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.2.11 [*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

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.

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

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.

ICFP'12, September 9–15, 2012, Copenhagen, Denmark.
Copyright © 2012 ACM 978-1-4503-1054-3/12/09...$10.00

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[2] 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.

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

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 parameter `a` is the type of the *semantic value* of the expression; i.e. the value obtained by evaluating the expression. (For $\texttt{Expr}_1$, 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:

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

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 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)
```

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
```

[3] http://www.cse.chalmers.se/~emax/documents/axelsson2012generic.lhs

```
*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 `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 $\texttt{Expr}_1$ 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 T_{ref} of the form

$$\mathsf{T}_{ref}\ \mathtt{a} \rightarrow \mathsf{T}_{ref}\ \mathtt{b} \rightarrow \ldots \rightarrow \mathsf{T}_{ref}\ \mathtt{x}$$

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 $\texttt{Expr}_2$) 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, $\texttt{Op}_1$ is a *non-terminal* that needs two sub-trees in order to make a complete syntax tree. $\texttt{Op}_2$ 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)
```

Without `(:→)` 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 :: Expr_3 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 `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 :: Expr_3 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 $\texttt{Expr}_1$, to extending non-recursive types, such as `NUM`. Fortunately, this problem has already been solved in Data Types à la Carte (DTC). DTC 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:

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

data If a where     -- Conditional expression
  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
  ...
```

```
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
```

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

infixr :+:
```

Listing 1: Composition of symbol domains (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          = Inj_L
  prj (Inj_L a) = Just a
  prj _        = Nothing

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

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

Listing 2: Symbol subsumption (part of DTC interface)

The symbols are now tagged with injection constructors, and the amount of injections will only grow as the domain gets larger. Fortunately, 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 $\texttt{Inj}_L/\texttt{Inj}_R$ constructors in *construction*, `prj` avoids a nest of injection constructors in *pattern matching* (see section 3.2). 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 List-

ing 3. Note that the types have now become more general. For example, the type

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

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

```
num :: (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

num             = 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 3: Extensible language front end

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 = (num 5 ⊕ num 0) ⊙ num 6
```

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

```
*Main> eval_NUM ex₂
30
```

Still, the type is general enough that we are free to use $\texttt{ex}_2$ 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, $\texttt{ex}_3$ is a valid expression in *any language* that includes `Logic` and `NUM`.

3.1 Functions over extensible languages

The evaluation function $\texttt{eval}_{NUM}$ 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. If we want to count the number of additions in an expression, say, 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 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 (such as $\texttt{ex}_3$).
- 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 $x + 0 \rightarrow x$:

```
optAddTop :: (NUM :<: dom) ⇒ ASTF dom a → ASTF dom a
optAddTop (add :$ a :$ b)
  | Just Add     ← prj add
  , Just (Num 0) ← prj b   = a
optAddTop a = a
```

(This function only rewrites the top-most node; in section 6.2, we will see how to apply the rewrite across the whole expression.) Note the sequencing of the pattern guards. An alternative is to use the `ViewPatterns` extension to GHC instead:

```
optAddTop
  ((prj→ Just Add) :$ a :$ (prj→ Just (Num 0))) = a
```

```
optAddTop a = a
```

While view patterns have the advantage that they can be nested, doing so tends to lead to long lines. For this reason, it is ofter 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 Löh [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 `countAdds` works for all types `(AST dom)` where the constraint `(NUM :<: dom)` is satisfied.[5] For `size`, all cases are covered by the default cases, while `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 `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 $\texttt{eval}_{NUM}$ and $\texttt{render}_{NUM}$ 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 do not 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)`:

```
*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     = (*)
```

[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.

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
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 $\texttt{evalSym}_{NUM}$ 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 sub1, Eval sub2)
      ⇒ Eval (sub1 :+: sub2) where
  eval (InjL s) = eval s
  eval (InjR 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 (ex3 :: 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 $\texttt{render}_{NUM}$ 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.

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:

```
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.

```
*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:

```
*Main> putStr $ 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.[6] 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> putStr $ 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 = 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-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 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 4: 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 5: 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 6: Extensible compiler: type-specific code

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 custom recursion schemes, whenever needed. As an example of a function that does not suit the `fold` pattern, we define term equality. The generic code is 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 (Inj_L s1) (Inj_L s2) = equal s1 s2
  equal (Inj_R s1) (Inj_R s2) = equal s1 s2
  equal _ _                   = False
```

And, 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
```

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 ex-

pected 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`:

```
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`:

```
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:

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

termines the number of elements in the list.

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

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

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

```
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:

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

```
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 `c` 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`:

```
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 the original `fold`. 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
```

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 subexpressions. 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]:

```
everywhere :: (∀a . ASTF dom a → ASTF dom a)
           → (∀a . ASTF dom a → ASTF dom a)
everywhere f = typedFold (λs → f ∘ appArgs (Sym s))

appArgs :: AST dom sig → Args (AST dom) sig
                       → ASTF dom (Result 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 `optAddTop` from section 3.2 bottom-up over a whole expression:

```
*Main> render (ex3 :: Expr Bool)
"(((5 + 0) * 6) == ((5 + 0) * 6))"
*Main> render $ everywhere optAddTop (ex3::Expr Bool)
"((5 * 6) == (5 * 6))"
```

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 finally define a version of `Render` that avoids refutable pattern matching (here showing only the `NUM` instance):

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

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
```

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 by new instances. The recursive calls are made in the instances for `AST` and `(:+:)`, and these are not affected by the instances for the symbol types. To have full freedom in writing generic recursive functions, one needs 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)
```

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 change the previously defined optimization `everywhere 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:

```
class OptAdd sym dom where
  optAddSym :: sym a → Args (AST dom) a
                     → AST dom (Full (Result a))
```

(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 `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:

```
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:

```
optAddDefault :: (sym :<: dom, OptAdd dom dom)
              ⇒ sym a → Args (AST dom) a
                      → AST dom (Full (Result a))
optAddDefault s = appArgs (Sym (inj s))
                ∘ 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:

```
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.

```
instance (NUM :<: dom, OptAdd dom dom)
      ⇒ OptAdd NUM dom where
  optAddSym Add (a :* zero :* Nil)
    | Just (Num 0) ← prj zero = optAdd a
  optAddSym s as = optAddDefault s as

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:

```
instance (If :<: dom, OptAdd dom dom)
      ⇒ OptAdd If dom where
  optAddSym If (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`:

```
instance (OptAdd sub1 dom, OptAdd sub2 dom)
      ⇒ OptAdd (sub1 :+: sub2) dom where
  optAddSym (Inj_L a) = optAddSym a
  optAddSym (Inj_R 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:

```
instance (If :<: dom, NUM :<: dom, OptAdd dom dom)
      ⇒ OptAdd If dom where
  optAddSym If (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:

```
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.

```
data E a  -- Expression tag
data S    -- Statement tag

data ExprDom a where
  NumSym  :: Int → ExprDom (Full (E Int))
  AddSym  :: ExprDom (E Int :→ E Int :→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)

type Expr_enc a = ASTF (ExprDom :+: StmtDom) (E a)
type Stmt_enc   = ASTF (ExprDom :+: StmtDom) S
```

For example, `ExecSym` has the signature `(S :→ ...)`, which means that its argument must be one of the symbols from `StmtDom`, since these are the only symbols that result in `Full S`. Because the tags above reflect the structure of the `Expr` and `Stmt` types, we conclude that $\texttt{Expr}_{enc}$ and $\texttt{Stmt}_{enc}$ are isomorphic to those types. Following

this recipe, it is possible to model arbitrary mutually recursive syntax trees using `AST`.

9. 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` 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

- 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:

```
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 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 untyped 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.

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

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 Zippers [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

`Args`—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.

Outside the Haskell world, an interesting approach to implementing EDSLs is Modelyze [6]. The Modelyze language is specifically designed to be a host for embedded languages. It has built-in support for open data types, and such types can be traversed generically by pattern matching on symbolic applications in much the same way as our `countAdds` example (section 3.1). However, generic traversals

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

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. How-

ever, 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 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. 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 Founda-

tion 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

[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*, volume 7241 of *LNCS*. Springer, 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] D. Broman and J. G. Siek. Modelyze: a gradually typed host language

for embedding equation-based modeling languages. Technical Report UCB/EECS-2012-173, EECS Department, University of California, Berkeley, Jun 2012.

[7] 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.

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

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

[11] 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.

[12] R. Lämmel and S. P. Jones. Scrap more boilerplate: reflection, zips, and generalised casts. In *Proceedings of the ninth ACM SIGPLAN international conference on Functional programming*, ICFP '04, pages 244–255. ACM, 2004.

[13] 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.

[14] 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.

[15] 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.

[16] A. Persson, E. Axelsson, and J. Svenningsson. Generic monadic constructs for embedded languages. In *23rd International Symposium on Implementation and Application of Functional Languages, IFL 2011*, volume 7257 of *LNCS*, 2012.

[17] 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.

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