Abstract Syntax Graphs for

Bruno C. d. S. Oliveira
National University of Singapore
oliveira@comp.nus.edu.sg

Abstract

This paper presents a representation for embedded domain specific languages (EDSLs) using *abstract syntax graphs* (ASGs). The purpose of this representation is to deal with the important problem of defining operations that require *observing* or *preserving* sharing and recursion in EDSLs in an expressive, yet easy-to-use way. In contrast to more conventional representations based on abstract syntax trees, ASGs represent sharing and recursion explicitly as binder constructs. We use a functional representation of ASGs based on *structured graphs*, where binders are encoded with *parametric higher-order abstract syntax*. We show how adapt to this representation to *well-typed* ASGs. This is especially useful for EDSLs, which often reuse the type system of the host language. We also show an alternative *class-based encoding* of (well-typed) ASGs that enables *extensible* and *modular* well-typed EDSLs while allowing the manipulation of sharing and recursion.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Functional Languages

General Terms Languages

Keywords Observable Sharing, DSLs, Graphs, Haskell.

1. Introduction

A *domain-specific language* (DSL) is a programming language targeted at a particular problem domain. DSLs offer a vocabulary, language constructs and a semantics crafted for that domain.

An *embedded DSL* (EDSL) [14] is a DSL that is implemented by reusing various elements of a (general-purpose) host language (such as the syntax, type-checker, or binding constructs). While being somewhat less flexible than writing a dedicated compiler or interpreter for a DSL, the embedded approach greatly reduces the cost of the implementation. Furthermore, integration with the host language comes for free, and mixing the DSL with the host language or other DSLs is easy.

Representation of the syntax of an EDSL in the host language are typically positioned between two extremes: a *shallow* embedding provides a very thin layer over the host language, implementing the DSL constructs directly by their semantics. As a result, there is no support for inspecting the syntax and manipulations of the DSL programs are difficult. *Deep* embeddings solve this problem by making the syntax of the DSL explicit – usually as an *abstract*

PEPM'13, January 21–22, 2013, Rome, Italy.

Domain Specific Languages

Andres Löh
Well-Typed LLP
andres@well-typed.com

syntax tree (AST). One or several semantics of the DSL can be given by defining interpretation functions over the AST, and transformations of the AST prior to interpretation are possible. The AST approach is well supported by functional languages such as Haskell and has been used to implement several EDSLs [4, 19].

In many real-life EDSLs, preserving and observing sharing and recursion are essential for implementing domain-specific transformations and optimizations. However, ASTs need to be complemented with *explicit* environments to allow transformations that rely on observing sharing or recursion. Furthermore, doing so, we suddenly need to keep track of names and binding, forcing us to a lower level of programming, where we have to worry about problems such as avoiding name capture in substitutions or preventing dangling references.

Abstract Syntax Graphs This paper suggests using an *abstract syntax graph* (ASG) representation for EDSLs. ASGs make it easy to guarantee that terms are well-scoped. They allow the observation and preservation of sharing and recursion. Furthermore, functions on ASGs can be defined in a natural way, using pattern matching.

Technically speaking ASGs are realized using Oliveira and Cook's *structured graphs* [20]. Such structured graphs offer a generic purely functional representation of cyclic structures in pure functional languages such as Haskell. Structured graphs use binders, represented using Chlipala's *parametric higher-order abstract syntax* (PHOAS) [7], to model cycles and sharing.

Related Work With ASTs, a possible approach to help with the issues regarding the management of explicit environment and generation of fresh labels [4] is to use *monads* [26]. However, while monads can make name management more bearable, they cannot completely hide the fact that we have to work on a low level, and monads alone cannot ensure the well-scopedness of a program.

Sometimes, one would like to ensure not only well-scoped, but also well-typed expressions in an EDSL. Guaranteeing well-typedness becomes even harder in the presence of explicit environments. Both the ASTs and the respective environments need to be enriched with additional type and binding information. Examples of such approaches include well-typed and well-scoped analysis and transformation of grammars, which have been a hot topic recently [2, 3, 9]. Baars et al. [2, 3] use well-typed ASTs in combination *typed references* and *typed environments* in their typed transformations. The relationship between a reference and an environment is statically enforced in a similar way to well-scoped/typed de Bruijn indices [1]. All this infrastructure relies on sophisticated type-level machinery and several Haskell extensions.

Another option is to represent sharing and recursion implicitly, by relying on the sharing of the host language. This is great from the usability point of view because we can reuse the host language syntax to create sharing. Unfortunately, this is usually too fragile or precludes the possibility of observation. To overcome the need for observing sharing and recursion in implicit representations, it is possible to use pointer or reference equality. This approach has

been used in many DSL implementations [8, 11, 18]. However, the use of references breaks *referential transparency* and significantly complicates reasoning [25]. In a language such as Haskell, we would then be forced to use monadic interfaces. Furthermore defining operations by working with references is fragile and prone to errors. In contrast, our ASGs are completely functional and avoid the need for observing and comparing pointers.

More recently, both Devriese et al. [9, 10] and Kiselyov [16] have proposed the use of recursive binders to implement explicit sharing using a type-class based representation. However a problem of such type-class based representations is that pattern matching is not supported. All operations are essentially defined as folds, making it difficult to define many transformations and optimizations that rely on both observing sharing and use more complex recursion patterns that would be most naturally expressed using nested pattern matching.

Contributions The contributions of this paper[1] are:

ASGs for EDSLs Neither Oliveira and Cook nor Chlipala considered the application of PHOAS and ASGs to EDSLs. Yet, we believe that the ASG representation is particularly valuable to EDSL developers, providing the best solution to date to the problem of observing sharing and recursion.

We make a case for the use of these techniques and we also fill in some gaps that are not covered in the earlier works on PHOAS and ASGs, that are of relevance in the context of EDSLs.

Well-typed ASGs We show how to represent deeply embedded well-typed terms with ASGs. Well-typed terms allow DSL designers to reuse (parts of) the type system of the host language for the type system of the DSL.

Oliveira and Cook develop techniques only for representing *untyped* abstract syntax using structured graphs. Chlipala's original work on PHOAS does cover a form of well-typed abstract

syntax using dependent types in the Coq theorem prover. However, conventional functional languages like Haskell do not have full-blown dependent types (although recent extensions get us very close to that [28]), so these techniques have to be adapted to use GADTs [23] instead. Also, Chlipala does not cover recursive binders and their mutually recursive generalization. While well-typed encodings of simple recursive binders are relatively straightforward to encode, encoding well-typed mutually recursive binders elegantly requires more work. To solve this problem we use *typed lists*: a generalization of both heterogeneous and homogeneous lists. With typed lists not only can we deal with well-typed mutually recursive binders, but also enforce certain size invariants that the untyped representation does not statically enforce.

Extensible and modular ASGs The issue of modularity is not considered neither by Oliveira and Cook nor by Chlipala. A popular representation of EDSLs that deals with this problem uses a class-based (typed) Church encoding representation [5, 12, 13, 22] (although this representation makes the definition of operations that use nested pattern matching more difficult). We show that ASG-based techniques can be adapted to a class-based representation, similar to the representations proposed by Devriese et al. [9, 10] and Kiselyov [16]. However, differently from these approaches, we use a PHOAS-based representation of binders and provide a simple encoding of mutually recursive binders using typed lists.

2. Sharing in EDSLs

In this section, we use a small example language to demonstrate the differences between shallow and deep embeddings as well as the issues with representing sharing.

[1] The code for this paper is available at `http://ropas.snu.ac.kr/~bruno/papers/ASGs.zip`.

In order to keep the examples as small as possible, we use an EDSL with just two constructs: the constant one, and a binary addition operator. The Haskell interface of our DSL is:

```
data Expr   -- abstract

one :: Expr
(⊕) :: Expr → Expr → Expr
```

The primary semantics we are interested in is evaluation:

```
eval :: Expr → Int
```

We are now going to contrast a shallow embedding with a deep embedding for this language.

Shallow embedding A *shallow* embedding for this language is:

```
type Expr = Int

one = 1
(⊕) = (+)
eval = id
```

We use the type Int as the representation of the expression type. Building a term in the expression language evaluates it automatically. The evaluation function eval is then just the identity function.

The shallow approach is appealing because it is so simple. Constructing terms in the DSL is as easy as constructing Haskell terms. We even inherit many features from the host-language Haskell. For example, we can use a Haskell function to generate a term in our DSL, as shown on the left hand side of Figure 1.

The term $\mathsf{tree}_\mathsf{l}\ \mathsf{n}$ (Figure 1) describes a binary tree of additions, with occurrences of one in the leaves. The function tree_l is recur-

sive, and it makes use of sharing via **let**. Both recursion and sharing are properties we do not have available in the interface of our expression DSL, yet they are available to us via the embedding into Haskell.

The use of sharing is essential here for efficient evaluation of the term. Without sharing, $\mathsf{tree}_\mathsf{I}\ \mathsf{n}$ would contain exponentially many additions and constants in n. By using sharing, the term is internally represented as a graph of just linear size. The identifier shared is bound to an Expr represented as an Int, and even though shared is being used twice, it is being evaluated only once. The evaluation of $\mathsf{eval}\ (\mathsf{tree}_\mathsf{I}\ 2)$ is sketched on the left hand side of Figure 2. Note how $1 + 1$ is evaluated only once, and its result (2) is shared.

However, shallow embeddings come at a price. We are committing to a specific semantics – in this case, evaluation. Often, that is too limited in practice. We may want to do other things with expressions: for example, show the original term via a function

$$\mathsf{text} :: \mathsf{Expr} \to \mathsf{String}$$

or transform the expression into a different (perhaps optimized) form, or translate the expression into a different language with a different set of constructs available. With a shallow embedding, we are out of luck. Our implementation picks one semantics and once we construct a term, we interpret the expression according to that semantics, losing the original structure of the expression.

Deep embedding A *deep* embedding solves this problem:

$$\mathbf{data}\ \mathsf{Expr} = \mathsf{One} \mid \mathsf{Add}\ \mathsf{Expr}\ \mathsf{Expr}$$

$$\mathsf{one} = \mathsf{One}$$
$$(\oplus) = \mathsf{Add}$$

$$\mathsf{eval}\ \mathsf{One} = 1$$
$$\mathsf{eval}\ (\mathsf{Add}\ \mathsf{e}_1\ \mathsf{e}_2) = \mathsf{eval}\ \mathsf{e}_1 + \mathsf{eval}\ \mathsf{e}_2$$

We now choose to represent the language constructs by their *abstract syntax*. A value of type Expr corresponds to the abstract syntax tree of a term in our DSL. We thus retain the structure of the terms we construct and can interpret them in various ways. We can,

```
treeI :: Int → Expr
treeI 0 = one
treeI n = let shared = treeI (n − 1) in shared ⊕ shared
```

```
treeE :: Int → Expr
treeE 0 = one
treeE n = let_ (treeE (n − 1)) (λshared → shared ⊕ shared)
```

Figure 1. Contrasting building a massively shared tree either using Haskell's implicit sharing (left) or explicit sharing in our DSL (right)

```
  eval (treeI 2)
= let shared = treeI (2 − 1) in shared + shared
= let shared = let shared' = treeI (1 − 1) in shared' + shared'
  in shared + shared
= let shared = let shared' = 1 in shared' + shared'
  in shared + shared
= let shared = 1 + 1 in shared + shared
= let shared = 2 in shared + shared
= 2 + 2
= 4
```

```
  eval (treeI 2)
= eval (let shared = treeI (2 − 1) in Add shared shared)
= let shared = treeI (2 − 1) in eval shared + eval shared
= let shared = let shared' = treeI (1 − 1) in Add shared' shared'
  in eval shared + eval shared
= let shared' = treeI (1 − 1)
  in (eval shared' + eval shared') + (eval shared' + eval shared')
= let shared' = One
  in (eval shared' + eval shared') + (eval shared' + eval shared')
= (1 + 1) + (1 + 1)
= 2 + 2
= 4
```

Figure 2. Contrasting evaluation of eval (treeI 2) using both the shallow (left) and deep (right) embedding

for example, evaluate it as shown in the definition of eval above, but we can also show it in textual form:

```
text :: Expr → String
text One         = "1"
text (Add e1 e2) = "(" ++ text e1 ++ " + " ++ text e2 ++ ")"
```

In a similar way, we could define additional interpretation functions such as an optimizer or a translator to a different language. Typically, the interpretation functions are *folds* (also known as *catamorphisms*), i.e., functions that traverse the structure of the underlying input datatype (here Expr) closely and recurse exactly where we encounter a recursive subterm in the datatype definition.

However, the greater flexibility comes at a price. Consider treeI again, defined exactly as before (that is, the treeI definition in the left side of Figure 1). The identifier shared now is a term of the datatype Expr, no longer of type Int. If we evaluate the term treeI n

using eval, we traverse the structure of the Expr, thereby destroying the sharing. The term will take exponentially long to evaluate (or to show, or to transform). The evaluation of eval (tree_1 2) in the deep setting is sketched on the right side of Figure 2. Note how the pattern matching in eval destroys the sharing introduced by **let**, and how $1 + 1$ is evaluated twice.

Haskell's **let** still allows us to construct implicitly shared terms of type Expr, but this sharing is not observable and is also quite fragile. Traversing such an implicitly shared term using any interpretation function will destroy all sharing.

Explicit sharing A solution is to make sharing *explicit* in the embedded language. This will enable us to observe and preserve the sharing that we wish to have in a term in a robust way.

It is quite clear that we need to add a **let**-like construct, but there is quite some design flexibility in the detail. We would like to avoid having to deal with names, binding and substitution ourselves, as this is tedious and error-prone, and would make the DSL much more tricky to use or at least to implement.

One promising approach to model binding in the embedded language is *higher-order abstract syntax* (HOAS) [24]. With HOAS the function space of the implementation language Haskell is used in order to express a shared term in the embedded language:

```
data Expr = One | Add Expr Expr | Let Expr (Expr → Expr)
```

We no longer have to use Haskell's **let** in order to express sharing in the embedded language. Next to one and $(\oplus)$ (that can be defined

as before) we have to augment the interface of our language with an explicit sharing construct:

```
let_ :: Expr → (Expr → Expr) → Expr
let_ = Let
```

We have to adapt the construction of shared terms to use this explicit sharing construct. The resulting modification of function tree_I, called tree_E, is shown on the right side of Figure 1.

However there is a problem: How do we extend the evaluator to cover the case for Let? Here is an attempt:

$$\mathsf{eval}\ (\mathsf{Let}\ \mathsf{e}_1\ \mathsf{e}_2) = \textbf{let}\ \mathsf{shared} = \mathsf{eval}\ \mathsf{e}_1\ \textbf{in}\ \mathsf{eval}\ (\mathsf{e}_2\ (\ldots\ \mathsf{shared}))$$

We would like to feed the evaluated shared shared expression to e_2, but it has the wrong type! The body of the Let expects an Expr, but we have an Int. At the position of ..., we need a function that can quote the interpreted term back into the original language [17]. Alternatively, we have to add another constructor to Expr, because the existing constructors are not really expressive enough (we have One, but not arbitrary integer literals). Note that other interpretation functions such as text would need other quotation functions.

But before we delve too deep into this issue, we should point out another problem with higher-order abstract syntax: the space of type Expr $\rightarrow$ Expr is too large. In order to express binding faithfully, we want the syntactic shape of the resulting expression to be independent of the expression being shared. However, a Haskell function of Expr $\rightarrow$ Expr allows us to plug in functions that **case**-analyze the incoming value and return different expressions depending on the outcome of that analysis.

Abstract syntax graphs Making sharing explicit means that the abstract syntax representation becomes a graph rather than a tree. Although our effort to use HOAS to model ASGs has some problems, Oliveira and Cook [20] have shown a functional representation of graphs that solves these problems. The idea is to use *parametric* higher-order abstract syntax (PHOAS) [7] instead of HOAS to model binders.

```
data Expr a = One | Add (Expr a) (Expr a)
            | Var a | Let (Expr a) (a → Expr a)
```

With PHOAS the whole expression datatype is now parameterized by the type of shared expressions a. We have two new constructors compared to our original type, one for variables that embeds a value

of type a in Expr, and one for Let. The body of the Let now receives a variable of type a rather than a value of type Expr.

If we now require expressions in our language to make no assumption about the variables, i.e., to be *polymorphic* in a, then (unlike HOAS) we cannot analyze the shared expression. Furthermore, Var serves as a generic way to quote intermediate results of interpretation functions. We can thus make the following definition for closed expressions, i.e., expressions with no free variables:

```
type ClosedExpr = ∀a.Expr a
```

We use ClosedExpr to explicitly refer to closed terms in our DSL and Expr a to construct terms or write interpreter functions.

We define one and $(\oplus)$ as before:

```
one = One
(⊕) = Add
```

In addition, we define a function let_ that wraps Let:

```
let_ :: Expr a → (Expr a → Expr a) → Expr a
let_ e1 e2 = Let e1 (λx → e2 (Var x))
```

The PHOAS underpinning guarantees that we cannot do anything with the argument we obtain in the body of the Let but to use it as a variable. But having to invoke Var explicitly at every use site is somewhat tedious – the wrapper performs this work for us.

With these definitions in place, we can define our explicitly shared tree_E function again. It looks just like the definition on the right side of Figure 1, but its type becomes Int → ClosedExpr. We now have the choice whether to use Haskell's host-language **let** construct while doing meta-programming by writing a term like on the left side of Figure 1, or if we explicitly want to express sharing in the embedded language using let_ like on the right side.

Preserving sharing The evaluator can now be defined as follows:

```
eval :: Expr Int → Int
```

```
eval One          = 1
eval (Add e1 e2)  = eval e1 + eval e2
eval (Var n)      = n
eval (Let e1 e2)  = eval (e2 (eval e1))
```

The interpreter expects an Expr Int – it thus assumes that variables are of type integer for the purpose of evaluating an expression. However, a ClosedExpr is polymorphic in the variable type, so it will naturally be accepted by eval. In the Var case, we find an integer and can return it. In the Let case, we have to provide an integer for the value of the bound variable: we pass eval e_1. Note that this achieves sharing, because lambda-bound terms in Haskell are automatically shared. Therefore calling eval ($tree_E$ 30) now will return the result 1073741824 almost immediately.

Observing sharing Furthermore, it is easy to write other interpretation functions for expressions. Here is a function that computes a textual representation of the given term. Here, rather than *preserving* the sharing, we are interested in *observing* it:

```
text :: ClosedExpr → String
text e = go e 0
  where
    go :: Expr String → Int → String
    go One          _ = "1"
    go (Add e1 e2) c =
      "(" ++ go e1 c ++ " + " ++ go e2 c ++ ")"
    go (Var x)      _ = x
    go (Let e1 e2)  c =
      "(let " ++ v ++ " = " ++ go e1 (c + 1) ++
      " in " ++ go (e2 v) (c + 1) ++ ")"
      where v = "v" ++ show c
```

In text, we internally use an interpretation of type Int → String, maintaining a counter. In the case for Let, we actually print a let-construct rather than unfolding the expression. Evaluating text (tree_E 2) yields

```
"(let v0 = (let v1 = 1 in (v1 + v1)) in (v0 + v0))"
```

Inlining As a final example, let us look at a transformation that removes explicit sharing again, effectively inlining all let-bound variables:

```
inline :: Expr (Expr a) → Expr a
inline One          = One
inline (Add e1 e2)  = Add (inline e1) (inline e2)
inline (Var x)      = x
inline (Let e1 e2)  = inline (e2 (inline e1))
```

This operation produces the original expression, but unfolds Let constructs. For the purposes of inline, variables are themselves expressions. For text (inline (tree_E 2)), we obtain

```
"((1 + 1) + (1 + 1))"
```

again, and eval (inline (tree_E 30)) takes forever to compute.

Summary We have shown that there are situations where we need to observe or preserve sharing in an embedded DSL. Preserving sharing may be needed for performance reasons (as in the tree_E example), or it may be needed for operations that inspect shared terms and treat them in a particular way (as in the text example).

PHOAS offers a safe yet convenient way to make sharing explicit and encode ASGs. The user can reuse Haskell's own scoping rules and does not have to worry about managing names. Differ-

ently from classic HOAS encoding, terms that perform case analysis on bound variables is forbidden.

3. (Mutual) recursion

In this section, following Oliveira and Cook [20], we will extend the solution to sharing presented in Section 2 to recursive and mutually recursive bindings.

To this end, we extend our example language with a few new constructs. For now, let us move from the constant "one" to allowing arbitrary integer literals, add a construct for checking if a term is equal to "zero", and add lambdas and application:

```
type ClosedExpr = ∀a.Expr a

data Expr a = Lit Int | Add (Expr a) (Expr a)
            | IfZero (Expr a) (Expr a) (Expr a)
            | Var a | Let (Expr a) (a → Expr a)
            | Lam (a → Expr a) | App (Expr a) (Expr a)
```

The constructor Lit takes an arbitrary integer literal. Addition is exactly as before. In IfZero, we take a condition, a then-part and an else-part. Variables (Var) and Let are unchanged. A lambda (Lam) is a binding construct. It therefore takes a function of type $a \to Expr\ a$ in the same way as the body of Let. Application (App) takes a function and an argument.

We define a few "smart constructors" to facilitate constructing terms again:

```
(⊕)          = Add
(⊙)          = App
let_ e1 e2 = Let e1 (λx → e2 (Var x))
lam_ e     = Lam (λx → e (Var x))
```

Evaluation Let us look at how to extend the evaluator. We no

longer have the luxury that all terms of our embedded language evaluate to integers. Instead, terms of our language now have a type τ where the type language is as follows:

$$\tau ::= \mathsf{Int} \mid \tau \to \tau$$

We have some flexibility encoding the type system when we embed the language: we can encode the types of the terms dynamically, and allow the language to represent ill-typed terms that will fail at run-time; or we can use Haskell's type system to enforce that terms in the language must be well-typed. Both settings have some merit. We will therefore look at the dynamic approach here and deal with the static encoding of the types later, in Section 5.

The result of evaluation is now a tagged value:

```
data Value = N Int | F (Value → Value)
```

Functions are represented as Haskell functions in this simple setting – we might move to a representation using an explicit closure using an environment in a larger setting. The evaluator changes slightly as a consequence, and now looks as follows:

```
eval :: Expr Value → Value
eval (Lit i)              = N i
eval (Add e1 e2)          = add (eval e1) (eval e2)
eval (IfZero e1 e2 e3)    = ifZero (eval e1) (eval e2) (eval e3)
eval (Var x)              = x
eval (Let e1 e2)          = eval (e2 (eval e1))
eval (Lam e)              = F (λv → eval (e v))
eval (App e1 e2)          = app (eval e1) (eval e2)
```

We now have to tag values whenever we produce them, such as in the cases for Lit and Lam. For constructors such as Add, IfZero and App we write wrapper functions that check (at run time) whether the arguments have the correct types and throw an error if not:

```
add (N m) (N n)  = N (m + n)
```

```
ifZero (N n) v1 v2 = if n == 0 then v1 else v2
app (F f) v        = f v
```

Of course, we could also define a monadic evaluator that would be a total function and return Maybe Value instead of Value.

Here is a small example:

```
example =
  let_ (lam_ (λx → x ⊕ Lit (− 1))) (λdec →
  let_ (lam_ (λf → lam_ (λx →
       f ⊙ (f ⊙ x)))) (λtwice →
      (twice ⊙ twice ⊙ dec ⊙ Lit 10)))
```

This expression encodes the term

```
let dec     x = x − 1
    twice f x = f (f x)
in  twice twice dec 10
```

Note that the two uses of twice are at different types. Evaluating the expression eval example yields N 6 as expected.

Recursion Recursion is simple to add, by introducing an additional constructor that represents fixed points:

```
data Expr a = ...    -- as before
            | Mu (a → Expr a)
```

This binding construct is very similar to Lam. Both constructs introduce a bound variable that scopes over the entire body of the expression.

The idea is that using Mu, we can encode a recursive function such as multiplication (in terms of addition) as follows:

```
mu_ e = Mu (λx → e (Var x))

mul :: ClosedExpr
mul = lam_ (λm → mu_ (λrec → lam_ (λn →
        ifZero n (Lit 0) (m ⊕ (rec ⊙ (n ⊕ Lit (− 1)))))))
```

The evaluator must of course be adapted as well:


```
eval :: ClosedExpr → Value
eval ...      = ...    -- as before
eval (Mu e) = fix (λv → eval (e v))
```

The new case maps Mu to Haskell recursion using the fix function:

```
fix :: (a → a) → a
fix f = let r = f r in r
```

Using the **let** here for the result introduces additional sharing.

As we did in Section 2, we can also write other semantic functions on our DSL such as a function text to display the expression. Semantic functions can now observe and preserve recursion as needed.

It is also possible to define a recursive let-construct in terms of Let and Mu:

```
letrec :: (Expr a → Expr a) → (Expr a → Expr a) → Expr a
letrec e1 e2 = Let (Mu (λx → e1 (Var x))) (λx → e2 (Var x))
```

Mutually recursive definitions The Mu construct is sufficient for expressing simple recursion, but we cannot easily express the definition of several mutually recursive bindings. For languages with an expressive internal structure we might be able to encode mutual recursion in terms of simple recursion within the DSL, but we want our techniques to be widely applicable and not impose strong requirements on the DSLs.

When defining mutually recursive definitions we need to bind several variables at once (one for each mutually recursive defini-

tion). As an example, consider the following Haskell term:

```
let dec  x     = x − 1
    even x t e = if x == 0 then t  else odd  (dec x) t e
    odd  x t e = if x == 0 then e else even (dec x) t e
in even 4 1 0
```

The function even takes a number and two continuations. If the number is even, the first continuation is returned, if it is odd, then the second continuation is returned instead. The given call returns 1, because 4 is even.

The functions even and odd are mutually recursive, and both depend on dec. This kind of mutually recursive binding is commonplace in a language like Haskell.

To deal with mutually recursive bindings, we add a new constructor called LetRec:

```
data Expr a = ...    -- as before
            | LetRec ([a] → [Expr a]) ([a] → Expr a)
```

We have a list of declarations now. Each of the declarations can refer to each of the others. So all declarations are parameterized by a list of inputs. The body also can refer to each of the bindings, therefore it is parameterized over the same list. The type system cannot express the intuition that all three lists that occur in the type above are supposed to have the same length. We will be able to make this precise in Section 5.

We also define a wrapper that applies Var to all the variables:

```
letrec_ :: ([Expr a] → [Expr a]) →
           ([Expr a] → Expr a) → Expr a
letrec_ es e = LetRec (λxs → es (map Var xs))
                      (λxs → e  (map Var xs))
```

Now we can define our example term as follows:

$$
\begin{aligned}
&\mathsf{evenOdd} = \mathsf{letrec\_}\ (\lambda{\sim}[\mathsf{dec}, \mathsf{even}, \mathsf{odd}] \rightarrow \\
&\quad [\mathsf{lam\_}\ (\lambda \mathsf{x} \rightarrow \mathsf{x} \oplus \mathsf{Lit}\ (-1)) \\
&\quad , \mathsf{lam\_}\ (\lambda \mathsf{x} \rightarrow \mathsf{lam\_}\ (\lambda \mathsf{t} \rightarrow \mathsf{lam\_}\ (\lambda \mathsf{e} \rightarrow \\
&\qquad \mathsf{IfZero\ x\ t}\ \ (\mathsf{odd}\ \ \odot (\mathsf{dec} \odot \mathsf{x}) \odot \mathsf{t} \odot \mathsf{e})))) \\
&\quad , \mathsf{lam\_}\ (\lambda \mathsf{x} \rightarrow \mathsf{lam\_}\ (\lambda \mathsf{t} \rightarrow \mathsf{lam\_}\ (\lambda \mathsf{e} \rightarrow
\end{aligned}
$$

```
    lfZero x e (even ⊙ (dec ⊙ x) ⊙ t ⊙ e))))
  ])
  (λ[dec, even, odd] → even ⊙ Lit 4 ⊙ Lit 1 ⊙ Lit 0)
```

The only slightly tricky point is that we need to delay the pattern match on the list of variables in the first argument to letrec_ (using $\sim$), because in an interpretation function, Haskell will not be able to determine the number of elements in this list before looking at the body of the lambda.

We can extend an interpretation function such as the evaluator to cope with the presence of LetRec as follows:

```
eval :: ClosedExpr → Value
eval ...                = ...    -- as before
eval (LetRec es e) = eval (e (fix (map eval ∘ es)))
```

Reusing native let syntax It can be argued that despite the advantages of using explicit sharing, it is still less convenient to use let_ or letrec_ than to use Haskell's native **let** construct.

Many EDSLs therefore use Haskell's **let**, but recover the sharing information by inspecting the internal representation of the term, using an impure function. The function reifyGraph, from the `data-reify` package [11], provides such functionality. This function returns a graph representing subterms using numbers – a representation that is neither particularly safe nor directly suitable for further computations.

We can combine reification with our ASG approach. We start with arithmetic expressions with just literals and addition:

```
data Expr_D = Lit_D Int | Add_D Expr_D Expr_D
```

The goal is to convert an implicitly shared term such as $\text{tree}_\text{I}\ 3$ (using tree_I from Figure 1 with type $\text{Int} \to \text{Expr}_\text{D}$, with obvious definitions of one and $\oplus$) into an explicitly shared term of type Expr. In order to be able to use `data-reify` on terms of type Expr_D, we have to define a *pattern functor* [15] for expressions

```
data Expr_F r = Lit_F Int | Add_F r r
```

that has the same structure as $\mathsf{Expr_D}$, but abstracts from recursive calls. We furthermore have to instantiate a class MuRef to make the relationship between Expr and $\mathsf{Expr_F}$ precise.

Using the function reifyGraph we can convert a value of type $\mathsf{Expr_D}$ into a conventional graph representation based on a list of type $[(\mathsf{Int}, \mathsf{Expr_F}\ \mathsf{Int})]$ associating integer labels with partial terms. For example, reifyGraph $(\mathsf{tree_I}\ 1)$ returns the graph

```
Graph [(1, Add_F 2 2), (2, Lit_F 1)] 1
```

where the final 1 points to the root node.

We now define a function build that transforms such a list of nodes into an explicitly shared ClosedExpr:

```
build :: [(Int, Expr_F Int)] → Int → ClosedExpr
build env root =
  letrec_ (λvs → let go (Lit_F x)       = Lit x
                     go (Add_F v1 v2) =
                       Add (var vs v1) (var vs v2)
                 in map (go ∘ snd) env)
          (λvs → var vs root)
  where
    var vs n =
      fromJust (lookup n (zipWith (λ(i, _) x → (i, x)) env vs))
```

In this definition, var associates the integer labels with a variable from the list vs, and then looks up the label n. We convert between values of type $\mathsf{Expr_F}$ a and $\mathsf{Expr_D}$ a using the function go.

Using build, we can now write programs like

```
test = do (Graph env r) ← reifyGraph (tree_I 3)
          print (text (build env r))
```

where we create an implicitly shared term of type Expr_D with tree_I and then convert it to a value of type ClosedExpr using reifyGraph and build. We can then process the resulting ASG with functions that observe sharing (such as text).

Summary Our ASG representation is suitable for representing various binding constructs in Haskell DSLs. However, there are at least two situations in which the type safety we are able to obtain is not satisfactory yet. Firstly, if the language itself has a type system, then we might want to have a datatype explicitly encoding well-typed terms, which has consequences on how we have to define the binding constructs. Secondly, for mutually recursive bindings we can either add on a constructor for each number of bindings and go via tuples, or we can add one constructor working with lists as we have done. However, this requires maintaining an implicit invariant that we match on no more bindings than we are defining, and we have to perform a lazy pattern match.

In the following, we will show how to fix these issues by assigning more precise types to our language constructs.

4. Typed Lists

This section presents *typed lists*. Typed lists are a generalization of both homogeneous and heterogeneous lists of statically known length. We will make use of typed lists for encoding *well-typed* mutually recursive bindings in Sections 5 and 6.

Typed lists are defined using the following datatype:

```
data TList :: (∗ → ∗) → ∗ → ∗ where
  TNil :: TList f ()
  (:::) :: f t → TList f ts → TList f (t, ts)
```

A typed list TList f ts is parameterized by a type constructor f of kind $* \to *$ and indexed by a *signature* of types ts. The signature encodes a type-level list, with () representing the empty list and (t, ts) representing the list with t as the head and ts as the tail.[2] The signature determines both the length of the typed list and the types of its elements. Where the signature contains a type t, the corresponding element has type f t.

Heterogeneous and homogeneous lists Typed lists can be viewed as a generalization of heterogeneous lists of statically known length. Heterogeneous lists correspond to the case where f = I, and I is the identity type constructor:

```
newtype I a = I {unI :: a}
```

Using I we can encode the following heterogeneous list:

```
hlist :: TList I (Int, (Int → Int, (Bool, ())))
hlist = I 3 ::: I (λx → x) ::: I False ::: TNil
```

In this case hlist is an heterogeneous list that contains values of type Int, Int → Int and Bool as elements, and the types of the elements are reflected in the signature.

Typed lists are also a generalization of homogeneous lists. Homogeneous lists correspond to the case where a = K b, and K b is the constant type constructor:

```
newtype K b a = K {unK :: b}
```

For example, we can encode the list [1,2,3] as follows:

---

[2] Alternatively, we could use recent GHC extensions that allow kind polymorphism and datatype *promotion* [28] to provide a more direct definition of typed lists:

```
data TList :: (k → ∗) → [k] → ∗ where
  TNil :: TList f '[]
  (:::) :: f t → TList f ts → TList f (t ': ts)
```

```
list :: TList (K Int) (t, (t1, (t2, ())))
list = K 1 ::: K 2 ::: K 3 ::: TNil
```

The use of the constant functor means that all elements are of type Int. The concrete types that occur in the signature become irrelevant; the signature merely encodes the length of the list.

Basic operations We can access the head and the tail of non-empty typed lists:

```
thead :: TList f (t, ts) → f t
thead (x ::: xs) = x

ttail :: TList f (t, ts) → TList f ts
ttail (x ::: xs) = xs
```

Unlike for regular head and tail, no pattern matching errors can occur in thead and ttail, because the type signature specifies that the input list must have at least one element.

Another useful operation is tlength, which returns the number of elements in a typed list:

```
tlength :: TList v t → Int
tlength TNil       = 0
tlength (x ::: xs) = 1 + tlength xs
```

Mapping and zipping Operations like map or zipWith have counterparts in the world of typed lists. Where map lifts a function of type a → b to a function on lists, the corresponding tmap operates on a *natural transformation* of type ∀t.f t → g t:

```
tmap :: (∀t.f t → g t) → TList f t → TList g t
tmap h TNil       = TNil
tmap h (x ::: xs) = h x ::: tmap h xs
```

Apart from the more general type, the code of tmap is the same as that for map. We can easily obtain a specialized version for homogeneous lists:

```
tmapK :: (a → b) → TList (K a) ts → TList (K b) ts
tmapK f = tmap (K ∘ f ∘ unK)
```

A generalization of zipWith for typed lists can be obtained in a similar fashion:

```
tzipWith :: (∀t.f t → g t → h t) →
            TList f ts → TList g ts → TList h ts
tzipWith f TNil       TNil       = TNil
tzipWith f (x ::: xs) (y ::: ys) = f x y ::: tzipWith f xs ys
```

Note that the type signature of tzipWith dictates that both input lists as well as the output list share a common signature and therefore must in particular be of the same length. As a result, we have to provide only two cases, where either both input lists are empty, or both input lists are non-empty.

Producers of typed lists We will also need a version of iterate that operates on typed lists. This operation is interesting because it *produces* a typed list, whereas all the functions we have defined above are *consumers* of typed lists.

While the conventional iterate function produces an infinite list, we now have to produce a list of a statically given signature, and in particular length. We therefore have to define our typed version of iterate by induction over the signature ts. As a consequence, the function cannot simply be of type

```
(a → a) → a → TList (K a) ts
```

because we have to produce a result that is polymorphic in ts, and we have no way in Haskell to analyze ts. We can, however, use a well-known type-level programming technique [6] to reflect the structure of the signature to the value level and then perform induction over the reflected signature:

```
data RList :: ∗ → ∗ where
  RNil  :: RList ()
  RCons :: RList ts → RList (t, ts)
```

Using RList, it is now straight-forward to define a version of iterate for typed lists:

```
titerate′ :: RList ts → (a → a) → a → TList (K a) ts
titerate′ RNil f n          = TNil
titerate′ (RCons xs) f n = K n ::: titerate′ xs f (f n)
```

Using type classes for producers Using titerate′ is inconvenient, because in order to invoke it, we have to pass a term of type RList ts, and constructing such a term is tedious. We can, however, use a type class to build a value of the appropriate type automatically and pass it implicitly, so that we can define a more convenient function titerate as follows:

```
titerate :: CList ts ⇒ (a → a) → a → TList (K a) ts
titerate = titerate′ cList
```

The type class CList and its instances are:

```
class CList t where
  cList :: RList t

instance CList () where
  cList = RNil

instance CList ts ⇒ CList (t, ts) where
  cList = RCons cList
```

The resulting function titerate can be used almost in the same way as iterate:

```
tenumFrom :: CList ts ⇒ Int → TList (K Int) ts
tenumFrom n = titerate (+ 1) n

test :: TList (K Int) (t1, (t2, ()))
test = tenumFrom 0
```

The main difference is that the type is important to determine how many elements will be generated. For example, test generates a list with the elements K 0 and K 1, because the signature of test is a type-level list with two elements t_1 and t_2.

5. Typed ASGs and DSLs

This section shows how to define well-typed abstract syntax graphs. We will illustrate this by adapting the interpreter presented throughout Section 3 to ensure that all terms are well-typed by construction. As for the untyped interpreter, observing sharing and recursion is possible. Because mutually recursive LetRec subsumes normal Let and Mu, we drop the latter two from the language.

Well-typed Abstract Syntax Graphs If we want to model well-typed ASGs, we have to first introduce an additional type argument that serves as the index for the type of the value being represented, and then adapt the types of the constructors in order to establish the typing rules of the embedded language.

But how do we represent variables? As the embedded language is now indexed by a type argument, variables can be of different (Haskell) types. Therefore, we change the type parameter for variables from kind $*$ to kind $* \to *$: we pass in a type function that, given a type of the embedded language, returns the associated type of variables. If we apply this strategy to our example expression language, we end up with the following datatype:

```
type ClosedExpr t = ∀f.Expr f t

data Expr (f :: * → *) :: * → * where
  Lit    :: Int → Expr f Int
  Add    :: Expr f Int → Expr f Int → Expr f Int
  IfZero :: Expr f Int → Expr f t → Expr f t → Expr f t
  Var    :: f t → Expr f t
  Lam    :: (f t1 → Expr f t2) → Expr f (t1 → t2)
  App    :: Expr f (t1 → t2) → Expr f t1 → Expr f t2
  LetRec :: CList ts ⇒ (TList f ts → TList (Expr f) ts) →
                       (TList f ts → Expr f t) → Expr f t
```

In the Var case, we pass the type t to the parameter function f to obtain a suitable variable type, as was our plan. The case for Lam shows that apart from adding type arguments everywhere, the structure of representing binders remains the same.

The case for mutually recursive bindings LetRec is more interesting. We now use typed lists (as introduced in Section 4) rather than ordinary lists. They keep track of the types of all the elements in the list, and at the same time determine the length of the list. Therefore, by using the same signature ts three times for the three occurrences of TList, we now establish *statically* that all three occurrences have exactly the same shape. This is a big improvement over the untyped encoding which does not provide such guarantees.

Furthermore, the CList ts constraint in LetRec guarantees that expressions built with this constructor support reifying the type-level list into a value of type RList ts. This is useful when we want to use producer functions like titerate to define functions over Expr.

As in the untyped setting, parametricity still ensures that we cannot inspect variables as long as an expression is polymorphic in the variable type function f. We define ClosedExpr as an abbreviation for such closed terms again.

Well-typed evaluator The code for the well-typed evaluator is:

```
eval :: Expr I t → t
eval (Lit i)              = i
eval (Add e1 e2)          = eval e1 + eval e2
eval (IfZero e1 e2 e3)    = if eval e1 == 0 then eval e2 else eval e3
eval (Var x)              = unI x
eval (Lam e)              = eval ∘ e ∘ I
eval (App e1 e2)          = (eval e1) (eval e2)
eval (LetRec es e)        = eval (e (fix (tmap (I ∘ eval) ∘ es)))
```

Unlike the interpreter we defined in Section 3, there is no need for a separate Value datatype for values. Since we used the Haskell type constructors Int and $(\rightarrow)$ to model the type language of the embedded language, we can simply use t as value type of a term that has type Expr f t. For the purposes of evaluation, we have to instantiate f with a type function that makes this relation explicit: the identity type constructor I.

The resulting interpreter is *untagged*. There are no constructors wrapping the values, and we do not need to perform any type-checking at run-time. We statically know that in each construct, the arguments we obtain are of the correct types.

While the code for the "normal" language constructs becomes simpler, the code for the binding constructs remains nearly unchanged: we only have to sprinkle coercion functions unI and I to help the type checker along.

As before, we can define wrappers for certain constructors to make the use of the language a bit more convenient. For example:

```
(⊕) = Add
one = Lit 1

lam_ :: (Expr f t1 → Expr f t2) → Expr f (t1 → t2)
lam_ e = Lam (λx → e (Var x))

letrec_ :: CList ts ⇒ (TList (Expr f) ts → TList (Expr f) ts) →
                      (TList (Expr f) ts → Expr f t) → Expr f t
```

```
letrec_ es e = LetRec (λxs → es (tmap Var xs))
                      (λxs → e   (tmap Var xs))
```


If we want non-recursive let-bindings or a simple fixed-point construct back, we can easily define these in terms of letrec_:

```
let_ :: Expr f t1 → (Expr f t1 → Expr f t2) → Expr f t2
let_ e1 e2 = letrec_ (λ_ → e1 ::: TNil) (λ(x ::: TNil) → e2 x)
```

Using the definitions above, we can still distinguish between implicitly shared and explicitly shared terms as before. The two versions tree_I and tree_E defined in Figure 1 are valid in the typed setting without any change of the code – only the type becomes

```
Int → ClosedExpr Int
```

in both cases. The implicitly shared version will still lose sharing, whereas the explicitly shared version still evaluates quickly.

In summary, the same properties regarding observable sharing and recursion apply to well-typed terms: the addition of typing information does not affect the preservation of sharing and recursion.

On the other hand, we now can no longer define terms that are ill-typed according to the type system of our DSL. For instance, example from Section 3 fails to type check, because it uses twice at two different types, but our DSL has only monomorphic types.

Printing terms Let us also look at how we have to adapt the function text that we have introduced in Section 2. Here, the relation between DSL types and result types is different compared to evaluation: regardless of the DSL type that a variable has, they are all printed as strings. We need the K type constructor instead of I:

```
text :: ClosedExpr t → String
```

```
text e = go e 0
  where
    go :: Expr (K String) t → Int → String
    ...    -- cases for Lit, Add, IfZero, App as before
    go (Var x)        _ = unK x
    go (Lam e)        c =
      "(\\ " ++ v ++ " -> " ++ go (e (K v)) (c + 1)
      where v = "v" ++ show c
    go (LetRec es e) c =
      "(let { " ++ intercalate "; " ds ++
      " } in " ++ go (e vs) c' ++ ")"
      where
        vs = tmapK (λi → "v" ++ show i) (tenumFrom c)
        c' = c + tlength vs
        ds = ttoList $
             tzipWith (λ(K v) e → K (v ++ " = " ++ go e c'))
                      vs (es vs)
```

Similarly to the evaluator code, we must add a few coercion functions (K and unK) throughout the pretty printer code.

The LetRec case is interesting again, because we have to deal with typed lists. In vs, we define the strings representing each of the bound variables. First, we generate numbers starting from the current counter c using tenumFrom. Then we map over the list, moving from type K Int to K String. How many variables are generated is determined by the type context! In the declaration of ds, we pass our typed list of strings to the declaration function es, and the type of LetRec dictates that the input lists of variables must have the same shape as the output list of bindings.

Note that tenumFrom works only for result types that actually are list types, as witnessed by the CList constraint – this is an

example for why we need to put a CList constraint in the type of the LetRec constructor.

In ds, we then take the list of variables vs and the list of expressions es vs and generate strings representing each of the bindings using tzipWith. We end up with a typed list containing elements of type K String, but we would actually like to have a list of strings at this point. The function ttoList achieves this:

```
ttoList :: TList (K a) ts → [a]
ttoList TNil          = []
ttoList (K x ::: xs) = x : ttoList xs
```

Finally, we separate each of the bindings by "; " by using the standard list function intercalate and append everything together in a single string.

6. Encodings of ASGs

In this section, we discuss an encoding of ASGs using type classes. This approach is interesting because it stands somewhere in-between a *shallow* and a *deep* embedding. Like for deep embeddings, it is possible to have multiple interpretations and perform a form of syntactic analysis. Like for shallow embeddings, it is possible to to create (but not observe) sharing using Haskell's built-in **let**. Moreover, it is easy to extend the language and add new constructs without touching existing code. For the deep embeddings we have been using in Sections 2, 3 and 5, adding a new constructor requires modifying all the interpretation functions.

Uses of sharing in the embedded language can still be explicit and therefore can be observed and as needed and we can maintain the level of type safety established in Section 5.

An additional advantage of the class-based encoding is that it is possible to provide reusable code for the binding constructs we have presented. As binding constructs are useful and similar throughout many DSLs, being able to reuse code reduces the implementation burden on DSL designers.

Encoding datatypes as type classes Hinze [12] showed that type classes provide a convenient way to define Church encodings of datatypes. Moving from a (generalized) algebraic datatype to a class is an entirely mechanical process [21]. As an example, let us consider how to encode simple well-typed arithmetic expressions like the ones presented in Section 5, i.e., we base this construction on the type Expr from Section 5, but we consider only the Lit and

Add constructors for now:

```
class ArithAlg expr (f :: * → *) where
  lit  :: Int → expr f Int
  (⊕) :: expr f Int → expr f Int → expr f Int
```

Looking at the transformation from a syntactic perspective, all we did was: change the datatype declaration to a class, transform the data constructors into methods of the class, and use a class parameter expr wherever the original datatype Expr was being used.

Semantically, ArithAlg encodes the signature of algebras of the original datatype. Instances of ArithAlg correspond to fold-like functions over that datatype. For the reader interested in knowing more about this technique we suggest several resources available elsewhere [5, 12, 21].

Encoding Binders We can follow the same recipe to encode binders. Let us again consider the datatype Expr from Section 5, ignoring all but the two binding-related constructors Var and LetRec. We obtain the following type class:

```
class BindAlg expr f where
  var    :: f t → expr f t
  letrec :: CList ts ⇒ (TList f ts → TList (expr f) ts) →
                       (TList f ts → expr f t) → expr f t
```

As before, it is possible to define wrappers that allow more convenient use of binding constructs, or that define simpler binding constructs in terms of letrec. As an example, here is the code for non-recursive let-bindings:

```
let_ :: BindAlg expr f ⇒ expr f t1 →
                         (expr f t1 → expr f t2) → expr f t2
let_ e1 e2 = letrec (λ_ → e1 ::: TNil) (λ(x ::: TNil) → e2 (var x))
```

Generic behavior for binders As observed by Oliveira and Cook [20], there are many operations that share a common definition for the binding constructs. We use this observation to capture this generic behavior by providing a "default" instance for BindAlg. This instance can be reused when defining suitable operations:

```
newtype Default f t = D {unD :: f t}

instance BindAlg Default f where
  var x       = D x
  letrec es e = e (fix (tmap unD ∘ es))
```

This definition turns out to be useful for operations such as evaluation or inlining – whenever we do not need to observe sharing. For functions such as text, we want to observe the binding structure and will require a different instance.

Extensibility and Modularity As Oliveira et al. [22] shows, an advantage of using the class-based approach is that in contrast to datatypes, which are closed to extension, we can add new cases to a language simply by defining another class. In other words this technique can be used to solve the *expression problem* [27].

Several DSLs share a number of common components. For example, many DSLs will have the arithmetic expressions and recursive binders that we already discussed. Adding a new set of DSL constructs is as simple as defining a new type class. For example, we can create a third class LamAlg for lambda and application:

```
class LamAlg expr f where
  lam :: (f t1 → expr f t2) → expr f (t1 → t2)
  app :: expr f (t1 → t2) → expr f t1 → expr f t2
```

If we want to state that expressions are given by the combination of the three classes we have defined above, we can denote that with

```
class (BindAlg expr f, ArithAlg expr f, LamAlg expr f) ⇒
      ExprAlg expr f
```

We can build terms in this language by applying and combining class methods from each of the three classes. Closed expressions are overloaded in the instantiation of ExprAlg:

```
type ClosedExpr t = ∀expr f.ExprAlg expr f ⇒ expr f t
```

Evaluation In order to define evaluation, we define instances for each of the classes separately. If desired, we could define these instances at different times, when we decide to extend the language with new constructs, and without touching existing code.

No instance for BindAlg is needed – we can reuse the default instance defined above. We have to provide instances for ArithAlg and LamAlg, however. A ClosedExpr t evaluates to a t, so we could choose I as the instantiation of the f arguments of the classes. However, while several functions on expressions might share the same type signature, there can be only a single instance each for ArithAlg Default I and LamAlg Default I. Therefore, we define a new type isomorphic to I specifically for the evaluation function:

```
newtype Eval t = E {unE :: t}

eval :: Default Eval t → t
eval = unE ∘ unD

toE :: t → Default Eval t
toE = D ∘ E
```

The instances are then straightforward:

```
instance ArithAlg Default Eval where
  lit i       = toE i
```

$$e_1 \oplus e_2 \quad = \mathsf{toE}\ (\mathsf{eval}\ e_1 + \mathsf{eval}\ e_2)$$

```
instance LamAlg Default Eval where
  lam f       = toE (eval ∘ f ∘ E)
  app e1 e2 = toE ((eval e1) (eval e2))

instance ExprAlg Default Eval
```

Note that eval can be applied directly to a term of type ClosedExpr.

Shared trees If we abbreviate one = lit 1, and use Int → ClosedExpr Int as the type signature, then the two versions tree_I and tree_E from Figure 1 work once again. However, the behavior here is similar to what we discussed in Section 2 for shallow DSLs: both versions preserve sharing. There is no indirection of data constructors when using the class-based encoding: a term is directly encoded as its interpretation (or actually, all possible interpretations).

Still, there are advantages to using explicit sharing, as implicit sharing remains rather fragile: if we, for example, define a function

```
double :: ClosedExpr t → ClosedExpr t
```

that traverses an expression and doubles all literals, then the traversal over an implicitly shared term will destroy the sharing. We also need to be explicit whenever we have to *observe* sharing.

Printing terms As an example of an operation that observes sharing and does not make use of the default instance for BindAlg, we return to our text function. We only show the instance for BindAlg:

```
newtype Text (f :: ∗ → ∗) t = T {text′ :: Int → String}

instance BindAlg Text (K String) where
  var x = T (λ_ → unK x)
  letrec es e = T (λc →
    let vs = tmapK (λi → "v" ++ show i) (tenumFrom c)
        c′ = c + tlength vs
        ds = ttoList $
               tzipWith (λ(K v) e → K (v ++ " = " ++ text′ e c′))
                        vs (es vs)
    in "(let { " ++ intercalate "; " ds ++
```

```
    " } in " ++ text' (e vs) c' ++ ")")
```

The code is nearly the same as that given in Section 5. The actual text function wraps text′:

```
text :: ∀t.ClosedExpr t → String
text e = text' (e :: Text (K String) t) 0
```

Summary Using the class-based encoding of ASGs is recommended whenever extensibility is a must. It is easily possible to encode well-typed terms in the class-based setting, but actually not necessary. The untyped ASGs of Section 3 can easily be translated into the class-based setting as well. A disadvantage of the class-based encoding is that it forces interpretation functions to be folds – if functions require nested pattern matches or have strange recursion behavior, they can be tricky to encode as algebras.

7. Conclusion

We propose using ASGs instead of ASTs to provide a more convenient representation of abstract syntax for EDSLs. ASGs internalize the information about sharing and recursion directly in the representation. As such, environments can in most cases be avoided, and there is no need to deal with other binding-related issues such as α-equivalence, name capture or generation of fresh variables.

We show that ASGs are flexible: they extend nicely to the generalized setting of terms with statically encoded type information, and they are suitable for class-based as well as datatype-based encodings. The ability to encode well-typed terms is particularly interesting, because many DSLs have type systems that can be encoded directly in the host language. The ability to modularize DSL constructs is important to provide flexibility and reuse.

References

[1] T. Altenkirch and B. Reus. Monadic presentations of lambda terms using generalized inductive types. In *CSL '99*, 1999.

[2] A. I. Baars and S. D. Swierstra. Type-safe, self inspecting code. In *Haskell '04*, 2004.

[3] A. I. Baars, S. D. Swierstra, and M. Viera. Typed transformations of typed grammars: The left corner transform. *Electron. Notes Theor. Comput. Sci.*, 253(7), 2010.

[4] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: hardware design in Haskell. In *ICFP '98*, 1998.

[5] J. Carette, O. Kiselyov, and C. Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *J. Funct. Program.*, 19(5), 2009.

[6] J. Cheney and R. Hinze. A lightweight implementation of generics and dynamics. In *Haskell 2002*, 2002.

[7] A. Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *ICFP'08*, 2008.

[8] K. Claessen and D. Sands. Observable sharing for functional circuit description. In *In Asian Computing Science Conference*. Springer Verlag, 1999.

[9] D. Devriese and F. Piessens. Finally tagless observable recursion for an abstract grammar model. *Journal of Functional Programming*, 22(6):757–796, November 2012.

[10] D. Devriese, I. Sergey, D. Clarke, and F. Piessens. Fixing idioms: A recursion primitive for applicative dsls. In *PEPM'13*, 2013.

[11] A. Gill. Type-safe observable sharing in Haskell. In *Haskell'09*, 2009.

[12] R. Hinze. Generics for the masses. *J. Funct. Program.*, 16(4-5), 2006.

[13] C. Hofer, K. Ostermann, T. Rendel, and A. Moors. Polymorphic

embedding of dsls. In *GPCE '08*, 2008.

[14] P. Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28, 1996.

[15] P. Jansson and J. Jeuring. Polyp – a polytypic programming language extension. In *POPL'97*, 1997.

[16] O. Kiselyov. Implementing explicit and finding implicit sharing in embedded DSLs. In *Proceedings IFIP Working Conference on Domain-Specific Languages*, 2011.

[17] E. Meijer and G. Hutton. Bananas in space: extending fold and unfold to exponential types. In *FPCA'95*, 1995.

[18] M. Might, D. Darais, and D. Spiewak. Parsing with derivatives: a functional pearl. In *ICFP '11*, 2011.

[19] J. T. O'Donnell. Overview of Hydra: A concurrent language for synchronous digital circuit design. In *IPDPS '02*, 2002.

[20] B. C. d. S. Oliveira and W. R. Cook. Functional programming with structured graphs. In *ICFP '12*, 2012.

[21] B. C. d. S. Oliveira and J. Gibbons. Typecase: a design pattern for type-indexed functions. In *Haskell '05*, 2005.

[22] B. C. d. S. Oliveira, R. Hinze, and Andres Löh. Extensible and Modular Generics for the Masses. In *TFP '06*, 2006.

[23] S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for gadts. In *ICFP'06*, 2006.

[24] F. Pfenning and C. Elliot. Higher-order abstract syntax. In *PLDI '88*, 1988.

[25] F. Pottier. Lazy least fixed points in ML. Unpublished, 2009.

[26] P. Wadler. The essence of functional programming. In *POPL'92*, 1992.

[27] P. Wadler. The expression problem. Note to Java Genericity mailing list, Nov. 1998.

[28] B. A. Yorgey, S. Weirich, J. Cretin, S. Peyton Jones, D. Vytiniotis, and J. P. Magalhães. Giving Haskell a promotion. In *TLDI '12*, 2012.