

Building Compilers by Combining Algebras *

Garrin Kimmell Perry Alexander
Ed Komp

Information and Telecommunications Technology Center
Department of Electrical Engineering and Computer Science
The University of Kansas, Lawrence, KS 66045
{kimmell,alex,komp}@ittc.ku.edu

November 19, 2004

Abstract

Embedded systems present a wide variety of challenges for developers of language tools. Verification of correctness, flexibility for adding new language features, and retargeting new architectures all present significant problems when developing a compiler for embedded systems. In this paper we present a domain-specific language based on modular monadic semantics which addresses many of these challenges.

1 Introduction

Embedded systems present unique challenges and opportunities for language tool builders. The high-reliability demands of embedded systems require verified systems, not just at the source level, but throughout the toolchain. Embedded systems require specialized capabilities for a large collection of varied tasks; it is suspect to believe a single language is appropriate for every task. This makes the flexibility of a compiler particularly important so that language constructs can added or removed as necessary for the system being developed. However, for a particular implementation it is desirable to have a unified language for expressing the system. When using hardware/software codesign techniques, it is desirable to target an implementation for execution either as software running on a general-purpose processor, or as dedicated hardware.

Traditional compiler construction techniques are difficult to adapt to these demands. Often the system consists of a large number of interdependent modules formed into a monolithic whole. A multi-pass compiler separates the phases of the compiler into a series of steps, but often common code is duplicated in

*This material is based upon work supported by the National Science Foundation under Grant No. 0209193

each pass. Retargeting the compiler for a new platform requires a substantial code modification.

Modular monadic semantics [11, 12, 3, 1] offers relief to the compiler writer by allowing independent specification of compilation schemes for various language constructs. Repetitive code, such as abstract syntax tree traversal, is handled automatically. In addition to simplifying the construction of a compiler, structuring the compiler in this manner provides an opportunity to simplify the verification of the completed compiler.

Unfortunately, structuring a compiler using modular monadic semantics is a mixed blessing. The compiler will sometimes need to apply one of a series of compilation schemes to a particular AST node, depending on a context. By relinquishing control of the traversal process it becomes very difficult to accomplish this task. In this paper we demonstrate a technique which allows the compiler writer to regain control of the traversal process, without sacrificing the modularity of the modular monadic semantics approach. This is accomplished by divorcing the control of the traversal process, implicit in the specification of a compilation scheme, from the compilation scheme itself.

The resulting framework is a domain-specific language [2] for creating a compiler from a set of small, independent parts. This framework has been successfully used to create a compiler for a functional language targeting a graph reduction machine implemented in FPGA.

2 Modular semantics

The semantics of a programming language is often specified as a *denotation* from the abstract syntax of the language to a value space. For example, the meaning of the expression $e_0 + e_1$ is defined by the following denotation function, where e_0 , $+$, and e_1 are abstract syntax tree (AST) elements, $+$ being the standard addition function, and \mathbf{n} is a metavariable ranging over numerals in the AST, and n is the number corresponding to the numeral \mathbf{n} .

$$\begin{aligned} \mathcal{E}[[e_0 + e_1]] &= \mathcal{E}[[e_0]] + \mathcal{E}[[e_1]] \\ \mathcal{E}[[\mathbf{n}]] &= n \end{aligned}$$

A more realistic language would include more complex forms. A language allowing definitions may include a “let” construct, of the form `let $v=e_0$ in e_1` , where v is an identifier name.

The denotation function for this language must be extended with an environment parameter ρ , for mapping identifiers to values.

$$\begin{aligned} \mathcal{E}[[\text{let } x=e_0 \text{ in } e_1]]\rho &= \mathcal{E}[[e_1]]\rho\{x \mapsto \mathcal{E}[[e_0]]\rho\} \\ \mathcal{E}[[v]] &= \rho v \end{aligned}$$

Unfortunately, this approach has the disadvantage that semantic specifications are monolithic; to combine the arithmetic and the `let` languages requires that we rewrite the original denotation of the arithmetic language to account for the extra environment parameter in the let denotation. This is particularly unappealing, because neither of the arithmetic constructs directly use the environment parameter.

$$\begin{aligned}
\mathcal{E}[\![e_0 + e_1]\!] \rho &= \mathcal{E}[\![e_0]\!] \rho + \mathcal{E}[\![e_1]\!] \rho \\
\mathcal{E}[\![n]\!] \rho &= n \\
\mathcal{E}[\![\mathbf{let} \ x=e_0 \ \mathbf{in} \ e_1]\!] \rho &= \mathcal{E}[\![e_1]\!] \rho \{x \mapsto \mathcal{E}[\![e_0]\!] \rho\} \\
\mathcal{E}[\![v]\!] &= \rho \ v
\end{aligned}$$

2.1 Monadic Semantics

We can eliminate the need to rewrite our semantics by structuring the definition differently. Observe that we could have, in the case of the **let** language, returned a *function* which mapped from an environment to a value and eliminated the environment parameter of the denotation function.

$$\begin{aligned}
\mathcal{E}[\![\mathbf{let} \ x=e_0 \ \mathbf{in} \ e_1]\!] &= \lambda \rho . \mathcal{E}[\![e_1]\!] \rho \{x \mapsto \mathcal{E}[\![e_0]\!] \rho\} \\
\mathcal{E}[\![v]\!] &= \lambda \rho . \rho \ v
\end{aligned}$$

This resolves the problem where different language constructs' denotations differ in their types due to varying arity. Both the original arithmetic denotation and the **let** denotation both map AST terms to values. However, the problem has not yet been solved – it has just been relocated. The value space is no longer simple numbers, but is either numbers (in the case of the arithmetic language) or functions from environments to values (for let expressions).

We can eliminate this problem by structuring our value space as a *monad*. A monad is a “computation” that once performed yields a value, in this case a number. Monads are frequently used in pure functional languages for structuring computations that appear to have side effects.

A monad is defined by two functions. The first, *return*, simply lifts a value into the monad. It represents a trivial computation, that does not take advantage of the effects of the monad. The second function, $\gg=$ (pronounced “bind”), sequences two computations, passing the result of the first into the second. This sequencing allows a pure language to dictate execution order, as is necessary when defining language constructs such as imperative features.

Using the functions *return* and $\gg=$, we can write a monadic semantics for the combined language:

$$\begin{aligned}
\mathcal{E}[\![n]\!] &= \text{return } n \\
\mathcal{E}[\![e_0 + e_1]\!] &= \mathcal{E}[\![e_0]\!] \gg= (\lambda v_0 . \mathcal{E}[\![e_1]\!] \gg= (\lambda v_1 . \text{return } (v_0 + v_1))) \\
\mathcal{E}[\![\mathbf{let} \ x=e_0 \ \mathbf{in} \ e_1]\!] &= \mathcal{E}[\![e_0]\!] \gg= (\lambda b . \text{extendEnvironment } x \ b \ \mathcal{E}[\![e_1]\!]) \\
\mathcal{E}[\![v]\!] &= \text{lookup } v
\end{aligned}$$

Haskell [5] provides a “do” syntax for working with monads. The above denotation function written using the do notation is shown below.

$$\begin{aligned}
e \ (\text{Num } x) &= \text{return } x \\
e \ (\text{Plus } e_0 \ e_1) &= \mathbf{do} \ \{ v_0 \leftarrow e \ e_0 \\
&\quad ; v_1 \leftarrow e \ e_1 \\
&\quad ; \text{return } (v_0 + v_1) \\
&\quad \} \\
e \ (\text{Let } x \ e_0 \ e_1) &= \mathbf{do} \ \{ b \leftarrow e \ e_0 \\
&\quad ; \text{extendEnvironment } x \ b \ (e \ e_1) \\
&\quad \} \\
e \ (\text{Var } v) &= \text{lookup } v
\end{aligned}$$

The above definitions take advantage of functions *extendEnvironment* and *lookup*, which operate on the environment. Both of these functions are easily defined.

```

extendEnvironment var value computation =
  λρ → computation (λv → if v ≡ var then value else (ρ v))
lookup var = λρ → return (ρ var)

```

Furthermore, we need to provide definitions for the *return* and $\gg=$ functions for the function type mapping environments to numbers, making this type a monad.

```

return x   = λρ.x
c1 >>= f   = λρ.f (c1 ρ) ρ

```

Fortunately, it is not necessarily to define a new monad each time a new language construct is added to the language. A small library of monads is available which support commonly found language constructs. Furthermore, the independent monads can be combined easily to generate a monad supporting the necessary features for the various language constructs. Modular monadic semantics have proven useful for defining the semantics of a wide range of language constructs [6, 7, 8]. We refer to the literature for a more complete description of the topic [3, 1].

3 Implicit AST Traversal

We have shown that, using modular monadic semantics, it is possible to largely separate the semantic definitions of orthogonal language constructs. There is further room for improvement; in the above definitions, the denotation function is coupled to a particular AST for the entire language. Moreover, the denotation specification is monolithic and recursively defined. This is counter to our goal of completely separating the specification of the orthogonal constructs.

While it is possible to develop language specifications this way by cutting-and-pasting from existing specifications, it is possible to further detach the denotation functions for various constructs. By doing so, we gain several benefits. First, a complete language specification can be constructed from existing specifications over its component language constructs. Secondly, the denotation functions are no longer recursive – the AST traversal is implicit in the construction of a complete language specification. Finally, the construction carries with it an implicit proof principle [4], simplifying induction proofs over the language.

3.1 Separating AST Elements

An AST datatype for the combined arithmetic and **let** languages can easily be written as a Haskell datatype. A semantics for this language was given in section 2.1.

```

data Expr = Num Int
          | Add Expr Expr

```

```

| Let String Expr Expr
| Var String

```

However, if we wish to add a new language construct such as a divide expression, we are forced to alter the datatype declaration.

```

data Expr = Num Int
          | Add Expr Expr
          | Div Expr Expr
          | Let String Expr Expr
          | Var String

```

The denotation function is no longer complete for the augmented AST type. It needs to be extended for the *Div* expression. A division can possibly throw an error when the divisor is zero, so the monad must support throwing exceptions.

The reason these changes are necessary is that both the AST datatype and the denotation function are expressed recursively and there is no way to “insert” a new term type. We can construct an isomorphic datatype by modeling it *nonrecursively*, then calculating the least fixed-point of that type. In doing so, we can separate related constructs into individual type constructors. Later we will demonstrate a method to combine the independent terms into a complete AST type.

```

data Expr' x = Num x
              | Add x x
              | Div x x
              | Let String x x
              | Var String

```

The least fixed-point of this type can be calculate with the (strict) *Rec* type constructor. This is similar to the least fixed point combinator for functions $fix\ f = f\ (fix\ f)$. The *inn* and *out* functions witness the isomorphism $Rec\ Expr' \simeq Expr$.

```

data Rec f = In (f (Rec f))
inn x = In x
out (In x) = x

```

We can deconstruct the *Expr'* type constructor into a set of type constructors, grouping similar constructs together.

```

data ArithExpr x = Num x
                  | Add x x

data LetExpr x = Let String x x
                | Var String

data DivExpr x = Div x x

```

To recombine the various datatypes together, we define a *Sum* type.

data *Sum* $f\ g\ x = \text{InjLeft}\ (f\ x) \mid \text{InjRight}\ (g\ x)$

The *Expr'* is then the *Sum* of the component types, yielding $\text{Expr}' \simeq \text{Sum}\ \text{ArithExpr}\ (\text{Sum}\ \text{LetExpr}\ \text{DivExpr})$. The original *Expr* type can then be recovered by calculating the least fixed-point of this type.

3.2 Implementing Traversal

The preceding section eliminated the problem of adding new syntax, but the problem of extending a denotation function remains. We now develop a solution to this problem.

The non-recursive types *ArithExpr*, *LetExpr*, *DivExpr*, and *Sum* above each model the category-theoretic notion of a *functor*. A functor is a “structure preserving map”, used for defining structure on categories. In this work, we are interested in the existence of a function *fmap* for a functor *F*, with type $(a \rightarrow b) \rightarrow (F\ a \rightarrow F\ b)$. This function can easily be defined for the above type constructors; verification that the definitions satisfy the requirements of functor are elided. We use the Haskell type system in our definition for handling overloading of the *fmap* function.

instance *Functor* *ArithExpr* **where**
fmap *f* (*Num* *x*) = *Num* *x*
fmap *f* (*Add* *x* *y*) = *Add* (*f* *x*) (*f* *y*)

instance *Functor* *LetExpr* **where**
fmap *f* (*Let* *n* *e*₀ *e*₁) = *Let* *n* (*f* *e*₀) (*f* *e*₁)
fmap *f* (*Var* *s*) = *Var* *s*

instance *Functor* *DivExpr* **where**
fmap *f* (*Div* *x* *y*) = *Div* (*f* *x*) (*f* *y*)

instance (*Functor* *f*, *Functor* *g*) \Rightarrow *Functor* (*Sum* *f* *g*) **where**
fmap *f* (*InjLeft* *x*) = *InjLeft* (*fmap* *f* *x*)
fmap *f* (*InjRight* *x*) = *InjRight* (*fmap* *f* *x*)

Because all of our term types are functors, we can now define a universal traversal function over all AST types. This will manifest itself as a *catamorphism*[13].

In the early semantic specification for the *Add* function, $e\ (\text{Add}\ x\ y) = ((e\ x) + (e\ y))$, we recursively apply the denotation function to the subterms of the *Add* term. The recursive call served to reduce the subterms to integers. We then substituted the *Add* constructor with the *+* function.

With a non-recursively defined AST type we can divide this into two steps. The first will perform the recursively traversal, so that the subterms are evaluated to integers. The second step, *which is the essence of the denotation*, replaces the *Add* constructor with its denotation.

(Add (Num 1) (Num 2))
 \Rightarrow evaluating subterms
 (Add 1 2)
 \Rightarrow replace Add by +
 3

We can define the denotation by composing these two steps. A first try yields the following specification. The function f , combined with the *carrier* of the value type integer, forms an algebra over the functor *ArithExpr*.

$$\begin{aligned} e (\text{Add } x \ y) &= f (\text{fmap } e (\text{Add } x \ y)) \\ f (\text{Add } x \ y) &= x + y \end{aligned}$$

This function is not quite right for two reasons. First, the AST is not of the form *Add x y*; rather it is *In (Add x y)*. We need to apply the *out* function to get the correct type, yielding $e \text{ term} = f (\text{fmap } e (\text{out term}))$. Secondly, not all subterms will use the particular f defined for *Add*. We can parameterize e over f to allow other algebras. We will call the corrected function *cata*, giving us $\text{cata } \phi \text{ term} = \phi (\text{fmap } (\text{cata } \phi) (\text{out term}))$.

Having separated the traversal from the algebra, we can define denotations for each algebra independently. It is necessary to define an algebra for the *Sum* functor, but this single definition suffices for all languages.

$$\begin{aligned} e_{arith} (\text{Add } e_0 \ e_1) &= \mathbf{do} \{ v_1 \leftarrow e_0 \\ &\quad ; v_2 \leftarrow e_1 \\ &\quad ; \text{return } (e_0 + e_1) \\ &\quad \} \\ e_{arith} (\text{Num } x) &= \text{return } x \end{aligned}$$

$$\begin{aligned} e_{let} (\text{Let } v \ e_0 \ e_1) &= \text{extendEnvironment } v \ e_0 \ e_1 \\ e_{let} (\text{Var } v) &= \text{lookup } v \end{aligned}$$

$$\begin{aligned} e_{div} (\text{Div } e_0 \ e_1) &= \mathbf{do} \{ v_1 \leftarrow e_0 \\ &\quad ; v_2 \leftarrow e_1 \\ &\quad ; \mathbf{if } v_2 \equiv 0 \\ &\quad \quad \mathbf{then fail "Divide-By-Zero"} \\ &\quad \quad \mathbf{else return } (v_1 / v_2) \\ &\quad \} \end{aligned}$$

$$\begin{aligned} e_{sum} \ e_{left} \ e_{right} (\text{InjLeft } f) &= e_{left} \ f \\ e_{sum} \ e_{left} \ e_{right} (\text{InjRight } g) &= e_{right} \ g \end{aligned}$$

The original denotation for the language (extended with a division construct) is $e = \text{cata } (e_{sum} \ e_{arith} \ (e_{sum} \ e_{let} \ e_{div}))$.

4 A Modular Monadic Compiler

The flexibility of the modular monadic approach to language specifications is demonstrated by the ease in developing a compiler for the language. Because we have separated traversal from the denotation, we just need to supply a new algebra, with the value space being lists of instructions for the abstract machine.

The machine is very simple; it consists of a stack of values upon which the instructions operate. The machine includes *AddI* and *DivI* instructions, that pop the top two values off the stack and push on the result of the appropriate operation with the two arguments. The *PushVar i* instruction gets the value *i* deep in the stack and pushes it on the stack. The *PushInt i* instruction pushes the value *i* onto the stack.

```
data Instruction = AddI
                | DivI
                | PushVar Int
                | PushInt Int
```

The environment used when compiling differs slightly from the environment in the earlier denotation because it maps identifiers to stack positions, rather than to values. Accordingly, it uses a function *incEnv* which increments the stack depth of each identifier in the environment.

$$incEnv\ c = \lambda\rho \rightarrow c\ (\lambda id \rightarrow \rho\ id + 1)$$

Using this definition, the compilation algebra is easy to write. The main differences are that we are generating a list of instructions rather than a value, and that the environment must be manipulated to manage the changing stack depth. Otherwise, the definitions are largely the same.

$$c_{arith}\ (Add\ e_0\ e_1) = \mathbf{do}\ \{ v_0 \leftarrow e_0 \\ \quad ; v_1 \leftarrow incEnv\ e_1 \\ \quad ; return\ (v_0 \# v_1 \# [AddI]) \\ \}$$

$$c_{arith}\ (Num\ x) = return\ [PushInt\ x]$$

$$c_{let}\ (Let\ x\ e_0\ e_1) = \mathbf{do}\ \{ v_0 \leftarrow e_0 \\ \quad ; extendEnv\ x\ 0\ (incEnv\ e_1) \\ \}$$

$$c_{let}\ (Var\ v) = \mathbf{do}\ \{ indx \leftarrow lookup\ v \\ \quad ; return\ [PushVar\ indx] \\ \}$$

$$c_{div}\ (Div\ e_0\ e_1) = \mathbf{do}\ \{ v_0 \leftarrow e_0 \\ \quad ; v_1 \leftarrow incEnv\ e_1 \\ \quad ; return\ (v_0 \# v_1 \# [DivI]) \\ \}$$

$$compiler = cata\ (e_{sum}\ c_{arith}\ (e_{sum}\ e_{let}\ e_{div}))$$

This section has demonstrated the development of a code generator for a simple language, targeting a particular abstract machine. By adding features to the abstract machine and additional compilation schemes, we can easily extend this compiler for a more complex language, without requiring a rewrite to the original compilation schemes.

While code generation represents a significant component of a complete compiler, it is but one part. Using the same framework it is easy to write other phases of compilation. Typechecking would use the same approach, with the algebra's carrier set being types; compiler optimizations based on source-to-source transformations can likewise be developed with the carrier set being the same as the AST datatype.

5 Controlling the traversal

The *cata* combinator is useful because it handles the recursive invocation of the compiler, leaving the algebras to focus only on the denotation of the term at hand. However, some compilation tasks require that the denotation be able to direct the traversal of the compiler. This usage is demonstrated in the example below.

Consider a compiler for a lazy language. A call to a function builds a thunk, which when evaluated will yield a value. There is a significant runtime cost to building the thunk, which should be eliminated if the compiler can infer that the thunk will always be evaluated. This distinction gives two compilation schemes, a lazy scheme where the a thunk is built, and a strict scheme which avoids building the thunk.

Strict and lazy compilation schemes for the let and addition languages are shown below. The lazy compilation of an addition expression builds a thunk of two arguments, using `makeThunk`, rather than generating an `AddI` instruction. The strict scheme for addition is the same as the original compilation, as is the lazy compilation scheme for let expressions. The strict let, however, compiles its binding's value in the *lazy* scheme, while the body of the let is compiled using the strict scheme.

$$\begin{array}{ll}
\mathcal{L}\llbracket e_0 + e_1 \rrbracket \rho & = \mathcal{L}\llbracket e_0 \rrbracket \rho ++ \mathcal{L}\llbracket e_1 \rrbracket \rho^{+1} ++ \text{makeThunk}(2, \text{Add}) \\
\mathcal{S}\llbracket e_0 + e_1 \rrbracket \rho & = \mathcal{S}\llbracket e_0 \rrbracket \rho ++ \mathcal{S}\llbracket e_1 \rrbracket \rho^{+1} ++ [\text{Add}] \\
\mathcal{L}\llbracket n \rrbracket \rho & = \text{PushInt } n \\
\mathcal{S}\llbracket n \rrbracket \rho & = \text{PushInt } n \\
\mathcal{L}\llbracket \text{let } v = e_0 \text{ in } e_1 \rrbracket \rho & = \mathcal{L}\llbracket e_0 \rrbracket \rho ++ \mathcal{L}\llbracket e_1 \rrbracket \rho^{+1} \{ v \mapsto 0 \} \\
\mathcal{S}\llbracket \text{let } v = e_0 \text{ in } e_1 \rrbracket \rho & = \mathcal{L}\llbracket e_0 \rrbracket \rho ++ \mathcal{S}\llbracket e_1 \rrbracket \rho^{+1} \{ v \mapsto 0 \} \\
\mathcal{L}\llbracket v \rrbracket \rho & = \text{PushVar } v \\
\mathcal{S}\llbracket v \rrbracket \rho & = \text{PushVar } v
\end{array}$$

Adapting the above schemes to the framework in section 2 presents a significant challenge. The \mathcal{S} scheme for let expressions explicitly invokes the \mathcal{L} scheme for the binding's value. The automatic traversal of the AST assumes that the *same* algebra will be used throughout the traversal, and provides no mechanism for one algebra to selectively invoke another. Furthermore, there is

a duplication of work as the \mathcal{L} and \mathcal{S} schemes are identical for both variables and numbers. This is less a problem than an annoyance, but in the next sections we present a solution to the first problem and solve the second along the way.

5.1 Updatable Algebras

In section 3.2 we treated algebras as simple functions. However, to solve the problem outlined above, we need them to be structured data objects[10] which allow introspection, but can be treated as functions by the *cata* combinator. This is accomplished in Haskell with a type class. The *alg* type variable is the data object, and *apply* is used to get the function behavior. The *f* type variable is the functor over which the algebra is defined, and *a* is the carrier set of the algebra.

```
class Algebra f alg a where
    apply :: alg → f a → a
```

The actual algebra structure is a simple record, with a field for each constructor of the term type. For convenience, an instance of *Algebra* is defined for function types, allowing the use of pure functions as algebras. Finally, a new version of the recursion combinator *cata* is defined using the *Algebra* class.

```
data ArithAlg a =
    ArithAlg { addExpr :: ArithExpr a → a,
              numExpr :: ArithExpr a → a }
data LetAlg a =
    LetAlg { letExpr :: LetExpr a → a,
            varExpr  :: LetExpr a → a }
```

```
instance Algebra ArithTerm ArithAlg a where
    apply alg term@(Add _ _) = addExpr alg term
    apply alg term@(Num _)  = numExpr alg term
```

```
instance Algebra LetTerm LetAlg a where
    apply alg term@(Let _ _ _) = letExpr alg term
    apply alg term@(Var _)     = varExpr  term
```

```
instance Algebra f (f a → a) a where
    apply alg term = alg term
```

```
cata φ = (apply φ) ∘ (fmap (cata φ)) ∘ out
```

The compilation schemes for the arithmetic language are shown below. Each constructor in the term type has a corresponding function that serves as the constructor's denotation. The *Algebra* class, combined with the record structure, insure that the algebra is defined over each constructor in the term type. The

number compilation scheme is only defined once, then used in both the strict and lazy algebra structures. The two different add compilation schemes are defined independently.

```

lazy_add = (Add e0 e1) = do { v0 ← e0
                                ; v1 ← incEnv e1
                                ; return

```

Combining the lazy and strict algebras into a single algebra requires augmenting the monad in the carrier set. To the existing monad facilities we add *getSwitch* and *withSwitch* functions. These functions are used in much the same way as the environment, but are used for controlling the compilation process. A *switchAlg* combinator controls which algebra will be applied to a term.

```

switchAlg switchFun term = do { switch ← getSwitch
                                ; switchFun switch term
                                }

```

For this compiler, the data type *LorR* is used to indicate which compilation scheme is used. The *select* function yields the correct algebra based on the switch value.

```

data LorS = Lazy | Strict
select lazyAlg strictAlg Lazy = apply lazyAlg
select lazyAlg strictAlg Strict = apply strictAlg

arith_alg = switchAlg (select lazy_arith strict_arith)

```

The let compilation scheme is more complex, because it must manipulate the switch, rather than just react to it. The compilation scheme is segmented into two parts. The first is a function that decorates each subterm with the appropriate switch. The second performs actual code generation and can be reused for both the lazy and strict let algebras. This common pattern is captured by the *SwitchAlg* algebra structure.

```

data Algebra f alg a ⇒ SwitchAlgebra s f alg a =
    Switch{ decorator :: (f a → f s),
           algFun :: alg }

instance (SwitchMonad s m, ZipFunctor f, Algebra f alg (m a)) ⇒
    Algebra f (SwitchAlgebra s f alg (m a)) (m a) where
    apply alg term = do { let selected = decorator alg term
                          ; term' ← zipFunctor mkSwitch selected term
                          ; apply (algFun alg) term'
                          }
    where mkSwitch switch comp = withSwitch switch comp

```

This algebra makes use of a *ZipFunctor* class, which combines two functors with different subterms type using a given function. It is necessary to define instances for this class for each term type, but there is exists a method for generating the instance automatically for all term types[9].

```

class ZipFunctor f where
  zipFunctor :: Monad m => (a -> b -> c) -> f a -> f b -> m (f c)

instance ZipFunctor LetTerm where
  zipFunctor f (Let v b0 body0) (Let w b1 body1) =
    return (Let v (f b0 b1) (f body0 body1))
  zipFunctor f (Var x) (Var y) = (Var x)
  zipFunctor _ _ _ = fail "Can't zip functors"

```

Using the *SwitchAlgebra*, the definition of the let compilation scheme becomes trivial. We can reuse the c_{let} compilation scheme from section 4.

```

clet (Let x e0 e1) = do { v0 ← e0
                          ; extendEnv x 0 (incEnv e1)
                          }

clet (Var v) = do { indx ← lookup v
                   ; return [PushVar indx]
                   }

```

```

let_decorate (Let v _ _) = Let v Lazy Strict
let_strict = SwitchAlg let_decorate clet
let_alg = switchAlg (select clet let_strict)

```

The complete compiler with strictness optimization is the combination of the *let_alg* and the *arith_alg*.

```

compiler = cata (sumAlg arith_alg let_alg)

```

Using the scheme described has an initial overhead due to the the need to define algebra structures *LetAlg* and *ArithAlg*, along with instances for *Functor*, *Algebra*, and *ZipFunctor*. However, this cost is mitigated in that this code only has to be written once and can be used for any tool operating on the language which uses this framework. The construction of the types and instances is algorithmic and can be automated, further reducing potential for programmer error by the compiler writer. Finally, the combinators *sumAlg* and *switchAlg* can be used for any algebra. If more use cases for combining algebras are observed, the combination routines can be localized in combinators similar to these.

6 Conclusions and Future Work

In this paper, we have described the development of a domain-specific language for building language processing tools such as interpreters and compilers. Language constructs are separated into logically connected group. Denotation functions, or algebras, define the meaning of each individual construct. An executable algebra is transcribed almost directly from the original specification.

Common patterns are factored out into a small set of combinators. The *cata* combinator captures the recursive nature of language processing, algebras are combined using the *sumAlg* combinator, and the *switchAlg* combinator reintroduces the ability to control traversal in an algebra.

Developing language processing tools using this framework is appealing for the embedded systems community for several reasons. The pattern of language processing shared by various tools, such as interpreters, compilers, and type-checkers, is factored. This has the potential to speed development both by facilitating reuse and by decoupling unrelated components. Additionally, because this framework is firmly grounded in category theory, the verification process is aided by results from that field.

The flexibility of the approach is interesting from a hardware/software code-design standpoint, as it allows multiple denotations of a language construct. Rather than generating target code for a single software architecture, a hardware and a software compiler can be combined in much the same manner as the strict and lazy compilation schemes, with the compiler statically deciding whether a hardware or a software implementation is more appropriate.

The framework described in this paper has already been used to successfully generate a compiler for a functional language, targeting an abstract machine implemented in hardware on an FPGA.

We plan to extend this work to address challenges in embedded systems design.

- Demonstrate the verification capabilities of the framework by verifying the compiler with respect to the operational semantics of the abstract machine
- Explore the hardware/software codesign space by targeting specific functionality for hardware, while generating abstract machine code for the remainder of the source program.

References

- [1] L. Duponcheel. Using catamorphisms, subtypes and monad transformers for writing modular functional interpreters.
- [2] P. Hudak. Building domain-specific embedded languages. *ACM Comput. Surv.*, 28(4es):196, 1996.

- [3] G. Hutton. Fold and unfold for program semantics. In *Proceedings of the third ACM SIGPLAN international conference on Functional programming*, pages 280–288. ACM Press, 1998.
- [4] G. Hutton. A tutorial on the universality and expressiveness of fold. *Journal of Functional Programming*, 9(4):355–372, 1999.
- [5] S. P. Jones. *Haskell 98 Language and Libraries*. Cambridge University Press, 2003.
- [6] J. E. Labra Gayo, J. M. C. L., M. C. L. D., and A. C. del Río. Reusable monadic semantics of object oriented programming languages, June 2002.
- [7] J. E. Labra Gayo, M. C. Luengo Díez, J. M. Cueva Lovelle, and A. Cernuda del Río. LPS: A language prototyping system using modular monadic semantics. In M. van der Brand and D. Parigot, editors, *Proceedings 1st Workshop on Language Descriptions, Tools and Applications, LDTA'01, Genova, Italy, 7 Apr 2001*, volume 44(2). Elsevier, Amsterdam, 2001.
- [8] J. E. Labra Gayo, M. C. Luengo Díez, J. M. Cueva Lovelle, and A. Cernuda del Río. Reusable monadic semantics of logic programs with arithmetic predicates. In *Proceedings 2001 APPIA-GULP-PRODE Joint Conf. on Declarative Programming, AGP'01, Évora, Portugal, 26–28 Sept. 2001*, pages 31–45. Dept. of Informatics, Univ. of Évora, 2001.
- [9] R. Lammel and S. P. Jones. Scrap more boilerplate: reflection, zips, and generalised casts. *SIGPLAN Not.*, 39(9):244–255, 2004.
- [10] R. Lammel, J. Visser, and J. Kort. Dealing with large bananas. In J. Jeuring, editor, *Workshop on Generic Programming*, 2000.
- [11] S. Liang and P. Hudak. Modular denotational semantics for compiler construction. In *Programming Languages and Systems – ESOP'96, Proc. 6th European Symposium on Programming, Linköping*, volume 1058, pages 219–234. Springer-Verlag, 1996.
- [12] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In ACM, editor, *Conference record of POPL '95, 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: San Francisco, California, January 22–25, 1995*, pages 333–343, New York, NY, USA, 1995. ACM Press.
- [13] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In J. Hughes, editor, *Proceedings 5th ACM Conf. on Functional Programming Languages and Computer Architecture, FPCA'91, Cambridge, MA, USA, 26–30 Aug 1991*, volume 523, pages 124–144. Springer-Verlag, Berlin, 1991.