Combining Deep and Shallow Embedding for EDSL

Josef Svenningsson and Emil Axelsson

Chalmers University of Technology
{josefs,emax}@chalmers.se

Abstract. 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. 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. In particular it helps keeping the deep embedding small and it makes extending the embedded language much easier. Our technique also has some unexpected but welcome knock-on 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

When compiling an embedded language it is natural to use an algebraic data type to represent the abstract syntax tree (AST). This is known as a *deep* embedding. Deep embeddings can be cumbersome: the AST can grow quite large in order to represent all the language features, which can make it rather unwieldy to work with. It is also laborious to add new language constructs as it requires changes to the AST as well as all functions manipulating the AST.

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. But if we wish to compile our embedded language we have little choice but having some form of AST — in particular if we not only want to compile it, but first transform the representation, or if we have another type of backend, say, a verification framework.

In this paper we present a technique for combining deep and shallow embeddings in order to achieve many of the advantages of both styles. This combination turns out to provide knock-on effects which we also 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 side-effects:

- The shallow interfaces can have properties not possessed by the deep embedding. For example, our vector interface (section 4.5) 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 section 4.4 and 4.5).

Extensibility Our technique can be seen as a partial solution to the expression problem[1] as it makes it easier to extend the embedded language with new language constructs and functions.

Before giving an overview of our technique (section 3) we will give an introduction to shallow and deep embeddings in section 2, including a comparison of the two methods (section 2.1).

Throughout this paper we will use Haskell [16] and some of the extensions provided by the Glasgow Haskell Compiler. While we will use many Haskell-specific functions and constructs the general technique and its advantages translates readily to other languages.

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) from [5] as an illustrating example.

```
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:

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

```
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 that the representation they use directly encode the operations that can be performed on them. 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.

The characteristic of deep embeddings is that they use an abstract syntax tree to represent the domain. 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 this 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 commonly referred to as the *expression problem*.

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 ($\cap$). 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 6).

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

3 Overview of the Technique

We assume a setting where we want an EDSL which generates code. Code generation tends to require intensional analysis of the AST, which is not directly possible with a shallow implementation (but see reference [4] for how to generate code in the final tagless style). Hence, we need 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.

These ideas have been partly described in our paper on the implementation of the Feldspar EDSL [1]. However, we feel that the ideas are important enough to be presented as a general technique, not tied to a particular language implementation.

In the sections that follow we will demonstrate our technique through a series of examples. For the sake of concreteness we have made some superficial choices which are orthogonal to our technique. In particular, we use a typed embedded language and employ higher order abstract syntax to deal with binding constructs. Neither of these choices matter for the applicability of our technique.

4 Examples

To demonstrate our technique we will use a small embedded language called FunC as our running example. The data type describing the FunC abstract syntax tree can be seen below.

```
data FunC a where
  LitI     :: Int  → FunC Int
  LitB     :: Bool → FunC Bool
  If       :: FunC Bool → FunC a → FunC a → FunC a
  While    :: (FunC s → FunC Bool) → (FunC s → FunC s) → FunC s → FunC s
  Pair     :: FunC a → FunC b → FunC (a,b)
  Fst      :: FunC (a,b) → FunC a
  Snd      :: FunC (a,b) → FunC b
  Prim1    :: String → (a → b) → FunC a → FunC b
  Prim2    :: String → (a → b → c) → FunC a → FunC b → FunC c
  Value    :: a → FunC a
  Variable :: String → FunC a
```

FunC is a low level, pure functional language which has a straightforward translation to C. It is meant for embedding low level programs and is inspired by the Core language used in the language Feldspar [2]. We use a GADT to give precise types to the different constructors. We have also chosen Higher Order Abstract Syntax [17] to represent constructs with variable binding. In the above data type, the only higher-order construct is `While`. We will add another one in section 4.5.

FunC has constructs for integer and boolean literals and an if-expression for testing booleans. The while expression models while loops. Since FunC is pure, the body of the loop cannot perform side-effects, so instead the while loop passes around a state. The third argument to the `While` constructor is the initial value of the state. The state is then updated each iteration of the loop by the second argument. In order to determine when to stop looping the first argument

is used, which performs a test on the state. Furthermore, FunC has pairs which are constructed with the Pair constructor and eliminated using Fst and Snd. The constructs Prim1 and Prim2 are used to create primitive functions in FunC. The string argument is the name of the primitive function which is used when generating code from FunC and the function argument is used during evaluation. It is possible to simply have a single constructor for primitive functions of an arbitrary number of arguments but that would complicate the presentation unnecessarily for the purpose of this paper. The two last constructors, Value and Variable , are not part of the language. They are used internally for evaluation and printing respectively.

The exact semantics of the FunC language is given by the eval function.

```
eval :: FunC a → a
eval (LitI i)        = i
eval (LitB b)        = b
eval (While c b i)   = head $ dropWhile (eval ∘ c ∘ Value) $
                       iterate (eval ∘ b ∘ Value) $ eval i
eval (If c t e)      = if eval c then eval t else eval e
eval (Pair a b)      = (eval a, eval b)
eval (Fst p)         = fst (eval p)
eval (Snd p)         = snd (eval p)
eval (Prim1 _ f a)   = f (eval a)
eval (Prim2 _ f a b) = f (eval a) (eval b)
eval (Value a)       = a
```

4.1 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 our FunC language. We call the type class Syntactic .

```
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 of FunC. This is what the associated type Internal is for. The two functions toFunC and fromFunC translates back and forth between the type T and its internal representation. The fromFunC method is needed when defining user interfaces based on the Syntactic class. 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
```

4.2 User Interface

Now that we have the Syntactic class we can give FunC a nice extensible interface which we can present to the programmer using FunC. This interface will mirror the deep embedding and its constructor 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 .

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

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 (c ∘ fromFunC)
                              (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.

We have not shown any interface for integers. One way to implement that would be to provide a function equivalent to the LitI constructor. In Haskell there is a nicer way: provide an instance of the type class Num. By instantiating Num we get access to Haskell's overloaded syntax for numeric literals so that we don't have to use a function for lifting numbers into FunC. Additionally, Num contains arithmetic functions which we also gain access to. Similarly, we instantiate the Integral class to get an interface for integral operations. The primive functions of said type classes are implemented using the constructors Prim1 and Prim2. We refrain from presenting the code as it is rather Haskell-specific and unrelated to the main point of the paper.

We will also be using comparison operators in FunC. For tiresome reasons it is not possible to overload the methods of the corresponding type classes Eq and Ord: these methods return 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 operators are hidden and we will use definitions specific to FunC.

4.3 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 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.

4.4 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:

[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 [7]. This issue is, however, orthogonal to the ideas presented in this paper.

```
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 (Fst m) (Snd m)
    toFunC (Option b a)      = 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 very straight forward but when we try to implement functions for creating values of type Option we run into problems. Specifically it is hard to create an empty Option value, because we need some value to put into the second component of the pair. FunC is simply not expressive enough to encode this type as it stands. So we will have to extend FunC somehow to accommodate the Option type. There are several ways of doing this and we have chosen a very minimal extension. We add a notion of undefined values. To begin with we add an extra constructor to the FunC type.

```
Undef :: FunC a
```

Next we give semantics to Undef (undefined is part of the Haskell standard) and provide an overloaded function undef for convenience.

```
eval Undef = undefined

undef :: Syntactic a ⇒ a
undef = fromFunC Undef
```

Armed with undefined values we can now easily provide functions for constructing optional values:

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

none :: Syntactic a ⇒ Option a
none = Option false undef

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. It is the function none which uses the undef function we previously added to FunC. The function option acts as a case on optional values, allowing the programmer to test an Option value to see whether is contains something or not.

[3] Larger unions can be encoded using an integer instead of a boolean.

The function above provides 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 divO :: FunC Int → FunC Int → Option (FunC Int) which returns nothing in the case the divisor is zero. Then we can write functions such as the following:

```
divTest :: FunC Int → FunC Int → FunC Int → Option (FunC Int)
divTest a b c = do r1 ← divO a b
                   r2 ← divO a c
                   return (r1+r2)
```

4.5 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 → (FunC Int → FunC a) → FunC (Array Int a)
ArrLen :: FunC (Array Int a) → FunC Int
ArrIx  :: FunC (Array Int a) → FunC Int → FunC 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 l ixf) = listArray (0,lm1) [eval $ ixf $ value i | i ← [0..lm1]]
                     where lm1 = eval l − 1
eval (ArrLen a)  = (1 +) $ uncurry (flip (−)) $ bounds $ eval a
eval (ArrIx a i) = eval a ! eval i
```

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 arr = ArrLen arr

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

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)   = Arr l (toFunC ∘ 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 4.3. Finally, just as with the Option type in section 4.4 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 :: (Syntax 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.

5 Fusion

Choosing to implement vectors as a shallow embedding has a very powerful consequence: it provides a very lightweight implementation of fusion [12]. 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. 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 → Arr l (toFunC ∘ 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 :: (Syntax 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.

The Vector type is not the only type which can benefit from fusion. In Feldspar there is a library for streams which captures the notion of an infinite sequence of values [2]. We use a shallow embedding of streams which also enjoys fusion in the same way as the vector library we have presented here. In fact, fusion is only one example of the kind of compile time transformations that can be achieved. We have a shallow embedding of monads which provides normalization of monadic expressions by applying the third monad law [15]. Common to all of these transformations is that they come with very strong guarantees that the transformations are actually performed.

6 Discussion

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. There is only one problem: Quite often, shallow language extensions are not efficiently (or not at all) expressible in the underlying deep implementation. When this happens, the deep implementation has to be extended. For example, when adding the Vector type (section 4.5), the core language had to be extended with the constructors Arr, ArrLen and ArrIx. Note, however, that this is a quite modest extension compared to the wealth of operations available in the vector library. (This paper has only presented a small selection of the operations.)

Furthermore, it is probable that, once the deep implementation has reached a certain level of completeness, new shallow extensions will not require any new changes to the deep implementation. As an example, when adding the Option type (section 4.4), the only deep extension needed was the Undef constructor. But this constructor does not have much to do with optional values; it could easily have been in the language already for other reasons.

In this paper we have shown a small but 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. The extensions implemented include libraries for finite and infinite streams, fixed-point numbers and bit vectors. Recently, we have also used the technique to embed monadic computations in order to enrich Feldspar with controlled side-effects [15].

7 Related Work

A practical example of the design pattern we have outlined here is the embedded DSL Hydra which targets Functional Hybrid Modelling [13]. Hydra has a shallow embedding of signal relations on top of a deep embedding of equations. However, they do not have anything corresponding to our `Syntactic` class. Furthermore, they don't take advantage of any fusion-like properties of their embedding nor do they make any instances of standard Haskell classes.

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 meta language. Another popular technique is the Finally Tagless technique [4]. 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.

The implementation of Kansas Lava [10] 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 in-

tended 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 [11] 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.

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 [6] 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 [3].

In their seminal work on compiling embedded languages, Elliott et al. use a type class Syntactic whose name gave inspiration to our type class [9]. However, their class is only used for overloading if expressions, and not as a general mechanism for extending the embedded language.

The way we provide fusion for vectors was first reported in [8] for the language Feldspar. The same technique was used in the language Obsidian [18] 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 [14], 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.

As in [9] 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.

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. This research is funded by Ericsson and the Swedish Foundation for Strategic Research. The Feldspar project is an initiative of and is partially funded by Ericsson Software Research and is a collaboration between Chalmers, Ericsson and ELTE University.

References

1. Axelsson, E., Claessen, K., Sheeran, M., Svenningsson, J., Engdal, D., Persson, A.: The design and implementation of Feldspar – an embedded language for digital signal processing. In: IFL 2010. LNCS, vol. 6647 (2011)

2. Axelsson, E., Dévai, G., Horváth, Z., Keijzer, K., Lyckegård, B., Persson, A., Sheeran, M., Svenningsson, J., Vajda, A.: Feldspar: A Domain Specific Language for Digital Signal Processing algorithms. In: Formal Methods and Models for Codesign, MemoCode. IEEE Computer Society (2010)
3. Bjesse, P., Claessen, K., Sheeran, M., Singh, S.: Lava: Hardware Design in Haskell. In: ICFP '98: Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming. pp. 174–184. ACM (1998)
4. Carette, J., Kiselyov, O., Shan, C.: Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. Journal of Functional Programming 19(05), 509–543 (2009)
5. Carlson, W., Hudak, P., Jones, M.: An experiment using Haskell to prototype "geometric region servers" for navy command and control. R. R. 1031 (1993)
6. Claessen, K., Pace, G.: An embedded language framework for hardware compilation. Designing Correct Circuits 2 (2002)
7. Claessen, K., Sands, D.: Observable sharing for functional circuit description. In: Proc. of Asian Computer Science Conference (ASIAN). Lecture Notes in Computer Science, Springer Verlag (1999)
8. Dévai, G., Tejfel, M., Gera, Z., Páli, G., Nagy, G., Horváth, Z., Axelsson, E., Sheeran, M., Vajda, A., Lyckegård, B., Persson, A.: 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)
9. Elliott, C., Finne, S., de Moor, O.: Compiling embedded languages. Journal of Functional Programming 13:3, 455– 481 (2003)
10. Farmer, A., Kimmell, G., Gill, A.: What's the matter with Kansas Lava? In: Trends in Functional Programming (2010)
11. Gill, A., Bull, T., Farmer, A., Kimmell, G., Komp, E.: 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. pp. 118–133. Springer-Verlag (2010)
12. Gill, A., Launchbury, J., Peyton Jones, S.: A short cut to deforestation. In: Proc. Int. Conf. on Functional programming languages and computer architecture (FPCA). pp. 223–232. ACM (1993)
13. Giorgidze, G., Nilsson, H.: 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. Lecture Notes in Computer Science, vol. 6559. Springer (2010)
14. Keller, G., Chakravarty, M., Leshchinskiy, R., Jones, S.P., Lippmeier, B.: Regular, shape-polymorphic, parallel arrays in haskell. In: Proceedings of ICFP 2010. ACM SIGPLAN (2010)
15. Persson, A., Axelsson, E., Svenningsson, J.: Generic monadic constructs for embedded languages. Implementation of Functional Languages (IFL'11) (2011)
16. Peyton Jones, S.: Haskell 98 language and libraries: the revised report. Cambridge University Press (2003)
17. Pfenning, F., Elliott, C.: Higher-order abstract syntax. In: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation. pp. 199–208. PLDI '88, ACM (1988)
18. Svensson, J., Sheeran, M., Claessen, K.: Obsidian: A domain specific embedded language for parallel programming of graphics processors. In: Implementation and Application of Functional Languages 2008. LNCS, vol. 5836. Springer (2011)