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.3.3 [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

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

1. Introduction

In 1998, Philip Wadler coined the “expression problem”:

“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 domain-specific languages (EDSLs), 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 EDSLs 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 constructs.

This paper proposes a simple model of typed abstract syntax trees that is extensible and supports generic traversals. The model is partly derived from Swierstra’s Data Types à la Carte (DTC) [18] 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 recent development [4]) already provides extensible data types and generic traversals, our paper makes the following additional contributions (see also the comparison in section 10):

- We confirm the versatility of the original DTC invention by using it in an alternative setting (section 3).
- Our model provides direct access to the recursive structure of the data types, leading to simpler generic traversals that do not rely on external generic programming mechanisms (section 4).
- We explore the use of explicit recursion in addition to predefined recursion schemes (sections 5, 6 and 7), demonstrating that generic traversals over extensible data types are not restricted to predefined recursive patterns.

Our model is available in the SYNTACTIC library together with a lot of utilities for EDSL implementation (section 9). It has been successfully used in the implementation of Feldspar [3] (section 9.1), an EDSL aimed at programming numerical algorithms in time-critical domains.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.


Copyright © 2012 ACM 978-1-4503-1054-3/12/09. . . $10.00

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


Copyright © 2012 ACM 978-1-4503-1054-3/12/09. . . $10.00

2 http://hackage.haskell.org/package/syntactic-1.0
The code in this paper is available as a literate Haskell file. 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 Expr a where
  Num : Int → Expr Int
  Add : Expr Int → Expr Int → Expr Int
  Mul : Expr Int → Expr Int → Expr Int
```

Expr is a type of numerical expressions with integer literals, addition and multiplication. The parameter a is the type of the semantic value of the expression; i.e. the value obtained by evaluating the expression. (For Expr, the semantic value type happens to always be Int, but we will soon consider expressions with other semantic types.) Evaluation is defined as a simple recursive function:

```
evalExpr :: Expr a → a
evalExpr (Num n) = n
evalExpr (Add a b) = evalExpr a + evalExpr b
evalExpr (Mul a b) = evalExpr a * evalExpr b
```

The problem with types such as Expr is that they are not extensible. It is perfectly possible to add new interpretation functions in the same way as evalExpr, 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 Expr, but also all existing interpretation functions. Another problem with Expr is 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 tree structure from the symbols is to make symbol application explicit:

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

Here, Add2 and Mul2 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 App2. As an example, here is the tree for the expression 3 + 4:

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

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
```

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 Expr2 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 Expr2 type:

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

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

Expr3 is isomorphic to Expr2 (modulo strictness properties). This correspondence can be seen by defining smart constructors corresponding to the constructors of the Expr2 type:

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

num = Sym o 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.

3 http://www.cse.chalmers.se/~emax/documents/axelsson2012generic.lhs
A step-by-step construction of the expression \(a + b\) illustrates how the type gradually changes as arguments are added to the symbol:

\[
\begin{align*}
& a, b :: \text{AST} \text{ NUM} \text{ (Full a)} \\
& \text{Add} :: \text{NUM} \text{ (Int \rightarrow Int \rightarrow Full Int)} \\
& \text{Sym Add} :: \text{AST} \text{ NUM} \text{ (Int \rightarrow Int \rightarrow Full Int)} \\
& \text{Sym Add} :$ a :: \text{AST} \text{ NUM} \text{ (Int \rightarrow Full Int)} \\
& \text{Sym Add} :$ a :$ b :: \text{AST} \text{ NUM} \text{ (Full Int)} \\
\end{align*}
\]

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

\[
\text{type ASTF dom a} = \text{AST dom} \ (\text{Full a})
\]

In general, a symbol has a type of the form

\[
\text{T} (a : b : \ldots : \rightarrow \text{Full x})
\]

Such a symbol can be thought of as a model of a constructor of a recursive reference type \(\text{T ref} \ a \rightarrow \text{T ref} \ b \rightarrow \ldots \rightarrow \text{T ref} \ x\)

Why is \text{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 \((::):\)

\[
\begin{align*}
& (::) :: \text{AST dom} \ (a \rightarrow \text{sig}) \rightarrow \text{AST dom} \ (\text{Full a}) \\
& \rightarrow \text{AST dom} \ \text{sig}
\end{align*}
\]

The \(a\) type in the first argument is mapped to \((\text{Full a})\) 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 \((\rightarrow)\) and \text{Full} (in contrast to how it was done in \text{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:

\[
\begin{align*}
\text{data \ Lang} \ a \ \text{where} \\
& \text{Op}_1 :: \text{Lang} \text{ Int} \rightarrow \text{Lang} \text{ Int} \rightarrow \text{Lang} \text{ Int} \\
& \text{Op}_2 :: \text{Lang} \text{ (Int \rightarrow Int \rightarrow Int)}
\end{align*}
\]

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

\[
\begin{align*}
\text{data \ LangDom} \ a \ \text{where} \\
& \text{Op}_1' :: \text{LangDom} \text{ (Int \rightarrow Int \rightarrow Full Int)} \\
& \text{Op}_2' :: \text{LangDom} \text{ (Full (Int \rightarrow Int \rightarrow Int))}
\end{align*}
\]

\[
\text{type Lang'} \ a = \text{AST} \text{ LangDom} \ (\text{Full a})
\]

Without \((\rightarrow)\) and \text{Full}, the distinction would be lost.

### 2.3 Simple interpretation

Just as we have used \text{Sym} and \((::)\) to construct expressions, we can use them for pattern matching:

\[
\begin{align*}
& \text{eval}_\text{NUM} :: \text{Expr}_3 \ a \rightarrow a \\
& \text{eval}_\text{NUM} \ (\text{Sym} \ (\text{Num} \ n)) = n \\
& \text{eval}_\text{NUM} \ (\text{Sym Add} :$ a :$ b) = \text{eval}_\text{NUM} \ a + \text{eval}_\text{NUM} \ b \\
& \text{eval}_\text{NUM} \ (\text{Sym Mul} :$ a :$ b) = \text{eval}_\text{NUM} \ a + \text{eval}_\text{NUM} \ b \\
\end{align*}
\]

Note the similarity to \text{evalExpr}_3. Here is a small example to show that it works:

\[
\begin{align*}
& \text{Main} \> \text{eval}_\text{NUM} \ (\text{num} \ 5 \ '\text{mul}' \ \text{num} \ 6) \\
& \Rightarrow 30
\end{align*}
\]

For later reference, we also define a rendering interpretation:

\[
\begin{align*}
& \text{render}_\text{NUM} :: \text{Expr}_3 \ a \rightarrow \text{String} \\
& \text{render}_\text{NUM} \ (\text{Sym} \ (\text{Num} \ n)) = \text{show} \ n \\
& \text{render}_\text{NUM} \ (\text{Sym Add} :$ a :$ b) = \\
& \quad \quad " \text{'++ render}_\text{NUM} \ a + ' + " \text{'++ render}_\text{NUM} \ b + '"
\end{align*}
\]

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 \text{AST 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 \text{Expr}_1, to extending non-recursive types, such as \text{NUM}. Fortunately, this problem has already been solved in Data Types \textit{à la Carte} (DTC). DTC defines the type composition operator in \texttt{Listing 1}, which can be seen as a higher-kindred version of the \textit{Either} type. We demonstrate its use by defining two new symbol domains:

\[
\begin{align*}
& \text{data Logic a where} \quad -- \text{Logic expressions} \\
& \quad \text{Not} :: \text{Logic} \ (\text{Bool} \rightarrow \text{Full Bool}) \\
& \quad \text{Eq} :: \text{Eq} \ a \Rightarrow \text{Logic} \ (a \rightarrow a \rightarrow \text{Full Bool})
\end{align*}
\]

\[
\begin{align*}
& \text{data If a where} \quad -- \text{Conditional expression} \\
& \quad \text{If} (\text{Bool} : a \rightarrow a : \rightarrow \text{Full a})
\end{align*}
\]

These can now be combined with \text{NUM} to form a larger domain:

\[
\text{type Expr a = ASTF (NUM :+: Logic :+: If) a}
\]

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

\[
\begin{align*}
& \text{data ExprRef a where} \\
& \quad \text{Num} :: \text{Int} \rightarrow \text{ExprRef} \text{ Int} \\
& \quad \text{Add} :: \text{ExprRef} \text{ Int} \rightarrow \text{ExprRef} \text{ Int} \rightarrow \text{ExprRef} \text{ Int} \\
& \ldots \\
& \quad \text{Not} :: \text{ExprRef} \text{ Bool} \rightarrow \text{ExprRef} \text{ Bool} \\
& \ldots \\
& \quad \text{If} :: \text{ExprRef} \text{ Bool} \rightarrow \text{ExprRef} \text{ a} \rightarrow \text{ExprRef} \text{ a} \\
& \quad \quad \rightarrow \text{ExprRef} \text{ a}
\end{align*}
\]

Unfortunately, the introduction of \((::+):\) means that constructing expressions becomes more complicated:

\[
\begin{align*}
& \text{not :: Expr Bool} \rightarrow \text{Expr Bool} \\
& \text{not a = Sym} \ (\text{Inj}_R \ (\text{Inj}_L \ \text{Not})) :$ a \\
& \text{cond :: Expr Bool} \rightarrow \text{Expr a} \rightarrow \text{Expr a} \rightarrow \text{Expr a} \\
& \text{cond c t f = Sym} \ (\text{Inj}_L \ (\text{Inj}_R \ \text{If})): \text{c :$ t :$ f}
\end{align*}
\]
Fortunately, DTC has a solution to this problem too. The amount of injections will only grow as the domain gets larger. For-
The symbols are now tagged with injection constructors, and the
will be used qualified in this paper.

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

Listing 1: Composition of symbol domains (part of DTC interface)

```
data (dom₁ :+: dom₂) a where
  Injₗ :: dom₁ a -> (dom₁ :+: dom₂) a
  Injₘ :: dom₂ a -> (dom₁ :+: dom₂) a

infixr :+:;
```

Listing 2: Symbol subsumption (part of DTC 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 (sym ::<: (sym :+: dom)) where
  inj = Injₗ
  prj (Injₗ a) = Just a
  prj _ = Nothing

instance (sym₁ ::<: dom) => (sym₁ ::<: (sym₂ ::<: dom)) where
  inj = Injₘ o inj
  prj (Injₘ a) = prj a
  prj _ = Nothing

-- Additional instance for AST
instance (sub ::<: sup) => (sub ::<: AST sup) where
  inj = Sym o inj
  prj (Sym a) = prj a
  prj _ = Nothing
```

The symbols are now tagged with injection constructors, and the
amount of injections will only grow as the domain gets larger. For-
tunately, DTC 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. The final instance also
takes care of injecting the Sym constructor from the AST type. We
can now define not as follows:

```
not :: (Logic ::<: dom) => ASTF dom Bool -> ASTF dom Bool
  not a = inj Not <$> a
```

The prj function in Listing 2 is the partial inverse of inj. Just
like inj allows one to avoid a nest of Injₗ/Injₘ constructors in
construction, prj avoids a nest of injection constructors in pattern
matching (see section 3.2). The instances of (:<:) essentially per-
form 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 List-
ing 3. Note that the types have now become more general. For ex-
ample, the type

```
  (≡) :: (NUM ::<: dom) => ASTF dom Int -> ASTF dom Int
```

says that (≡) works with any domain dom that contains NUM. Inform-
ally, 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 = (num 5 :+: num 0) :+: num 6
```

This means that such expressions can be evaluated by the earlier
function evalNUM, which only knows about NUM:

```
-<Main> evalNUM ex₂
  30
```

Still, the type is general enough that we are free to use ex₂ 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, ex₃ is a valid expression in any language that includes
Logic and NUM.

3.1 Functions over extensible languages

The evaluation function evalNUM 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
```

```
num :: (NUM ::<: dom) => Int -> ASTF dom Int
  (⊕) :: (NUM ::<: dom)
      => ASTF dom Int -> ASTF dom Int 
  (⊙) :: (NUM ::<: dom)
      => ASTF dom Int -> ASTF dom Int 
  (⇒) :: (Logic ::<: dom, Eq a)
      => ASTF dom a -> ASTF dom Bool 
```

```
  condition :: (If ::<: dom)
      => ASTF dom Bool
      => ASTF dom a -> ASTF dom dom a 
```

```
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
```

```
inflix 6 ⊕
inflix 7 ⊙
```

Listing 3: Extensible language front end
In the symbol case, the Add to the general as possible, with only a it accepts terms from any language that includes symbol is an addition. This is where the in an expression, say, we need to be able to tell whether a given symbols involved. If we want to count the number of additions the function that performs the optimization x + 0 → x:

\[
\text{optAddTop} :: \text{NUM} <: \text{dom} \rightarrow \text{ASTF} \text{dom} a \rightarrow \text{ASTF} \text{dom} a
\]

| Just Add | \( \text{prj} s = 1 \) |
| otherwise | \( = 0 \)

\[
\text{countAdds} (s :$ a) = \text{countAdds s} + \text{countAdds a}
\]

In the symbol case, the \( \text{prj} \) 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:

\[
\text{countAdds} :: \text{NUM} <: \text{dom} \Rightarrow \text{AST dom} a \rightarrow \text{Int}
\]

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 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 \text{countAdd} is completely independent of Logic and If. Still, it can be used with expressions containing those constructs (such as \text{ex}i).
- We have not sacrificed any type-safety.

3.2 Pattern matching

The encoding we use does come with a certain overhead. This is particularly visible when doing nested pattern matching. Here is a function that performs the optimization \( x + 0 \rightarrow x \):

\[
\text{optAddTop} :: \text{NUM} <: \text{dom} \Rightarrow \text{ASTF} \text{dom} a \rightarrow \text{ASTF} \text{dom} a
\]

\[
\begin{align*}
\text{optAddTop} &= (\text{sym}) \cdot (\text{prj} :\text{Just Add}) :\text{Just Add} :$ a :$ (\text{prj} :\text{Just (Num 3))) = a \\
\text{optAddTop} &= a = a
\end{align*}
\]

While view patterns have the advantage that they can be nested, doing so tends to lead to long lines. For this reason, it is often preferable to use a sequence of pattern guards.

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 and 7, we will look at more general types of traversals.

According to Hinze and L"oh [9], support for generic programming consists of two essential ingredients: (1) a way to write overloaded functions, and (2) 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 some generic functions in this paper. For example, size works for all possible AST types, and \text{countAdds} works for all types (AST dom) where the constraint (NUM <: dom) is satisfied.3 For size, all cases are covered by the default cases, while \text{countAdds} has one special case, 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 \text{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 \text{evalG} and \text{renderG} are defined for a single, closed domain. To make them extensible, we need to make the domain abstract, just like we did in \text{countAdds}. However, we do not want to use \text{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 simple case for Sym and one for (\text{:}):\text{ Int}

\[
\begin{align*}
\text{evalG} :: (\forall a . \text{domain} a \rightarrow \text{Denotation} a) & \\
\text{evalG} f (\text{Sym s}) &= f s \\
\text{evalG} f (\text{s :$ a}) &= \text{evalG} f s s f \text{evalG} f a \\
\text{type family} \quad \text{Denotation} \text{sig} \\
\text{type instance} \quad \text{Denotation} \text{sig} = a \\
\text{type instance} \quad \text{Denotation} (\text{Full a}) = a \\
\ \quad \rightarrow \quad \text{Denotation} a \\
\text{type instance} \quad \text{Denotation} \text{sig} = a & \\
\text{kind} \quad \text{Denotation} \text{sig} = \text{a} \\
\text{evalSym} :: \text{NUM} a \rightarrow \text{Denotation} a \\
\text{evalSym} \text{mul} = (\ast ) \\
\text{evalSym} \text{add} = (+)
\end{align*}
\]

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

\[
\text{Main} \cdot \text{kind}! \text{Denotation} (\text{Int :\text{Full Bool}) = \text{Int} :\text{Bool}
\]

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

\[
\begin{align*}
\text{evalSym} :: \text{NUM} a \rightarrow \text{Denotation} a \\
\text{evalSym} \text{mul} &= (\ast ) \\
\text{evalSym} \text{add} &= (+)
\end{align*}
\]

3 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.
Because this definition only has to deal with non-recursive symbols, it is very simple compared to evalNUM. We can now plug the generic and the type-specific functions together and use them to evaluate expressions:

```haskell
+Main+> evalG evalSymNUM ex2
30
```

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 evalSymNUM function extensible. The way to do this is to put it in a type class:

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

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

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

```haskell
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:

```haskell
instance (Eval sub1, Eval sub2) ⇒ Eval (sub1 :+: sub2) where
eval (Inj₁ s) = eval s
eval (Inj₂ s) = eval s
```

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

```haskell
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:

```haskell
+Main+> eval (ex₂) :: 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 :$ num 5) evaluates to the denotation (5 +). We call such interpretations spine-compositional.

When making a generic version of renderNUM we might try to use the following interface:

```haskell
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:

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

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):

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

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

```haskell
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.

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

```haskell
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-terms. The Render instance for (:+:) is analogous to the Eval instance, so we omit it. This concludes the definition of rendering for extensible languages.

```haskell
+Main+> render (ex₂ :: Expr Int)
"((5 + 0) + 6)"
```

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 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:

```haskell
+Main+> putStrLn $ compile (ex₂ :: Expr Int)
v3 = 5
v4 = 0
v1 = (v3 + v4)
v2 = 6
v0 = (v1 + v2)
```

Listing 4 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. The monad also has a state in order to be able to generate fresh variables.

Listing 5 defines the fully generic parts of the compiler. Note the similarity between the types of compileArgs and renderArgs. One 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 analogous to those of the Render class. Finally, the compile function takes care of running the CodeGen and extracting the written program.

The code in Listings 4 and 5 is completely generic—it does not mention anything about the symbols involved, apart from the assumption of them being instances of Compile. In Listing 6 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.

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

```
  ex₄ = condition (num 1 == num 2) (num 3) ex₂
  *Main* putStrLn $ compile (ex₄ :: Expr Int)
  v₂ = 1
  v₃ = 2
  v₁ = (v₂ == v₃)
  if v₁ then
    v₀ = 3
  else
    v₆ = 5
    v₇ = 0
    v₄ = (v₆ + v₇)
    v₅ = 6
    v₀ = (v₄ * v₅)
```

5. Implicit and explicit recursion

So far, our functions have all been defined using explicit recursion. But there is nothing stopping us from defining convenient recursion schemes as higher-order functions. For example, the AST instances for renderArgs and compileArgs (see section 4) both perform the same kind of fold-like bottom-up traversal which can be captured by the general combinator 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 = fold f 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-terms. Despite the traversal of the spine, fold should not be confused with a “spine fold” such as gfoldl from Scrap Your Boilerplate [11]. Rather, we are folding over the whole syntax tree, and go is just used to collect the sub-results in a list. This way of using ordinary lists to hold the result of sub-terms is also used in the Uniplate library [15] (see the para combinator).

6 Thanks to Déval Gergely for the technique of parameterizing the compiler on the result location.
As a demonstration, we show how to 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.

Despite the usefulness of functions like fold, it is important to stress that our traversals are by no means restricted to fold-like patterns. We can fall back to explicit recursion, or define new instances for AST to suit our needs. As an example of a function that does not suit the fold pattern, we define term equality. The generic code is as follows:

```haskell
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 (Inj₁ s1) (Inj₁ s2) = equal s1 s2 && equal a1 a2
equal _ _ = False

instance (Equality sub₁, Equality sub₂) ⇒ Equality (sub₁ :+: sub₂) where
equal (Inj₂ s1) (Inj₂ s2) = equal s1 s2
equal (Inj₃ s1) (Inj₃ s2) = equal s1 s2
equal _ _ = False
```

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

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

We see that term equality comes out very naturally as an explicitly recursive function. Expressing this kind of recursion (simultaneous traversal of two terms) in terms of fold is possible, but quite tricky (for a general method, see the generic version of `zipWith` in reference [12]). In section 7, we will see another example where explicit recursion is useful.

6. Regaining type-safety

The use of a list to hold the interpretation of sub-terms (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 the refutable pattern matching in implementations of renderArgs).
- All intermediate results are required to have the same type and cannot depend on the type of the individual sub-expressions.

We can make the problem concrete by looking at the local function go that traverses the spine in fold:

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

Now, consider folding an expression with Add as its top-level symbol: fold f (Sym Add :$ x :$ y), for some algebra f and sub-expressions x and y. This leads to the following unfolding of go:

```haskell
  go (Sym Add :$ x :$ y) [] =
  go (Sym Add :$ x) [fold f y] =
  go (Sym Add) [fold f x, fold f y]
```

In this sequence of calls, go is used at the following types:

```haskell
  go :: AST dom (Full Int) → [b] → b
  go :: AST dom (Int :-: Full Int) → [b] → b
  go :: AST dom (Int :-: Full Int) → Int :-: Full Int) → [b] → b
```

We see that the type of the term gradually changes to reflect that sub-terms are stripped away; the number of arrows (:-:) determines the number of missing sub-terms. However, the type of the list remains the same, even though its contents grows in each iteration. This is the root of the problem with fold. What we need instead is a list-like type—we will call it Args—indexed by a symbol signature, and with the property that the number of arrows determines the number of elements in the list.

With such a list type, the go function will get a type of this form:

```haskell
  go :: ∀ a . AST dom a → Args a → ...
```

Specifically, in the last recursive call in the above example, go will have the type:

```haskell
  go :: AST dom (Int :-: Int :-: Full Int) → Args (Int :-: Int :-: Full Int) → ...
```

The first argument is an expression that is missing two sub-terms, and the intention is that the second argument is a two-element list containing the result of folding those particular sub-terms.

6.1 Typed argument lists

A definition of Args that fulfills the above specification is the following:

```haskell
data Args c sig where
  Nil :: Args c (Full a)
  (:=} :: c (Full a) -> Args c sig
  infixr *:
```

Here we have added a parameter c which is the type constructor for the elements. The elements are of type c (Full a) where a varies with the position in the signature. Each cons cell (:=) imposes an additional arrow (:->) in the signature, which shows that the number of elements is equal to the number of arrows. Here is an example of a list containing an integer and a Boolean, using Maybe as type constructor:

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

The reason for making the elements indexed by Full a rather than just a is to be able to have lists with expressions in them. It is not possible to use (ASTF dom) as the type constructor because ASTF is a type synonym, and, as such, cannot be partially applied. But because the elements are indexed by Full a, we can instead use (AST dom) as type constructor. Lists of type Args (AST dom) are used, for example, when using recursion schemes to transform expressions as we will do in the following section.
6.2 Type-safe fold

We are now ready to define a typed version of fold:

\[
\text{typedFold} :: \forall \text{dom} c \,
\begin{array}{l}
  \forall a . \text{dom} a \rightarrow \text{Args} c a \rightarrow c (\text{Full} (\text{Result} a)) \\
  \rightarrow \forall a . \text{ASTF} \text{ dom} a \rightarrow c (\text{Full} a)
\end{array}
\]

typedFold f a = go a Nil

where

\[
\begin{array}{l}
go :: \forall a . \text{ASTF} \text{ dom} a \rightarrow \text{Args} c a \\
  \rightarrow c (\text{Full} (\text{Result} a)) \\
  \text{go (Sym s) as} = f s \text{ as} \\
  \text{go (s :$ a) as} = \text{go s (typedFold f a :+ as)}
\end{array}
\]

Note the close correspondence to the definition of the original fold. The Result type function simply gives the result type of a signature:

\[
\begin{array}{l}
\text{type family} \quad \text{Result} \quad \text{sig} \\
\text{type instance} \quad \text{Result} (\text{Full} a) = a \\
\text{type instance} \quad \text{Result} (\text{a :+ sig}) = \text{Result sig}
\end{array}
\]

The Args list ensures 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 define the function everywhere that applies a function uniformly across an expression. It corresponds to the combinator with the same name in Scrap Your Boilerplate [11]:

\[
\begin{array}{l}
\text{everywhere} :: \forall a . \text{ASTF} \text{ dom} a \rightarrow \text{ASTF dom a} \\
  \rightarrow \forall a . \text{ASTF} \text{ dom} a \rightarrow \text{ASTF dom a}
\end{array}
\]

everywhere f = typedFold (\lambda s \rightarrow f \circ \text{appArgs} (\text{Sym s}))

appArgs :: \text{ASTF dom sig} \rightarrow \text{Args (AST dom) sig} \\
\rightarrow \text{ASTF} \text{ dom (Result} \text{ sig})

appArgs a Nil = a \\
appArgs s (a :+ as) = appArgs (s :$ a) as

The algebra receives the symbol and its transformed arguments. The general function appArgs is used to apply the symbol to the folded arguments, and f is applied to the newly built expression.

We can now use everywhere to apply \text{optAddTop} from section 3.2 bottom-up over a whole expression:

\[
\begin{array}{l}
\text{query} :: \forall \text{dom} a c \\
  \cdot \text{query f a = go a Nil}
\end{array}
\]

In typedFold, the function f is applied across all nodes, which is why it is polymorphic in the symbol signature. In the case of query, f is only used at the top-level symbol, which is why we can allow the constraint (a ~ Result b) (the scope of a is now the whole definition). This constraint says that the top-most symbol has the same result type as the whole expression. By reducing the required polymorphism, we make query applicable to a larger set of functions. We note in passing that typedFold can be defined in terms of query, but leave the definition out of the paper.

One example where query is useful is when defining generic context-sensitive traversals. As a slightly contrived example, imagine that we want to replace the previously defined optimization everywhere \text{optAddTop} so that it is performed everywhere, except in certain sub-expressions. Also imagine that we want each symbol to decide for itself whether to perform the optimization in its sub-terms, and we want to be able to add cases for new symbol types in a modular way.

Because we need to be able to add new cases, we use a type class:

\[
\begin{array}{l}
\text{class OptAdd sym dom where} \\
\text{optAddSym :: sym a \rightarrow \text{Args (AST dom) a}} \\
  \rightarrow \text{AST dom (Full (Result a))}
\end{array}
\]

(The need for the second parameter will be explained shortly.) The idea is that the class method returns the optimized expression given the top-level symbol and its sub-terms. However, we do not want to use \text{optAddSym} as the algebra in typedFold. This is because typedFold traverses the expression bottom-up, and when
the function is to join the results of a symbol and its sub-terms, it is already too late to decide that certain sub-terms should remain unoptimized. Rather, we have to let optAddSym receive a list of unoptimized sub-terms, so that it can choose whether or not to recurse depending on the symbol.

We can now use query to lift optAddSym to operate on a complete syntax tree:

```haskell
optAdd :: OptAdd dom dom ⇒ ASTF dom a → ASTF dom a
optAdd = query optAddSym
```

Before we define instances of the OptAdd class we need a default implementation of its method:

```haskell
optAddDefault :: (sym :<: dom, OptAdd dom dom)
⇒ sym a → Args (AST dom) a
⇒ AST dom (Full (Result a))
opAddDefault s as = appArgs (Sym (inj s))
  o mapArgs optAdd
```

This function calls optAdd recursively for all arguments and then applies the symbol to the optimized terms. The `mapArgs` function is used to map a function over an `Args` list:

```haskell
mapArgs :: ∀ a . c1 (Full a) → c2 (Full a))
→ ∀ a . Args c1 a → Args c2 a)
mapArgs f Nil = Nil
mapArgs f (a ::* as) = f a :* mapArgs f as
```

In the optimization of `NUM`, we make a special case for addition with zero, and call the default method for all other cases. The optimization of `Logic` uses only the default method.

```haskell
instance (NUM :<: dom, OptAdd dom dom)
⇒ OptAdd NUM dom where
  optAddSymAdd (a :*: zero :*: Nil) = Just (Num 0)
  prj t = prj t = ...
  | Just (Num 0) ← prj t = ...

instance (If :<: dom, NUM :<: dom, OptAdd dom dom)
⇒ OptAdd If dom where
optAddSymIf (cond :*: t :*: f :*: Nil)
⇒ Just (Num 0) ← prj t = ...
```

8. **Mutually recursive types**

Many languages are naturally defined as a set of mutually recursive types. For example, the following is a language with expressions and imperative statements:

```haskell
type Var = String

data Expr a where
  Num :: Int → Expr Int
  Add :: Expr Int → Expr Int → Expr Int
  Exec :: Var → Stmt → Expr a

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

The purpose of the `Exec` construct is to return the contents of the given variable after executing the imperative program. `Assign` writes the result of an expression to the given variable. In the AST model, it is not directly possible to group the symbols so that only some of them are available at a given node. However, it is possible to use type-level “tags” to achieve the same effect. In the encoding below, the types in the symbol signatures are tagged with `E` or `S` depending on whether they represent expressions or statements.

```haskell
data ExprDom a where
  NumSym :: Int → ExprDom (Full (E Int))
  AddSym :: ExprDom (E Int) → ExprDom (E Int) → ExprDom (S : Full (E Int))
  ExecSym :: Var → ExprDom (S : Full (E a))

data StmtDom a where
  AssignSym :: Var → StmtDom (E a : Full S)
  SeqSym :: StmtDom (S : S : Full S)
```

```haskell
instance (If :<: dom, NUM :<: dom, OptAdd dom dom)
⇒ OptAdd If dom where
optAddSymIf (cond :*: t :*: f :*: Nil)
⇒ Just (Num 0) ← prj t = ...
```

The purpose of the second parameter of the OptAdd class is to let instances declare constraints on the whole domain. This is needed, for example, to be able to pattern match on the sub-terms, as the NUM instance does. As a nice side effect, it is even possible to pattern match on constructors from a different symbol type. For example, in the `If` instance, we can pattern match on `Num` simply by declaring `(NUM :<: dom)` in the class context:

```haskell
instance (Logic :<: dom, OptAdd dom dom)
⇒ OptAdd Logic dom where
optAddSym = optAddDefault
```

Now, to show the point of the whole exercise, imagine we want to avoid optimization in the branches of a conditional. With the current setup, this is completely straightforward:

```haskell
instance (If :<: dom, OptAdd dom dom)
⇒ OptAdd If dom where
  optAddSymIf (c :*: t :*: f :*: Nil) = appArgs (Sym (inj If))
  (optAdd c :*: t :*: f :*: Nil)
```

This instance chooses to optimize only the condition, while the two branches are passed unoptimized.

The instance for `(:*:)` concludes the definition of OptAdd:

```haskell
instance (OptAdd sub1 dom, OptAdd sub2 dom)
⇒ OptAdd (sub1 :+: sub2) dom where
  optAddSym (InjI a) = optAddSym a
  optAddSym (InjII a) = optAddSym a
```

The purpose of the second parameter of the OptAdd class is to let instances declare constraints on the whole domain. This is needed, for example, to be able to pattern match on the sub-terms, as the NUM instance does. As a nice side effect, it is even possible to pattern match on constructors from a different symbol type. For example, in the `If` instance, we can pattern match on `Num` simply by declaring `(NUM :<: dom)` in the class context:

```haskell
instance (If :<: dom, NUM :<: dom, OptAdd dom dom)
⇒ OptAdd If dom where
optAddSymIf (cond :*: t :*: f :*: Nil)
⇒ Just (Num 0) ← prj t = ...
```

9. **The Syntactic library**

The abstract syntax model presented in this paper is available in the **Syntactic** library, available on Hackage. In addition to the AST type and the generic programming facilities, the library provides various building blocks for implementing practical EDSLs:

- Language constructs (conditionals, tuples, etc.)
- Interpretations (evaluation, equivalence, rendering, etc.)

---

7. [http://hackage.haskell.org/package/syntactic-1.0](http://hackage.haskell.org/package/syntactic-1.0)
• Transformations (constant folding, code motion, etc.)
• Utilities for host-language interaction (the Syntactic class \[2, 16\], observable sharing, etc.)

Being based on the extensible AST type, these building blocks are generic, and can quite easily be customized for different languages. A particular aim of SYNTACTIC is to simplify the implementation of languages with binding constructs. To this end, the library provides constructs for defining higher-order abstract syntax, and a number of generic interpretations and transformations for languages with variable binding.

9.1 Practical use-case: Feldspar

Feldspar \[3\] is an EDSL for high-performance numerical computation, in particular for embedded digital signal processing applications. Version 0.5.0.1\(^8\) 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 \[16\], where we show how to add monadic constructs and support for mutable data structures to Feldspar without changing the existing implementation.

As a concrete example from the implementation, here is a functional for loop used for iterative computations:

```haskell
data Loop a where
  ForLoop :: Type st =>
    Loop (Length -- # iterations
      :-> st    -- initial state
      :-> (Index -> st -> st) -- step function
      :-> Full a) -- final state
```

The first argument is the number of iterations; the second argument is the initial state. The third argument is the step function which, given the current loop index and state, computes the next state. The third argument is of function type, which calls for a way of embedding functions as AST terms. SYNTACTIC provides different ways of doing so, but the nice thing—and a great advantage of using SYNTACTIC—is that the embedding of functions is handled completely independently of the definition of `ForLoop`.

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.

10. Related work

Data Types à la Carte \[18\] (DTC) is an encoding of extensible data types in Haskell. Our syntax tree model inherits its extensibility from DTC. Bahr and Hvitved \[4\] 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. Given that instances for said type classes can be generated automatically (as Bahr and Hvitved do), the difference is by no means fundamental. Still, our method can generally be considered to be more lightweight with slightly less encoding overhead. The original DTC paper only considered untyp'd expressions. Bahr and Hvitved extend the model to account for typed syntax trees (as all trees in this paper are). This change also lets them handle mutually recursive types in essentially the same way as we describe in section 8.

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 \[18\]), the combination of explicit recursion and generic traversals appears to be rather unexplored. In this paper we have shown how to support this combination, demonstrating that generic traversals over extensible data types are not restricted to predefined recursive patterns.

Lämmel and Ostermann \[13\] 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 \[7\], 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. In contrast to DTC and our model, this technique limits interpretations 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 \[10\].)

There exist a number of techniques for data-type generic programming in Haskell (see, for example, references \[11, 14\]). An extensive, though slightly dated, overview is given in reference \[17\]. However, these techniques do not qualify as solutions to the expression problem, as they do not provide a way to extend existing types with new constructors. Rather, the aim is to define generic algorithms that work for many different types. The spine view \[9\] is a generic view for the Scrap Your Boilerplate \[11\] 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 different in practice, and solve different problems.

Another use of a spine data type is found in Adams’ Scrap Your Zipper \[1\], which defines a generic zipper data structure. The Left data type—similar to our AST—holds the left siblings of the current position. Just like for AST, its type parameter specifies what arguments it is missing. The Right data type—reminiscent of our Arg—holds the right siblings of the current position, and its type parameter specifies what arguments it provides. This similarity suggests that it might be possible to implement a similar generic zipper for the AST type.

\(^8\) See slides by Lämmel and Kisselyov “Spin-offs from the Expression Problem” http://userpages.uni-koblenz.de/~laemmel/TheEagle/resources/xproblem2.html

\(^9\) http://hackage.haskell.org/package/feldspar-language-0.5.0.1
require resorting to dynamic typing (for that particular fragment of the code), which makes the approach slightly less type-safe than ours.

11. Discussion
In this paper we have focused on the AST model and the basic principles for programming with it. To remain focused, we have left out many details that are important when implementing an embedded language but still not fundamental to the underlying syntax model. Such details include how to deal with variable binding and syntactic annotations. The SYNTACTIC library has support for these aspects (with varying degree of stability), but it is important to stress that all of this extra functionality can be implemented on top of the existing AST type. So while SYNTACTIC is still developing, the AST model appears to be rather mature.

One important aspect of extensible syntax that we have not treated in this paper is the ability to ensure that certain constructs are present or absent at certain passes in a compiler. Bahr and Hvitved have demonstrated how to do this with Data Types à la Carte, using a desugaring transformation as example. The example is directly transferable to our model.

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 these changes to affect the underlying AST type.

Our syntax tree encoding imposes a certain run-time overhead over ordinary data types. Although we have not investigated the extent of this overhead, we have not noticed any performance problems due to the encoding in the Feldspar implementation. Still, the performance impact should be investigated, as it may become noticeable when dealing with very large programs.

12. Conclusion
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 patterns. 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. Even though our model comes with convenient recursion schemes, it is by no means restricted to fixed traversals. The user has essentially the same freedom as when programming with ordinary data types to define general recursive traversals.

Acknowledgments
This work has been funded by Ericsson, the Swedish Foundation for Strategic Research (SSF) and the Swedish Basic Research Agency (Vetenskapsrådet). The author would like to thank the following people for valuable discussions, comments and other input: Jean-Philippe Bernardy, Koen Claessen, Dévai Gergely, Patrik Jansson, Oleg Kiselyov, Anders Persson, Norman Ramsey, Mary Sheeran, Josef Svenningsson, Wouter Swierstra and Meng Wang. The anonymous reviewers also helped improving the paper.

References