator requires an additional environment feature for moving information down the abstract syntax tree, while the sequenceAlgebra requires an environment feature as well as an output feature for moving information up the tree alongside the value returned by the monadic computation. The monad transformers supporting both combinators are implemented as thin wrappers around the same monad transformer, RWT. Similarly, the type classes and run functions are thin wrappers around MonadRW and runRWT. The RWT monad transformer supports environment and output effects and avoids a potential conflict with the standard monad libraries. In particular, the transformer uses a phantom type which is omitted from the body of the paper for brevity—it is an orthogonal concern. All details are given in Appendix B.

6.1 Implementing switchAlgebra

The switchAlgebra combinator maps directly to the environment monadic feature. It takes a choice function, that, when applied to the current value of the “switch” identifying the phases of the semantic algebra, yields an Algebra. The resulting algebra is then applied to the current term, x.

switchAlgebra ::
  (Functor f, MonadSwitch sw w m)
⇒ (sw → Algebra f (m a)) → Algebra f (m a)
switchAlgebra choice x = RW.ask >>= λsw → (choice sw) x

localSwitch :: MonadRW sw w m ⇒ (sw → sw) → m a → m a
localSwitch = RW.local

runSwitchT :: Monad m ⇒ (SwitchT sw m) a → sw → m a
runSwitchT m sw = liftM fst (runRWT m sw)

The definition of each algebra indexed by the choice function includes the localSwitch to determine which algebra is applied to the sub-computations. The runSwitchT function removes the SwitchT layer of the monad. It ignores the second component of the pair returned by runRWT, corresponding to the final monadic output, which, as detailed below, is the useless () value in this case.

The MonadSwitch type class and the SwitchT transformer are thin wrappers around MonadRW and RWT. The () type serves as a vacuous monoid since the output features of MonadRW are not used by the switch combinator.

type SwitchT r = RWT r ()

class MonadRW r () m ⇒ MonadSwitch r m | m → r
instance MonadRW r () m ⇒ MonadSwitch r m

6.2 Implementing sequenceAlgebra

The challenge in implementing the sequenceAlgebra combinator originates in the need to make the results of the first algebra available to the second algebra without losing the monadic encapsulation. Our implementation assumes that the second algebra will only access those results of the first algebra that are forced via the >>= method in a monadic computation. This restriction is best described by a counterexample.

Recall the definition of evaluation of If expression from Figure 2(b). The function phi uses the monadic computation of either the mtru or mfls branch of the If.

phi (If mcond mtru mfls) = do
    Const cond ← checkTagM mcond
    if cond then mtru else mfls

Consequently, if an algebra like phiIdx below is sequenced after this one, the sequenceAlgebra combinator will only have a result available for one of those branches. The result will be defined for either truVal or flsVal, but not both. If phiIdx is strict in both, then a run-time exception will be raised if an If
constructor appears anywhere in the term being evaluated.\footnote{An exception will also be raised if \texttt{phiIdx} is strict in both \texttt{tru} and \texttt{fls}, as these \texttt{phiIdx} are (as detailed below) actually generated as a side-effect of \texttt{mtru} and \texttt{mfils} in the evaluation algebra.}

\[
\phiIdx p (\text{If } \text{condVal} \text{ truVal } \text{flsVal}) (\text{If } \text{cond} \text{ tru } \text{fls}) = \ldots
\]

InterpreterLib does not handle this exception monadically; therefore, any use of the combinator must be statically proven valid, since the exception raised is fatal.

We say that any algebra pair that guarantees that no exception will be raised satisfies the \textit{sequencing restriction}. For each constructor in the AST traversed by the second algebra, the first algebra must produce exactly one value. If an algebra produces exactly one value for every constructor, then it is guaranteed to satisfy the sequencing restriction with any second algebra.

As a consequence of implementing \texttt{sequenceAlgebra} as a monadic layer of the first algebra’s carrier, the production of a value corresponds to the inclusion of a monadic computation via the monadic method \texttt{>>=}. In a first algebra, the holes within the functor argument to the algebra contain monadic computations computed from the respective sub-terms of the AST. If each of those computations are included in the resulting monadic computation via \texttt{bind} exactly once, the algebra is safe for use with \texttt{sequenceAlgebra}. If any computation is not included, the algebra is not necessarily safe for \texttt{sequenceAlgebra} and may therefore only satisfy the sequencing restriction with a second algebra that does not inspect the value corresponding to the omitted computation. If any computation is included multiple times, the single value that gets passed along is deterministic given the implementation, but is specified to be arbitrary.

The following strategy overviews the implementation of the \texttt{sequenceAlgebra} combinator. First, provide a mechanism for recording the result of a monadic computation and the syntactic position (context) where the computation was generated. Second, provide a mechanism to extract a recorded result. Third, augment the first algebra in the sequence so that all results are recorded. Finally, augment the second algebra to extract recorded results and apply the algebra to those recorded results.

Two functor properties, captured as type constructor classes, are used to implement this strategy. These properties allow the combinator to be overloaded over all functors. Both classes capture general structural operators provided by many functors. However, both classes are also subject to coherence conditions that are not directly expressible within the Haskell type system. We introduce both classes and their associated coherence conditions before proceeding with
The algebra combinator.

The \texttt{fzip} operator lifts the notion of product to the level of a functor: it specifies how to merge two functor values.

\begin{verbatim}
class Functor f ⇒ FunctorZip f where
   -- partial functions
   fzip :: f a → f b → f (a, b)
   fzipWith :: (a → b → c) → f a → f b → f c
   fzip = fzipWith (,)
   fzipWith f x y = fmap (uncurry f) (fzip x y)
\end{verbatim}

We assume the following coherence condition for \texttt{fzip}. It specifies that the two arguments must have the same functor structure in order for the operator to behave as a product.

\begin{enumerate}
\item \texttt{shapeOf x = fmap (const ()) x}
\end{enumerate}

\begin{verbatim}
   fmap fst (fzip a b) = a ← shapeOf a = shapeOf b
   fmap snd (fzip a b) = b ← shapeOf a = shapeOf b
\end{verbatim}

Instead of including this precondition as a coherence condition, the partiality of the \texttt{fzip} function could be modeled with the \texttt{Maybe} monad. However, the \texttt{fzip} operator is only used in the definition of the combinator and in such a way that there is no meaningful way to handle the exceptional case. Since the error is always fatal, there is no reason to model it monadically.

The second functor property exhibits the \texttt{fcontexts} operator for generating contextually situated values. For each hole within the functor value, the entire functor value is duplicated with that hole distinguished as the only \texttt{Nothing} value.

\begin{verbatim}
type FContext f a = f (Maybe a)

fillCtxt :: Functor f ⇒ (a → b) → b → FContext f a → f b
fillCtxt f self = fmap (maybe self f)
\end{verbatim}

\begin{verbatim}
class Functor f ⇒ FunctorContext f where
   fcontexts :: f a → f (a, FContext f a)
\end{verbatim}

The \texttt{FunctorContext} type class also has coherence conditions not expressible within the Haskell type system. We use the \texttt{foldr} method of the \texttt{Foldable} class, which generalizes the reduction operator from lists to other functors. The \texttt{Foldable} type class is a super-class of \texttt{Traversable} and so safe to assume for all syntactic functors.
The first coherence condition specifies that all of the contexts generated by \texttt{fcontexts} are well-formed. Specifically, it requires that there be one \texttt{Nothing} per \texttt{FContext} value. If the functor value actually contains no holes, then the limitation is moot; the coherence condition as stated still holds in this degenerate case. The second coherence condition specifies that \texttt{fillCtxt} reduces a value and its context, as generated by \texttt{fcontexts}, to a whole value by plugging the situated value back into the single hole in the contextual value.

Since these two functor properties are required for the use of the combinator, all syntactic functors must be instances of the two classes. The InterpreterLib library includes the following instances for \texttt{FSum} and \texttt{FVoid}. Once again, the \texttt{FVoid} instances are safe because it is only a sentinel type, and no values actually exist. InterpreterLib also declares instances of the \texttt{Traversable} class.

We now define the \texttt{sequenceAlgebra} algebra combinator. Instead of abandoning the monadic encapsulation, we put it to use. We add features to the monad of the first algebra's carrier in order to support the second algebra's traversal as a monadic side-effect of the first. The Haskell qualified type system allows us to hide these additional features behind monadic class interfaces. If the first algebra were to specify its exact monad instead of requiring only support for various interfaces, the combinator could not require the additional \texttt{MonadSequence} interface. If the combinator instead added the \texttt{SequenceT} trans-
former to the first algebra’s monad, it would have to duplicate some effects in order to apply the first algebra without losing access to the effects of the sub-terms’ computations, since the algebra is contravariant in the carrier. Using the monadic interface causes the algebra to unknowingly build the necessary effects into its own result on behalf of the combinator (hence the coherence conditions on the first algebra).

The key concept behind the monadic sequence algebra combinator is to modify the computations defined by the first algebra to store their result in an output that travels alongside the carrier up the data structure. This monadic feature is commonly called a Writer monad. The first algebra builds a computation \((m \ a)\) from the sub-computations resulting from the traversal of sub-terms \((f (m \ a))\). The execution of the sub-computations yields a list of their results \(([a])\), from which the current computation can reconstruct the functor structure \((f \ a)\). We first develop functions in support of such decomposition and reconstruction of functor values, and then use a monadic interface to perform the traversal of the second algebra as a side-effect of the first algebra.

A list of values does not provide enough information to reconstruct a functor value, since there is ambiguity regarding where each value in the list belongs in the structure of the functor. Thus, instead of a list of values, we store a list of situated values. A situated value is a value accompanied by an indication of where it belongs in the functor structure. For reasons we will see later, we call such an index into a functor structure a turn.

```haskell
type Turn f = f Bool

childTurns :: FunctorContext f ⇒ f a → f (Turn f)
childTurns = fmap (fillCtxt (const False) True ∘ snd) ∘ fcontexts
```

We specify a Turn as a functor holding booleans. If the functor contains sub-terms, then exactly one hole is True. If the functor contains no sub-terms, and hence no holes, then the Turn will contain no Bool values at all. To generate turns from a functor value, we assume that the functor supports the type class for generating contexts. childTurns first uses fcontexts to duplicate the structure and then ignores the values in the holes, replacing them with booleans to specify the turn associated with the context. The fcontexts-fillCtxt coherence conditions imply the well-formedness of turns: there will be exactly one True hole, or no holes.

Turns support a method for reconstructing a functor value from a list of situated values. The Turns all have the same functor structure, but each has its one True in a different hole. If we were to overlay all of the turns atop one another, each would contribute one True hole to form a functor value full of
By merging two functors with the `fzip` method, we can repeat the process to overlay all of the turns within our list.

```haskell
overlay :: FunctorZip f ⇒ [(Turn f, v)] → f v → f v
overlay [] = id
overlay ((tn, v):ps) = fzipWith upd tn ◦ overlay ps
  where upd b prevHole = if b then v else prevHole
```

The implementation of the `sequenceAlgebra` combinator can be understood by examining its resulting algebra. The resulting algebra has two primary steps: first, provide the sub-computations with their turn within the structure of the current term (via `childTurns`), and second, augment the computation for the current term to report its results in the `Writer` output (`addReport`). Appendix D traces this code during the evaluation of a concrete example.

```haskell
sequenceAlgebra :: MonadSequence f a b m
  ⇒ Algebra f (m a) → (a → f a → Algebra f b)
  → Algebra f (m (a, b))
sequenceAlgebra alg idxAlg f_mb =
  addReport f_mb idxAlg
  (alg (fmap (liftM fst) (fzipWith (RW.local ◦ const) (childTurns f_mb) f_mb)))
```

```haskell
addReport :: MonadSequence f a b m
  ⇒ f x → (a → f a → Algebra f b) → m a → m (a, b)
addReport shape idxAlg m = RW.pass (do
  (a, output) ← RW.listen m
  let f_ab = overlay output (fmap (const ⊥) shape)
      b = idxAlg a (fmap fst f_ab) (fmap snd f_ab)
      results = (a, b)
      turn ← RW.ask
  return (results, const [(turn, results)]))
```

To accomplish the first step, the environment monadic feature `RW.local` is used to inform the sub-computations of their position in the current term. The computations use the turn from the environment to specify the position of their result within the current term. The current term gathers these results from the `Writer` output and repeats the process at its layer of the traversal.

The `addReport` function defines the more complicated second step and deserves further explanation. The `shape` argument is used only for its structure, not its contents. The second argument of the combinator is an `idxAlg`, an algebra which is indexed by the results of the first. The `m` argument is the computation

5 That is assuming our list of turns is a complete description of the functor value. If not, then there would be some holes not containing `True` for the turns that were missing from the list.
generated by the first algebra for the current node. Because of the recursive
nature of the algebra, the sub-computations within \( m \) were already manipulated
by \texttt{addReport}, so they will generate their results as \texttt{Writer} output. The use of
\texttt{pass} will be explained last, though it appears first.

The computation first uses the \texttt{Writer} feature \texttt{listen} to collect the results
of the sub-computations in \texttt{output}. This list of turns and values is used to
reconstruct via the \texttt{overlay} function the results of both the first and the second
algebra for the sub-terms. Only if the results are incomplete will the \( \perp \)-filled
\texttt{shape} be used; otherwise, the actual result values will replace all error values.
The indexed algebra generates the result for this node with the reconstructed
results of each algebra for the sub-terms – the second traversal takes place
within the monadic side-effects of the first (though not in a way that it affects
the first algebra). Finally, the \texttt{Writer} feature \texttt{pass} uses the second part of the
return pair to write the current situated results to the output and consume the
output of the children via \texttt{const}. If the children’s output were not consumed,
it would be passed along to all ancestors of the current node, which would
pollute our intended use of the \texttt{Writer} output.

\[
\text{type Output } f \ a \ b = [(\text{Turn } f, (a, b))] \\
\text{type SequenceT } f \ a \ b = \text{RWT } (\text{Turn } f) \ (\text{Output } a \ f \ b)
\]

\[
\text{class } (\text{FunctorContext } f, \text{FunctorZip } f, \\
\quad \text{MonadRW } (\text{Turn } f) \ (\text{Output } f \ a \ b) \ m ) \\
\Rightarrow \text{MonadSequence } f \ a \ b \ m \ |
\quad m \rightarrow f \ a \ b
\]

\[
\text{instance } (\text{FunctorContext } f, \text{FunctorZip } f, \\
\quad \text{MonadRW } (\text{Turn } f) \ (\text{Output } f \ a \ b) \ m ) \\
\Rightarrow \text{MonadSequence } f \ a \ b \ m
\]

\[
\text{runSequenceT :: Monad } m \Rightarrow (\text{SequenceT } f \ a \ b \ m) \ (a, b) \rightarrow m \ b
\]

\[
\text{runSequenceT } m = \\
\quad \text{liftM } (\text{snd } \circ \text{fst}) \ (\text{runRWT } m \ (\text{error} \ \text{‘runSequenceT: top turn’}))
\]

The \texttt{SequenceT} monad transformer is a thin wrapper around \texttt{RWT}. Note that
\texttt{runSequenceT} provides an initial environment and ignores the first component
of the computation’s result. The \( \perp \)-valued environment is never forced by
correct use of the combinator. The first component of the result is introduced
only to implement the combinator; ignoring it prevents implementation details
from polluting the carrier space. The \texttt{MonadSequence} class simplifies the pro-
grammer interface by collecting the \texttt{FunctorContext}, \texttt{FunctorZip}, and \texttt{MonadRW}
class constraints. The class has no methods because, unlike \texttt{switchAlgebra}, the
user does not interact with the \texttt{MonadSequence} interface.
7 Related Work

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. Cartwright and Felleisen [1994] achieve a similar result, allowing orthogonal extension of language semantics, using extended direct semantics rather than monads and monad transformers, to structure the computational effects. In this and the existing MMS work, the focus is on combining semantics for orthogonal language constructs, but for a common task, such as typechecking, interpretation, or compilation.

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

Meijer and Jeuring [1995] demonstrate a combination of the catamorphism and monad abstractions. They introduce the \texttt{mfoldr} recursion operator, a variant of the catamorphism that performs a fixed pattern of monadic binds on behalf of the algebra. The operator captures the form of semantic algebras that are unconcerned with the monadic features of the sub-terms’ computations, like \texttt{Evaluation.Products.phi} from Figure 2(c). An equivalent operator can be defined in InterpreterLib.

\begin{verbatim}
mfoldr :: (Traversable f, Monad m) \Rightarrow (f a \rightarrow m a) \rightarrow Fix f \rightarrow m a
mfoldr mphi = cata (\lambda x \rightarrow Traversable.sequence x >>= phi)
\end{verbatim}

In accord with the focus on manipulating semantic algebras, InterpreterLib realizes \texttt{mfoldr} as an application of \texttt{cata} to a derived semantic algebra. The semantic algebras with the same expressive power as \texttt{mfoldr} are those that factor through \texttt{Traversable.sequence}. The generic distribution of the monad over the syntactic functor emphasizes that the argument to \texttt{mfoldr} is unaware of the monadic features of the sub-terms’ computations. Though this special form of semantic algebra is, as Meijer and Jeuring [1995] lamented, “often too specific to be useful,” InterpreterLib smoothly captures it.

Lämmel and Visser [2002] use first-class strategies to represent common traversal patterns. This work is similar to the Strafunski [Lämmel and Visser, 2003] Haskell library and to Stratego [Visser, 2001], a domain-specific language for program transformations. Of particular note, the \texttt{letTU} and \texttt{letTP} strategy
combinators are similar in purpose to the sequenceAlgebra algebra combi-
nator introduced in this article. Those combinators can be understood to se-
quencer two algebras so that the result at the current node from one monadic
algebra is available to a second monadic algebra. The sequenceAlgebra combi-
nator does the same, but it also provides the results at the current node’s
sub-terms, and thus is more expressive. For example, the indexed netlist syn-
thesis algebra from Subsection 5.3 must access the types of the sub-terms. To
achieve this with the letTU combinator, the type-checking algebra must carry
the derivation of a type instead of just the type. This is disappointing because
the type-checking algebra must now be obfuscated in order to be reused. The
sequenceAlgebra combinator derives the extra information from the natural
definition of the first algebra, avoiding the trade-off between concision and
reuse.

A more expressive interface for the sequenceAlgebra combinator has been
proposed.

data Annotate a f x = Annotate a (f x)
type Decorated f a = Fix (Annotate f a)

sequenceAlgebra ::
... ⇒ Algebra f (m a)
   → (Decorated f a → Algebra f a b)
   → Algebra f (m b)

This interface allows the indexed algebra to inspect not only the first alge-
bra’s intermediate results at the current location within the AST and at the
immediate sub-terms, but also at every location within the entire term rooted
at the current location. The argument to the second algebra is a decorated
version of the current term where every constructor also has an associated
value of type a. This version of the combinator subsumes the expressiveness
of our implementation and enriching the given version requires little extra
code. We present the simpler one both because it conveys the same essen-
tial concepts and implementation strategy and because our research projects
employing InterpreterLib have not yet demanded the richer interface.

Lämmel et al. [2000] also use the terminology “algebra combinators,” but ap-
ply it to what we call generic algebras, algebras that exhibit polymorphism
over the functor. Lämmel’s paper defines only one syntactic signature, thus
functor-polymorphism is not evident. We did not discuss the primary generic
programming facilities of InterpreterLib in this article; they can be summa-
rized as an integration of the “updateable fold algebras” concept from [Lämmel
et al., 2000] with the sum functor approach to extensible syntax.

We have shown how using the sequenceAlgebra combinator can augment the
catamorphic recursion operator to effect the paramorphism recursion operator.
In this regard, the concerted use of the algebra combinator and the catamorphism shares a purpose with the zygomorphism recursion operator [Malcolm, 1990], which takes two algebras and provides the results of the first to the second. The zygomorphism does not support the sequencing of multiple algebras, since it is not an operation on algebras. It also does not directly support monads. The sequenceAlgebra combinator provides an interesting comparison with the zygomorphism recursion operator because they achieve similar expressiveness by dissimilar means, the most predominant difference being the support of, and reliance on, monads by the algebra combinator.

The generation of contexts in the implementation of the sequenceAlgebra combinator strongly resembles the partial differentiation operator for data types, introduced by Abott et al. [2005], and consequently the zipper data types of Huet [1997]. Differentiation in InterpreterLib is specialized to the requirements of implementing sequenceAlgebra. As a result, the FContext type and FunctorContext class are straight-forward mechanisms for expressing derivatives but far more loosely-typed than the constructions of Abott et al. [2005]. Improving the implementation along that dimension would complicate the interface InterpreterLib presents to the user by exposing the richer data types for derivatives. The current implementation is both fully functional and well encapsulated behind the library interface: only FunctorContext f constraints are exposed. The developing Associated Types extension from Chakravarty et al. [2005] may offer strongly-typed derivatives without propagating the derivative types beyond the library interface.

PolyP [Jansson and Jeuring, 1997] uses non-recursive functor representations of data types to define a variety of generic programs. A pre-processor is used to generate instances of Haskell type classes for data structures; InterpreterLib includes a similar tool. An alternative approach published in a series of “Scrap Your Boilerplate” papers [Lämmel and Jones, 2003, 2004] utilizes a dynamic cast operator to embed generic programming combinators within Haskell, eliminating the need for the non-recursive representations that enable static generic programming.

Sheard and Pasalic [2004] use the fixed-point-of-functor representation of data types (using the terminology “two-level types”) to develop a generic version of an efficient unification algorithm. The addMeta semantics from Section 5 is a similar example of this technique. In a discussion of the history of two-level types, Sheard and Pasalic note that “uniform control structures, like cata, turned out to be too inflexible”. It is the inflexibility of cata that motivated the development of the switchAlgebra and sequenceAlgebra combinators.
8 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 denotation functions in a modular fashion. In this work we have shown how algebra combinators extend this approach to defining compositions of different semantics 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 semantics 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


A Syntactic Functor Type class Instances

Declaring instances of requisite functor classes involves a significant amount of boilerplate. We deliver alongside InterpreterLib a code generator that processes a simple specification language representing syntactic functors and generates Haskell data type definitions and functor class instances integrating the syntactic functor into the InterpreterLib framework. This appendix presents some of the generated code.

Presented below are the Arith functor and instances for the standard Functor, Traversable, and Foldable type classes. The fmapDefault and foldMapDefault functions are defined in the standard Haskell libraries witnessing how the Traversable property implies the Functor and Foldable properties.

data Arith carrier = Num Int | Add carrier carrier

instance Functor Arith where
  fmap = fmapDefault

instance Traversable Arith where
  traverse f (Num arg0) = pure Num <$> pure arg0
  traverse f (Add arg0 arg1) = pure Add <$> (f arg0) <*> (f arg1)

instance Foldable Arith where foldMap = foldMapDefault

Instances for the FunctorContext and FunctorZip type classes are generated for using sequenceAlgebra.

instance FunctorZip Arith where
  fzip (Num arg0) (Num _) = Num arg0
  fzip (Add arg0L arg1L) (Add arg0R arg1R) =
    Add ((,)) arg0L arg0R ((,)) arg1L arg1R

50
instance FunctorContext Arith where
fcontexts (Num arg0) = Num arg0
fcontexts (Add arg0 arg1) =
    Add (aug_arg0 (fc_REC arg0)) (aug_arg1 (fc_REC arg1))
where fc_REC v = (v, Self)
aug_arg0 (hole, ctxt) = (hole, Add ctxt (Sibl arg1))
aug_arg1 (hole, ctxt) = (hole, Add (Sibl arg0) ctxt)

And lastly, the “smart constructors” are generated using the injF method of the SubFunctor class.

mkNum :: SubFunctor Arith f ⇒ Int → Fix f
mkNum arg0 = (In ◦ injF) (Num arg0)
mkAdd :: SubFunctor Arith f ⇒ Fix f → Fix f → Fix f
mkAdd arg0 arg1 = (In ◦ injF) (Add arg0 arg1)

B The Indexed Reader/Writer Monad

We designed InterpreterLib to inter-operate with the Monad Transformer Library (mtl), which is part of the Haskell standard libraries. The use of functional dependencies within the mtl conflicts with the intended use of the algebra combinators. We introduce an new monad transformer with a phantom type to resolve the issue.

The instance declarations in the mtl reflect the assumption that any occurrence of a monad transformer in a transformer stack is the only occurrence. For example, if ReaderT Int appears in the stack defining a monad M, then the monad satisfies the constraint MonadReader Int M. This assumption saves the mtl user numerous type annotations, but it also has a negative effect on modularity since one monad cannot satisfy two different MonadReader constraints. The algebra combinators both use a monadic environment feature and sequenceAlgebra also uses a monadic output feature. If the combinators were to use the corresponding MonadReader and MonadWriter classes from the mtl, then the monads of the combined algebras could not also use those features. If they did, the repeated constraints would violate the functional dependencies of the mtl instance declarations. The use of the algebra combinators would conflict with most algebras, since the use of MonadReader is so common.

As a workaround, we introduce a new monad transformer and type class, duplicating the monadic features of MonadReader and MonadWriter.
module RW where

class (Monoid w, Monad m) ⇒ MonadRW tag r w m | tag m → r w where
  ask :: tag → m r
  local :: tag → (r → r) → m a → m a
  tell :: tag → w → m ()
  listen :: tag → m a → m (a, w)
  pass :: tag → m (a, w → w) → m a

newtype RWT tag r w m a
  = RWT (r → m (a, w))

runRWT :: tag → RWT tag r w m a → r → m (a, w)
runRWT _ (RWT f) = f

instance (Monoid w, Monad m) ⇒ MonadRW tag r w (RWT tag r w m) where
  ask _ = RWT (λr → return (r, mempty))
  local _ g (RWT f) = RWT (f ∘ g)
  tell _ w = RWT (λr → return (((), w))
  listen _ (RWT f) = RWT (λr → do
                              ~(a, w) ← f r
                              return ((a, w), w))
  pass _ (RWT f) = RWT (λr → do
                          ~(a, fltr), w) ← f r
                          return (a, fltr w))

commute ((a, w’), w) = ((a, w), w’)

instance (Monoid w’, MonadRW tag r w m) ⇒ MonadRW tag r w (RWT tag’ r’ w’ m) where
  ask tag = lift (ask tag)
  local f (RWT g) = RWT (local f ∘ g)
  tell tag = lift ∘ tell tag
  listen tag (RWT f) = RWT (λr → commute ‘liftM’ listen tag (f r))
  pass tag (RWT f) = RWT (λr → pass tag (commute ‘liftM’ f r))

The first MonadRW instance is for when the tag phantom types match, and the second is for when they do not. The details of the method implementations are conventional. We have omitted the instances for lifting all of the mtl monadic interface classes, such as MonadReader, through the RWT transformer; they are essential but also numerous and straightforward.

Other than the phantom type argument tag, this monad is a conventional type for implementing the environment and output side-effects. The phantom type allows the same class to be reused by both algebra combinators; it is used to avoid code duplication without violating functional dependencies. In the body
of the article, we have omitted the use of the phantom type from all code for simplicity. Both combinators have a corresponding token type that serves as the phantom argument to differentiate the respective \texttt{MonadRW} constraints.

\begin{verbatim}
data SwitchToken = SwitchToken
data SequenceToken = SequenceToken
\end{verbatim}

Every occurrence in the article of \texttt{MonadRW}, its methods, \texttt{RWT}, and \texttt{runRWT} should actually be immediately followed by the appropriate token type argument.

These token types and the algebra combinator interfaces have been provided only as simplified defaults. The algebra combinators themselves, their monad interface classes, and their run functions can also expose the phantom type, as more configurable variants. This is indeed necessary for applications of \texttt{sequenceAlgebra} to be left-nested, as this requires one concrete monad to satisfy multiple, separate \texttt{MonadSequence} constraints. Admitting left-nesting enables more flexible re-use of sequenced algebras.

\section*{C Monadic Utilities}

Both type-checking and evaluation manage an environment for binding names to parts of the value space and potentially raise exceptions. We capture both the environment and exceptions with monads. Each denotation uses the same interface, so we create two small modules of utility functions.

\begin{verbatim}
module Env where

import Prelude hiding (lookup)

We will import this module qualified, so we introduce \texttt{lookup} in this module as a small wrapper around the function from the \texttt{Prelude} of the same name.

An environment of \texttt{v}s is a list of (\texttt{String},\texttt{v}) pairs. The function \texttt{lookup} yields a computation of a \texttt{String}’s associated \texttt{v}.

\begin{verbatim}
type Env v = [(String, v)]

lookup ::
  (Error e, MonadError e m, MonadReader (Env v) m)
  ⇒ String → m v
lookup n = do
  env ← ask
  case Prelude.lookup n env of
    Just x → return x
    Nothing → Error.throwStr ("Undefined:Env.lookup: " ++ n)
\end{verbatim}

53
The function \texttt{with} executes a computation with a new environment, and the function \texttt{extend} executes a computation with an additional binding.

\texttt{with :: MonadReader (Env v) m \Rightarrow Env v \to m a \to m a}

\texttt{with e = local (const e)}

\texttt{extend :: MonadReader (Env v) m \Rightarrow (String, v) \to m a \to m a}

\texttt{extend = local \circ (:)}

In accord with our intent of maximizing reuse, the data structure for exceptions should also be extensible. However, the extensibility of exceptions in Haskell is an unsolved problem. We raise either a simple message or an empty exception for the purposes of this example.

\texttt{module Error where}

\texttt{throwStr :: (Error e, MonadError e m) \Rightarrow String \to m a}

\texttt{throwStr x = throwError (strMsg x)}

\texttt{raise :: (Error e, MonadError e m) \Rightarrow m a}

\texttt{raise = throwError noMsg}

This minimal interface for exceptions is particularly appropriate since we do not catch any exceptions.

\textbf{D A Concrete Sequence Algebra Example}

Consider a \texttt{simplify} function implementing an optimization over the \texttt{Arith} syntactic functor that eliminates additions involving zero. For instance, the optimization simplifies \((0+1)+(2+3)\) to \(1+(2+3)\). (Throughout this appendix, expressions are written in concrete syntax for brevity.) The essence of the optimization can be defined as an indexed algebra.

\texttt{simplifyPhi :: SubFunctor Arith f \Rightarrow f (Fix f) \to f (Fix f) \to Fix f}

\texttt{simplifyPhi sub_terms f_x = maybe (inn f_x) arith (prjF f_x)}

where

\texttt{arith (Num i) = mkNum i}

\texttt{arith (Add x' y') =}

\texttt{case fromS x of}

\texttt{Just (Num 0) \to y}

\texttt{_ \to case fromS y of}

\texttt{Just (Num 0) \to x}

\texttt{_ \to mkAdd x' y'}

\texttt{where Add x y = unsafePrjF sub_terms}
The paramorphism of \(\text{simplifyPhi}\) defines \(\text{simplify}\). Recall from Section 5.2 that the paramorphism can be implemented in terms of the \(\text{sequenceAlgebra}\) combinator. Accordingly, the catamorphism of \(\phi\) followed by the appropriate monadic run functions also defines \(\text{simplify}\). The eavesdrop function is explained immediately below.

\[
\phi = \text{eavesdrop} \ "\text{entire}" \\
( \text{eavesdrop} \ "\text{first}\" \ (\text{liftMAlg} \ \text{idAlg}) \ "\text{sequenceAlgebra}\" \\
\lambda \ \text{me} \ \text{sub_terms} \ f_x \rightarrow \text{simplifyPhi} \ \text{sub_terms} \ f_x )
\]

The eavesdrop function exposes the bookkeeping of the \(\text{sequenceAlgebra}\) combinator. Functionally, it is the identity function on the algebra \(\phi\). It introduces a monadic output effect by peeking into the \(\text{sequenceAlgebra}\) combinator’s monadic layer and copying what it sees into a separate \(\text{Writer}\) output.

\[
\text{eavesdrop} ::
( \text{MonadSequence} \ f \ \text{a} \ \text{b} \ \text{m}, \ \text{MonadWriter} \ \text{String} \ \text{m}, \\
\text{Show} \ (\text{Turn} \ f), \ \text{Show} \ a, \ \text{Show} \ b, \ \text{Show} \ c \\
) \Rightarrow \text{String} \rightarrow \text{Algebra} \ f \ (\text{m} \ \text{c}) \rightarrow \text{Algebra} \ f \ (\text{m} \ \text{c})
\]

\[
\text{eavesdrop} \ s \ \phi \ x = \ do \\
\text{turn} ← \ \text{RW.ask} \\
(z, \ \text{output}) ← \ \text{RW.listen} \ (\phi \ x) \\
\text{Writer.tell} \ (\text{show} \ (s, \ \text{turn}, \ \text{output}, \ z)) \\
\text{return} \ z
\]
The table in Figure D.1 lists the output generated by applying the cata-
morphic extension of the phi algebra to the term \((0+1)+(2+3)\). Each column

corresponds to a part of the tuple include in eavesdrop’s monadic output. The

rows in the table are in the same order they are generated. In this table, ⊥ abbreviates error "runSequenceT::top::turn".

The rows tagged with first are those generated as output due to the evaluation

of the first algebra. Since that is the identity algebra idAlg, the output
column indicates which of the term tree’s nodes the row records the traversal of. Similarly, for the entire rows, the first part of the z pair is the result of the identity algebra at the same node, and so can be used to identify the corresponding node. Thus, it can be observed that the order of the results are in accord with the “bottom-up” catamorphic recursion scheme. Since the first and entire tags occur in lock-step, it is clear that the combinator’s result algebra invokes the first algebra once as part of its own monadic computation per node.

The turn column is the data used by the addReport function to situate its results within the structure of the parent node. For each column, we can validate the correctness of the Turn value. For instance, the 0 node occurs in the original term on the left of a + node, as its row’s turn value indicates. On the other hand, the 1 occurs on the right. Also, note that all turns have exactly one True hole, as required.

The terminal nodes of the original tree can be identified by the first rows with an empty output value. Since there are no sub-terms to report their results, the first algebra sees no output.

The first rows record information seen before the combinator has acted at the current node, and the entire rows record information from afterwards. Since the eavesdrop "entire" function is applied to an application of sequenceAlgebra, this is the value of the combinator’s output as it is being generated by the algebra at the current node and before it is consumed by the node above it. The eavesdrop "first" function, on the other hand, records the output (generated by the algebra at the child nodes) as seen by the node consuming it. This is why the situated output in the entire rows’ output column is always both a singleton list and includes the value from the z column. Similarly, the z column records only the results of the first algebra in the first rows, but records the results of the index algebra as well in the entire rows.

The rows corresponding to the root of the term tree include ⊥ values, all of which are Turns. This is because there is no turn corresponding to the context of the root: it has no parent node to be situated within. These undefined values are never forced by the combinator because the overlay function is
only applied during the traversal of a parent node to the output gathered from the childrens’ computations. There is no parent of the root node and hence no call to overlay. The root’s turn is never forced for the same reason that it is undefined.

Consider the output column of the penultimate row, which corresponds to the first algebra’s evaluation at the root node. This node, as a + term, has two children and hence two elements in the output list – one generated by each child. This information is consumed by the combinator’s evaluation at the root node, which corresponds to the last row in the table. The computation applies the overlay function to the output seen by the first algebra (from the penultimate row) yielding the value Add (0+1, 1) (2+3, 2+3). Together with the first algebra’s results (the z column of penultimate row), these are the arguments to the indexed algebra: me is bound to (0+1)+(2+3), sub_terms is bound to Add (0+1) (2+3) (via fmap fst), and f_x to Add 1 (2+3) (via fmap snd). The indexed algebra yields 1+(2+3), which the combinator pairs with the first algebra’s result.