# A Generic Abstract Syntax Model for Embedded Languages

Emil Axelsson
Chalmers University of Technology
emax@chalmers.se

## Abstract

Representing a syntax tree using a data type often involves having many similar-looking constructors. Functions operating on such types often end up having many similar-looking cases. Different languages often make use of similar-looking constructions. We propose a generic model of abstract syntax trees capable of representing a wide range of typed languages. Syntactic constructs can be composed in a modular fashion enabling reuse of abstract syntax and syntactic processing within and across languages. Building on previous methods of encoding extensible data types in Haskell, our model is a pragmatic solution to Wadler's "expression problem". Its practicality has been confirmed by its use in the implementation of the embedded language Feldspar.

***Categories and Subject Descriptors*** D.2.10 [*Design*]: Representation; D.2.11 [*Software Architectures*]: Languages; D.2.13 [*Reusable Software*]: Reusable libraries; D.3.2 [*Language Classifications*]: Extensible languages; D.3.3 [*Language Constructs and Features*]: Data types and structures

***General Terms*** Design, Languages

***Keywords*** the expression problem, generic programming, embedded domain-specific languages

## 1. Introduction

In 1998, Philip Wadler coined the "expression problem":[1]

> "The Expression Problem is a new name for an old problem. The goal is to define a datatype by cases, where one can add new cases to the datatype and new functions over the datatype, without recompiling existing code, and while retaining static type safety (e.g., no casts)."

This is not just a toy problem. It is an important matter of making software more maintainable and reusable. Being able to extend existing code without recompilation means that different features can be developed and verified independently of each other. Moreover, it gives the opportunity to extract common functionality into a library for others to benefit from. Having a single source for common functionality not only reduces implementation effort, but also leads to more trustworthy software, since the library can be verified once and used many times.

Our motivation for looking at the expression problem is highly practical. Our research group has developed several embedded languages, for example, Lava [5], Feldspar [3] and Obsidian [8]. There are several constructs and operations that occur repeatedly, both between the languages and within each language. We are interested in factoring out this common functionality in order to simplify the implementations and to make the generic parts available to others. A modular design also makes it easier to try out new features, which is important given the experimental state of the languages.

In addition to the requirements stated in the expression problem, a desired property of an extensible data type model is support for generic traversals. This means that interpretation functions should only have to mention the "interesting" cases. For example, an analysis that counts the number of additions in an expression should only have to specify two cases: (1) the case for addition, and (2) a generic case for all other constructs.

Our vision is a library of generic building blocks for embedded languages that can easily be assembled and customized for different domains. Modular extensibility (as stated in the expression problem) is one aspect of this vision. Support for generic programming is another important aspect, as it can reduce the amount of boilerplate code needed to customize interpretation functions for specific language constructs.

This paper proposes a simple model of abstract syntax trees that is extensible and supports generic traversals. The model is partly derived from Swierstra's *Data Types à la Carte* (DTC) [19] which is an encoding of extensible data types in Haskell. DTC is based on fixed-points of extensible functors. Our work employs the extensibility mechanism from DTC, but uses an application tree (section 2.2) instead of a type-level fixed-point. Given that DTC (including its recent development [4]) already provides extensible data types and generic traversals, our paper makes the following additional contributions (see also the comparison in section 9):

- We confirm the versatility of the original DTC invention by using it in an alternative setting (section 3).
- We provide direct access to the recursive structure of the data types, leading to simple generic traversals that do not rely on external generic programming mechanisms (section 4).
- We show how to support the freedom of explicit recursion in addition to predefined recursion schemes (sections 5, 6 and 7).

Our model is available in the SYNTACTIC library[2] together with a lot of utilities for implementing embedded languages (section 8). It has been successfully used in the implementation of the embedded language Feldspar [3] (section 8.1). Feldspar is a significantly-sized

[1] http://www.daimi.au.dk/~madst/tool/papers/expression.txt

[2] http://hackage.haskell.org/package/syntactic-1.0

language aimed at programming numerical algorithms in time-critical domains.

The code in this paper is available as a literate Haskell file.[3] It has been tested using GHC 7.4.1 (and the `mtl` package). A number of GHC-specific extensions are used; see the source code for details.

## 2. Modeling abstract syntax

It is common for embedded languages to implement an abstract syntax tree such as the following:

```
data Expr1 a where
  Num1 :: Int → Expr1 Int
  Add1 :: Expr1 Int → Expr1 Int → Expr1 Int
  Mul1 :: Expr1 Int → Expr1 Int → Expr1 Int
```

$\texttt{Expr}_1$ is a type of numerical expressions with integer literals, addition and multiplication. The type parameter `a` is the type of the *semantic value* of the expression; i.e. the value obtained by evaluating the expression. Evaluation is defined as a simple recursive function over expressions:

```
evalExpr1 :: Expr1 a → a
evalExpr1 (Num1 n)   = n
evalExpr1 (Add1 a b) = evalExpr1 a + evalExpr1 b
evalExpr1 (Mul1 a b) = evalExpr1 a * evalExpr1 b
```

For $\texttt{Expr}_1$, the semantic value type is always `Int`, but we will soon consider expressions with other semantic types.

The problem with types such as $\texttt{Expr}_1$ is that they are not extensible. It is perfectly possible to add new interpretation functions in the same way as $\texttt{evalExpr}_1$, but unfortunately, adding new constructors is not that easy. If we want to add a new constructor, say for subtraction, not only do we need to edit and recompile the definition of $\texttt{Expr}_1$, but also all existing interpretation functions. Another problem with $\texttt{Expr}_1$ is the way that the recursive structure of the tree has been mixed up with the symbols in it: It is not possible to traverse the tree without pattern matching on the constructors, and this prevents the definition of generic traversals where only the "interesting" constructors have to be dealt with. We are going to deal with the problem of generic traversal first, and will then see that the result also opens up for a solution to the extensibility problem.

### 2.1 Exposing the tree structure

One way to separate the recursive tree structure from the symbols is to make symbol application explicit:

```
data Expr2 a where
  Num2 :: Int → Expr2 Int
  Add2 :: Expr2 (Int → Int → Int)
  Mul2 :: Expr2 (Int → Int → Int)
  App2 :: Expr2 (a → b) → Expr2 a → Expr2 b
```

Here, $\texttt{Add}_2$ and $\texttt{Mul}_2$ are *function-valued symbols* (i.e. symbols whose semantic value is a function), and the only thing we can do with those symbols is to apply them to arguments using $\texttt{App}_2$. As an example, here is the tree for the expression $3 + 4$:

```
ex1 = App2 (App2 Add2 (Num2 3)) (Num2 4)
```

[3] http://www.cse.chalmers.se/~emax/documents/AST.lhs

What we have gained with this rewriting is the ability to traverse the tree without necessarily mentioning any symbols. For example, this function computes the size of an expression:

```
sizeExpr2 :: Expr2 a → Int
sizeExpr2 (App2 s a) = sizeExpr2 s + sizeExpr2 a
sizeExpr2 _          = 1
```

```
*Main> sizeExpr2 ex1
3
```

However, even though we have achieved a certain kind of generic programming, it is limited to *a single type*, which makes it quite uninteresting. Luckily, the idea can be generalized.

### 2.2 The `AST` model

If we lift out the three symbols from $\texttt{Expr}_2$ and replace them with a single symbol constructor, we reach the following syntax tree model:

```
data AST dom sig where
  Sym  :: dom sig → AST dom sig
  (:$) :: AST dom (a :→ sig) → AST dom (Full a)
                             → AST dom sig
infixl 1 :$
```

The `AST` type is parameterized on the *symbol domain* `dom`, and the `Sym` constructor introduces a symbol from this domain. The type `(:→)` is isomorphic to the function arrow, and `Full a` is isomorphic to `a`:

```
newtype Full a  = Full {result :: a}
newtype a :→ b = Partial (a → b)
infixr :→
```

As will be seen later, these types are needed to be able to distinguish function-valued expressions from partially applied syntax trees.

The `AST` type is best understood by looking at a concrete example. `NUM` is the symbol domain corresponding to the $\texttt{Expr}_1$ type:

```
data NUM a where
  Num :: Int → NUM (Full Int)
  Add :: NUM (Int :→ Int :→ Full Int)
  Mul :: NUM (Int :→ Int :→ Full Int)

type Expr3 a = AST NUM (Full a)
```

$\texttt{Expr}_3$ is isomorphic to $\texttt{Expr}_1$ (modulo strictness properties). This correspondence can be seen by defining smart constructors corresponding to the constructors of the $\texttt{Expr}_1$ type:

```
num :: Int → Expr3 Int
add :: Expr3 Int → Expr3 Int → Expr3 Int
mul :: Expr3 Int → Expr3 Int → Expr3 Int

num     = Sym ∘ Num
add a b = Sym Add :$ a :$ b
mul a b = Sym Mul :$ a :$ b
```

Symbol types, such as `NUM` are indexed by *symbol signatures* built up using `Full` and `(:→)`. The signatures of `Num` and `Add` are:

```
Full Int
Int :→ Int :→ Full Int
```

The signature determines how a symbol can be used in an AST by specifying the semantic value types of its arguments and result. The first signature above specifies a terminal symbol that can be used to make an `Int`-valued AST, while the second signature specifies a non-terminal symbol that can be used to make an `Int`-valued AST node with two `Int`-valued sub-terms. The `Num` constructor also has an argument of type `Int`. However, this (being an ordinary Haskell integer) is to be regarded as a parameter to the symbol rather than a syntactic sub-term.

A step-by-step construction of the expression $a + b$ illustrates how the type gradually changes as arguments are added to the symbol:

```
a, b :: AST NUM (Full Int)

Add               :: NUM (Int :→ Int :→ Full Int)
Sym Add           :: AST NUM (Int :→ Int :→Full Int)
Sym Add :$ a      :: AST NUM (Int :→ Full Int)
Sym Add :$ a :$ b :: AST NUM (Full Int)
```

We recognize a fully applied symbol by a type of the form `AST dom (Full a)`. Because we are often only interested in complete trees, we define the following shorthand:

```
type ASTF dom a = AST dom (Full a)
```

In general, a symbol has a type of the form

```
T (a :→ b :→ ... :→ Full x)
```

Such a symbol can be thought of as a model of a constructor of a recursive reference type $\mathtt{T}_{ref}$ of the form

```
T_ref a → T_ref b → ... → T_ref x
```

Why is `Full` only used at the result type of a signature and not the arguments? After all, we expect all sub-terms to be complete syntax trees. The answer can be seen in the type of `(:$)`:

```
(:$) :: AST dom (a :→ sig) → AST dom (Full a)
                           → AST dom sig
```

The `a` type in the first argument is mapped to `(Full a)` in the second argument (the sub-term). This ensures that the sub-term is always a complete AST, regardless of the signature.

The reason for using `(:→)` and `Full` (in contrast to how it was done in $\mathtt{Expr}_2$) is that we want to distinguish non-terminal symbols from function-valued terminal symbols. This is needed in order to model the following language:

```
data Lang a where
  Op1 :: Lang Int → Lang Int → Lang Int
  Op2 :: Lang (Int → Int → Int)
  ...
```

Here, $\mathtt{Op}_1$ is a *non-terminal* that needs two sub-trees in order to make a complete syntax tree. $\mathtt{Op}_2$ is a function-valued terminal. This distinction can be captured precisely when using `AST`:

```
data LangDom a where
  Op1' :: LangDom (Int :→ Int :→ Full Int)
  Op2' :: LangDom (Full (Int → Int → Int))
  ...

type Lang' a = AST LangDom (Full a)
```

Whithout `(:→)` and `Full`, the distinction would be lost.

### 2.3 Simple interpretation

Just as we have used `Sym` and `(:$)` to construct expressions, we can use them for pattern matching:

```
eval_NUM :: Expr3 a → a
eval_NUM (Sym (Num n))       = n
eval_NUM (Sym Add :$ a :$ b) = eval_NUM a + eval_NUM b
eval_NUM (Sym Mul :$ a :$ b) = eval_NUM a * eval_NUM b
```

Note the similarity to $\mathtt{evalExpr}_1$. Here is a small example to show that it works:

```
*Main> eval_NUM (num 5 `mul` num 6)
30
```

For later reference, we also define a rendering interpretation:

```
render_NUM :: Expr3 a → String
render_NUM (Sym (Num n))       = show n
render_NUM (Sym Add :$ a :$ b) =
    "(" ++ render_NUM a ++ " + " ++ render_NUM b ++ ")"
render_NUM (Sym Mul :$ a :$ b) =
    "(" ++ render_NUM a ++ " * " ++ render_NUM b ++ ")"
```

A quick intermediate summary is in order. We have shown a method of encoding recursive data types using the general `AST` type. The encoding has a one-to-one correspondence to the original type, and because of this correspondence, we intend to define languages only using `AST` without the existence of an encoded reference type. However, for any type `(ASTF dom)`, a corresponding reference type can always be constructed. So far, it does not look like we have gained much from this exercise, but remember that the goal is to enable extensible languages and generic traversals. This will be done in the two following sections.

## 3. Extensible languages

In the quest for enabling the definition of extensible languages, the `AST` type has put us in a better situation. Namely, the problem has been reduced from extending recursive data types, such as $\mathtt{Expr}_1$, to extending non-recursive types, such as `NUM`. Fortunately, this problem has already been solved in Swierstra's *Data Types à la Carte* [19]. Swierstra defines the type composition operator in Listing 1, which can be seen as a higher-kinded version of the `Either` type. We demonstrate its use by defining two new symbol domains:

```
-- Logic expressions
data Logic a where
  Not :: Logic (Bool :→ Full Bool)
  Eq  :: Eq a ⇒ Logic (a :→ a :→ Full Bool)

-- Conditional expression
data If a where
  If :: If (Bool :→ a :→ a :→ Full a)
```

These can now be combined with `NUM` to form a larger domain:

```
type Expr a = ASTF (NUM :+: Logic :+: If) a
```

A corresponding reference type (which we do not need to define) has all constructors merged at the same level:

```
data Expr_ref a where
  Num :: Int → Expr_ref Int
  Add :: Expr_ref Int → Expr_ref Int → Expr_ref Int
```

```
data (dom1 :+: dom2) :: * → * where
  Inj_L :: dom1 a → (dom1 :+: dom2) a
  Inj_R :: dom2 a → (dom1 :+: dom2) a

infixr :+:
```

**Listing 1:** Composition of symbol domains (part of Data Types à la Carte interface)

```
class (sub :<: sup) where
  inj :: sub a → sup a
  prj :: sup a → Maybe (sub a)

instance (expr :<: expr) where
  inj = id
  prj = Just

instance (sym1 :<: (sym1 :+: sym2)) where
  inj          = Inj_L
  prj (Inj_L a) = Just a
  prj _        = Nothing

instance (sym1 :<: sym3)
      ⇒ (sym1 :<: (sym2 :+: sym3)) where
  inj          = Inj_R ∘ inj
  prj (Inj_R a) = prj a
  prj _        = Nothing
```

**Listing 2:** Symbol subsumption (part of Data Types à la Carte interface)

```
  ...
  Not :: Expr_ref Bool → Expr_ref Bool
  ...
  If  :: Expr_ref Bool → Expr_ref a → Expr_ref a
                                    → Expr_ref a
```

Unfortunately, the introduction of `(:+:)` means that constructing expressions becomes more complicated:[4]

```
not :: Expr Bool → Expr Bool
not a = Sym (Inj_R (Inj_L Not)) :$ a

cond :: Expr Bool → Expr a → Expr a → Expr a
cond c t f = Sym (Inj_R (Inj_R If)) :$ c :$ t :$ f
```

The symbols are now tagged with injection constructors, and the amount of injections will only grow as the domain gets larger. Fortunately, Data Types à la Carte has a solution to this problem too. The `(:<:)` class, defined in Listing 2, provides the `inj` function which automates the insertion of injections based on the types. This lets us define `not` as follows:

```
not :: (Logic :<: dom) ⇒
       ASTF dom Bool → ASTF dom Bool
not a = Sym (inj Not) :$ a
```

Or, with the addition of the instance in Listing 3, simply as

[4] Here we override the `not` function from the `Prelude`. The `Prelude` function will be used qualified in this paper.

```
instance (sub :<: sup) ⇒ (sub :<: AST sup) where
  inj         = Sym ∘ inj
  prj (Sym a) = prj a
  prj _       = Nothing
```

**Listing 3:** Subsumption for `AST`

```
int :: (NUM :<: dom) ⇒ Int → ASTF dom Int

(⊕) :: (NUM :<: dom)
    ⇒ ASTF dom Int → ASTF dom Int → ASTF dom Int

(⊙) :: (NUM :<: dom)
    ⇒ ASTF dom Int → ASTF dom Int → ASTF dom Int

(≡) :: (Logic :<: dom, Eq a) ⇒
       ASTF dom a → ASTF dom a → ASTF dom Bool

condition :: (If :<: dom)
          ⇒ ASTF dom Bool
          → ASTF dom a → ASTF dom a → ASTF dom a

int             = inj ∘ Num
a ⊕ b           = inj Add :$ a :$ b
a ⊙ b           = inj Mul :$ a :$ b
a ≡ b           = inj Eq  :$ a :$ b
condition c t f = inj If  :$ c :$ t :$ f

infixl 6 ⊕
infixl 7 ⊙
```

**Listing 4:** Extensible language front-end

```
not a = inj Not :$ a
```

The `prj` function in Listing 2 is the partial inverse of `inj`. It will be used for pattern matching on expressions. The instances of `(:<:)` essentially perform a linear search at the type level to find the right injection. Overlapping instances are used to select the base case.

The remaining constructs of the `Expr` language are defined in Listing 4. Note that the types have now become more general. For example, the type

```
(⊕) :: (NUM :<: dom) ⇒
       ASTF dom Int → ASTF dom Int → ASTF dom Int
```

says that `(⊕)` works with *any* domain `dom` that contains `NUM`. Informally, this means any domain of the form

```
... :+: NUM :+: ...
```

Expressions only involving numeric operations will only have a `NUM` constraint on the domain:

```
ex2 :: (NUM :<: dom) ⇒ ASTF dom Int
ex2 = int 5 ⊙ int 6 ⊕ int 7
```

This means that such expressions can be evaluated by the earlier function $\texttt{eval}_{NUM}$, which only knows about `NUM`:

```
*Main> eval_NUM ex2
37
```

Still, the type is general enough that we are free to use $ex_2$ together with non-numeric constructs:

```
ex3 = ex2 ≡ ex2
```

The class constraints compose as expected:

```
*Main> :t ex3
ex3 :: (Logic :<: dom, NUM :<: dom) ⇒ ASTF dom Bool
```

That is, $ex_3$ is a valid expression in *any language* that includes `Logic` and `NUM`.

### 3.1 Functions over extensible languages

The evaluation function $eval_{NUM}$ is closed and works only for the `NUM` domain. By making the domain type polymorphic, we can define functions over open domains. The simplest example is `size`, which is completely parametric in the `dom` type:

```
size :: AST dom a → Int
size (Sym _)  = 1
size (s :$ a) = size s + size a

*Main> size (ex2 :: Expr3 Int)
5
*Main> size (ex3 :: Expr Bool)
11
```

But most functions we want to define require some awareness of the symbols involved. For example, if we want to count the number of additions in an expression, we need to be able to tell whether a given symbol is an addition. This is where the `prj` function comes in:

```
countAdds :: (NUM :<: dom) ⇒ AST dom a → Int
countAdds (Sym s)
    | Just Add ← prj s = 1
    | otherwise        = 0
countAdds (s :$ a)     = countAdds s + countAdds a
```

In the symbol case, the `prj` function attempts to project the symbol to the `NUM` type. If it succeeds (returning `Just`) and the symbol is `Add`, 1 is returned; otherwise 0 is returned. Note that the type is as general as possible, with only a `NUM` constraint on the domain. Thus, it accepts terms from any language that includes `NUM`:

```
*Main> countAdds (ex2 :: Expr3 Int)
1
*Main> countAdds (ex3 :: Expr Bool)
2
```

We have now fulfilled all requirements of the expression problem:

- We have the ability to extend data types with new cases, and to define functions over such open types.
- We can also add new interpretations (this was never a problem).
- Extension does not require recompilation of existing code. For example, the `NUM`, `Logic` and `If` types could have been defined in separate modules. The function `countAdds` is completely independent of `Logic` and `If`. Still, it can be used with expressions containing those constructs (e.g. $ex_3$).
- We have not sacrificed any type-safety.

## 4. Generic traversals

We will now see how to define various kinds of generic traversals over the `AST` type. In this section, we will only deal with fold-like traversals (but they are defined using explicit recursion). In sections 5.1 and 7, we will look at more general types of traversals.

According to Hinze and Löh [10], support for generic programming consists of two essential ingredients:

- A way to write overloaded functions
- A way to access the structure of values in a uniform way

Together, these two components allow functions to be defined over a (possibly open) set of types, for which only the "interesting" cases need to be given. All other cases will be covered by a single (or a few) default case(s).

We have already encountered two generic functions in this paper: `size` and `countAdds`. The `size` function works for all possible `AST` types while `countAdds` works for types `AST dom` where the constraint `(NUM :<: dom)` is satisfied.[5] For `size`, all cases are covered by the default cases. For `countAdds`, a special case is given for `Add`, and all other cases have default behavior.

An important aspect of a generic programming model is whether or not new interesting cases can be added in a modular way. The `countAdds` function has a single interesting case, and there is no way to add more of them. We will now see how to define functions for which the interesting cases can be extended for new types. We begin by looking at functions for which *all cases* are interesting.

### 4.1 Generic interpretation

The interpretation functions $eval_{NUM}$ and $render_{NUM}$ are defined for a single, closed domain. To make them extensible, we need to make the domain abstract, just like we did in `countAdds`. However, we don't want to use `prj` to match out the interesting cases, because now all cases are interesting. Instead, we factor out the evaluation of the symbols to a user-provided function. What is left is a single case for `Sym` and one for `(:$)`:

```
evalG :: (∀a . dom a     → Denotation a)
      → (∀a . AST dom a → Denotation a)
evalG f (Sym s)  = f s
evalG f (s :$ a) = evalG f s $ evalG f a

type family   Denotation sig
type instance Denotation (Full a)    = a
type instance Denotation (a :→ sig) =
                  a → Denotation sig
```

The `Denotation` type function strips away `(:→)` and `Full` from a signature. As an example, we let GHCi compute the denotation of `(Int :→ Full Bool)`:

[5] One can argue that these functions are not technically generic, because they only work for instances of the `AST` type constructor. However, because we use `AST` as a way to encode hypothetical reference types, we take the liberty to call such functions generic anyway.

```
*Main> :kind! Denotation (Int :→ Full Bool)
Denotation (Int :→ Full Bool) :: *
= Int → Bool
```

Next, we define the evaluation of `NUM` symbols as a separate function:

```
evalSym_NUM :: NUM a → Denotation a
evalSym_NUM (Num n) = n
evalSym_NUM Add     = (+)
evalSym_NUM Mul     = (*)
```

Because this definition only has to deal with non-recursive symbols, it is very simple compared to $\texttt{eval}_{NUM}$. We can now plug the generic and the type-specific functions together and use them to evaluate expressions:

```
*Main> evalG evalSym_NUM ex_2
37
```

Our task is to define an extensible evaluation that can easily be extended with new cases. We have now reduced this problem to making the $\texttt{evalSym}_{NUM}$ function extensible. The way to do this is to put it in a type class:

```
class Eval expr where
  eval :: expr a → Denotation a

instance Eval NUM where
  eval (Num n) = n
  eval Add     = (+)
  eval Mul     = (*)

instance Eval Logic where
  eval Not = Prelude.not
  eval Eq  = (==)

instance Eval If where
  eval If = λc t f → if c then t else f
```

Now that we have instances for all our symbol types, we also need to make sure that we can evaluate combinations of these types using `(:+:)`. The instance is straightforward:

```
instance (Eval sub_1, Eval sub_2)
      ⇒ Eval (sub_1 :+: sub_2) where
  eval (Inj_L s) = eval s
  eval (Inj_R s) = eval s
```

We can even make an instance for `AST`, which then replaces the `evalG` function:

```
instance Eval dom ⇒ Eval (AST dom) where
  eval (Sym s)  = eval s
  eval (s :$ a) = eval s $ eval a
```

Now everything is in place, and we should be able to evaluate expressions using a mixed domain:

```
*Main> eval (ex_3 :: Expr Bool)
True
```

### 4.2 Finding compositionality

One nice thing about `eval` is that it is completely compositional over the application spine of the symbol. This means that even partially applied symbols have an interpretation. For example, the partially applied symbol `(inj Add :$ int 5)` evaluates to the denotation `(5 +)`. We call such interpretations *spine-compositional*.

When making a generic version of $\texttt{render}_{NUM}$ we might try to use the following interface:

```
class Render expr where
  render :: expr a → String
```

However, the problem with this is that rendering is not spine-compositional. It is generally not possible to render a partially applied symbol as a monolithic string. For example, a symbol representing an infix operator will join its sub-expression strings differently from a prefix operator symbol. A common way to get to a spine-compositional interpretation is to make the renderings of the sub-expressions explicit in the interpretation. That is, we use `([String] → String)` as interpretation:

```
class Render expr where
  renderArgs :: expr a → ([String] → String)

render :: Render expr ⇒ expr a → String
render a = renderArgs a []
```

Now, the joining of the sub-expressions can be chosen for each case individually. The following instances use a mixture of prefix (`Not`), infix (`Add`, `Mul`, `Eq`) and mixfix rendering (`If`):

```
instance Render NUM where
  renderArgs (Num n) [] = show n
  renderArgs Add [a,b] = "(" ++ a ++ " + " ++ b ++ ")"
  renderArgs Mul [a,b] = "(" ++ a ++ " * " ++ b ++ ")"

instance Render Logic where
  renderArgs Not [a]  = "(not " ++ a ++ ")"
  renderArgs Eq [a,b] = "(" ++ a ++ " == " ++ b ++ ")"

instance Render If where
  renderArgs If [c,t,f] = unwords
      ["(if", c, "then", t, "else", f ++ ")"]
```

Although convenient, it is quite unsatisfying to have to use refutable pattern matching on the argument lists. We will present a solution to this problem in section 6. It is worth noting that this way of using ordinary lists to hold the result of sub-terms is also used in the Uniplate library [16] (see the `para` combinator).

The instance for `AST` simply traverses the spine, collecting the rendered sub-expressions in a list that is passed on to the rendering of the symbol:

```
instance Render dom ⇒ Render (AST dom) where
  renderArgs (Sym s)  as = renderArgs s as
  renderArgs (s :$ a) as = renderArgs s (render a:as)
```

Note that the case for `(:$)` has two recursive calls. The call to `renderArgs` is for traversing the application spine, and the call to `render` is for rendering the sub-expressions.

Finally, just like for `Eval`, we need an instance to dispatch on combined domains:

```
instance (Render sub1, Render sub2)
      ⇒ Render (sub1 :+: sub2) where
  renderArgs (Inj_L s) = renderArgs s
  renderArgs (Inj_R s) = renderArgs s
```

This concludes the definition of rendering for extensible languages.

```
*Main> render (ex_2 :: Expr Int)
"((5 * 6) + 7)"
```

The functions `eval` and `render` do not have any generic default cases, because all cases have interesting behavior. The next step is to look at an example of a function that has useful generic default cases.

### 4.3 Case study: Extensible compiler

Will now use the presented techniques to define a simple compiler for our extensible expression language. The job of the compiler is to turn expressions into a sequence of variable assignments:

```
*Main> putStr $ compile (ex_2 :: Expr Int)
v3 = 5
v4 = 6
v1 = (v3 * v4)
v2 = 7
v0 = (v1 + v2)
```

Listing 5 defines the type `CodeGen` along with some utility functions. A `CodeGen` is a function from a variable identifier (the result location) to a monadic expression that computes the program as a list of strings.[6] The monad also has a state in order to be able to generate fresh variables.

Listing 6 defines the fully generic parts of the compiler. Note the similarity between the types of `compileArgs` and `renderArgs`. The difference between the `Compile` and `Render` classes is that `Compile` has a default implementation of its method. The default method assumes that the symbol represents a simple expression, and uses `renderArgs` to render it as a string. The rendered expression is then assigned to the result location using `(=:=)`. The instances for `AST` and `(:+:)` are essentially the same as in the `Render` class. Finally, the `compile` function takes care of running the `CodeGen` and extracting the written program.

The code in Listings 5 and 6 is *completely generic*—it does not mention anything about the symbols involved, apart from the assumption of them being instances of `Compile`. In Listing 7 we give the specific instances for the symbol types defined earlier. Because `NUM` and `Logic` are simple expression types, we rely on the default behavior for these. For `If`, we want to generate an if statement rather than an expression with an assignment. This means that we cannot use the default case, so we have to provide a specific case for `If`. Note that the two branches (`tGen` and `fGen`) are given the same result location.

Note that the instances of `Compile` are completely independent, and could easily live in separate modules.

A simple test will demonstrate that the compiler works as intended:

```
ex_4 :: (NUM :<: dom, Logic :<: dom, If :<: dom)
    ⇒ ASTF dom Int
```

[6] Thanks to Dévai Gergely for the technique of parameterizing the compiler on the result location.

```
type VarId     =  Integer
type ResultLoc =  VarId
type Program   =  [String]
type CodeMonad =  WriterT Program (State VarId)
type CodeGen   =  ResultLoc → CodeMonad ()

freshVar      ::  CodeMonad VarId
var           ::  VarId → String
(=:=)         ::  VarId → String → String
indent        ::  Program → Program

freshVar       =  do v ← get; put (v+1); return v
var v          =  "v" ++ show v
v =:= expr     =  var v ++ " = " ++ expr
indent         =  map ("   " ++)
```

**Listing 5:** Extensible compiler: interpretation and utility functions

```
class Render expr ⇒ Compile expr where
  compileArgs :: expr a → ([CodeGen] → CodeGen)
  compileArgs expr args loc = do
      argVars ← replicateM (length args) freshVar
      zipWithM ($) args argVars
      tell [loc =:= renderArgs expr (map var argVars)]

instance Compile dom ⇒ Compile (AST dom) where
  compileArgs (Sym s) args loc =
      compileArgs s args loc
  compileArgs (s :$ a) args loc = do
      compileArgs s (compileArgs a [] : args) loc

instance (Compile sub1, Compile sub2)
      ⇒ Compile (sub1 :+: sub2) where
  compileArgs (Inj_L s) = compileArgs s
  compileArgs (Inj_R s) = compileArgs s

compile :: Compile expr ⇒ expr a → String
compile expr = unlines
             $ flip evalState 1
             $ execWriterT
             $ compileArgs expr [] 0
```

**Listing 6:** Extensible compiler: generic code

```
instance Compile NUM
instance Compile Logic

instance Compile If where
  compileArgs If [cGen,tGen,fGen] loc = do
      cVar ← freshVar
      cGen cVar
      tProg ← lift $ execWriterT $ tGen loc
      fProg ← lift $ execWriterT $ fGen loc
      tell $ [unwords ["if", var cVar, "then"]]
          ++ indent tProg
          ++ ["else"]
          ++ indent fProg
```

**Listing 7:** Extensible compiler: type-specific code

```
ex4 = condition (int 1 ≡ ex2)
                (int 0)
                (ex2 ⊙ int 2)

*Main> putStr $ compile (ex4 :: Expr Int)
v2 = 1
v6 = 5
v7 = 6
v4 = (v6 * v7)
v5 = 7
v3 = (v4 + v5)
v1 = (v2 == v3)
if v1 then
   v0 = 0
else
   v12 = 5
   v13 = 6
   v10 = (v12 * v13)
   v11 = 7
   v8 = (v10 + v11)
   v9 = 2
   v0 = (v8 * v9)
```

## 5. Implicit and explicit recursion

So far, our functions have been using explicit recursion. But there is nothing stopping us from defining convenient recursion schemes. The recursive pattern used in the `AST` instance for `renderArgs` and `compileArgs` (see section 4) generalizes nicely to a fold:

```
fold :: ∀dom b . (∀a . dom a → [b] → b)
             → (∀a . ASTF dom a   → b)
fold f a = go a []
  where
    go :: ∀a . AST dom a → [b] → b
    go (Sym s)  as = f s as
    go (s :$ a) as = go s (fold f a : as)
```

Note, again, the two recursive calls in the case for `(:$)`: the call to `go` for traversing the spine, and the call to `fold` for folding the sub-expressions. The function `f` is commonly called an *algebra*.

As a demonstration, it suffices to show that we can redefine `render` and `compile` in terms of `fold`:

```
render2 :: Render dom ⇒ ASTF dom a → String
render2 = fold renderArgs

compile2 :: Compile dom ⇒ ASTF dom a → String
compile2 a = unlines
           $ flip evalState 1
           $ execWriterT
           $ fold compileArgs a 0
```

Here, `renderArgs` and `compileArgs` are only used as algebras (of type `(dom a → [...] → ...)`), which means that the `Render` and `Compile` instances for `AST` are no longer needed.

### 5.1 Explicit recursion

TODO: *Patrik: The order of things is strange here: First we do it the right way and then the wrong way. Maybe skip this sub-section completely, or move to appendix?*

TODO: *Reviewer: In section 5.1, you should mention and cite the twin traversals from "Scrap More Boilerplate". (I might consider omitting section 5.1 as I'm not sure how much it adds to the overall presentation.)*

Most of the functions we have seen so far can easily be expressed in terms of `fold`. However, there are cases where explicit recursion is preferred. One such situation is when doing simultaneous recursion over two terms; for example, when checking if two expressions are equal. The generic code for equality is defined as follows:

```
class Equality expr where
  equal :: expr a → expr b → Bool

instance Equality dom ⇒ Equality (AST dom) where
  equal (Sym s1)   (Sym s2)   = equal s1 s2
  equal (s1 :$ a1) (s2 :$ a2) =
      equal s1 s2 && equal a1 a2
  equal _ _ = False

instance (Equality sub1, Equality sub2)
      ⇒ Equality (sub1 :+: sub2) where
  equal (InjL s1) (InjL s2) = equal s1 s2
  equal (InjR s1) (InjR s2) = equal s1 s2
  equal _ _                 = False
```

Once the generic code is in place, the type-specific instances are trivial; for example:

```
instance Equality NUM where
  equal (Num n1) (Num n2) = n1 == n2
  equal Add      Add      = True
  equal Mul      Mul      = True
  equal _ _               = False
```

Expressing this kind of recursion in terms of `fold` is much trickier. One way is to have the fold return a function that can decide whether an expression (with arbitrary type) is equal to the folded expression:

```
type EqChecker dom = AST∃ dom → Bool

data AST∃ dom = ∀a . AST∃ (ASTF dom a)
```

Because the type of the other expression is unrelated to that of the folded expression, we have to hide the type using existential quantification in `AST∃`.

For simplicity, we will only consider the algebra for `NUM`:

```
equalArgs_NUM :: NUM a → [EqChecker NUM]
                       → EqChecker NUM

equalArgs_NUM (Num n1) [] b
    | AST∃ (Sym (Num n2)) ← b = n1 == n2

equalArgs_NUM Add [a1,b1] b
    | AST∃ (Sym Add :$ a2 :$ b2) ← b =
        a1 (AST∃ a2) && b1 (AST∃ b2)

equalArgs_NUM Mul [a1,b1] b
    | AST∃ (Sym Mul :$ a2 :$ b2) ← b =
        a1 (AST∃ a2) && b1 (AST∃ b2)

equalArgs_NUM _ _ _ = False
```

```
equal_NUM :: ASTF NUM a → ASTF NUM b → Bool
equal_NUM a b = fold equalArgs_NUM a (AST∃ b)
```

Unfortunately, the clarity from the explicitly recursive version has been lost.

## 6. Regaining type-safety

The technique of making the arguments part of the interpretation (used by, for example, `renderArgs` and `fold`) has the problem that it loses type information about the context. This has two problems:

- The algebra function can never know whether it receives the expected number of arguments (see, for example, the pattern matching in `renderArgs` for `NUM`).
- All intermediate results are required to have the same type and cannot depend on the type of the expression.

This can be fixed by making a version of `fold` that uses a typed heterogeneous list [12] instead of an ordinary list. We define the type of heterogeneous lists as follows:

```
data Args c sig where
  Nil  :: Args c (Full a)
  (:*) :: c (Full a) → Args c sig
                     → Args c (a :→ sig)
infixr :*
```

TODO: *Reviewer: Maybe explain the purpose of the container type.*

TODO: *Reviewer: Connection between `AST/Args` and `Left/Right` from Adams [1]*

An `Args` list is indexed by a symbol signature. The elements are of type `c a` (`c` for "container") where `a` varies with the position in the signature. For example, here is an example of a list containing an integer and a Boolean, using `Maybe` as container:

```
argEx :: Args Maybe (Int :→ Bool :→ Full Char)
argEx = Just (Full 5) :* Just (Full False) :* Nil
```

(Because `Args` lists are only meant to hold the arguments of a symbol, the result type—`Char` in the above example—does not affect the contents of the list.) The reason for making the elements indexed by `Full a` rather than just `a` is to be able to have lists with `ASTF` expressions in them—i.e. lists of type `Args (AST dom)`. This is used, for example, when using recursion schemes transform expressions; see section 6.1.

Using `Args`, we can define a typed fold as follows:

```
typedFold :: ∀dom c
  .  (∀a . dom a → Args c a → c (Full (Result a)))
  → (∀a . ASTF dom a → c (Full a))
typedFold f a = go a Nil
  where
    go :: ∀a . AST dom a → Args c a
                         → c (Full (Result a))
    go (Sym s)  as = f s as
    go (s :$ a) as = go s (typedFold f a :* as)
```

Note the close correspondence to the definition of `fold` in section 5. The `Result` type function simply gives the result type of a signature:

```
type family   Result sig
type instance Result (Full a)    = a
type instance Result (a :→ sig) = Result sig
```

```
*Main> :kind! Result (Int :→ Full Bool)
Result (Int :→ Full Bool) :: *
= Bool
```

TODO: *Reviewer: Is `typedFold` equivalent to Scrap Your Boilerplate's `gfoldl`? In the paper, you should mention that it is or why it isn't.*

Because the `Args` list is indexed by the type of the symbol, we now know that the algebra will always receive the expected number of arguments. Furthermore, the elements in the `Args` list are now indexed by the type of the corresponding sub-expressions. In particular, this means that we can use `typedFold` to transform expressions without losing any type information. As a demonstration, we give a trivial "transformation" that just rebuilds an expression:

```
rebuild :: ASTF dom a → ASTF dom a
rebuild = typedFold (appArgs ∘ Sym)
```

The function `appArgs` applies a symbol to a list of arguments:

```
appArgs :: AST dom a → Args (AST dom) a
                     → ASTF dom (Result a)
appArgs a Nil       = a
appArgs s (a :* as) = appArgs (s :$ a) as
```

For the cases when we are not interested in type-indexed results, we define a version of `typedFold` with a slightly simplified type:

```
newtype Const a b = Const { unConst :: a }

typedFoldSimple :: ∀dom b
  .  (∀a . dom a → Args (Const b) a → b)
  → (∀a . ASTF dom a                → b)
typedFoldSimple f =
    unConst ∘ typedFold (λs → Const ∘ f s)
```

Using `typedFoldSimple`, we can now define a version of `Render` that avoids partial pattern matching:

```
class Render_safe sym where
  renderArgs_safe ::
      sym a → Args (Const String) a → String

instance (Render_safe sub_1, Render_safe sub_2)
      ⇒ Render_safe (sub_1 :+: sub_2) where
  renderArgs_safe (Inj_L s) = renderArgs_safe s
  renderArgs_safe (Inj_R s) = renderArgs_safe s

instance Render_safe NUM where
  renderArgs_safe (Num n) Nil = show n
  renderArgs_safe Add (Const a :* Const b :* Nil) =
      "(" ++ a ++ " + " ++ b ++ ")"
  renderArgs_safe Mul (Const a :* Const b :* Nil) =
      "(" ++ a ++ " * " ++ b ++ ")"

render_safe :: Render_safe dom ⇒ ASTF dom a → String
render_safe = typedFoldSimple renderArgs_safe
```

### 6.1 Typed transformations

As an example of folding with type-indexed intermediate results, we define a transformation that just flips the arguments of all additions in an expression. First, the generic code:

```
class FlipAdds sym dom where
  flipAddsArgs :: sym a → Args (AST dom) a
                        → ASTF dom (Result a)

instance (FlipAdds sub₁ dom, FlipAdds sub₂ dom)
      ⇒ FlipAdds (sub₁ :+: sub₂) dom where
  flipAddsArgs (Inj_L s) = flipAddsArgs s
  flipAddsArgs (Inj_R s) = flipAddsArgs s

flipAddsArgsDefault = appArgs ∘ inj
```

In contrast to our previous interpretation classes, this is a *two-parameter* class. The first parameter is the type of symbol in the usual manner. The second parameter is the type of the whole domain. The domain type has to be exposed in order for the type-specific instances to put constraints on it (see below). Unfortunately, the default case (which just reapplies the symbol without flipping) cannot be added as a default method. This would impose a `(sym :<: dom)` super-class constraint, which cannot be satisfied by the instance for `(:+:)`. This means that instances with default behavior become slightly more verbose:

```
instance (Logic :<: dom) ⇒ FlipAdds Logic dom where
  flipAddsArgs = flipAddsArgsDefault

instance (If :<: dom) ⇒ FlipAdds If dom where
  flipAddsArgs = flipAddsArgsDefault
```

The only interesting case is that for `NUM`:

```
instance (NUM :<: dom) ⇒ FlipAdds NUM dom where
  flipAddsArgs Add (a :* b :* Nil) = inj Add :$ b :$ a
  flipAddsArgs s args = flipAddsArgsDefault s args
```

Finally, we use `typedFold` to apply the transformation bottom-up:

```
flipAdds :: FlipAdds dom dom
         ⇒ ASTF dom a → ASTF dom a
flipAdds = typedFold flipAddsArgs


*Main> render (flipAdds ex₃ :: Expr Bool)
"((7 + (5 * 6)) == (7 + (5 * 6)))"
```

### 6.2 Crossing modular boundaries

A nice effect of having the whole domain type as a class parameter in `FlipAdds` is that it makes it possible to add awareness of the whole domain by declaring stronger constraints. Let us say we only want to flip the addition arguments when the second one is a conditional. The `FlipAdds` instance for `NUM` cannot do this, because it is not allowed to assume that the domain contains `If`. However, by simply adding an `If` constraint on the domain, pattern matching works just fine:

```
instance (NUM :<: dom, If :<: dom)
      ⇒ FlipAdds NUM dom where
  flipAddsArgs Add (a :* b :* Nil)
    | cond :$ _ :$ _ :$ _ ← b
    , Just If             ← prj cond
    = inj Add :$ b :$ a
  flipAddsArgs s args = flipAddsArgsDefault s args
```

Note that the domain is still maximally open—it is *only* constrained by `NUM` and `If`.

## 7. Controlling the recursion

All generic recursive functions that we have seen so far have one aspect in common: The recursive calls are fixed, and cannot be overridden. For example, for `renderArgs`, the recursive calls are made in the instances for `AST` and `(:+:)`, and these are not affected by the instances for the symbol types. This is often a good thing, making the programs less error-prone. However, sometimes, it is useful to be able to control the recursive calls on a case-by-case basis. This can be achieved by a simple change to `typedFold`: simply drop the recursive call to `typedFold` and replace it with the unchanged sub-term.

```
query :: ∀dom a c
  .  (∀b . (a ~ Result b)
        ⇒ dom b → Args (AST dom) b → c (Full a)
     )
  → ASTF dom a → c (Full a)
query f a = go a Nil
  where
    go :: (a ~ Result b)
       ⇒ AST dom b → Args (AST dom) b → c (Full a)
    go (Sym a)  as = f a as
    go (s :$ a) as = go s (a :* as)
```

TODO: *Patrik: Explain `(a ~ Result b)`*

The important difference here is that `query` lets the function `f` take care of the recursive call. The function `f` receives the symbol and its argument list, and produces a query result (which could be anything, including a transformed expression). The advantage of using `query` over direct recursion over an `AST` is that `query` permits the definition of fully extensible functions by making `f` overloaded on the symbol type—analogous to how, for example, $\texttt{render}_{safe}$ is extensible by means of the $\texttt{Render}_{safe}$ class.

The `query` combinator also comes with a version with a simpler type, analogous to `typedFoldSimple`:

```
querySimple :: ∀dom a c
  .  (∀b . (a ~ Result b)
        ⇒ dom b → Args (AST dom) b → c
     )
  → ASTF dom a → c
querySimple f = unConst ∘ query (λs → Const ∘ f s)
```

In our experience with the Feldspar implementation (section 8.1), we have found it convenient to have full control over the recursion when defining interpretations and transformations that perform simultaneous top-down and bottom-up traversals.

## 8. The SYNTACTIC library

The abstract syntax model presented in this paper is available in the SYNTACTIC library, available on Hackage[7]. In addition to the `AST`

[7] http://hackage.haskell.org/package/syntactic-1.0

type and the generic programming facilities, the library provides various building blocks for implementing practical embedded languages:

- Language constructs (conditionals, tuples, variable binding, etc.)
- Interpretation functions (evaluation, rendering, equivalence, etc.)
- Transformation functions (constant folding, code motion, etc.)
- Utilities for host-language interaction (the `Syntactic` class [2, 17], observable sharing [7, 9], etc.)

Being based on the extensible `AST` type, these building blocks are generic, and can quite easily be customized for different languages.

TODO: *Reviewer: Mention how to deal with binding.*

### 8.1 Practical use-case: Feldspar

Feldspar [3] is an embedded language for high-performance numerical computation, in particular for embedded digital signal processing applications. The latest version[8] is implemented using SYNTACTIC. Some details about the implementation can be found in reference [2]. A demonstration of the advantage of a modular language implementation is given in reference [17], where we show how to add monadic constructs and support for mutable data structures to Feldspar without changing the existing implementation.

Feldspar has a back-end for generating C code. It is divided in two main stages: (1) generating an intermediate imperative representation (used for low-level optimization, etc.), and (2) generating C code. It is worth noting that the first of these two stages uses the same basic principles as the compiler in section 4.3.

## 9. Related work

Our `AST` model inherits its extensibility from Data Types à la Carte [19] (DTC). Bahr and Hvitved [4] have show that DTC supports generic traversals with relatively low overhead using the `Foldable` and `Traversable` classes. Our model differs by providing generic traversals directly, without external assistance. This somewhat pedantic distinction becomes slightly more important when considering traversal of typed syntax trees. For this, Bahr and Hvitved have to redefine `Foldable` and `Traversable` for higher-order functors, which means that these classes are no longer automatically derivable by the compiler.

The DTC literature has focused on using recursion schemes rather than explicit recursion for traversing data types. Although examples of explicit recursion exist (see the `render` function in reference [19]), it is not immediately clear how to combine explicit recursion with generic traversals based on `Foldable`/`Traversable`.

Lämmel and Ostermann [14] give a solution to the expression problem based on Haskell type classes. The basic idea is to have a non-recursive data type for each constructor, and a type class representing the open union of all constructors. Interpretations are added by introducing sub-classes of the union type class. This method can be combined with existing frameworks for generic programming.[9] One drawback with the approach is that expression types reflect the exact structure of the expressions, and quite some work is required to manage these heterogeneous types.

Yet another method for defining fully extensible languages is *Finally Tagless* [6], which associates each group of language constructs with a type class, and each interpretation with a semantic domain type. Extending the language constructs is done by adding new type classes, and extending the interpretations is done by adding new instances. This technique limits traversals to compositional bottom-up traversals. (Note, though, that this limit is mostly of practical interest. With a little creativity, it is possible to express even apparently non-compositional interpretations compositionally [11].)

There exist a number of techniques for *data-type generic programming* in Haskell (for example, see references [13, 15]). An extensive overview is given in reference [18]. However, they do not qualify as solutions to the expression problem, as they do not provide a way to extend existing types with new constructors. Many of these approaches provide a *one-layer* generic view of data types, where the top-most constructor can be traversed generically and all substructures are left as values in the original type. Hence the generic view is no more extensible than the type it represents.

*The spine view* [10] is a generic view for the "Scrap your boilerplate" [13] style of generic programming. The `Spine` type has strong similarities to our `AST` type. The main difference is that `Spine` is a *one-layer* view, whereas `AST` is a complete view of a data type. This means that the `Spine` type is not useful on its own—it merely provides a way to define generic functions over other existing types. It should be pointed out that the one-layer aspect of `Spine` is a *good thing* when it comes to ordinary generic programming, but it does mean that `Spine` alone cannot provide a solution to the expression problem. So, although `Spine` and `AST` rely on the same principle for generic constructor access, they are quite different in practice, and they solve different problems.

TODO: *David Broman's MKL: http://www.bromans.com/david/publ/thesis-2010-david-broman.pdf*

## 10. Discussion

Our goal with this work is to make a library of generic building blocks for implementing embedded languages. Any such attempt is bound to run into the expression problem, because the library must provide extensible versions of both syntactic constructs and interpretation functions. The `AST` model provides a pleasingly simple and flexible basis for such an extensible library. Its distinguishing feature is the *direct* support for generic recursive functions—no additional machinery is needed. For extensibility, some extra machinery had to be brought in, but the overhead is quite small compared to the added benefits.

The `AST` model has one major drawback: It cannot model mutually recursive types. For example, `AST` is not suited to model the following pair of data types:

```
type Var = String

data Expr a where
```

[8] http://hackage.haskell.org/package/feldspar-language-0.5.0.1

[9] See slides by Lämmel and Kiselyov *"Spin-offs from the Expression Problem"* http://userpages.uni-koblenz.de/~laemmel/TheEagle/resources/xproblem2.html.

```
  Var  :: Var → Expr a
  Num  :: Int → Expr Int
  Add  :: Expr Int → Expr Int → Expr Int
  Exec :: Stmt → Var → Expr a

data Stmt where
  Assign :: Var → Expr a → Stmt
  Seq    :: Stmt → Stmt → Stmt
  Loop   :: Expr Int → Stmt → Stmt
```

TODO: *Give the full story on mutually recursive types (and similarity to CompData/MultiRec).*

Here, `Expr` is an expression language capable of embedding imperative code using the `Exec` constructor. `Stmt` is an imperative language using the `Expr` type for pure expressions. In the `AST` type, all symbols are first-class, which means that we cannot easily group the symbols as in the example above.

Note, however, that the above language can easily be modeled as a single data type with monadic expressions (for example by representing `Stmt` as pure expressions with semantic value type `IO ()`). In fact, the latest Feldspar release has support for mutable data structures with a monadic interface. Their implementation is described in reference [17].

It is also important to be aware that many of the reusable components provided by SYNTACTIC assume that the language being implemented has a pure functional semantics. However, this is not a limitation of the `AST` type itself, but rather of the surrounding utility library. There is nothing preventing adding utilities for different kinds of languages if the need arises.

TODO: *Jean-Philippe found the above paragraph to be a bit "written in a hurry".*

TODO: *Reviewer: Give a more detailed account of the difficulties of using* SYNTACTIC.

Our experience with implementing Feldspar has shown that, while the resulting code is quite readable, developing code using SYNTACTIC can be quite hard due to the heavy use of type-level programming. In the future, we would like to look into ways of hiding this complexity, by providing a simpler user interface, and, for example, using Template Haskell to generate the tricky code. However, we do not expect this to affect the underlying `AST` type.

TODO: *Check that listings don't appear (too long) before they're referred to.*

TODO: *Reviewer: Avoid page breaks after a colon or between text and following math/code.*

TODO: *Jean-Philippe asks: "What is the specification of* SYNTACTIC *(independent of its implementation in Haskell)?"*

TODO: *Reviewer: In writing a compiler with several passes it is often useful for the output intermediate language to differ in minor or substantial ways from the input intermediate language. Is this easy to do with the* SYNTACTIC *library? An example that illustrates this would be useful.*

TODO: *Reviewer: It would be nice to know how much compile-time and run-time overhead is involved when using the* SYNTACTIC *library instead of ordinary datatypes.*

## Acknowledgments

This work has been funded by Ericsson. The author would like to thank the following people for valuable discussions, comments and other input: Koen Claessen, Dévai Gergely, Oleg Kiselyov, Anders Persson, Mary Sheeran, Josef Svenningsson and Meng Wang.

TODO: *Reviewers, etc.*

## References

[1] M. D. Adams. Scrap your zippers: a generic zipper for heterogeneous types. In *Proceedings of the 6th ACM SIGPLAN workshop on Generic programming*, WGP '10, pages 13–24. ACM, 2010.

[2] E. Axelsson and M. Sheeran. Feldspar: Application and implementation. In *Lecture Notes of the Central European Functional Programming School (to appear)*, volume 7241 of *LNCS*. 2012.

[3] E. Axelsson, K. Claessen, G. Dévai, Z. Horváth, K. Keijzer, B. Lyckegård, A. Persson, M. Sheeran, J. Svenningsson, and A. Vajda. Feldspar: A domain specific language for digital signal processing algorithms. In *8th ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2010)*, pages 169–178. IEEE Computer Society, 2010.

[4] P. Bahr and T. Hvitved. Compositional data types. In *Proceedings of the seventh ACM SIGPLAN workshop on Generic programming*, WGP '11, pages 83–94. ACM, 2011.

[5] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: Hardware Design in Haskell. In *ICFP '98: Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, pages 174–184. ACM, 1998.

[6] J. Carette, O. Kiselyov, and C.-c. Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming*, 19(05):509–543, 2009.

[7] K. Claessen and D. Sands. Observable sharing for functional circuit description. In *Proc. of Asian Computer Science Conference (ASIAN)*, Lecture Notes in Computer Science. Springer Verlag, 1999.

[8] K. Claessen, M. Sheeran, and B. J. Svensson. Expressive array constructs in an embedded GPU kernel programming language. In *Proceedings of the 7th workshop on Declarative aspects and applications of multicore programming*, DAMP '12, pages 21–30. ACM, 2012.

[9] A. Gill. Type-safe observable sharing in Haskell. In *Haskell '09: Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, pages 117–128. ACM, 2009.

[10] R. Hinze and A. Löh. "Scrap Your Boilerplate" Revolutions. In *Mathematics of Program Construction*, volume 4014, pages 180–208. Springer, 2006.

[11] O. Kiselyov. Typed tagless final interpreters. In *Lecture Notes of the Spring School on Generic and Indexed Programming (to appear)*. 2010.

[12] O. Kiselyov, R. Lämmel, and K. Schupke. Strongly typed heterogeneous collections. In *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, Haskell '04, pages 96–107. ACM, 2004.

[13] R. Lämmel and S. P. Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation*, TLDI '03, pages 26–37. ACM, 2003.

[14] R. Lämmel and K. Ostermann. Software extension and integration with type classes. In *Proceedings of the 5th international conference*

*on Generative programming and component engineering*, GPCE '06, pages 161–170. ACM, 2006.

[15] J. P. Magalhães, A. Dijkstra, J. Jeuring, and A. Löh. A generic deriving mechanism for Haskell. In *Proceedings of the third ACM Haskell symposium on Haskell*, Haskell '10, pages 37–48. ACM, 2010.

[16] N. Mitchell and C. Runciman. Uniform boilerplate and list processing. In *Proceedings of the ACM SIGPLAN workshop on Haskell workshop*, Haskell '07, pages 49–60. ACM, 2007.

[17] A. Persson, E. Axelsson, and J. Svenningsson. Generic monadic constructs for embedded languages. In *IFL '11: Implementation and Application of Functional Languages*, 2011. To be published.

[18] A. Rodriguez, J. Jeuring, P. Jansson, A. Gerdes, O. Kiselyov, and B. C. d. S. Oliveira. Comparing libraries for generic programming in Haskell. In *Proceedings of the first ACM SIGPLAN symposium on Haskell*, Haskell '08, pages 111–122. ACM, 2008.

[19] W. Swierstra. Data types à la carte. *Journal of Functional Programming*, 18(4):423–436, 2008.