Writing Composable Software with InterpreterLib

Mark Snyder, Nicolas Frisby, Garrin Kimmell, Perry Alexander

Information and Telecommunication Technology Center
The University of Kansas, Lawrence, KS USA 66045
{marks,nfrisby,kimmell,alex}@ittc.ku.edu

Abstract. InterpreterLib is a Haskell library for building and composing modular syntactic and semantic definitions. We identify two forms of composition within this domain. Composition *along syntax* combines semantics for denoting differing term spaces to a common denotational domain. Composition *along semantics* combines semantics for denoting a common term space to differing domains. We demonstrate these composition mechanisms by applying InterpreterLib to examples and then relate our experiences with InterpreterLib implementing tools for the Rosetta language.

1 Introduction

A denotational semantics maps terms in a term space to a denotational domain, but traditional composition of denotations is prohibited by mismatches in either the term space or the domain. Accordingly, we identify two separate kinds of composition that address this problem: composition along syntax composes semantics denoting differing term spaces to a common denotational domain. Composition along semantics composes semantics for denoting a common term space to differing domains. InterpreterLib defines composition mechanisms for both. While the library was developed as a tool for denotational semantics, it supports definition and composition of all analyses over inductive datatypes.

InterpreterLib implements two existing techniques for composition along syntax. Liang et al. [13] demonstrated monad transformers as a solution to the lack of modularity in semantic domains. Gayo et al. [4] then introduced an extensible syntax solution derived from the initial algebra semantics of inductive datatypes. As a result, the library enables the definition of syntactic and semantic components and their re-use.

Both techniques derive modularity from qualified types [6, 26], as implemented in Haskell [15]. In modular monadic semantics, type classes serve as the modular interface to monadic operators. Semantic definitions remain modular so long as the monadic type remains qualified, admitting composition by collecting type constraints over the monadic interface classes. Once the developer determines a concrete monad by composing suitable monad transformers, the library of type class instances relating the monad transformers and monadic interfaces automatically derives the executable implementation of the monadic operators at the cost of further modularity.

Gayo et al. [4] use a type class to overload the constructors and destructors of the concrete term space, requiring only that particular syntactic constructs be embedded in that term space. Again, syntactic definitions can be composed so long as the term space type remains qualified. The developer specifies a concrete term space using a composition operator for syntactic constructs and an associated set of type class instances derives the implementation of the syntactic modularity interface at the cost of further modularity. Section 3.1 includes a concrete example.

InterpreterLib also implements algebra combinators [27, 28] for composition along semantics. Algebra combinators build composite semantics from component semantics over a common term space. In the library, semantics are specified by algebras, which are functions of a particular shape. The modularity is derived not from qualified types, but from an algebra's implicit recursion. Since an algebra is not directly recursive, it admits more manipulation.

2 Basic Semantics

We demonstrate composition via InterpreterLib and present the library's definitions with a running example. We first develop two distinct syntaxes (an integer language and a Boolean language) and a common analysis (evaluation). We compose the syntax and semantics along syntax to construct an interpreter for a combined language over integers and Booleans. Next, we demonstrate the usefulness of composition along semantics by adding a syntactic construct for overloaded operators and defining its evaluation semantics in terms of a type-checking semantics using an algebra combinator.

We start by defining two separate evaluation semantics for language constructs over integers and Booleans. InterpreterLib represents syntactic constructs with *syntactic functors* and semantics with *semantic algebras*. This is a consequence of adopting the initial algebraic semantics for the term space. In Haskell, functors are data types of kind $* \rightarrow *$ that are instances of the `Functor` class, providing the operation `fmap :: (a → b) → (f a → f b)`.

2.1 Integers

We define the syntax and semantics of the integer language (fig. 1) with a syntactic functor and a semantic algebra. Both definitions are standard except that they are not directly recursive. The recursion will be introduced using a fixed-point operator as the final step.

The `Integers` functor introduces the syntactic constructors for basic integer arithmetic. Using the type argument `t` to represent recursively-defined subterms rather than specifying that they specifically be `Integers` or any other specific functor allows us to define the `Integers` functor as a single entity while still allowing us to later construct syntax containing `Integers` and other functors.

```
-- Integers language
data Integers t =
    Add t t | Sub t t | Num Int
$(derive makeAll ''Integers)

phi :: Algebra Integers Int
phi (Add x y) = x + y
phi (Sub x y) = x - y
phi (Num i) = i
```

```
--InterpreterLib definitions
data Fix f = In (f (Fix f))

type Algebra f a = f a -> a

cata ::
  Functor f => Algebra f a
            -> Fix f -> a
cata phi (In x) =
  phi (fmap (cata phi) x)
```

Fig. 1. Syntax and semantics for the `Integers` language and some `InterpreterLib` definitions

For example, an addition `Add x y` does not require `x` and `y` to necessarily be `Integers` themselves–they may be function calls, record lookups, or any other syntax that, if we define the usual evaluation semantics, we'd expect to represent numerical quantities. In this way, we consider `Integers` to be one open component of a BNF grammar that we later construct by combining all the functors that fully define all alternatives of the grammar, as we shall see in section 3.1.

The type-level fixed point operator `InterpreterLib.Fix` calculates the type of terms in the language generated by a syntactic functor's analogous grammar. Thus, the term $3 + 4$ is represented as `In (Add (In (Num 3)) (In (Num 4))) :: Fix Integers`.

The type synonym `InterpreterLib.Algebra` specifies the form of semantic algebras. An `f`-*algebra* captures the semantics of the structure of a functor `f` by specifing how to fold that structure into the algebra's *carrier* `a`. The `phi` algebra over `Integers` gives meaning to the structure of `Integers` by mapping the constructors directly to arithmetic operations.

As defined, the semantic algebra `phi` can only be applied to a single layer of `Integers` structure: `phi (Add 3 4)` reduces to 7. The catamorphic recursion operator, `InterpreterLib.cata`, can extend an `f`-algebra to a function applicable to terms in the language generated by the syntactic functor `f`.

With these definitions, `phi` denotes terms in `Fix Integers` to values.

```
*Integers> let denote = cata phi
*Integers> denote (In (Num 3))
3
*Integers> denote (In (Add (In (Num 3)) (In (Num 4))))
7
```

The Template Haskell [21] splice `$(derive makeAll ''Integers)` invokes the InterpreterLib code generators. The `makeAll` code generator derives as much boilerplate as possible for a functor. The derived code supports other operators provided by InterpreterLib for manipulating and composing functors and algebras; some of these will be demonstrated later in this article.

2.2 Booleans

Next we define the syntax and semantics for Booleans in figure 2. The computational and polymorphic concerns regarding the conditional construct necessitate a more intricate semantic algebra. The semantics of the conditional exposes both that there may be other types of value in the value space besides Booleans and that the computations of each branch may have side-effects. Thus, the semantics must explicitly manage the qualified types for the monad and the value space.

```
data Booleans t = Tru | Fls | If t t t
$(derive makeAll ''Booleans)

data VBool v = VBool Bool deriving (Show, Eq)

phi :: (Monad m, SubFunctor VBool v) => Algebra Booleans (m (Fix v))
phi Tru = return $ toS $ VBool True
phi Fls = return $ toS $ VBool False
phi (If mc mt mf) = do
    v <- mc
    case fromS v of
      Nothing -> fail "if-guard was not Boolean"
      Just (VBool b) -> if b then mt else mf
```

Fig. 2. Syntax and semantics for the `Booleans` language

The `Booleans` syntactic functor represents another group of alternatives from a single-sorted BNF grammar. We describe the `VBool` functor below. The Template Haskell splice is exactly the same as for `Integers`.

The Boolean semantics are complicated by two modularity concerns. First, the conditional expression is traditionally non-strict in its alternatives. The evaluation of a branch must be guarded by the conditional. Even though Booleans themselves introduce no side-effects, the semantics must explicitly manage the monadic carrier in order to respect the side-effects of any language constructs it may be composed with. Second, the algebra `phi` allows the branches to compute a value other than a Boolean. The `InterpreterLib.SubFunctor` constraint (to be discussed shortly) requires merely that the value space `Fix v` includes Booleans instead of requiring that the value space be exactly Booleans. The value functor `VBool` is solely defined to make this embedding precise.

The semantic algebra for `Integers` does not suffer these complications because arithmetic operations have no traditional interactions with side-effects and do not involve other types. In fact, encapsulating side-effects in a monad and the embedding of one value space in a larger one via `SubFunctor` allows the pure, uniformly-typed `Integers.phi` to be promoted in a natural way into a richer algebra carrying any monad and any value space that embeds Integers. This promotion is a necessary step in the composition of the Boolean and Integer semantics, which we demonstrate next.

3 Composing along Syntax

Finally, we compose the syntax, semantics and value spaces of the previously defined sub-languages. The composition mechanism for syntactic functors corresponds closely with the modularity mechanism from the `SubFunctor` class. The promotion of the `Integers.phi` algebra necessary for the composition with `Booleans.phi` is carried out via two re-usable algebra operators, `pureAlg` and `embedAlg`. This sort of operator motivates the foundation of InterpreterLib: first-class syntax and semantics.

3.1 Syntactic Composition along Syntax

Syntactic functors are composed with the functor sum operator and the void functor, `InterpreterLib.:$:` and `InterpreterLib.FVoid` respectively. `f:$:g` is a syntactic functor with all the properties shared by `f` and `g`, such as `Functor` and `Traversable`.

The `InterpreterLib.SubFunctor` class relates one type with another that embeds it. An embedding is witnessed by the two class methods `injF` and `prjF`. This class is a modularity interface that directly corresponds to the composition mechanism of `:$:`. InterpreterLib declares instances of `SubFunctor` for `:$:` and `FVoid` such that `SubFunctor f fs` holds if `fs` is a right-nested sum functor terminated by the void functor and `f` occurs as the left operand of one of the functor sums. In other words, the derivation of the `SubFunctor` relation is automatic if the type hosting the embedding is structured as a list of possible functors.

The composition and modularity mechanisms of the term space should be re-used to achieve extensibility in the value space, as `Booleans.phi` demonstrates above. This re-use comes as no surprise given the convention of defining the value space as a sort in a BNF grammar.

We compose the syntactic functors `Integers` and `Booleans` as `Integers :$: Booleans :$: FVoid`. While the ordering of the functors does induce a particular type to which we must adhere when writing functions over the functor sum, the ordering of the functors (other than FVoid at the end) is arbitrary. The mechanism for projecting out of the sum is a matter of chasing labels, providing an appropriate interface to extracting the value that is unaffected by the order of functors within the sum.

3.2 Semantic Composition along Syntax

InterpreterLib defines a composition mechanism for semantic algebras that corresponds to the functor sum syntactic mechanism. `InterpreterLib.(@+@)` (fig. 3) applies its first algebra if the sum functor value is an `L`, and its second if it is an `R`.

```
infixr 5 :$:                   class SubFunctor f g where
data (f :$: g) a =               injF :: f a -> g a
   L (f a) | R (g a)             prjF :: g a -> Maybe (f a)
unFSum (L x) = Left x
unFSum (R x) = Right x         toS = inn . injF
                               fromS = prjF . out
data FVoid a

infixr 5 @+@
(@+@) :: Algebra f a -> Algebra g a -> Algebra (f :$: g) a
fAlg @+@ gAlg = either fAlg gAlg . unFSum

voidAlg :: Algebra FVoid a
voidAlg = undefined
```

Fig. 3. `InterpreterLib` modularity interface and mechanisms for composition along syntax

The `InterpreterLib.voidAlg` algebra is necessarily and sufficiently undefined as there is no way to construct an `FVoid` term.

The composition of `Integers.phi` and `Booleans.phi` cannot be directly achieved with the algebra sum operator. Its type shows that the carriers of the two algebras must be the same, which is not yet the case. As discussed above, `Integers.phi` is unaware of monadic side-effects and values other integers, since it is a simpler denotation. It must be promoted to handle the concerns before being summed with `Booleans.phi`.

Algebras can be composed even if they do not all utilize monads for side-effects or use precisely the same result type. Algebra carriers can be lifted from pure to monadic forms via `InterpreterLib.pureAlg`, and from a concrete type to a larger 'host' type via `InterpreterLib.embedAlg` (fig. 4). `pureAlg` converts the pure carrier of an `f`-algebra to a monadic carrier if the functor is an instance of the `Traversable` class, which provides the distributive operator `sequence :: Monad m => t (m a) -> m (t a)`. The `makeAll` code generator derives instances of the `Traversable` class for syntactic functors that do not include function spaces.

The `embedAlg` operator promotes the carrier from a concrete type to a host type that embeds the concrete type, relying on the monad for handling projection failures. The first two arguments identify which functor will be used to represent the pure carrier in the embedding value space. The third argument specifies which exception to raise in the monad on a projection failure.

Composing these two operators promotes the carrier of the `Integers.phi` algebra so that it can unify with the carrier of `Booleans.phi`. The composite algebra for the combined integers and Booleans language can now be defined.

With these definitions, `composite_phi` can be used to denote terms in the language `Fix (Integers :$: Booleans :$: FVoid)` to monadic computations. The `Either String` monad satisfies the monadic type constraints on the de-

```
pureAlg :: (Traversable f, Monad m) => Algebra f a -> Algebra f (m a)
pureAlg phi = liftM phi . Data.Traversable.sequence

embedAlg :: ( Functor f, MonadError e m, SubFunctor g v
            ) => (a -> g (Fix v)) -> (g (Fix v) -> a) -> e
              -> Algebra f (m a) -> Algebra f (m (Fix v))
embedAlg ctor dtor exn phi =
    liftM (toS . ctor) . phi . fmap project
  where project m = do
          v <- m
          case fromS v of
            Nothing -> throwError exn
            Just g -> return (dtor g)
```

Fig. 4. Some `InterpreterLib` Algebra promotion operators

noted computation, and the `Fix (VNum :$: VBool :$: FVoid)` value space satisfies the `SubFunctor` constraints. These types must be ascribed because the constraints accumulated via the `SubFunctor` and `MonadError` modularity mechanisms only restrict the types without determining a specific type. Semantics defined with InterpreterLib derive their modularity from these type classes and lose that modularity once concrete types have been specified. They remain extensible via the composition operators (functor and algebra sum operators) until recursion is introduced with `Fix` and `cata`.

```
*Interpreter> let denote t = cata composite_phi t
*Interpreter> denote test1 :: M V
Right (In (L (VNum 1)))
*Interpreter> denote test2 :: M V
Right (In (L (VNum 2)))
*Interpreter> denote test3 :: M V
Left "if-guard was not Boolean"
```

The `test1`, `test2`, `test3` terms as defined in figure 5 below are defined using "smart constructors." Deriving with `makeAll` generates these definitions for any functor. Each smart constructor wraps the base data constructor in calls to the injection facilities of the `SubFunctor` class. They are named by prepending "mk" to each constructor name. As a result of using smart constructors, these test definitions specify terms in any language that includes both `Integers` and `Booleans`.

3.3 Summary

To add new features to this language, we would: (1) write syntactic functors for the new language constructors; (2) derive the boilerplate for those functors; (3) write semantic algebras for the new functors; (4) extend the composite algebra

```
data VNum x = VNum Int deriving (Show, Eq)

phiIntegers :: ( MonadError e m, Error e
               , SubFunctor VNum v
               ) => Algebra Integers (m (Fix v))
phiIntegers =
  embedAlg VNum (\(VNum i) -> i) (strMsg "not an integer")
                (pureAlg Integers.phi)

composite_phi ::
  ( MonadError e m, Error e
  , SubFunctor VBool v, SubFunctor VNum v
  ) => Algebra (Integers :$: Booleans :$: FVoid) (m (Fix v))
composite_phi = phiIntegers @+@ Booleans.phi @+@ voidAlg

type M = Either String
type V = Fix (VNum :$: VBool :$:  FVoid)

test1, test2, test3 ::
    (SubFunctor Integers t, SubFunctor Booleans t) => Fix t
test1 = mkIf mkTru (mkSub (mkNum 5) (mkNum 4)) (mkNum 1)
test2 = mkAdd (mkNum 1) (mkIf mkFls (mkNum 2) test1)
test3 = mkIf (mkNum 0) (mkNum 0) (mkNum 0)
```

Fig. 5. Interpreter for the combined `Integers :$: Booleans` language

by summing it with the new algebras. This last step corresponds to extending the composite syntactic functor by summing it with the new syntactic functors.

Notice that none of the original syntactic or semantic definitions require changes (or even recompilation). We also did not have to write any boilerplate code. Old terms defined using smart constructors do not require any changes because their type is only constrained by `SubFunctor` constraints. The results are excellent – modularity is preserved in the term space, in the algebras, and in the value space, and yet the tasks required for changing the language are straightforward, requiring change practically only where actual change is intended.

Compositionality is achieved via modularity interfaces and composition mechanisms. Semantic algebras abstract over the value space and computational structures using the `SubFunctor` and monadic interface type classes. Unification of these restricted type variables collects constraints instead of merging concrete types. The InterpreterLib user uses these constraints to calculate a suitable concrete monad and value space once extensibility is no longer required. The syntactic and semantic definitions themselves allow composition by relying on recursion combinators in lieu of direct recursion. The non-recursive definitions also admit manipulations, such as the carrier promotion achieved via `pureAlg` and `embedAlg`. The user again only introduces recursion once extensibility is no longer required. Because InterpreterLib defines instances of the modularity interface type classes for each of the composition mechanisms, compositionality comes nearly for free, most often requiring only the application of recursion operators and type ascriptions.

4 Composition along Semantics

The composition mechanisms from the previous section, `:$:` and `@+@`, both change the syntactic functor without affecting the carrier of the semantic algebra, as can be observed directly from the types involved in each operator's Haskell signature. These compositon operators are used to combine semantic algebras defining the same semantic analysis for disparate syntactic functors into a composite algebra for the corresponding composite functor; it is composition along syntax. InterpreterLib also provides composition mechanisms for algebras over the same functor but with heterogenous carriers; this is composition along semantics.

4.1 Sequencing Algebras

The principle InterpreterLib mechanism for composition along semantics is the sequence algebra combinator, `InterpreterLib.seqMAlg` (fig. 6). The implementation of the sequence algebra combinator is omitted for this discusion. Instead, we will specify the semantics in two phases. First, we discuss a pure variation of the combinator, `seqAlg`, with no specialized handling of monads. Second, we motivate the support for monadic carriers.

The `seqAlg` combinator builds a composite algebra from an algebra and an *indexed algebra*. The result of the first algebra at each node is made available to the second algebra.

```
seqAlg :: Functor f => Algebra f a -> (a -> f a -> Algebra f b)
                    -> Algebra f (a, b)
seqAlg phi psi f_ab = (a, psi a f_a (fmap snd f_ab))
    where f_a = fmap fst f_ab;   a = phi f_a
```

Fig. 6. The pure sequence algebra combinator

4.2 Example: Checking for the AVL Property

We define an analysis that checks a binary tree for the AVL property. The composition operator used in this example is the pure variation of `seqMAlg` where the monad is assumed to be the trivial identity monad, since the semantics require no side-effects. (The example in section 4.4 makes use of the monad.) This emphasizes the composition semantics of the sequence algebra combinator without involving the preservation of monadic encapsulation.

The AVL property requires that subtree heights differ by at most one. The `Node` functor (fig. 7) is introduced as the branching functor for binary trees such that the shape of a binary tree has type `Fix Node`. An analysis for checking this property can be formulated as the composition of a `Node`-algebra for determining tree height, `heightPhi`, and an indexed `Node`-algebra for requiring a property to hold at all nodes, `everywhere`. The essence of the AVL property is checked by the `within1` predicate, which is applied to the results of the `heightPhi` algebra at each node.

```
data Node x = Leaf | Branch x x

heightPhi :: Algebra Node Int
heightPhi Leaf = 0
heightPhi (Branch l r) =
  1 + max l r

within1 :: Node Int -> Bool
within1 Leaf = True
within1 (Branch l r) =
  abs (l - r) <= 1
```

```
everywhere :: Bool -> Algebra Node Bool
everywhere _ Leaf = True
everywhere here (Branch l r) =
  l && r && here

isAVL :: Fix Node -> Bool
isAVL = snd . cata phi
  where phi = heightPhi ‘seqAlg‘
          (\h f_h ->
              everywhere (within1 f_h))
```

Fig. 7. An AVL check for `Fix Node` trees

The algebra `phi` that determines the `isAVL` catamorphic extension is defined by composing the re-usable component algebras `heightPhi` and `everywhere` with the sequence algebra combinator. Informally, the algebra combinator decorates the tree with the results of the algebra and then applies the indexed algebra. In this case, each node is decorated with the height of the subtree it roots, as calculated by `heightPhi`. Then the `everywhere` indexed algebra is applied to require that the `within1` predicate holds at every node in the tree.

It is more accurate to identify the sequence combinator as a `let` construct for algebras. The bindings are context-dependent in that they take various values as the eventual catamorphic traversal applies the resulting algebra to each node within the regular data type. The concept of decoration misleadingly implies that the traversal has already taken place, whereas the sequence algebra combinator yields an algebra, not a catamorphic extension. The correspondence between the decoration and the `let` construct interpretations of the combinator is in accord with the duality between products and exponents found in curried function semantics.

4.3 Sequencing Algebras with Monadic Carriers

The `seqMAlg` combinator is a specialization of `seqAlg` for the case where the algebra has a monadic carrier (fig 8). Using `seqAlg` would name each monadic computation for use within the index algebra. In contrast, `seqMAlg` names the *result* of those monadic computations. The `FunctorContext` and `FunctorZip` type classes are modularity interfaces for operations that `seqMAlg` performs on the functor; instances are generated by the `makeAll` code generator.

```
seqMAlg :: ( FunctorContext f, FunctorZip f, MonadSequence a f b m
           ) => Algebra f (m a) -> (a -> f a -> Algebra f b)
             -> Algebra f (m (a, b))

class MonadSequence a f b m | m -> a f b

data SequenceT a f b m = ...
runSequenceT :: SequenceT a f b m (a, b) -> m b
```

Fig. 8. The interface to the monadic sequence algebra combinator

The combinator crosses the monadic boundary without compromising the monadic encapsulation: computations are not repeated or run out of order; instead, the algebra combinator introduces a set of monadic interface constraints on the monad and side-conditions on the algebra. The constraints are represented by the `InterpreterLib.MonadSequence` class, and InterpreterLib provides a corresponding monad transformer, `InterpreterLib.SequenceT`. The methods of this type class are only used by the combinator, so the InterpreterLib user need not see them. It is essentially a combination of the `Reader` and `Writer` monads. InterpreterLib also provides the necessary instances for lifting this interface through other transformers and vice versa.

Informally, the side-condition requires that the algebra be a complete, static analysis: at each node, it must bind each sub-computation exactly once. An algebra for type-checking is an archetypal example of such an analysis. Formalizing this condition is current research.

4.4 Example: Resolving Overloaded Operators

The semantics of overloaded operators can be resolved to specific operations once the concrete type is known. We assume an algebra for type checking and compose it with an indexed algebra for evaluation of overloaded operators (fig. 9).

This type-checking algebra uses monads to encapsulate the side-effect of raising an error. Type-checking algebras also commonly uses the monad to maintain the typing environment for bound variables.

We can sequence an indexed evaluation algebra for overloaded operators after the type checking algebra, as in figure 10. Thus, the indexed algebra can determine the resolution of the operator syntax from the type of the result.

We have defined an evaluation algebra for overloaded operators by composing an independent specification of a type checking algebra with a specification of a `Ty`-indexed evaluation algebra for operators using the `seqMAlg` combinator. This example motivates the use of `seqMAlg` to index across the monadic boundary.

```
data Ty = TInt | TBool

phi :: (MonadError e m, Error e) =>
    Algebra (Operators :$: Integers :$: Booleans :$: FVoid) (m Ty)
```

Fig. 9. Interface to an assumed type checker for the language with overload operators

5 Applications

In this section, we provide examples of some uses of InterpreterLib beyond toy examples. We show that InterpreterLib is a valid basis of a language description and semantic implementation.

5.1 Rosetta Type Checker

Rosetta is a systems specification language under standardization by IEEE (IEEE 1694) that allows for the specifier to design separate modules, attribute various model constraints to them, and reason about the system's behavior, specifically focusing on the interaction of elements from different domains. The Rosetta parser uses the Parsec [12] parser combinator library to target the recursive AST, which can then be converted into the non-recursive AST (the syntactic functors). Type checking starts with the parser's output and sequences its algebras to construct the complete typing analysis.

The type checker uses InterpreterLib to break the analysis into a series of sequenceable, simple algebras. These can be viewed as various passes over the structure, similar to multiple passes in a compiler. The initial two algebras record scopable items and build up a symbol table for each node. The third algebra performs traditional type checks, generating constraints on types. Unification occurs over the set of constraints, not over the structure of the AST, so the type checker performs unification prior to sequencing the final algebra, a substitution of actual types into the type variables and symbol tables at each node.

Sequencing algebras provides the results of previous algebras per-node, and this allows for a simple structure of the type checker. First, symbol table items are recorded in-place. A second algebra passes these items up and down the AST structure to fill out the symbol tables of defined items at each node. At this point, we have decorated the AST with symbol tables of all legal names that are in scope. The symbol table entries include type variables to represent each item as well, so we also have a representation of the type of each node available. This names analysis can now be composed with other algebras.

The type checker composes the symbol table analysis with a constraints generation analysis. With type variables available through the symbol table, the type checker generates constraints on the types of nodes based on their environment and subterms' types. For instance, an `if` term will require its guard to be a `boolean`, and will require the `if` type to be the least upper bound of its branches' types. These constraints are unified by an algorithm based on that

```
data Operators t = Plus t t | Times t t
$(derive makeAll ''Operators)

evalOperators :: ( Monad m, SubFunctor VNum v, SubFunctor VBool v
                 ) => Ty -> Algebra Operators (m (Fix v))
evalOperators TInt (Plus x y) = intOp (+) x y
evalOperators TBool (Plus x y) = boolOp (||) x y
evalOperators TInt (Times x y) = intOp (*) x y
evalOperators TBool (Times x y) = boolOp (&&) x y `asTypeOf` x

type F = Operators :$: Integers :$: Booleans :$: FVoid

evalPhi ::
  ( MonadError e m, Error e, , MonadError e1 m1, Error e1
  , SubFunctor VBool v, SubFunctor VNum v
  , MonadSequence F Ty (m1 (Fix v)) m
  ) => Algebra F (m (Ty, m1 (Fix v)))
evalPhi = TypeCheck.phi `seqMAlg`
             (\t f_t -> evalOperators t @+@ Interpreters.composite_phi)

eval :: Fix F -> IO ()
eval t = case runSeqT (cata evalPhi t) of
           Left e -> putStr "Type error:" >> putStrLn e
           Right (ty, m) ->
             case m of
               Left e -> putStr "Evaluation error:" >> putStrLn e
               Right v -> putStr "Value:" >> putStrLn (show v)
```

Fig. 10. Syntax and semantics for the overloaded operators and an interpreter for the combined language

found in [16], but modified to address subtyping issues. Unification operates over the set of constraints, not over the AST itself, and occurs outside of the algebras. Not everything has to be an algebra when using InterpreterLib. Unification generates a list of appropriate substitutions for type variables, which are then substituted into the types and symbol tables at each node in another algebra. It composes the names analysis and constraints analysis, but as the replacement of type variables with concrete types is very regular, we use a generic traversal, which InterpreterLib also provides.

The type checker can be improved and used for different purposes with minimal changes. As the type checker must sometimes generate the symbol tables for imported packages (in other files), a mechanism for storing symbol tables and re-using the results is employed to avoid duplication of effort. Also, as Rosetta provides overloading of basic operators, simulation efforts must know when to cast primitive values. Even if we know that adding an `int` and a `real` will be performed as `(+)::real`→`real`→`real`, we must still convert each element to a `real` value. A coercion algebra sequences the type checker to find types, and then identifies which elements need casting. There is no need to modify the type checker, only to sequence its algebras to make type information available.

The type checker composes different algebras to define type correctness in Rosetta. Not only is it defined via separate semantics for scoping, typing constraints, and substitutions, but these algebras can be used without modification when composing new semantics that rely on type information. InterpreterLib facilitates straightforward reasoning about types and type checking while providing modularity.

5.2 Oread

InterpreterLib has been used extensively in the development of Oread [7], a functional language that can be compiled either as software, to be executed on a general purpose CPU, or as hardware, to be synthesized to a FPGA netlist. A wide range of language analyses for Oread have been implemented as semantic algebras, including a type checker, optimizations, and an evaluator. Moreover, three compilation target backends have been implemented, one which uses C as a high-level assembly, one that uses the LLVM intermediate representation, and finally a hardware backend which generates structural VHDL. Using InterpreterLib to construct the language analyses for Oread demonstrates the usefulness of using composition to combine individual component analyses.

First, Oread is an experimental language, and during the development of the toolset language features were often added or removed. The Oread type checking algebra, for example, is defined as the composition of separate type checking algebras, each independently of the other constructs. Integrating the type checking logic for a new language construct requires no change to the existing algebras.

Second, the backend compiler algebras, generating C or VHDL, utilize the results of the type checking algebra. This can be trivially accomplished using the `seqMAlg` algebra combinator for composing heterogeneous semantics. If the `seqMAlg` combinator were not used, the compiler algebras would either replicate the type checking implementation, resulting in a duplication of that logic, or the implementor would have to manually define the "plumbing" to propogate the type checking results to the compiler algebra. Rather than define that plumbing on an *ad hoc* basis, the `seqMAlg` algebra implements a re-usable mechanism.

6 Related Work

Liang et al. [13] build upon a line of research into monad transformers [3, 22, 25]. Cartwright and Felleisen [1] achieve a similar result, allowing orthogonal extension of language semantics, using *extended direct semantics* rather than monad transformers to structure the computational effects. The orthogonal contributions of Gayo et al. [4] are most related to *generic programming.* See [20] for a comparison of the most mature techniques.

Reps and Teitelbaum [19] represent languages via attribute grammars and focus on source transformations. Mosses [14] extends the work on Plotkin's structural operational semantics [17, 18], introducing modularity in defining operational semantics of languages. Syntax is modularly defined with SDF [5]. Deursen et al. [2] algebraically specify languages in ASF+SDF, and the Meta-Environment [23] provides an open-source platform and libraries for language implementations that follow ASF+SDF and other formalisms. InterpreterLib provides similar modularity in a denotational semantics.

Strategic programming [9, 24] adopts term rewriting strategies as a programming methodology in order to separate the concerns of traversal and semantics. Such separation yields re-usable semantics and traversal schemes. Algebra combinators are an instantiation of *strategy combinators* in the domain of modular denotational semantics. The `seqMAlg` combinator's management of monadic encapsulation is a synthesizing contribution at the juncture of modular monadic semantics and strategic programming. In particular, it is strictly more expressive than the `letTU` strategy combinator from the Strafunski library [10] because it also provides the results computed at the subterms (and potentially all descendents) to the indexed algebra.

Lämmel et al. [11] also use the terminology "algebra combinators," but apply it to what we call *generic algebras*, algebras that exhibit polymorphism over the functor. Lammel's paper defines only one syntactic signature, thus functor-polymorphism is not evident. InterpreterLib implements the "updateable fold algebras" concept from [11] with the `SubFunctor` modularity interface.

7 Conclusions / Future Work

InterpreterLib combines the benefits of modular monadic programming with generic programming techniques to provide a flexible environment for creating composable data transformations. In this paper we described two forms of composition implemented in IntepreterLib. The first allows composition of multiple syntactic elements and their associated processors using modular monadic techniques. The second allows composition of multiple semantic interpretations using algebra combinators. The net result is a mechanism for composing semantic interpretations over distinct syntactic elements along with a mechanism for composing complete interpretations.

The approach has been successfully demonstrated in a variety of settings. A pedagogical example of a traditional interpreter written using InterpreterLib is presented in detail to illustrate its use. Additionally, two examples from type checking and mixed system synthesis are included as anecdotal evidence of InterpreterLib's effectiveness. We continue to use InterpreterLib extensively, applying it to the synthesis of analysis models from specifications, a prototype comonadic simulator, and other tools in the Raskell [8] Rosetta analysis suite. Although difficult to quantify, we believe our suite of representative applications provides substantial anecdotal evidence of savings in both development and testing time.

Future work related to InterpreterLib application includes more extensive applications for type checking, analysis model synthesis, and system synthesis from Rosetta specifications. We have also proposed using InterpreterLib's compositional techniques for synthesizing secure systems, although this work is largely speculative at this time. Finally, planned extensions include development of similar techniques for graph transformations and exploration of modular comonadic semantics.

References

[1] R. Cartwright and M. Felleisen. Extensible denotational language specifications. In *Symposium on Theoretical Aspects of Computer Software, number 789 in LNCS*, pages 244–272. Springer-Verlag, 1994.

[2] A. v. Deursen, J. Heering, and P. Klint, editors. *Language Prototyping: An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*. World Scientific Publishing Co., 1996.

[3] D. Espinosa. *Semantic Lego*. PhD thesis, Columbia University, 1995.

[4] J. E. L. Gayo, J. M. C. Lovelle, M. C. L. Díez, and A. C. del Río. Modular development of interpreters from semantic building blocks. *Nordic J. of Computing*, 8(3):391–407, 2001. ISSN 1236-6064.

[5] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism sdf—reference manual—. *SIGPLAN Not.*, 24(11):43–75, 1989. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/71605.71607.

[6] M. P. Jones. *Qualified types: theory and practice*. Cambridge University Press, New York, NY, USA, 1995. ISBN 0-521-47253-9.

[7] G. Kimmell. *System Synthesis from a Monadic Functional Language*. PhD thesis, University of Kansas, 2008.

[8] E. Komp, G. Kimmell, J. Ward, and P. Alexander. The Raskell Evaluation Environment. Technical report, The University of Kansas Information and Telecommunications Technology Center, 2335 Irving Hill Rd, Lawrence, KS, USA, November 2003.

[9] R. Lämmel. Typed Generic Traversal With Term Rewriting Strategies. *Journal of Logic and Algebraic Programming*, 54, 2003. Also available as arXiv technical report cs.PL/0205018.

[10] R. Lämmel and J. Visser. A Strafunski application letter. In V. Dahl and P. Wadler, editors, *PADL*, volume 2562 of *Lecture Notes in Computer Science*, pages 357–375. Springer, 2003.

[11] R. Lämmel, J. Visser, and J. Kort. Dealing with large bananas. In J. Jeuring, editor, *Proceedings of WGP'2000, Technical Report, Universiteit Utrecht*, pages 46–59, July 2000.

[12] D. Leijen. Parsec, a fast combinator parser, 2001.

[13] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In ACM, editor, *22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: San Francisco, California, January 22–25, 1995*, pages 333–343, New York, NY, USA, 1995. ACM Press.

[14] P. D. Mosses. Modular structural operational semantics. *J. Log. Algebr. Program.*, 60-61:195–228, 2004.

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

[16] B. C. Pierce. *Types and Programming Languages.* The MIT Press, Cambridge, MA, 2002.

[17] G. Plotkin. The origins of structural operational semantics, 2003. URL `citeseer.ist.psu.edu/plotkin03origins.html`.

[18] G. D. Plotkin. A structural approach to operational semantics, 1981.

[19] T. Reps and T. Teitelbaum. The synthesizer generator. *SIGSOFT Softw. Eng. Notes*, 9(3):42–48, 1984. ISSN 0163-5948. doi: http://doi.acm.org/10.1145/390010.808247.

[20] A. Rodriguez, J. Jeuring, P. Jansson, A. Gerdes, O. Kiselyov, and B. C. d. S. Oliveira. Comparing libraries for generic programming in haskell. In *Haskell '08: Proceedings of the first ACM SIGPLAN symposium on Haskell*, pages 111–122, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-064-7.

[21] T. Sheard and S. Peyton Jones. Template metaprogramming for Haskell. In M. M. T. Chakravarty, editor, *ACM SIGPLAN Haskell Workshop 02*, pages 1–16. ACM Press, Oct. 2002.

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

[23] M. van den Brand, J. Heering, H. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. Olivier, J. Scheerder, J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In *Proceedings of Compiler Construction 2001 (CC 2001)*, LNCS. Springer, 2001.

[24] E. Visser. Language independent traversals for program transformation. In J. Jeuring, editor, *Workshop on Generic Programming (WGP 2000)*, Ponte de Lima, Portugal, July 2000. Technical Report UU-CS-2000-19, Department of Information and Computing Sciences, Universiteit Utrecht.

[25] P. Wadler. The essence of functional programming. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–14, Albequerque, New Mexico, 1992.

[26] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 60–76, New York, NY, USA, 1989. ACM.

[27] P. Weaver, G. Kimmell, N. Frisby, and P. Alexander. Modular and generic programming with interpreterlib. In R. E. K. Stirewalt, A. Egyed, and B. Fischer, editors, *ASE*, pages 473–476. ACM, 2007.

[28] P. Weaver, G. Kimmell, N. Frisby, and P. Alexander. Constructing language processors with algebra combinators. In *GPCE '07: Proceedings of the 6th international conference on Generative programming and component engineering*, pages 155–164, New York, NY, USA, 2007. ACM.