Combining Deep and Shallow Embedding of Domain-Specific Languages

Josef Svenningsson, Emil Axelsson

Chalmers University of Technology

Abstract

We present a technique to combine deep and shallow embedding in the context of compiling embedded languages in order to provide the benefits of both techniques. When compiling embedded languages it is natural to use an abstract syntax tree to represent programs. This is known as a *deep* embedding and it is a rather cumbersome technique compared to other forms of embedding, typically leading to more code and being harder to extend. In *shallow* embeddings, language constructs are mapped directly to their semantics which yields more flexible and succinct implementations. But shallow embeddings are not well-suited for compiling embedded languages.

Our technique uses a combination of deep and shallow embedding, which helps keeping the deep embedding small and makes extending the embedded language much easier. The technique also has some unexpected but welcome secondary effects. It provides fusion of functions to remove intermediate results for free without any additional effort. It also helps to give the embedded language a more natural programming interface.

1. Introduction

Domain specific languages (DSLs) provide an effective means of increasing programmer productivity [25]. In order to lessen the initial cost of implementing the DSL, many implementors choose to *embed* the language in a *host language*. Embeddings can come in many shapes and forms [19], partly dictated by the purpose of the language. This article focuses on DSLs which are designed to generate code. In this situation it is natural to use an algebraic data type to represent the abstract syntax tree (AST) of the DSL. This is known as a *deep* embedding. Deep embeddings can be cumbersome: the AST definition can grow large when each language construct has its own constructor. It is also laborious to add new language constructs as it requires changes to the AST as well as all functions manipulating the AST.

Email addresses: `josefs@chalmers.se` (Josef Svenningsson), `emax@chalmers.se` (Emil Axelsson)

In contrast, *shallow* embeddings don't require an abstract syntax tree and all the problems that come with it. Instead, language constructs are mapped directly to their semantics. Nevertheless, there are many situations in which it is convenient to have access to an AST – especially when we wish to transform expressions and generate code from them.

In this paper we present a technique for combining deep and shallow embeddings in order to achieve many of the advantages of both styles. Concretely, we propose to define DSLs using shallow embeddings which generate a deeply embedded AST. This combination turns out to provide surprising but welcome secondary effects which we explore. In particular, our technique has the following advantages:

Simplicity. By moving functionality to shallow embeddings, our technique helps keep the AST small without sacrificing expressiveness.

Abstraction. The shallow embeddings are based on *abstract data types* leading to better programming interfaces (more like ordinary APIs than constructs of a language). This has important additional benefits:

- The shallow interfaces can have properties not possessed by the deep embedding. For example, our vector interface (Section 4.9) guarantees removal of intermediate structures (see Section 5).
- The abstract types can sometimes be made instances of standard Haskell type classes, such as Functor and Monad, even when the deep embedding cannot (demonstrated in Sections 4.8, 4.9 and 6).

Extensibility. Our technique can be seen as a partial solution to the expression problem [43] as it makes it easier to extend the embedded language with new language constructs and functions.

1.1. Organization

The paper is organized as follows: In Section 2 we start by giving a more detailed introduction to shallow and deep embeddings, including a comparison of the two methods (Section 2.1). Section 3 gives a detailed description of our technique. Section 4 demonstrates the technique by defining a deep embedding and showing a number of examples of how it can be extended with new shallow language constructs. Section 5 describes how fusion comes for free as a consequence of our technique and explain in detail what guarantees it provides. Section 6 describes how to embed arbitrary monads and shows a monad for mutable data structures as an example. Finally, Section 7 discusses how the presented techniques can be scaled up to a full EDSL implementation.

Throughout this paper we will use Haskell [32] and some of the extensions provided by the Glasgow Haskell Compiler. Code from this paper be found in the following repository: `https://github.com/josefs/deep-shallow-paper`.

This article is an extended version of our paper "Combining Deep and Shallow Embedding for EDSL" which appeared in Trends in Functional Programming 2012 [38]. New material presented here includes Sections 6 and 7 which are completely new. Section 5 has been expanded with more examples of fusable data structures. The deep embedding in Section 4 has been changed in some ways: Binding is now handled using the `Lam` and `:$` constructors. We have included a `Syntactic` instance for functions, which simplifies the definition of smart constructors. Literals have been generalized to more closely match what an actual implementation would look like. We have also added a function for rendering the generated ASTs in Section 4.10. Finally, bugs have been fixed in the evaluator in Section 4 and the description of the `Option` type in Section 4.8.

2. Shallow and Deep – Pros and Cons

To explain the meaning of "deep" and "shallow" we will use the following small embedded domain specific language (EDSL) by Carlson et al. [9] as an illustrating example.

```
type Region

inRegion :: Point  → Region → Bool
circle   :: Radius → Region
outside  :: Region → Region
(∩)      :: Region → Region → Region
(∪)      :: Region → Region → Region
```

This piece of code defines a small language for regions, i.e. two-dimensional areas. It only shows the interface; we will give two implementations, one deep and one shallow.

The type `Region` defines the type of regions which is the domain we are concerned with in this example. We can interpret regions by using `inRegion`, which allows us to check whether a point is within a region or not. We will refer to functions such as `inRegion` which interpret values in our domain as *interpretation functions*. The function `inRegion` takes an argument of type `Point` and we will just assume there is such a type together with the expected operations on points.

Regions can be constructed using `circle` which creates a region with a given radius (again, we assume a type `Radius` without giving its definition). The functions `outside`, `(∩)` and `(∪)` take the complement, intersection and union of regions. As an example of how to use the language, we define the function `annulus` which can be used to construct donut-like regions given two radii:

```
annulus :: Radius → Radius → Region
annulus r1 r2 = outside (circle r1) ∩ (circle r2)
```

The first implementation of our small region EDSL will use a *shallow* embedding. The code is shown below.

```
type Region = Point → Bool

p `inRegion` r = r p
circle  r      = λp → magnitude p ≤ r
outside r      = λp → not (r p)
r1 ∩ r2        = λp → r1 p && r2 p
r1 ∪ r2        = λp → r1 p || r2 p
```

Our concrete implementation of the type Region is the type Point → Bool. We will refer to the type Point → Bool as the *semantic domain* of the shallow embedding. It is no coincidence that the semantic domain is similar to the type of the function inRegion. The essence of shallow embeddings is this:

Definition 1. *A* shallow embedding *represents language constructs as their semantics in the host language.*

In our case Region is represented exactly as a test whether a Point is within the region or not.

The implementation of the function inRegion becomes trivial; it simply uses the function used to represent regions. This is common for shallow embeddings; interpretation functions like inRegion, can make direct use of the operations used in the representation. All the other functions encode what it means for a point to be inside the respective region.

We characterize deep embeddings as follows:

Definition 2. *A* deep embedding *represents language constructs as constructors in an abstract syntax tree.*

Below is how we would represent our example language using a deep embedding.

```
data Region = Circle Radius  | Intersect Region Region
            | Outside Region | Union     Region Region

circle  r = Circle    r
outside r = Outside   r
r1 ∩ r2   = Intersect r1 r2
r1 ∪ r2   = Union     r1 r2

p `inRegion` (Circle  r)        = magnitude p ≤ r
p `inRegion` (Outside r)        = not (p `inRegion` r)
p `inRegion` (Intersect r1 r2) = p `inRegion` r1 && p `inRegion` r2
p `inRegion` (Union     r1 r2) = p `inRegion` r1 || p `inRegion` r2
```

The type Region is here represented as a data type with one constructor for each function that can be used to construct regions.

Writing the functions for constructing new regions becomes trivial. It is simply a matter of returning the right constructor. The hard work is instead done in the interpretation function inRegion which has to interpret the meaning of each constructor.

2.1. Brief Comparison

As the above example EDSL illustrates, a shallow embedding makes it easier to add new language constructs – as long as they can be represented in the semantic domain. For instance, it would be easy to add a function `rectangle` to our region example. On the other hand, since the semantic domain is fixed, adding a different form of interpretation, say, computing the area of a region, would not be possible without a complete reimplementation.

In the deep embedding, we can easily add new interpretations (just add a new function like `inRegion`), but it comes at the price of having a fixed set of language constructs. Adding a new construct to the deep implementation requires updating the `Region` type as well as all existing interpretation functions.

This comparison shows that shallow and deep embeddings are dual in the sense that the former is extensible with regards to adding language constructs while the latter is extensible with regards to adding interpretations. The holy grail of embedded language implementation is to be able to combine the advantages of shallow and deep in a single implementation. This is an instance of the *expression problem* [43].

One way to work around the limitation of deep embeddings not being extensible is to use "derived constructs". An example of a derived construct is `annulus`, which we defined in terms of `outside`, `circle` and `(∩)`. Derived constructs are shallow in the sense that they do not have a direct correspondence in the underlying embedding. Shallow derived constructs of a deep embedding are particularly interesting as they *inherit most advantages of both shallow and deep embeddings.* They can be added with the same ease as constructs in a fully shallow embedding. Yet, the interpretation functions only need to be aware of the deep constructs, which means that we retain the freedom of interpretation available in deep embeddings. There are, of course, limitations to how far these advantages can be stretched. We will return to this point in the concluding discussion (Section 9).

The use of shallow derived constructs is quite common in deeply embedded DSLs. However, the technique presented in this paper is novel and goes beyond "simple" derived constructs to extensions with new interface types leading to drastically different interfaces.

One existing solution to the problem of extending deep embeddings is Data Types à la Carte [41]. It makes it possible to define several independent data types and combine them to a single deep embedding in a modular way. However, regardless of this modularity, extending a deep embedding makes the language larger and increases the number of cases that need to be handled when traversing expressions. In contrast, our approach allows the definition of rich languages on top of simple deep embeddings. It is often possible to extend the language with no or minimal changes to the compiler when using our approach.

To be clear, our technique is not in competition with Data Types à la Carte. The two techniques complement each other and can be combined just fine [2, 31].

3. Overview of the Technique

We assume a setting where we want an EDSL that generates code. Code generation tends to require intensional analysis of the AST, which is not directly possible with a shallow implementation. Hence, we will start with a deep embedding as a basis. Our technique can be summarized in the following steps:

1. Implement a deeply embedded core language. The aim of the core language is *not* to act as a convenient user interface, but rather to support efficient generation of common code patterns in the target language. For this reason, the core language should be kept as simple as possible.
2. Implement user-friendly interfaces as shallow embeddings on top of the core language. Each interface is represented by a separate type and operations on this type.
3. Give each interface a precise meaning by giving a translation to and from a corresponding core language program. In other words, make the deep embedding the semantic domain of the shallow embedding. This is done by means of type class instantiation. If such a translation is not possible, or not efficient, extend the core language as necessary.

In the sections that follow we will demonstrate our technique by defining a deep embedding and showing a number of examples of shallow extensions. For the sake of concreteness we have made some superficial choices which are orthogonal to our technique. In particular, we use a typed representation of the deep embedding and employ higher order abstract syntax to deal with binding constructs. Neither of these choices matter for the applicability of our technique.

4. Demo: Deep Embedding with Shallow Extensions

To demonstrate our technique we will use a small embedded language called FunC as our running example.

4.1. Deep Embedding

The data type describing the FunC abstract syntax tree can be seen in Figure 1. [1] FunC is a low level, pure functional language which has a straightforward translation into C. It is meant for embedding low level programs and is inspired by the core language used in Feldspar [4]. We use a GADT to give precise types to the different constructors. We have also chosen Higher Order Abstract Syntax (HOAS) [33] for the `Lam` constructor.

The first two constructors, `:$` and `Lam` correspond to application and abstraction in the lambda calculus. Then there are a number of symbols for different language constructs: `Lit` introduces a literal; `If` introduces a function for testing booleans; `While` introduces a functional while loop (explained in Section 4.3);

[1] We use a serif font to refer to the language FunC, and sans serif to refer to the data type implementation `FunC`.

```
data FunC a where
  (:$) :: FunC (a → b) → FunC a → FunC b      -- Application
  Lam  :: (FunC a → FunC b) → FunC (a → b)    -- Abstraction

  -- Symbols
  Lit   :: Show a ⇒ a → FunC a
  If    :: FunC (Bool → a → a → a)
  While :: FunC ((s → Bool) → (s → s) → s → s)
  Pair  :: FunC (a → b → (a,b))
  Fst   :: FunC ((a,b) → a)
  Snd   :: FunC ((a,b) → b)
  Prim  :: String → a → FunC a

  -- Interpretation of variables
  Value    :: a → FunC a         -- Value of a variable
  Variable :: String → FunC a    -- Name of a variable
```

Figure 1: A deep embedding of FunC

`Pair`, `Fst` and `Snd` are for constructing and eliminating pairs; `Prim` introduces a primitive function.

The last two constructors, `Value` and `Variable`, are not part of the language. They are used internally for evaluation and printing respectively (see Sections 4.4 and 4.10). It would be possible to avoid these odd constructors by using a parameteric HOAS representation [10], but we have opted for a simpler representation in this paper.

Instead of letting the user write explicit applications, we can define smart constructors corresponding to the different symbols:

```
ifC :: FunC Bool → FunC a → FunC a → FunC a
ifC c t f = If :$ c :$ t :$ f

pair :: FunC a → FunC b → FunC (a,b)
pair a b = Pair :$ a :$ b
```

4.2. Primitive Functions and Literals

The `Prim` symbol introduces a primitive function from a name (used for printing and code generation) and a semantic function (used for evaluation). We can use `Prim` and `Lit` to instantiate the `Num` class for `FunC`:

```
instance (Num a, Show a) ⇒ Num (FunC a) where
  fromInteger = Lit . fromInteger
  a + b       = Prim "(+)" (+) :$ a :$ b
  a − b       = Prim "(−)" (−) :$ a :$ b
  a ∗ b       = Prim "(∗)" (∗) :$ a :$ b
  abs a       = Prim "abs" abs :$ a
  ...
```

Note that the arity of the semantic function passed to `Prim` determines the number of applications needed. With the above `Num` instance, we can write FunC expressions that look like ordinary Haskell; for example, `10 + 5 :: FunC Int`.

While numeric literals are conveniently introduced using the Num instance, boolean literals are written using the following definitions:

```
true, false :: FunC Bool
true  = Lit True
false = Lit False
```

We will also be using comparison and integral operators in FunC. For tiresome reasons it is not possible to overload the methods of the corresponding type classes Eq, Ord and Integral: for example, the == operator returns a Haskell Bool and there is no way we can change that to fit the types of FunC. Instead we will simply assume that the standard definitions of the comparison and integral operators are hidden and we will use definitions specific to FunC.

4.3. Higher-Order Functions

The While symbol has a *higher-order* type: (s → Bool) → (s → s) → s → s. Seen as an ordinary Haskell function, it is supposed to work as follows: the first argument is a function that determines whether or not to continue based on the current state (of type s); the second argument is the step function that computes the next state from the current state; the third argument is the initial state; the result is the final state. The reason for having a step function is that FunC is pure, so the body of the loop cannot perform side-effects.

A first attempt to make a smart constructor for While might lead to the following definition:

```
while :: FunC (s → Bool) → FunC (s → s) → FunC s → FunC s
while cont step init = While :$ cont :$ step :$ init
```

The problem with this function is that it expects FunC expressions of function types as argument. Such expressions can be created using Lam or some of the symbols of FunC. However, using symbols to construct the function expression is generally not a good idea, because when analyzing or compiling expressions we usually want the state of the while loop to be associated with a variable. So, since we always want to use Lam for these arguments, it is convenient to let the smart constructor insert Lam automatically for us:

```
while :: (FunC s → FunC Bool) → (FunC s → FunC s) → FunC s → FunC s
while cont step init = While :$ Lam cont :$ Lam step :$ init
```

Now the while loop starts to look like an ordinary higher-order Haskell function, and we can even write some examples with it. The following toy program computes the smallest multiple of 2 that is greater than 100:

```
ex₁ :: FunC Int
ex₁ = while (≤ 100) (*2) 1
```

Being based on the lambda calculus, FunC can represent arbitrary higher-order expressions. This is problematic if we want to generate efficient low-level code from FunC. In Section 7, we will discuss how to restrict the use of higher-order expressions, so that efficient code can be generated.

4.4. Evaluation

The exact semantics of the FunC language is given by the `eval` function which maps a FunC expression to the corresponding Haskell expression:

```
eval :: FunC a → a
eval (f :$ a)    = eval f $! eval a
eval (Lam f)     = eval ∘ f ∘ Value
eval (Lit l)     = l
eval If          = λc t f → if c then t else f
eval While       = λc b i → head $ dropWhile c $ iterate b i
eval Pair        = (,)
eval Fst         = fst
eval Snd         = snd
eval (Prim _ f)  = f
eval (Value a)   = a
```

Evaluation of the `Lam` constructor uses a standard technique for folding HOAS terms [18]. The argument `f` is of type `FunC a → FunC b` and we need to return something of type `a → b`. This is done by using `Value` to convert `a` to `FunC a`, apply the function `f`, and finally use `eval` to convert the resulting `b` to `FunC b`. Our only use of the `Value` constructor is in `eval`.

There is no case for `Variable` in `eval`. This is because `Variable` is not part of the FunC language, but only a technicality used for inspecting AST (see Section 4.10).

Note that application maps to *strict* application in Haskell, reflecting the fact that FunC is meant to be compiled to targets without support for lazy evaluation.

4.5. Extensible User Interfaces – the `Syntactic` Class

So far our presentation of FunC has been a purely deep embedding. Our goal is to be able to add shallow embeddings on top of the deep embedding and in order to make that possible we will make our language extensible using a type class. This type class will encompass all the types that can be compiled into the FunC language. We call the type class `Syntactic` (inspired by a less general class of the same name in Pan [16]).

```
class Syntactic a where
    type Internal a
    toFunC   :: a → FunC (Internal a)
    fromFunC :: FunC (Internal a) → a
```

When making an instance of the class `Syntactic` for a type `T` one must specify how `T` will represented internally, in the already existing deep embedding. This is what the associated type `Internal` is for. The two functions `toFunC` and `fromFunC` translate back and forth between an element of type `T` and its FunC term representation.

It is generally not the case that `toFunC` and `fromFunC` are each other's inverses; however, they must preserve the semantics of the expression:

Law 1. *For all types* t *in the* Syntactic *class and all expressions* a :: FunC (Internal t) *the following must hold:*

```
eval a == eval (toFunC (fromFunC a :: t))
```

The first instance of Syntactic is simply FunC itself, and the instance is completely straightforward.

```
instance Syntactic (FunC a) where
    type Internal (FunC a) = a
    toFunC    ast = ast
    fromFunC ast = ast
```

In Section 4.1, we defined smart constructors to make it easier to construct FunC expressions. Now that we have the Syntactic class we can give an even nicer extensible interface to the programmer. This interface will mirror the deep embedding and its constructors but will use the class Syntactic to overload the functions to make them compatible with any type that we choose to make an instance of Syntactic .

```
ifC :: Syntactic a ⇒ FunC Bool → a → a → a
ifC c t e = fromFunC (If :$ c :$ toFunC t :$ toFunC e)

c ? (t,e) = ifC c t e

while :: Syntactic s ⇒ (s → FunC Bool) → (s → s) → s → s
while c b i = fromFunC (While :$ Lam (c ∘ fromFunC)
                              :$ Lam (toFunC ∘ b ∘ fromFunC)
                              :$ toFunC i)
```

When specifying the types in our new interface we note that base types are not overloaded, they are still on the form FunC Bool. The big difference is when we have polymorphic functions. The function ifC works for any a as long as it is an instance of Syntactic . The advantage of the type Syntactic a ⇒ FunC Bool → a → a → a over FunC Bool → FunC a → FunC a → FunC a is two-fold: First, it is closer to the type that an ordinary Haskell function would have and so it gives the function a more native feel, like it is less of a library and more of a language. Secondly, it makes the language extensible. These functions can now be used with any type that is an instance of Syntactic . We are no longer tied to working solely on the abstract syntax tree FunC.

4.6. Embedding Pairs

We have not yet given an interface for pairs. The reason for this is that they provide an excellent opportunity to demonstrate our technique. We simply instantiate the Syntactic class for Haskell pairs:

```
instance (Syntactic a, Syntactic b) ⇒ Syntactic (a,b) where
    type Internal (a,b) = (Internal a, Internal b)
    toFunC (a,b)        = Pair :$ toFunC a :$ toFunC b
    fromFunC p          = (fromFunC (Fst :$ p), fromFunC (Snd :$ toFunC p))
```

In this instance, toFunC constructs an embedded pair from a Haskell pair, and fromFunC eliminates an embedded pair by selecting the first and second component and returning these as a Haskell pair.[2]

The usefulness of pairs comes in when we need an existing function to operate on a compound value rather than a single value. For example, the state of the while loop is a single value. If we want the state to consist of, say, two integers, we use a pair. Since functions such as ifC and while are overloaded using Syntactic, there is no need for the user to construct compound values explicitly; this is done automatically by the overloaded interface.

As an example of this, here is a for loop defined using the while construct with a compound state:

```
forLoop :: Syntactic s ⇒ FunC Int → s → (FunC Int → s → s) → s
forLoop len init step = snd $ while (λ(i,s) → i<len)
                                    (λ(i,s) → (i+1, step i s))
                                    (0,init)
```

The first argument to forLoop is the number of iterations; the second argument is the initial state; the third argument is the step function which, given the current loop index and current state, computes the next state. We define forLoop using a while loop whose state is a pair of an integer and a smaller state.

Note that the above definition only uses ordinary Haskell pairs: The continue condition and step function of the while loop pattern match on the state using ordinary pair syntax, and the initial state is constructed as a standard Haskell pair.

Another example of using the while loop and pairs together is the following implementation of the greatest common divisor algorithm:

```
gcd :: FunC Int → FunC Int → FunC Int
gcd a b = fst $ while (λ(a,b) → a /= b)
                      (λ(a,b) → a > b ? ( (a−b,b) , (a,b−a) ))
                      (a,b)
```

The state of the while loop is two integers where the smaller integer is subtracted from the larger until they are equal.

4.7. Embedding Functions

Writing smart constructors such as ifC and while is quite a boring task, and it would be nice to be able to automate it. Consider the While symbol and the corresponding non-overloaded smart constructor:

```
While :: FunC ((s → Bool) → (s → s) → s → s)

while :: (FunC s → FunC Bool) → (FunC s → FunC s) → FunC s → FunC s
while cont step init = While :$ Lam cont :$ Lam step :$ init
```

[2] Note that the argument p is duplicated in the definition of fromFunC. If both components are later used in the program, this means that the syntax tree will contain two copies of p. For this reason, having tuples in the language usually requires some way of recovering sharing (see Section 7.2). This issue is, however, orthogonal to the ideas presented in this paper.

We can say that the purpose of the smart constructor is to move all function arrows from inside the parameter of FunC to the outside. This is done because it is more convenient for the user to deal with ordinary Haskell functions than using explicit application and abstraction in FunC.

However, from a semantic point of view, the types (FunC a → FunC b) and FunC (a → b) are equivalent: we can use :$ and Lam to convert between the two without changing the meaning of the program. This correspondence can be captured by declaring a Syntactic instance for functions:

```
instance (Syntactic a, Syntactic b) ⇒ Syntactic (a → b) where
    type Internal (a → b) = Internal a → Internal b
    toFunC f   = Lam (toFunC . f . fromFunC)
    fromFunC f = λa → fromFunC (f :$ toFunC a)
```

Note that a and b can be any types in the Syntactic class. Somewhat magically, the above instance lets us derive the whole implementation of the smart constructor automatically:

```
while :: (FunC s → FunC Bool) → (FunC s → FunC s) → FunC s → FunC s
while = fromFunC While
```

Not only that; since the instance works for any a and b in Syntactic , we can also derive the overloaded version in the same way:

```
while :: Syntactic s ⇒ (s → FunC Bool) → (s → s) → s → s
while = fromFunC While
```

To see how this works, we show a stepwise expansion of the definition:

```
while :: Syntactic s ⇒ (s → FunC Bool) → (s → s) → s → s
while = fromFunC While

    -- Expand fromFunC
    = λc →
        fromFunC (While :$ toFunC c)

    -- Expand fromFunC
    = λc → λs →
        fromFunC ((While :$ toFunC c) :$ toFunC s)

    -- Expand fromFunC
    = λc → λs → λi →
        fromFunC (((While :$ toFunC c) :$ toFunC s) :$ toFunC i)

    -- Expand toFunC for functions
    = λc → λs → λi →
        fromFunC (While :$ Lam (toFunC ∘ c ∘ fromFunC)
                        :$ Lam (toFunC ∘ s ∘ fromFunC)
                        :$ toFunC i)

    -- toFunC = id for type FunC Bool
    = λc → λs → λi →
        fromFunC (While :$ Lam (c ∘ fromFunC)
                        :$ Lam (toFunC ∘ s ∘ fromFunC)
                        :$ toFunC i)
```

We recognize the last step as the definition we gave for `while` in Section 4.5.

From now on, we will only use smart constructors derived from `fromFunC`, and there will be no need for explicit uses of `:$` and `Lam`.

4.8. Embedding Option

If we want to extend our language with optional values, one may be tempted to make a `Syntactic` instance for `Maybe`. Unfortunately, there is no way to make this work, because `fromFunC` would have to decide whether to return `Just` or `Nothing` when the Haskell program is evaluated, which is one stage earlier than when the FunC program is evaluated. Instead, we can use the following implementation:

```
data Option a = Option { isSome :: FunC Bool, fromSome :: a }

instance Syntactic a ⇒ Syntactic (Option a) where
    type Internal (Option a) = (Bool, Internal a)
    fromFunC m               = Option (fromFunC Fst m) (fromFunC Snd m)
    toFunC (Option b a)      = fromFunC Pair b a
```

We have borrowed the name `Option` from ML to avoid clashing with the name of the Haskell type. The type `Option` is represented as a boolean and a value.[3] The boolean indicates whether the value is valid or whether it should simply be ignored, effectively interpreting it as not being there. The `Syntactic` instance converts to and from the representation in `FunC` which is a pair of a boolean and the value.

The definition of `Option` may seem straightforward, but when we try to create an empty `Option` value, we run into problems. We need some value to put into the second component of the pair. It is not important what value we put there, since it is not going to be looked at anyway, but the problem is that we need a polymorphic value, because we want to be able to create empty `Option` values of arbitrary types. One alternative would be to extend `FunC` with a bottom value, analogous to Haskell's `undefined`, but that seems quite unsatisfactory. A better alternative is to introduce a type class that lets us construct arbitrary "example" values of different types:[4]

```
class Inhabited a where
    example :: FunC a

instance Inhabited Bool where example = true
instance Inhabited Int  where example = 0
...
```

The `example` method just has to produce an example value of each type. What specific value it produces is irrelevant.

Armed with the `Inhabited` class, we can now provide functions for constructing and eliminating optional values:

[3]Larger unions can be encoded using an integer instead of a boolean.
[4]Thanks to Phil Wadler for the idea to use a type class rather than a bottom value.

```
some :: a → Option a
some a = Option true a

none :: (Syntactic a, Inhabited (Internal a)) ⇒ Option a
none = Option false (fromFunC example)

option :: (Syntactic a, Syntactic b) ⇒ b → (a → b) → Option a → b
option noneCase someCase opt = ifC (isSome opt)
                                   (someCase (fromSome opt))
                                   noneCase
```

The some function creates an optional value which actually contains a value whereas none defines an empty value using the newly introduced example method. The function option acts as a case on optional values, allowing the programmer to test an Option value to see whether it contains something or not.

The functions above provide a nice programmer interface but the real power of the shallow embedding of the Option type comes from the fact that we can make it an instance of standard Haskell classes. In particular we can make it an instance of Functor and Monad.

```
instance Functor Option where
    fmap f (Option b a) = Option b (f a)

instance Monad Option where
    return a  = some a
    opt >>= k = b { isSome = isSome opt ? (isSome b, false) }
      where b = k (fromSome opt)
```

Being able to reuse standard Haskell functions is a great advantage as it helps to decrease the cognitive load of the programmer when learning our new language. We can map any Haskell function on the element of an optional value because we chose to let the element of the Option type to be completely polymorphic, which is why these instances type check.

The advantage of reusing Haskell's standard classes is particularly powerful in the case of the Monad class because it has syntactic support in Haskell which means that it can be reused for our embedded language. For example, suppose that we have a function divF :: FunC Float → FunC Float → Option (FunC Float) which returns nothing in the case the divisor is zero. Then we can write a function for computing the resistance of two parallel resistors as follows:

```
resistance :: FunC Float → FunC Float → Option (FunC Float)
resistance r1 r2 = do rp1 ← divF 1 r1
                      rp2 ← divF 1 r2
                      divF 1 (rp1 + rp2)
```

4.9. Embedding Vector

Our language FunC is intended to target low level programming. In this domain most programs deal with sequences of data, typically in the form of arrays. In this section we will see how we can extend FunC to provide a nice interface to array programming.

The first thing to note is that FunC doesn't have any support for arrays at the moment. We will therefore have to extend FunC to accommodate this. The addition we have chosen is one constructor which computes an array plus two constructors for accessing the length and indexing into the array respectively:

```
Arr    :: FunC (Int → (Int → a) → Array Int a)
ArrLen :: FunC (Array Int a → Int)
ArrIx  :: FunC (Array Int a → Int → a)
```

The first argument of the Arr constructor computes the length of the array. The second argument is a function which given an index computes the element at that index. By repeatedly calling the function for each index we can construct the whole array this way. The meaning of ArrLen and ArrIx should require little explanation. The exact semantics of these constructors is given by the corresponding clauses in the eval function.

```
eval Arr    = λl ixf → let lm1 = l − 1
                        in  listArray (0,lm1) [ixf i | i  ← [0..lm1]]
eval ArrLen = λa → (1 +) $ uncurry (flip (−)) $ bounds a
eval ArrIx  = (!)
```

We will use two convenience functions for dealing with length and indexing: len which computes the length of the array and the infix operator (<!>) which is used to index into the array. As usual we have overloaded (<!>) so that it can be used with any type in the Syntactic class.

```
len :: FunC (Array Int a) → FunC Int
len = fromFunC ArrLen

(<!>) :: Syntactic a ⇒ FunC (Array Int (Internal a)) → FunC Int → a
(<!>) = fromFunC ArrIx
```

Having extended our deep embedding to support arrays we are now ready to provide the shallow embedding. In order to avoid confusion between the two embeddings we will refer to the shallow embedding as vector instead of array.

```
data Vector a where
    Indexed :: FunC Int → (FunC Int → a) → Vector a

instance Syntactic a ⇒ Syntactic (Vector a) where
    type Internal (Vector a) = Array Int (Internal a)
    toFunC (Indexed l ixf)   = fromFunC Arr l ixf
    fromFunC arr             = Indexed (len arr) (λix → arr <!> ix)
```

The type Vector forms the shallow embedding and its constructor Indexed is strikingly similar to the Arr construct. The only difference is that Indexed is completely polymorphic in the element type. One of the advantages of a polymorphic element type is that we can have any type which is an instance of Syntactic in vectors, not only values which are deeply embedded. Indeed we can even have vectors of vectors which can be used as a simple (although not very efficient) representation of matrices.

The Syntactic instance converts vectors into arrays and back. It is mostly straightforward except that elements of vectors need not be deeply embedded so they must in turn be converted using toFunC.

```
zipWithVec :: (Syntactic a, Syntactic b) ⇒
              (a → b → c) → Vector a → Vector b → Vector c
zipWithVec f (Indexed l1 ixf1) (Indexed l2 ixf2)
  = Indexed (min l1 l2) (λix → f (ixf1 ix) (ixf2 ix))

sumVec :: (Syntactic a, Num a) ⇒ Vector a → a
sumVec (Indexed l ixf) = forLoop l 0 (λix s → s + ixf ix)

instance Functor Vector where
    fmap f (Indexed l ixf) = Indexed l (f ∘ ixf)
```

The above code listing shows some examples of primitive functions for vectors. The call `zipWith f v1 v2` combines the two vectors `v1` and `v2` pointwise using the function `f`. The `sumVec` function computes the sum of all the elements of a vector using the for loop defined in Section 4.6. Finally, just as with the `Option` type in Section 4.8 we can define an instance of the class `Functor`.

Many more functions can be defined for our `Vector` type. In particular, any kind of function where each vector element can be computed independently will work particularly well with the representation we have chosen. However, functions that require sharing of previously computed results (e.g. Haskell's `unfoldr`) will yield poor code.

```
scalarProd :: (Syntactic a, Num a) ⇒ Vector a → Vector a → a
scalarProd a b = sumVec (zipWithVec (∗) a b)
```

An example of using the functions presented above we define the function `scalarProd` which computes the scalar product of two vectors. It works by first multiplying the two vectors pointwise using `zipWithVec`. The resulting vector is then summed to yield the final answer.

4.10. Rendering the AST

Figure 2 shows the conversion from `FunC` to a tree (from the standard Haskell module `Data.Tree`). The helper function `toTreeArgs` uses the `State` monad to be able to generate fresh variable names, and it takes a list of children as arguments. The purpose of the list is to accumulate applications so that all arguments of a function expression become children to the same node. The only case where `:$` shows up in the tree is when we have a `Lam` that is immediately applied. However, such expressions do not appear in the examples given in this paper.

Just like in `eval`, we need to pass an expression to the function in a `Lam` node in order to be able to examine the body. However, in this case we pass a `Variable` with a freshly generated name instead of a `Value`.

Figure 3 shows the tree produced from the expression `toFunC scalarProd`. It has been rendered using the `tree-view` package[5]. It is interesting to see that the generated tree is a simple expression with a single loop. In the next section we will see how this expression is obtained from the definition of `scalarProd`.

[5] http://hackage.haskell.org/package/tree-view

```
toTreeArgs :: FunC a → [Tree String] → State Int (Tree String)
toTreeArgs (f :$ a) as = do
    at ← toTreeArgs a []
    toTreeArgs f (at:as)
toTreeArgs (Lam f) as = do
    v ← get; put (v+1)
    let var = Variable ('v' : show v)
    body ← toTreeArgs (f var) []
    return $ case as of
        [] → Node ("Lam v" ++ show v) [body]
        _  → Node (":$") (body:as)
toTreeArgs (Variable v) as = return $ Node v as
toTreeArgs sym as = return $ Node (showSym sym) as
  where
    showSym :: FunC a → String
    showSym (Lit a) = show a
    showSym If      = "If"
    showSym While   = "While"
    ...

toTree :: FunC a → Tree String
toTree a = evalState (toTreeArgs a []) 0
```

Figure 2: Conversion from FunC to a tree.

```
Lam v0
 └ Lam v1
    └ Snd
       └ While
          ├ Lam v3
          │  └ (<)
          │     ├ Fst
          │     │  └ v3
          │     └ min
          │        ├ ArrLen
          │        │  └ v0
          │        └ ArrLen
          │           └ v1
          ├ Lam v2
          │  └ Pair
          │     ├ (+)
          │     │  ├ Fst
          │     │  │  └ v2
          │     │  └ 1
          │     └ (+)
          │        ├ Snd
          │        │  └ v2
          │        └ (*)
          │           ├ ArrIx
          │           │  ├ v0
          │           │  └ Fst
          │           │     └ v2
          │           └ ArrIx
          │              ├ v1
          │              └ Fst
          │                 └ v2
          └ Pair
             ├ 0
             └ 0
```

Figure 3: Abstract syntax tree of scalarProd .

We note in passing that most other examples in this paper result in ASTs that are too large to present in the paper. The reason is lack of sharing in the generated expressions and lack of syntactic simplification. This problem is solved by a combination of common sub-expression elimination and a number of mostly trivial simplification rules. We return to this point in Section 7.

5. Fusion

Choosing to implement vectors as a shallow embedding has a very powerful consequence: it provides a very lightweight implementation of fusion [23]. We will demonstrate this using the function `scalarProd` defined in the previous section. Upon a first glance it may seem as if this function computes an intermediate vector, the vector `zipWithVec (∗) a b` which is then consumed by the `sumVec`. This intermediate vector would be quite bad for performance and space reasons if we ever wanted to use the `scalarProd` function as defined.

Luckily the intermediate vector is never computed as we see in Figure 3. To see why this is the case consider what happens when we generate code for the expression `scalarProd v1 v2`, where `v1` and `v2` are defined as `Indexed l1 ixf1` and `Indexed l2 ixf2` respectively. Before generating an abstract syntax tree the Haskell evaluation mechanism will reduce the expression as follows:

```
  scalarProd v1 v2
= sumVec (zipWithVec (∗) v1 v2)
= sumVec (zipWithVec (∗) (Indexed l1 ixf1) (Indexed l2 ixf2))
= sumVec (Indexed (min l1 l2) (λix → ixf1 ix ∗ ixf2 ix)
= forLoop (min l1 l2) 0 (λix s → s + ixf1 ix ∗ ixf2 ix)
```

The intermediate vector has disappeared and the only thing left is a for loop which computes the scalar product directly from the two argument vectors.

In the above example, fusion happened because although `zipWithVec` constructs a vector, it does not generate an array in the deep embedding. In fact, all standard vector operations (`fmap`, `take`, `reverse`, etc.) can be defined in a similar manner, without using internal storage. Whenever two such functions are composed, the intermediate vector is guaranteed to be eliminated. This guarantee by far exceeds guarantees given by conventional optimizing compilers.

So far, we have only seen one example of a vector producing function that uses internal storage: `fromFunC`. Thus intermediate vectors produced by `fromFunC` (for example as the result of `ifC` or `while`) will generally not be eliminated.

There are some situations when fusion is not beneficial, for instance in a function which access an element of a vector more than once. This will cause the elements to be recomputed. It is therefore important that the programmer has some way of backing out of using fusion and store the vector to memory. For this purpose we can provide the following function:

```
memorize  :: Syntactic a ⇒ Vector a → Vector a
memorize (Indexed l ixf) = Indexed l (λn → fromFunC Arr l ixf <!> n)
```

The function `memorize` can be inserted between two functions to make sure that the intermediate vector is stored to memory. For example, if we wish to store the intermediate vector in our `scalarProd` function we can define it as follows:

```
scalarProd :: (Syntactic a, Num a) ⇒ Vector a → Vector a → a
scalarProd a b = sumVec (memorize (zipWithVec (∗) a b))
```

Strong guarantees for fusion in combination with the function memorize gives the programmer a very simple interface which still provides powerful optimizations and fine grained control over memory usage.

5.1. Other types which benefit from fusion

The Vector type is very useful for writing array computations in a compositional style. Unfortunately, not all computations are efficiently implementable with the Vector type. One example is to compute the scan of a vector. The following is an inefficient implementation:

```
scanVec :: Syntactic a ⇒ (a → b → a) → a → Vector b → Vector a
scanVec f z (Indexed l ixf) = Indexed (l+1) ixf'
  where ixf' i = forLoop (i−1) z $ λj s →
                   f s (ixf j)
```

This implementation will perform a lot of duplicate computations. An efficient implementation would avoid recomputations by linearly iterating through the array and use an accumulating parameter to store the intermediate results. The Vector type does not permit such an implementation because each element is computed and accessed independently; there is no way to impose a particular order in which the elements are traversed. We must turn to a different representation in order to implement scan efficiently.

The type of sequential vectors imposes a linear, left-to-right traversal order of the elements. We can construct a shallow embedding as follows:

```
data Seq a = ∀ s . Syntactic s ⇒ Seq s (s → (a,s)) (FunC Int)
```

The type Seq contains a hidden piece of state; the existentially bound type variable s. The first argument to the constructor is an initial state. The state is consumed by the stepper function, the second argument to Seq, which produces a new element in the vector and a new state. The last argument is the length of the vector.

The type Seq permits an efficient implementation of scan:

```
scanSeq :: Syntactic a ⇒ (a → b → a) → a → Seq b → Seq a
scanSeq f z (Seq init step l) = Seq init' step' (l+1)
  where init' = (z,init)
        step' (a,s) = let (b,s') = step s
                      in (a,(f a b,s'))
```

We refrain from going into the full details about how to implement a full library for the Seq type. In summary, we make the following observations:

- To construct a Syntactic instance for Seq requires a new constructor in the deep embedding:

  ```
  Sequential :: FunC (s → (s → (a,s)) → Int → Array Int a)
  ```

- The type Seq provides a complementary set of operations compared to Vector. For example, scanning is provided for Seq while random access indexing is not.
- Operations on Seq enjoys the same kind of fusion guarantees as the operations on Vector.
- It is possible to convert from Vector to Seq while still preserving guaranteed fusion, using the following function:

  ```
  vecToSeq :: Vector a → Seq a
  vecToSeq (Indexed l ixf) = Seq 0 step l
    where step i = (ixf i, i+1)
  ```

  Converting from Seq to Vector requires storing to memory (i.e. introducing a Sequential node in the deep embedding).

There is a third kind of vector which provides yet another, complementary set of operations; push vectors [13].

```
data Push a = Push ((FunC Int → a → M ()) → M ()) (FunC Int)
```

The the first argument to the Push constructor can be thought of as a program which writes an array to memory. Writing to memory is an inherently imperative operation and fits badly with the functional nature of the language we've presented so far. The solution is to use monads, and the type M is a monad for writing to memory. We return to explaining push vectors once we've covered how to embed monads.

6. Monads

It is sometimes useful to include monads in a domain specific language, for the same reason they are useful in Haskell: to isolate effectful computations from pure computations using the type system. In Section 4.8 we saw that the Option could be made an instance of the typeclass Monad. This monad was constructed using building blocks already available in the deep representation of the language. In this section we will see how to build generic support for monads where the monadic operations are represented as new explicit constructors.

The first step in adding support for monads in our language is to enrich our deep embedding with the constructors Return and Bind, corresponding to the two operations in the standard Monad class.

```
Return :: Monad m ⇒ FunC (a → m a)
Bind   :: Monad m ⇒ FunC (m a → (a → m b) → m b)
```

Although the types for these constructors are similar to the monad operations, they cannot be used directly as implementations in an instance for the Monad class, since they are merely symbols. We show how to get around that below.

Defining the semantics for Return and Bind is straightforward:

```
eval Return = return
eval Bind   = (>>=)
```

The user interface for monads consists of the type Mon, which provides a shallow embedding which lifts an arbitrary monad in Haskell into the embedded language, and a Monad instance for Mon.

```
data Mon m a = M { unM :: ∀ b . ((a → FunC (m b)) → FunC (m b)) }

instance Monad m ⇒ Monad (Mon m) where
    return a   = M $ λk → k a
   M m >>= f = M $ λk → m (λa → unM (f a) k)
```

Readers with prior knowledge about monads will recognize that Mon is similar to the continuation passing monad. The difference is that the answer type has been specialized to generate syntax trees.

Using the continuation monad on top of the deep embedding has a fortunate side effect: it normalizes monadic expression. Certain monads suffer an asymptotic slowdown if compositions of the >>= operator are associated to the left: writing e.g. m >>=(λa → f a >>= n) is more costly to evaluate than writing (m >>=f) >>= n. Luckily, the continuation monad transformer will associate >>= in the underlying monad to the right, ensuring efficient execution [42].

```
instance (Monad m, Syntactic a) ⇒ Syntactic (Mon m a) where
    type Internal (Mon m a) = m (Internal a)
    toFunC (M m) = m (fromFunC Return)
    fromFunC m   = M $ λk → fromFunC Bind m k
```

It is possible to provide a Syntactic instance for Mon as shown above. This makes monadic computations first class citizens in the DSL – a very powerful addition. Though, depending on what kind of target code we want to generate, we might not want to allow passing around monadic computations as that would entail creating closures. In Section 7.4, we elaborate on how to constrain the types of the symbols in FunC to rule out such problematic code.

The type Mon provides a generic building block for constructing particular monads in our DSL. As a concrete example, we will implement a monad which adds mutable arrays to our language. Haskell's standard library for mutable arrays will stand as a model for our extension.

```
  NewArray_   :: FunC (Int → IO (IOArray Int a))
  GetArray    :: FunC (IOArray Int a → Int → IO a)
  PutArray    :: FunC (IOArray Int a → Int → a → IO ())
  LengthArray :: FunC (IOArray Int a → IO Int)
  FreezeArray :: FunC (IOArray Int a → IO (Array Int a))
  ThawArray   :: FunC (Array Int a → IO (IOArray Int a))

eval NewArray_   = λi → newArray_ (0,i−1)
eval GetArray    = readArray
eval PutArray    = writeArray
eval LengthArray = getNumElements
eval FreezeArray = freeze
eval ThawArray   = thaw
```

The construct NewArray allocates a new, uninitialized array. The length is determined by the first argument. GetArray and PutArray reads from and writes to an array (the semantics for using an out of bounds index is undefined). The length of an array is given by LengthArray. FreezeArray and ThawArray provides a way to convert back and forth between mutable and immutable arrays.

Compared to previous language features, this list of constructors is a big addition to our deep embedding. This is not surprising, given that references and arrays are primitive types that require special primitive operations. In the following sub-sections, we will see how these primitive operations enable the definition of shallow high-level data structures. It turns out that these extra primitive operations give us quite a lot in return!

The code below provides the user interface for the array constructs. We define a new type M which captures computations with mutable arrays. The type Marr is a convenient alias when using arrays.

```
type M a = Mon IO a
type MArr a = FunC (IOArray Int a)

newArray    :: FunC Int → M (MArr a)
getArray    :: MArr a → FunC Int → M (FunC a)
putArray    :: MArr a → FunC Int → FunC a → M ()
lengthArray :: MArr a → M (FunC Int)
freezeArray :: MArr a → M (FunC (Array Int a))
thawArray   :: FunC (Array Int a) → M (MArr a)

newArray    = fromFunC NewArray_
getArray    = fromFunC GetArray
putArray    = fromFunC PutArray
lengthArray = fromFunC LengthArray
freezeArray = fromFunC FreezeArray
thawArray   = fromFunC ThawArray
```

The result type M () of putArray requires the existence of a Syntactic instance for (), something which is trivial to define.

When working with arrays it is crucial to be able to perform loops over them. Since we have a Monad instance for our embedded monad, it is natural to think that we can use the standard control operators for loops provided by the standard library in Haskell. But these control operators would be evaluated at compile time and there would be no loops left in the generated code. For that reason, the loops could not depend on any runtime data, which would be overly restrictive. So we are left with using looping constructs defined in our deep representation. The existing while loop is not directly suited to represent monadic loops, so instead we add two new constructs.

```
  ForM   :: Monad m ⇒ FunC (Int → (Int → m ()) → m ())
  WhileM :: Monad m ⇒ FunC (m Bool → m () → m ())
```

The user interface for monadic loops is as follows:

```
forM :: Monad m ⇒ FunC Int → FunC Int → (FunC Int → Mon m ()) → Mon m ()
forM start stop body = fromFunC ForM (stop − start) (body ∘ (+start))
```

```
whileM :: Monad m ⇒ Mon m (FunC Bool) → Mon m () → Mon m ()
whileM = fromFunC WhileM
```

An example of how to use the mutable array interface is the following implementation of in-place insertion sort (we assume the existence of mutable references implemented similarly to mutable arrays).

```
insertionSort :: Ord a ⇒ FunC Int → MArr a → M ()
insertionSort l arr = do
  forM 1 l $ λi → do
    value ← getArray arr i
    j ← newRef (i−1)
    let cond = do jv ← readRef j
                  aj ← getArray arr jv
                  return (jv ≥ 0 && aj > value)
    whileM cond $ do
      jv ← readRef j
      a ← getArray arr jv
      putArray arr (jv+1) a
      writeRef j (jv−1)
    jv ← readRef j
    putArray arr (jv+1) value
```

One might be worried about the fact that the FunC type keeps growing whenever we add new primitives. This problem can be alleviated by using Data Type à la Carte [41] to divide the primitive operations into several independent types. In fact, our earlier work on embedding monads was based on Data Type à la Carte [31].

6.1. Push vectors

As mentioned in the previous section, there is a form of vector which uses monads to write to memory: push vectors [13].

```
data Push a = Push ((FunC Int → a → M ()) → M ()) (FunC Int)
```

The implementation of Push contains a monadic program which writes the array to memory. It is parameterized on a function of type FunC Int → a → M (). This is the function that performs the actual writing, given an index and an array element. The monadic program is responsible for iterating over all the index-value-pairs of the array and call the writing function on each one of them. As a first example of how to construct push vectors we give a function which enumerates integers:

```
enum :: FunC Int → FunC Int → Push (FunC Int)
enum start stop = Push f (stop − start)
  where f w = forM start stop $ λi →
                w i i
```

Push solves two problems which neither Vector nor Seq can handle: efficient concatenation and computing several elements at once. Here's how we can implement concatenation of two push vectors:

```
(++) :: Push a → Push a → Push a
Push f1 l1 ++ Push f2 l2 = Push f (l1 + l2)
  where f w = do f1 w
                 f2 (λi a → w (i+l1) a)
```

Concatenation is given two push vectors, containing two monadic programs for writing them to memory, `f1` and `f2`. When constructing the program for the resulting push vector, `f`, we first run `f1` to write that vector to memory. Then `f2` is allowed to run, but the index where it writes its elements is adjusted so that they are written after the first vector.

An observation is that the two programs `f1` and `f2` write to completely separate memory locations. That means that they could be executed in parallel for increased speed. Push vectors support several operations which can be parallelized in this way [13, 1].

As an example of computing several elements at once, we use the `dup` function below. It performs the same operation as concatenating a vector with itself, but makes sure not to duplicate the computation of the elements, which can otherwise happen.

```
dup :: Push a → Push a
dup (Push f l) = Push g (2*l)
  where g w = f (λi a → w i a >> w (i + l) a)
```

Although `dup` is only meant as a pedagogical example, similar patterns happen in real life applications. For example, when scaling up an image to cover more pixels, several pixels are produced at every iteration in the computation.

```
store :: Push (FunC a) → M (FunC (Array Int a))
store (Push f l) = do
  arr ← newArray l
  f (putArray arr)
  freezeArray arr
```

Storing a push vector means allocating a mutable array in memory, then make the push vector program write to that array by feeding it a function which performs the write. Finally, the mutable array is frozen and the result is an immutable array. The whole computation lives in the `M` monad, since there is no way to escape it. It is possible to provide an embedding like the `ST` monad, which can encapsulate imperative algorithms in a purely functional interface [29]. Such an embedding would enable a purely functional interface to `store`, and would enable a `Syntactic` instance.

In summary, push vectors provide yet another useful abstraction for array processing. Their implementation is particularly convenient thanks to the embedding of monads, it's almost as writing normal Haskell.

6.2. Mutable data structures

The data structures we've seen so far, such as pairs, `Option`, `Vector`, `Seq`, and even `Push`, have had purely functional interfaces (with the exception of the `store` function). The introduction of monads in the language opens up for the possibility of creating mutable data structures. When writing streaming applications it is common to use a mutable cyclic buffer. We can implement such a buffer in our language as a shallow embedding:

```
data Buffer a = Buffer
    { indexBuf :: FunC Int → M a
    , putBuf    :: a → M ()
    }
```

The implementation of this buffer is reminicent of how classes are implemented in object oriented languages. The data type contains the public methods exposed to the programmer using the buffer. The hidden members and data are stored in the closure created when the buffer is constructed. The following function constructs a buffer:

```
initBuffer :: ∀ a . Syntactic a ⇒ Vector a → M (Buffer a)
initBuffer vec = do
    buf ← thawArray (toFunC vec)
    l   ← lengthArray buf
    ir  ← newRef 0
    let get j = do
          i ← readRef ir
          fmap fromFunC $ getArray buf $ calcIndex l i j
        put a = do
          i ← readRef ir
          writeRef ir ((i+1) `mod` l)
          putArray buf i $ toFunC a
    return (Buffer get put)
  where
    calcIndex l i j = (l+i−j−1) `mod` l
```

Constructing a Buffer begins with allocating a mutable array which will contain the payload, and a reference for keeping track of where the first element in the buffer is located in the array. The two functions get and put read and write to the appropriate locations in the mutable array using the reference.

As an example of how to use the circular buffer, the following program computes the nth fibonacci number.

```
fib :: FunC Int → M (FunC Int)
fib n = do
  let twoOnes = Indexed 2 $ λ_ → 1      −− Initial buffer [1,1]
  buf ← initBuffer twoOnes
  forM 1 n $ λ_ → do
    a ← indexBuf buf 0
    b ← indexBuf buf 1
    putBuf buf (a+b)
  indexBuf buf 0
```

7. Scaling up to a full implementation

In this paper we have used a simple implementation to be able to focus on the basic ideas. In order to scale up the method to a full-blown EDSL implementation such as Feldspar [4], there are a few things that need to be taken care of:

- The front end needs to be extended with more primitive functions. For example, in Feldspar we have reimplemented many of Haskell's Prelude functions, including most methods of classes such as `Eq`, `Ord`, `Integral`, `Floating`, etc.

- In order to avoid duplication of code and run-time computation, there has to be a way to discover and represent shared sub-expressions.

- Despite high-level optimizations in the shallow embedding (such as fusion; see Section 5), there are often opportunities to simplify the generated ASTs in order to generate more efficient code.

- We need a translator from expressions to C code (or similar). This requires making some changes to FunC to rule out higher-order terms that are not easily compiled to a low-level target.

The following sub-sections discuss the above points in a bit more detail.

7.1. AST representation

The three latter points in the above list involve traversing and transforming `FunC` expressions in various ways. This turns out to be very inconvenient when using higher-order abstract syntax (HOAS), as in this paper. Instead a first-order representation is generally preferred when the AST needs to be examined. At the same time, HOAS comes with some definite advantages:

- It makes it easy to define higher-order front end functions (such as the `while` loop in this paper).

- It makes evaluation both easy to define and very efficient due to the fact that substitution is performed directly by the function embedded in the AST.

One way to get the best of both worlds is to have two versions of the AST: a higher-order one and a first-order one with a function converting from the former to the latter. The higher-order one is used in the front end and possibly for evaluation, while the first-order one is used in the rest of the implementation. The disadvantage of this approach is that it requires two separate but very similar data types as representations of the same AST. Conversion from a higher-order to a first-order AST can be done using the same method as the rendering in Section 4.10.

Two EDSLs that use a combination of higher-order and first-order ASTs are Feldspar (until version 0.7) [4] and Accelerate [30].

In order to avoid having two separate representations of the same AST, it is possible to make a higher-order front end directly for a first-order AST by using a technique based on circular programming [3]. We plan to use this technique to get rid of the HOAS representation in future versions of Feldspar.

7.2. Handling Sharing

It is easy to write EDSL programs that result in duplicated sub-expressions. For example, the expression `let a = bigExpr in a+a` results in an AST that contains two copies of `bigExpr`. This is because Haskell bindings are inlined as part of Haskell's evaluation when generating an AST. This loss of sharing is problematic for two reasons: (1) it makes the AST larger which can slow down the compiler and lead to larger generated code, and (2) it leads to duplicated computations in the generated code which can increase its run time.

The problem with large ASTs is more severe than it may seem at first. An expression with nested duplications – for example

```
let a = ... in let b = x + x in let c = b + b in ...
```

– generates an AST which is exponentially larger than the corresponding Haskell expression.

Such expressions do actually occur in practice. The following innocent-looking function from Feldspar's source uses bit manipulation to compute the number of leading zeros in a machine word:

```
nlz x = bitCount $ complement
      $ foldl go x $ takeWhile (P.< bitSize' x)
      $ P.map (2 P.^) [(0::Integer)..]
  where
    go b s = b .|. (b .>>. value s)
```

Here, `foldl` is the normal left fold for Haskell lists, which means that `nlz` builds up an unrolled expression by repeatedly applying the `go` function. The problem with this is that the `b` parameter is used twice in the body of `go` which means that the size of the resulting expression is $O(2^n)$, where n is the number of calls to `go`. (The function has now been fixed by inserting an explicit sharing construct for `b`.)

Several techniques can be used to handle sharing in EDSLs:

Implicit sharing. Standard common sub-expression elimination (CSE) can be employed to remove duplications in the generated code. However, it does not fix the problem with large ASTs slowing down the compiler. This is because CSE has to traverse the whole expression in order to know which sub-expressions to share.

Observable sharing. By observing how the AST is stored in the heap, it is possible to recreate the sharing structure of the Haskell expression that generated the AST [12, 21]. The problem with observable sharing is that it is somewhat fragile: a Haskell compiler is free to store data structures as it likes, and the amount of sharing may very well depend on the implementation at hand, optimization flags, etc.

Explicit sharing. A different approach is to be completely explicit about sharing. In `FunC`, we could represent explicit sharing by the following construct:

```
Share :: FunC ((a → b) → a → b)
```

These three techniques can be combined in various ways. Kiselyov proposes using a combination of explicit and implicit sharing [28]. Hash-consing is used to introduce sharing of equal sub-expressions, and an explicit sharing construct can be used to manage the size of the expression. Similarly, it is possible to use observable sharing to speed up implicit sharing. This approach is taken by Elliott et al. in the Pan EDSL [16], and it has the advantage of not being sensitive to the unpredictable behavior of observable sharing.

There is a slight complication when using observable sharing together with HOAS: sharing has to be detected while converting the HOAS to a first-order representation [30]. It cannot be done before the conversion because a HOAS data structure is not easily inspectable, and it cannot be done after the conversion because by then the conversion has destroyed all sharing.

7.3. Simplification

Despite high-level optimizations in the shallow embedding (such as fusion, see Section 5), there are often opportunities to simplify the generated ASTs in order to generate more efficient code.

Many simplification rules can be performed directly in the front end using “smart constructors” [16]. For example, the following definitions of (+) and (<!>) can return simpler expressions depending on the form of the arguments:

```
(+) :: FunC Int → FunC Int → FunC Int
a     + Lit 0 = a
Lit 0 + b     = b
a     + b     = fromFunC (Prim "(+)" (+)) a b

(<!>) :: FunC (Array Int a) → FunC Int → FunC a
(Arr :$ l :$ Lam f) <!> i = f i
arr                 <!> i = fromFunC ArrIx arr i
```

The disadvantage of simplification in the front end is that it is limited to context-free rules. More sophisticated optimizations must therefore be done as separate passes on the generated AST (after conversion to a first-order representation in the case of HOAS).

Elliott et al. [16] and McDonnel et al. [30] give more information on optimization of EDSL programs.

7.4. Code Generation

If we want to generate efficient low-level code from FunC we are faced with a problem: since FunC is based on the lambda calculus, we may need to compile arbitrary higher-order terms which generally do not map well to efficient low-level code. However, none of the examples in this paper involves such problematic terms. In particular, the only sub-expressions of higher-order type (i.e. of the form `((t₁ → t₂) → t₃)`) are higher-order symbols like `While` and `Sequential`.

Generating code for the higher-order symbols is unproblematic. For example, the expression `While :$ Lam cont :$ Lam body :$ init` can be handled as follows:

- Generate a fresh name `s`.

- Recursively generate code for the sub-expressions `cont (Variable s)` and `body (Variable s)`.
- Put the resulting pieces of code together in a loop.

Here we see that the code generator does not view the function arguments of `While` as functions, but rather as expressions with one extra free variable – and this variable is the state of the loop. We can generate code for all the other higher-order symbols in a similar way.

7.4.1. Enforcing First-Order Target Code

We have seen that restricting programs so that higher-order types only appear for expressions that the code generator knows how to handle ensures that we can generate first-order code from FunC. This restriction can be enforced by constraining the type of `Lam`:

```
Lam :: Type a ⇒ (FunC a → FunC b) → FunC (a → b)
```

The `Type` class captures simple types that can be stored in variables in the target language (e.g. `Int`, `Bool`, `Float`, `Array Int Int`, etc.). This is a class without methods, and it is only used to restrict the set of expressions we can construct.

We will now argue for why the restricted type of `Lam` rules out arbitrary higher-order types.

Definition 3. *A* higher-order type *is a type of the form* $((t_1 \to t_2) \to t_3)$.

Definition 4. *A* compiler-known expression *is a FunC symbol applied to zero or more arguments. We assume that the compiler knows how to translate a compiler-known expression, even if it has a higher-order type.*

Proposition 1. *For all expressions* `e :: FunC a`*, if* `a` *is a higher-order type, then* `e` *is a compiler-known expression.*

Proof. All symbols are trivially compiler-known. `Lam` cannot result in a higher-order expression due to the `Type` constraint. The only form of expression left to examine is `e = (f :$ a) :: b`, where `f :: FunC (a → b)` and `a :: FunC a`. By induction, `f` is either a compiler-known expression, or `a → b` is a first-order type. Hence, either `e` is a compiler-known expression, or `b` is a first-order type.

The reasoning so far assumes that FunC has been designed so that all symbols can be handled by the code generator. For this to be the case, we also need to constrain the types of certain symbols. For example, `Lit` needs a `Type` constraint to ensure that we can only create literals of simple representable types:

```
Lit :: (Show a, Type a) ⇒ a → FunC a
```

As mentioned in Section 6, we may need to rule out code that entails representing monadic actions as values in the host language. Again, we can do this by placing a `Type` constraint on polymorphic symbols like `If` and `While`:

```
If    :: Type a ⇒ FunC (Bool → a → a → a)
While :: Type s ⇒ FunC ((s → Bool) → (s → s) → s → s)
```

Note that the above constraints will also show up in the user interface. For example, `ifC` will get the type:

```
ifC :: (Syntactic a, Type (Internal a)) ⇒ FunC Bool → a → a → a
```

8. Related Work

The Feldspar EDSL [5] makes use of the techniques described in this paper. We have found that Feldspar's design with a simple core language extended with shallow high-level libraries makes it easy to explore new ideas without investing a lot of implementation effort.

The Lightweight Modular Staging framework [35] for Scala enables the implementation of deeply embedded DSLs and offers significant infrastructure for optimization and code generation. Rompf et al. note the benefit of implementing parts of an EDSL as shallow embeddings that expand to a simpler core language – something which they call "deep linguistic reuse" [34].

Gibbons and Wu [20] give an insightful overview of deep and shallow embeddings and discuss their relation in depth. Inspired by our work, they also consider "intermediate embeddings", where a deeply embedded core language is extended using shallow embeddings.

A practical example of the combination of deep and shallow embedding is the embedded DSL Hydra which targets Functional Hybrid Modelling [24]. Hydra has a shallow embedding of signal relations on top of a deep embedding of equations. However, it does not have anything corresponding to our `Syntactic` class. Furthermore, it does not seem to take advantage of any fusion-like properties of the embedding nor make any instances of standard Haskell classes.

The work by Elliott et al. on compiling embedded languages [16] has been a great source of inspiration for us. In particular, they use a type class `Syntactic` whose name gave inspiration to our type class. However, their class is only used for overloading if expressions and not as a general mechanism for extending the embedded language. Just like Elliott et al., we note that deeply embedded compilation relates strongly to partial evaluation. The shallow embeddings we describe can be seen as a compositional and predictable way to describe partial evaluation.

The implementation of Kansas Lava [17] uses a combination of shallow and deep embedding. However, this implementation is quite different from what we are proposing. In our case, we use a *nested embedding*, where the deep embedding is used as the semantic domain of the shallow embedding. In Kansas Lava, the two embeddings exist in parallel – the shallow embedding is used for evaluation and the deep embedding for compilation. It appears that this design is not intended for extensibility: adding new interpretations is difficult due to the shallow embedding, and adding new constructs is difficult due to the deep embedding.

At the same time, Kansas Lava contains a type class Pack [22] that has some similarities to our Syntactic class. Indeed, using Pack, Kansas Lava implements support for optional values by mapping them to a pair of a boolean and a value. However, it is not clear from the publications to what extent Pack can be used to develop high-level language extensions and optimizations.

Deep embeddings have the disadvantage of leaking some implementation details to the user (e.g. a deeply embedded integer expression has type FunC Int while a Haskell integer is just an Int). In the Yin-Yang system, Jovanović et al. [26] use Scala macros to translate shallow EDSL programs to the corresponding deep EDSL programs. This allows the user to work in a friendlier shallow embedding while still reaping the benefits of the deep embedding (i.e. higher performance) when needed. Yin-Yang also simplifies EDSL development by automatically generating deep embeddings from shallow ones. In a similar line of work, Scherr and Chiba [36] propose a technique called *implicit staging* for Java which hides the implementation details of the deep embedding from the user.

While our work has focused on making shallow extensions of deep embeddings, it is also possible to have the extensions as deep embeddings. This approach was used by Claessen and Pace [11] to implement a simple language for behavioral hardware description. The behavioral language is defined as a simple recursive data type whose meaning is given as a function mapping these descriptions into structural hardware descriptions in the EDSL Lava [6].

Our focus in this paper has been on deep and shallow embeddings. But these are not the only techniques for embedding a language into a host language. Another popular method is the Finally Tagless technique [8]. The essence of Finally Tagless is to have an interface which abstracts over all interpretations of the language. In Haskell this is realized by a typeclass where each method corresponds to one language construct. Concrete interpretations are realized by creating a data type and making it an instance of the type class. For example, creating an abstract syntax tree would correspond to one interpretation and would have its own data type, evaluation would be another one. Since new interpretations and constructs can be added modularly (corresponds to adding new interpretation types and new interface classes respectively), Finally Tagless can be said to be a solution to the expression problem.

Our technique can be made to work with Finally Tagless as well. Creating a new embedding on top of an existing embedding simply amounts to creating a subclass of the type class capturing the existing embedding. However, care has to be taken if one would like to emulate a shallow embedding on top of a deep embedding and provide the kind of guarantees that we have shown in this paper. This will require an interpretation which maps some types to their abstract syntax tree representation and some types to their corresponding shallow embedding. Also, it is not possible to define general instances for standard Haskell classes for languages using the Finally Tagless technique. Instances can only be provided by particular interpretations.

As discussed in Section 2.1, Data Types à la Carte is a technique that enables modular definition of deep embeddings [41, 14]. It is complementary to the technique presented in this paper, and the two techniques can be usefully combined [2, 31].

The way we provide fusion for vectors was first used in the implementation of Feldspar [15]. The same technique was used in the language Obsidian [40] but it has never been documented that Obsidian actually supports fusion. The programming interface is very closely related to that provided by the Repa library [27], including the idea of guaranteeing fusion and providing programmer control for avoiding fusion. Although similar, the ideas were developed completely independently. It should also be noted that our implementation of fusion is vastly simpler than the one employed in Repa.

Section 6 presents a solution for the *monad reification problem*, i.e. observing the structure of monadic computations and converting them to a first order representation. Strictly speaking we don't solve the full problem here since we don't generate first order terms, but it is an easy step to add. The version presented here is a simplified and extended presentation of our previous work [31]. Other solutions include the so called "Björn and Benny" method [39], which has a particularly simple implementation but where the types become somewhat more contrived, and the normalization method of Sculthorpe et al. [37], which has applications beyond the monadic reification problem. Compiled EDSLs which feature a Monad instance include Feldspar [31], Obsidian [40] and Sunroof [7].

9. Conclusion

The technique described in this paper is a simple, yet powerful, method that gives a partial solution to the expression problem. By having a deep core language, we can add new interpretations without problem. And by means of the Syntactic class, we can add new language types and constructs with minimal effort.

The method offers an advantageous power-to-weight ratio: each construct in the deep embedding typically enables several new functions in the shallow embedding. For example, the three constructs related to immutable arrays (Section 4.9) enables us to define wide range of operations for the Vector type (only a few of which are shown in the paper).

Shallow embeddings allows for utilizing evaluation in the host language for optimization purposes. For example, pairs can be removed statically, operations on Vector can be fused automatically and monadic computations are normalized. These advantages come simply due to the fact that we use shallow embeddings, we do not have to make any extra effort to enable these optimizations.

We have presented a diverse selection of language extensions to demonstrate the idea of combining deep and shallow embeddings. The technique has been used with great success by the Feldspar team during the implementation of Feldspar.

Acknowledgments

The authors would like to thank the Feldspar team for comments and discussions on the ideas presented in this paper. The first author would like to thank the participants of the DSL conference 2011 for the their constructive feedback, when part of the material in this paper was presented as an invited tutorial. We also thanks the anonymous reviewers for constructive feedback. Thanks to Philip Wadler for pointing out that the evaluator should be strict and for identifying bugs in our implementation of `Option`. This research is funded by Ericsson and the Swedish Foundation for Strategic Research (which funds the Resource Aware Functional Programming (RAW FP) Project).

[1] Johan Ankner and Josef Svenningsson. An EDSL approach to high performance haskell programming. In *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell*, Haskell '13, pages 1–12, New York, NY, USA, 2013. ACM.

[2] Emil Axelsson. Syntactic library. http://hackage.haskell.org/package/syntactic.

[3] Emil Axelsson and Koen Claessen. Using circular programs for higher-order syntax: Functional pearl. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 257–262, New York, NY, USA, 2013. ACM.

[4] Emil Axelsson, Koen Claessen, Gergely Dévai, Zoltán Horváth, Karin Keijzer, Bo Lyckegård, Anders Persson, Mary Sheeran, Josef Svenningsson, and András Vajda. Feldspar: A domain specific language for digital signal processing algorithms. In *8th IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE 2010)*, pages 169–178. IEEE, 2010.

[5] Emil Axelsson, Koen Claessen, Mary Sheeran, Josef Svenningsson, David Engdal, and Anders Persson. The design and implementation of Feldspar. In Jurriaan Hage and MarcoT. Morazán, editors, *Implementation and Application of Functional Languages*, volume 6647 of *LNCS*, pages 121–136. Springer Berlin Heidelberg, 2011.

[6] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam 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.

[7] Jan Bracker and Andy Gill. Sunroof: A monadic DSL for generating JavaScript. In *Practical Aspects of Declarative Languages*, pages 65–80. Springer, 2014.

[8] Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming*, 19(05):509–543, 2009.

[9] William E. Carlson, Paul Hudak, and Mark P. Jones. An experiment using Haskell to prototype "geometric region servers" for navy command and control. *R. R.*, 1031, 1993.

[10] Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, pages 143–156, New York, NY, USA, 2008. ACM.

[11] Koen Claessen and Gordon Pace. An embedded language framework for hardware compilation. *Designing Correct Circuits*, 2, 2002.

[12] Koen Claessen and David Sands. Observable sharing for functional circuit description. In P.S. Thiagarajan and Roland Yap, editors, *Advances in Computing Science, ASIAN'99*, volume 1742 of *LNCS*, pages 62–73. Springer, 1999.

[13] Koen Claessen, Mary Sheeran, and Bo Joel 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*, pages 21–30. ACM, 2012.

[14] Laurence E. Day and Graham Hutton. Compilation à la carte. In *Proceedings of the 25th Symposium on Implementation and Application of Functional Languages*, IFL '13, pages 13:13–13:24, New York, NY, USA, 2014. ACM.

[15] Gergely Dévai, Máté Tejfel, Zoltán Gera, Gábor Páli, Gyula Nagy, Zoltán Horváth, Emil Axelsson, Mary Sheeran, András Vajda, Bo Lyckegård, and Anders Persson. Efficient Code Generation from the High-level Domain-specific Language Feldspar for DSPs. In *Proc. ODES-8: 8th Workshop on Optimizations for DSP and Embedded Systems*, 2010.

[16] Conal Elliott, Sigbjørn Finne, and Oege de Moor. Compiling embedded languages. *Journal of Functional Programming*, 13:3:455– 481, 2003.

[17] Andrew Farmer, Garrin Kimmell, and Andy Gill. What's the matter with kansas lava? In Rex Page, Zoltán Horváth, and Viktória Zsók, editors, *Trends in Functional Programming*, volume 6546 of *Lecture Notes in Computer Science*, pages 102–117. Springer Berlin Heidelberg, 2011.

[18] Leonid Fegaras and Tim Sheard. Revisiting catamorphisms over datatypes with embedded functions (or, programs from outer space). In *Annual Symposium on Principles of Programming Languages: Proceedings of the 23 rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, volume 21, pages 284–294, 1996.

[19] Martin Fowler. *Domain Specific Languages.* Addison-Wesley Professional, 2010.

[20] Jeremy Gibbons and Nicolas Wu. Folding domain-specific languages: Deep and shallow embeddings (functional pearl). In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 339–347, New York, NY, USA, 2014. ACM.

[21] Andy Gill. Type-safe observable sharing in Haskell. In *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, Haskell '09, pages 117–128, New York, NY, USA, 2009. ACM.

[22] Andy Gill, Tristan Bull, Andrew Farmer, Garrin Kimmell, and Ed Komp. Types and type families for hardware simulation and synthesis: The internals and externals of Kansas Lava. In *Proceedings of the 11th international conference on Trends in functional programming*, pages 118–133. Springer-Verlag, 2010.

[23] Andy Gill, John Launchbury, and Simon L. Peyton Jones. A short cut to deforestation. In *Proc. Int. Conf. on Functional programming languages and computer architecture (FPCA)*, pages 223–232. ACM, 1993.

[24] George Giorgidze and Henrik Nilsson. Mixed-level embedding and JIT compilation for an iteratively staged DSL. In *Revised selected papers of the 19th international workshop on Functional and (Constraint) Logic Programming, Madrid, Spain*, volume 6559 of *Lecture Notes in Computer Science*. Springer, 2010.

[25] Paul Hudak. Domain-specific languages. *Handbook of Programming Languages*, 3:39–60, 1997.

[26] Vojin Jovanović, Amir Shaikhha, Sandro Stucki, Vladimir Nikolaev, Christoph Koch, and Martin Odersky. Yin-yang: Concealing the deep embedding of DSLs. In *Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences*, GPCE 2014, pages 73–82, New York, NY, USA, 2014. ACM.

[27] Gabriele Keller, Manuel M.T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, and Ben Lippmeier. Regular, shape-polymorphic, parallel arrays in haskell. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 261–272, New York, NY, USA, 2010. ACM.

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

[29] John Launchbury and Simon L. Peyton Jones. Lazy functional state threads. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pages 24–35, New York, NY, USA, 1994. ACM.

[30] Trevor L. McDonell, Manuel M.T. Chakravarty, Gabriele Keller, and Ben Lippmeier. Optimising purely functional GPU programs. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 49–60, New York, NY, USA, 2013. ACM.

[31] Anders Persson, Emil Axelsson, and Josef Svenningsson. Generic monadic constructs for embedded languages. In Andy Gill and Jurriaan Hage, editors, *Implementation and Application of Functional Languages*, volume 7257 of *Lecture Notes in Computer Science*, pages 85–99. Springer Berlin Heidelberg, 2012.

[32] Simon Peyton Jones. *Haskell 98 language and libraries: the revised report.* Cambridge University Press, 2003.

[33] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, PLDI '88, pages 199–208. ACM, 1988.

[34] Tiark Rompf, Nada Amin, Adriaan Moors, Philipp Haller, and Martin Odersky. Scala-virtualized: linguistic reuse for deep embeddings. *Higher-Order and Symbolic Computation*, 25(1):165–207, 2012.

[35] Tiark Rompf and Martin Odersky. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled DSLs. *Commun. ACM*, 55(6):121–130, June 2012.

[36] Maximilian Scherr and Shigeru Chiba. Implicit staging of EDSL expressions: A bridge between shallow and deep embedding. In Richard Jones, editor, *ECOOP 2014 – Object-Oriented Programming*, volume 8586 of *Lecture Notes in Computer Science*, pages 385–410. Springer Berlin Heidelberg, 2014.

[37] Neil Sculthorpe, Jan Bracker, George Giorgidze, and Andy Gill. The constrained-monad problem. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 287–298, New York, NY, USA, 2013. ACM.

[38] Josef Svenningsson and Emil Axelsson. Combining deep and shallow embedding for edsl. In *Trends in Functional Programming*, pages 21–36. Springer, 2013.

[39] Josef Svenningsson and Bo Joel Svensson. Simple and compositional reification of monadic embedded languages. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 299–304, New York, NY, USA, 2013. ACM.

[40] Joel Svensson, Mary Sheeran, and Koen Claessen. Obsidian: A domain specific embedded language for parallel programming of graphics processors. In Sven-Bodo Scholz and Olaf Chitil, editors, *Implementation and Application of Functional Languages*, volume 5836 of *Lecture Notes in Computer*

Science, pages 156–173. Springer Berlin Heidelberg, 2011. Presented at IFL 2008.

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

[42] Janis Voigtländer. Asymptotic improvement of computations over free monads. In *Mathematics of Program Construction*, pages 388–403. Springer, 2008.

[43] Philip Wadler. The expression problem. http://www.daimi.au.dk/~madst/tool/papers/expression.txt, 1998.