Strategies and Tradeoffs in Designing and Implementing Embedded DSLs

A Report Submitted in Partial Satisfaction of the Requirements for the Ph.D Qualifying Examination in Computer Science, Indiana University.
2014

Thejaka Amila Kanewala
thejkane@iu.edu
School of Informatics and Computing
Indiana University, Bloomington, IN, USA

ABSTRACT

Domain Specific Language (DSL) is an elegant software engineering solution to fairly complex problems in specific subject areas. While DSLs provide apt solutions to many domain problems, developing a DSL from the scratch is a laborious task that consumes considerable amount of money and time. Recently embedding has become a widely used methodology to develop DSLs. Embedded DSLs (EDSLs) reduces time and cost by reusing host programming language features such as parser, type checker, etc. In this paper we will go through various strategies used to embed a DSL into a general purpose programming language. Also we will discuss several implementation strategies for each embedding type and a comparison between different embedding strategies.

Categories and Subject Descriptors

D.1.3 [**Software**]: Programming Languages—*Domain Specific Languages*

General Terms

Programming Languages

Keywords

Embedded DSL, Domain Specific Languages

1. INTRODUCTION

To program using *general purpose language* (GPL) programmer needs certain level of understanding and experience about the programming language. Many application areas develop their tools on top of general purpose programming languages. Following standard Software Engineering methodologies, developers in these application areas develop an infrastructure that suits their application domain (usually a set of libraries). Programmers in these environments, need to be build an expertise about application domains as well as need to be well aware about the general purpose programming language. For many application areas *Domain Specific Languages* (DSL) has been used successfully in implementing solutions to the problems in their domain with less effort, with less cost and with less expertise on general purpose languages. Some of the success stories are, *LATEX* in document formatting, *HTML* in markup, *SQL* in database querying and the list grows.

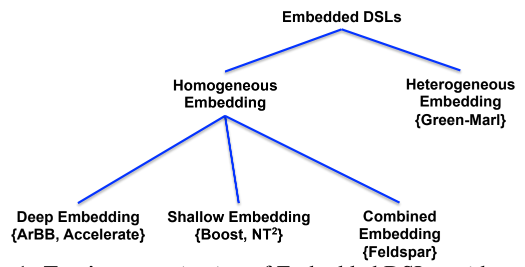
Regardless of all the advantages in DSLs, DSLs come with a certain cost. One common problem with DSLs is the performance. DSLs performs poorly compared to the equivalent programs implemented using general purpose languages. Further startup cost for a DSL might be considerable. It can be fairly complex process to design and implement a good DSL from the scratch (from parsing to fully functioning compiler that performs optimizations and code generation). May even take 1-2 years to get it to an usable state. Due to the same reason it may not be practical to develop a DSL for every application domain.

Embedded Domain Specific Languages (EDSLs) is a strategy to overcome limitations in DSLs. Sometimes EDSLs are referred as *Internal* DSLs. In EDSLs we do not build the language from scratch, instead we use an existing infrastructure to build the DSL. Usually the existing infrastructure is supported by a general purpose language. Since EDSLs are capable of re-using existing infrastructure the time and money needed to develop an EDSL is less.

The definition of the EDSL is not very precise. *Wikipedia* defines an embedded DSL as an implementation of libraries which exploits the syntax of their host general purpose language or a subset thereof, while adding domain-specific language elements (data types, routines, methods, macros etc.) [13]. Laurence Tratt presented a different way to categorize embedded DSLs in [11]. In his classification he divided EDSLs into 2 main categories. They are *Heterogeneous Embedding* and *Homogeneous Embedding*. Embedding is homogeneous when the host language and DSL uses the same compiler program. Embedding is heterogeneous when the system used to compile the host language, and the system used to implement the embedding is different. This classification is useful in analysing different strategies used to implement EDSLs.

There are several widely used techniques to do Homogeneous Embedding. An interesting special case is when embedding does not need any translation; i.e. the embedding language is syntactically and semantically a subset of the host language. In such case the EDSL would simply be a runtime library implementation. In DSL community we call this strategy *shallow embedding*. In shallow embedding all host language operations are translated to a target code. Another strategy is to get hold of the *Abstract Syntax Tree* generated by EDSL program and implement semantics. This strategy is called *deep embedding*. A mixed strategy of deep embedding and shallow embedding is also explained in [8].

On the other hand Heterogeneous Embedding does not necessarily mean that the host language is different from the language used to implement the embedding: it is possible to identify a heterogeneous embedding approach as one whose two separate systems just so happen to be written in the same language. Figure 1 summarizes the


Embedded DSLs
Homogeneous Embedding
Heterogeneous Embedding {Green-Marl}
Deep Embedding {ArBB, Accelerate}
Shallow Embedding {Boost, NT²}
Combined Embedding {Feldspar}
Figure 1: Tratt's categorization of Embedded DSLs, with examples

EDSL categorization.

We are interested in DSL implementations in parallel/distributed programming. Most of the recently developed Embedded DSLs for parallel/distributed programming belongs to Homogeneous Embedding category. Therefore our discussion is more focused on Homogeneous Embedding strategies.

In this paper we will discuss in detail about strategies used to implement EDSLs with several examples. For deep embedding we will discuss *Intel's ArBB* [6] and *Accelarate* [2], two DSLs that are implemented on C++ and Haskell. For shallow embedding we will use *Boost Graph Library* (BGL) [5] and NT^2 [10] as example implementations. Further we will discuss language features used by EDSLs with several examples. Finally we will compare each strategy and discuss when each strategy is suitable.

2. EMBEDDING DSLS

In this section we will discuss embedding strategies in detail. Throughout the section, to explain embedded strategies we will use a "very simple", example DSL when appropriate. In this simple DSL we will have integer literals and "Add", "Mult" and "Sub" words to represent integer addition, multiplication and subtraction. We may represent ((3+2)+5) in this language as follows;

$$(Add(Add(Num3)(Num2))(Num5))$$

2.1 Homogeneous Embedding

In Homogeneous Embedding we use the same compiler program to compile both DSL as well as the host language. In early days most of the DSL embedding is achieved via macros. An example is LISP macros. LISP's flexible syntax and powerful macro system opens path to develop DSLs. Only LISP compiler is used to compile LISP macro based DSLs.

Nowadays several other techniques are available for homogeneous embedding. But in general those strategies can be categorised into Deep Embedding and Shallow Embedding. Deep Embedding particularly popular in functional programming. Further Deep Embedding is used in many parallel DSLs, simply because deep embedding opens lot of avenues for optimizations.

On the other hand shallow embedded languages take a form of a library and an API. In this section we will discuss in detail about deep embedding and shallow embedding with examples whenever appropriate.

2.1.1 Deep Embedding

In Haskell like programming languages we can implement above described arithmetic DSL using *algebraic data types*. Listing 1 shows how we use Haskell algebraic data types to encode arithmetic constructs. For this example we use Haskell's *Generalized Algebraic Datatypes* (GADT). GADT is a generalization of *Abstract Data Types*.

Listing 1: Encoding arithmetic operators using Haskell GADTs

```haskell
module Deep
where
import Data.List

data Expr = Num Int
    | Add Expr Expr
    | Mult Expr Expr
    | Sub Expr Expr
```

Following are 2 valid arithmetic DSL statements;

1. $(Mult(Num3)(Num2))$
2. $(Add(Num1)(Num2))$

For a deep embedding language the evaluation function would look like in Listing 2.

Listing 2: Evaluation logic for deep embedding

```haskell
evalDsl :: Expr -> Int
evalDsl (Num n) = n
evalDsl (Add x y) = (evalDsl x) + (evalDsl y)
evalDsl (Mult x y) = (evalDsl x) ∗ (evalDsl y)
evalDsl (Sub x y) = (evalDsl x) - (evalDsl y)
```

We would execute the DSL as follows;
$*Deep > (evalDsl(Mult(Num3)(Num2)))$
6
$*Deep > (evalDsl(Add(Num1)(Num2)))$
3

As shown in above example a deep embedding uses a representation of AST in a form of algebraic data type and a function (evalDSL in above example) that assigns semantics to the algebraic data type by matching cases. The fact that deep embedding make uses of AST is important. Infact that is the main difference between deep embedding and shallow embedding. Though Haskell has a nice interface to work with generated AST some other general purpose programming languages does not provide features to easily access AST generated. For example a language like C++ does not provide a straightforward way to access the AST generated by the DSL. Later in Section 4.2 we will discuss ArBB which is a template based DSL for array processing in parallel programs; ArBB is a deeply embedded language that uses AST like structure to implement the DSL.

2.1.2 Shallow Embedding

Above described trivial DSL can be encoded in a shallow embedded way in Haskell as follows;

Listing 3: Shallow embedding arithmatic DSL

```haskell
module Shallow
where

import Data.List

type Expr = Int

num :: Int -> Int
num n = n

add :: Expr -> Expr -> Expr
add p q = p + q

mult :: Expr -> Expr -> Expr
mult p q = p ∗ q
```

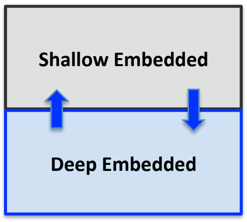


Figure 2: Combine embedding; layering deep and shallow embedding

```

sub :: Expr -> Expr -> Expr
sub p q = p - q

evalDsl :: Expr -> Int
evalDsl n = n
```

Then we would evaluate the DSL as follows;
$*Shallow > (evalDsl(mult(num3)(num2)))$
6
$*Shallow > (evalDsl(add(num1)(num2)))$
3

The literals "num", "add", "mult" and "sub" are correspond to the "Num", "Add", "Mult" and "Sub" in deeply embedded implementation. You may also notice that constructs "Num", "Add", "Mult" and "Sub" are data type definitions in deep embedding and "num", "add", "mult" and "sub" are function definitions in shallow embedding. In the case of deep embedding the definitions of Num, Add, Mult and Sub are part of data types. Those definitions help to build the AST. Further interpreting AST is completely a responsibility of evalDsl function. But in the case of shallow embedding the evalDsl function does not do much work. Most of the semantic interpretation are taken place at the definitions of "num", "mult" etc. Further shallow embedding does not deal with an AST.

2.1.3 *Combined Embedding*

In deep embedding adding new language features require changing the generated AST as well as changing the evaluation function. As most of the DSLs evolve they tend to generate quite large ASTs. Changing ASTs and adding new functionality become laborious on such situations. On the other hand shallow embedding does a direct mapping to semantics and shallow embedding not well suited when we need to do transformations. To overcome these limitations in shallow embedding and deep embedding [8] gave a combined strategy of embedding.

[8] preliminary assumes the need for deep embedded language in order to perform code generation. Under that assumption [8]'s approach of developing a combined embedded DSL is as follows;

1. The main language features (also called *core* language) are implemented using AST in a deep embedded style.
2. Implement user friendly interfaces with shallow embeddings on top of the core language.
3. Give each interface a precise meaning by giving a translation to and from a corresponding core language program.

One could think of combined embedding as a way of layering deep embedding and shallow embedding. Layering is depicted in Figure 2. In the case of Haskell the shallow embedding wrapping on top of deep embedded implementation is carried out using *type classes* in Haskell.

Feldspar [1] is a DSL which uses combined embedding strategy. We will discuss about Feldspar detail in a later section.

3. EMBEDDING STRATEGIES

In this section we will discuss several widely used techniques for embedding, with examples.

3.1 Deep Embedding Strategies

3.1.1 *Data Type representation of Syntax*

This is the strategy used in most of the functional languages such as Haskell. We already saw an example of this strategy in an earlier sub section.

In this methodology the syntax of the language is represented as a data type. Each constructor represent a grammar production of the language and its argument types correspond to non terminals that occur in the right hand side of the production. Constructors without arguments represent terminal symbols.

Semantics of the DSL is defined using a valuation function as shown in Listing 2. Compared to declarative languages functional languages such as Haskell provide better infrastructure to implement deeply embedded DSLs using data type based syntax.

3.1.2 *Multi-Stage Programming*

Multi-Stage programming (MSP) [9] is a program generation methodology for developing generic softwares. Type systems for MSP languages are designed to statically ensure that generated programs are type safe. A detail description about MSP is given in [9] using MetaOCaml (an extension of OCaml).

One of the main uses of MSP is to make it easy for user to develop DSLs. MSP languages provide a set of constructs for the *construction*, *combination* and *execution*. In reference [9] those constructs are referred as *Brackets*, *Escape* and *Run*. In following we will explain these constructs briefly.

Brackets (Construction) (written <...>): This basically says that code inside brackets should delay its execution. MetaOCaml generates source code at runtime for code within brackets. Following we show how code is generated for bracketed expressions. Left hand side of the arrow is the MetaOCaml code and right hand side we show the generated code.

let a = 1+2; ⟶ val a : int = 3
let a = .<1+2>.; ⟶ val a : int code = .<1+2>.

If brackets are not present, addition on 1+2 is performed immediately. If brackets are present we will get an expression like ".<1+2>". An interesting feature in MetaOCaml is the generated expressions (such as ".<1+2>") has a type associated. For above example the type of ".<1+2>" is *int code*.

Escape (Combination) (written .): This allows the combination of smaller delayed values to construct larger ones.

let b = .<.~a * .~a >. ; ⟶ val b : int code = .<(1 + 2) * (1 + 2)>.

The declaration binds b to delayed computation.

The last construct is *Run (written !))* . This allows us to compile and run DSL code.

MetaOCaml follows *staged interpreter* approach in DSL implementation. First an interpreter is implemented and tested for the DSL. Then, the three stage constructs are used to produce an implementation that performs the traversal of the DSL program. Let's go through an example to see how above constructs are used in DSL implementation.

The stage interpreter we are going to build will take a program AST as an input. In CodeListing 0, we present a variation of arithmatic DSL grammar to demonstrate MSP based on an example in [9], Section 1.5.

CodeListing 0 : Grammar for arithmatic DSL

```
type exp = Int of int | Var of string | App of string * exp
| Add of exp * exp | Sub of exp * exp
| Mul of exp * exp
type def = Declaration of string * string * exp
type prog = Program of def list * exp
```

An example (name it E_{AST}) program in the DSL in AST form is given in CodeListing 1.

CodeListing 1 : Example program AST

```
E_AST =Program ([Declaration
          ("sample_add","x", "y", Add(Var "x", Var "y"))
],
App ("sample_add", Int 10, Int 20))
```

Now we need to create an intepretter for staged code. Taha [9] explains a nice process to build stage interpretter. Briefly the strategy is to develop an interpretter for DSL then make use of three staging constructs to build the staged interpretter. We first show the intereprettter for DSL (eval1).

CodeListing 2 : eval1 Implementation; Raw interpretter for DSL

```
let rec eval1 e env fenv =
match e with
     Int i -> i
| Var s -> env s
| App (s,e2) -> (fenv s)(eval1 e2 env fenv)
| Add (e1,e2) -> (eval1 e1 env fenv)+(eval1 e2 env fenv)
| Sub (e1,e2) -> (eval1 e1 env fenv)-(eval1 e2 env fenv)
| Mul (e1,e2) -> (eval1 e1 env fenv)*(eval1 e2 env fenv)

let rec peval1 p env fenv=
     match p with
      Program ([],e) -> eval1 e env fenv
     |Program (Declaration (s1,s2,e1)::tl,e) ->
      let rec f x = eval1 e1 (ext env s2 x) (ext fenv s1 f)
      in peval1 (Program(tl,e)) env (ext fenv s1 f)
```

Next we use stageing constructors <...> and . to build the staged interpretter (eval2) as depicted in CodeListing 3.

We can evaluate example DSL program in CodeListing 1 (E_{AST}), with eval2 staged interpretter. The eval2 output is given in CodeListing 4.

The output code is very similar to the program that we would implement the example without the DSL. Above code can be executed using Run (!) construct.

The interpretter eval1 immediately evaluates the code. But interpretter eval2 does not immediately interpret the code instead it generates code in the form of a typed string. Compiler can do optimizations on the generated code (e.g:- *operator fusion*) to avoid duplicate computations and improve performance.

Since MSP works on an AST and semantics are defined using evaluation function Embedded DSLs developed using MSP technique belongs to Deep Embedding category.

3.1.3 Template Programming

Template programming is another generative programming mechanism used in EDSL development. We will discuss approaches

CodeListing 3 : Staged Interpretter Implementation

```
let rec eval2 e env fenv =
match e with
     Int i -> .<i>.
| Var s -> env s
| App (s,e2) -> .<.~(fenv s).~(eval2 e2 env fenv)>.
| Add (e1,e2)-> .<.~(eval2 e1 env fenv)+.~(eval2 e2 env fenv)>.
| Mult (e1,e2)-> .<.~(eval2 e1 env fenv)*.~(eval2 e2 env fenv)>.
| Sub (e1,e2)-> .<.~(eval2 e1 env fenv)-.~(eval2 e2 env fenv)>.

let rec peval2 p env fenv=
     match p with
      Program ([],e) -> eval2 e env fenv
     |Program (Declaration (s1,s2,e1)::tl,e) ->
      .<let rec f x = . (eval2 e1 (ext env s2 .<x>.)
                    (ext fenv s1 .<f>.))
       in . (peval2 (Program(tl,e)) env (ext fenv s1 .<f>.))>.
```

CodeListing 4 : Output of eval2 after processing E_{AST}

```
.<let f = fun x y -> x + y in (f 10 20)>.
```

used in two different programming languages to implement EDSLs. Haskell templates are based on functional approach and C++ templates have an imperative style of programming.

A. C++ Templates

Templates are a methodology used to achieve generic programming. Initially templates were used in data container implementations and algorithm implementations. But now C++ templates takes a stronger form. Latest C++ compilers contain an interpreter that is capable of interpreting a Turing -complete functional sublanguage of C++. For the same reason mix of two forms of programming strategies are used with C++ programs. Static code which is executed at compile time and dynamic code which is compiled and later executed at runtime. Static computations are encoded in the C++ type system, and writing static programs is usually referred to as *Template Meta Programming*.

Very briefly we will look at few C++ template concepts in order to carry out rest of the discussion. C++ supports two kinds of templates; 1. class templates 2. function templates. An example template definition is given in Listing 4.

Listing 4: Example class template definition

```
template<class T, class U>
class SomeClass {
    T getMember();
    T member;
    U member;
...
}
```

A template is instantiated when we actually use the template class. Usually the use is through a *typedef* statement.

```
typedef SomClass<int, long> IntLongClass;
```

A template can be *specialized* by providing a definition for a specialized implementation as in Listing 5.

Listing 5: Template specialization.

```
template <class U>
class SomeClass<int, U> {
...
}
```

In relation to DSL implementation C++ templates can be used in two different ways. 1. As a *type driven DSL* 2. As an *expression driven DSL*. As far as deep embedding implementations are concerned the relevant C++ template usage is expression driven DSL. Therefore in this section we will only discuss expression driven DSL. But in Section 3.2.4 (under shallow embedding) we will discuss type driven DSLs.

Expression driven EDSL implementations using C++ templates

Expression-level embedded DSL implementation using C++ is well explained in [3], [4], [12]. We briefly explain the strategy using our simple DSL definition.

In expression driven DSL implementation we create an AST using DSL language. The creation of AST is done through template definitions and avoids any form parsing. Then the created AST is interpreted using an evaluation function. In following we go through an example implementation;

First we define arithmetic operators as C++ structures as listed in 6.

Listing 6: Arithmatic operator definitions for DSL

```
struct Add {};
struct Mult {};
struct Sub {};
struct Num {
  Num(int n): value(n){}
  int value;
};
```

Then we use C++ structure *Node* to represent a node in AST. In our tiny DSL we only have binary operators. Therefore we only need to create an AST with 3 nodes maximum. The Node structure definition is in Listing 7.

Listing 7: AST Node definition

```
template <class OP, class Left, class Right>
struct Node {
  Node(const OP& oper, const Left& l, const Right& r):
    op(oper), left(l), right(r){}
  OP op;
  Left left;
  Right right;
};
```

AST representation of a simple DSL program would look as follows;

Node <Mult, Node<Add, Num, Num> Num>

More concretely a DSL expression would look like *Node<Add, Num, Num> exp(Add(), Num(1), Num(2))*. To carry out evaluation we create a template based *Eval* structure as in Listing 8;

Listing 8: Template structure representing evaluation

```
template <class T>
struct Eval {
};
```

To make the EDSL work we need to make sure the Add operation is defined for all the types of AST nodes. As an example Add should work with Nums as well as Add should work when there are nested Add nodes or Mult nodes. Therefore we overload all operators for all template specializations.(As in Listing 9).

Listing 9: Example template specialization for Add and Mult nodes

```
//E.g :- Node<Add, X, Y> would look like follows;
template <class L, class R>
struct Eval< Node<Add, L, R> > {
  static inline int evalAt(const Node<Add, L, R>& n) {
    return Eval<L>::evalAt(n.left) +
      Eval<R>::evalAt(n.right);
  }
};

//E.g :- Node<Mult, X, Y> would look like follows;
template <class L, class R>
struct Eval< Node<Mult, L, R> > {
  static inline int evalAt(const Node<Mult, L, R>& n) {
    return Eval<L>::evalAt(n.left) *
      Eval<R>::evalAt(n.right);
  }
};
...
```

A simple invocation of the program would look like follows;

...

Node<Add, Num, Num> exp(Add(), Num(1), Num(2));

int out = eval(exp);

std::cout « "out " « out « std::endl;

To govern the evaluation we have *eval* function. *eval* function will create an Eval structure with passed expression (*exp*) in above example and call the *evalAt* of root expression.

As shown above the DSL implementation did not use any form of parsing. The type parameters to C++ templates with specialization helped to implement the DSL without parsing.

B. Haskell Templates

Template Haskell is an extension of Haskell that support compile-time preprocessing of Haskell source programs. Using template haskell we can write haskell code that generates haskell code. Having said that, Template Haskell does not actually generate haskell code instead it creates ASTs.

The mechanism used to instruct Template Haskell to generate AST is via quotations. Template Haskell defines several types of *quotations* [3].

1. Expression quotations are used for generating regular Haskell expressions, and they have the syntax [|expression|].
2. Declaration quotations are used for generating top-level declarations for constants, types, functions, instances etc. and use the syntax [d|declaration|].
3. Type quotations are used for generating type values, such as [t|Int|]
4. Pattern quotations are used for generating patterns
5. The last type is the so called "quasi-quotation", which lets us build our own, custom quotations.

An example template Haskell porgram is given in Listing 10 (based on reference [3]).

Listing 10: Example Template Haskell Program

```
expand_power :: Int -> Q Exp -> Q Exp
expand_power n x =
if n==0
```

```
then [| 1 |]
else [| $x * $(expand_power (n-1) x ) |]
```

Above code generates an AST representation of the program. The quotation; [|e|] and splice; $ is somewhat similar to bracket and escape in MetaOcaml. The ASTs generated are contained in a monad called $QExp$. Template Haskell provides an algebraic data type for representing Haskell code as an AST. The DSL author can then implement a function which goes through the AST and generate appropriate target code.

3.1.4 *Dynamic Compilation using Intermediate Representation (IR)*

Interpreting generated AST is not the only methodology in implementing deep embedding of DSL. Some DSLs use a dynamic approach to generate an AST like structure. When DSL specific function is invoked during runtime they are not immediately executed; instead they go through a simulation phase. During the simulation phase, runtime gathers as much information about call paths and relevant data. Then the call path information is organized into a structure (mostly this structure is similar to AST). The created structure is then used to do certain optimizations and generate code.

This strategy is mostly used by programs that generates JIT code. Also sometimes this methodology is used in combination with template programming. ArBB DSL is based on this strategy. We will discuss more about this strategy when discussing ArBB.

3.2 Shallow Embedding Strategies

3.2.1 *Encoding DSL constructs as functions*

We already saw how this strategy is used to implement DSLs in Listing 3. In summary in this strategy semantics of each EDSL construct is interepretted as a function immediately.

3.2.2 *As an API*

Difference between an EDSL and an API is not evident in many cases. The API is distributed in a form of a library. The EDSL user links the library and program against the API. Object Oriented languages such as Java provides set of rich constructs (encapsulation, polymorphism etc.) to reduce *semantic noise* in API based EDSLs. One common question relevant to current discussion is whether library APIs are really EDSLs ? EDSLs attempts to restrict introducing new concepts to the language. In a similar fashion carefully designed APIs also restrict access and expose only necessary concepts.

Even though APIs are good at restricting semantic noise, APIs may not be good at restricting syntactic noise. The underlying host language constructs will be available for end users of the EDSL.

3.2.3 *Preprocessor Macros*

An easy approach to implement shallow embedded DSL is to use preprocessor macros. This strategy is mostly used together with API method. We demonstrate implementation of a DSL with our little example. We use C as host languages and we use C preprocessor macros to implement logic (Listing 11).

Listing 11: EDSL implementation for arithmatic DSL using preprocessor macros.

```
#include <stdio.h>

#define NUM(N) N
#define ADD(M, N) (M+N)
#define MULT(M, N) (M*N)
#define SUB(M, N) (M-N)

int main() {
    printf("%d\n", ADD(NUM(1), NUM(2)));
    printf("%d\n", MULT(NUM(3), NUM(2)));
}
```

In above example the constructs “ADD”, “MULT”, “SUB” does the actual evaluation; further program is not touching the AST at all.

3.2.4 *Using Templates*

In Section 3.1.3 we discussed about C++ templates. Also we discussed that there are 2 strategies to implement DSLs using templates. In Section 3.1.3 we discussed only one of those strategies. i.e. expression driven DSL implementation. The second strategy namely the type driven DSL implementation is more relevant to shallow embedding. In following we discuss type driven DSL implementation using C++ templates.

Type driven EDSL implementation using templates

Template metaprogramming gives a feeling of a functional programming language. Template programming provides 2 views to support the behaviour of a functional programming. Following example in reference [3] demonstrate this aspect;

In Haskell the List is defined as in Listing 12. The same list can be defined in a functional pattern in C++ as depicted in Listing 13.

Listing 12: List definition in Haskell

```
data List = Nil | Cons Int List
list = Cons 1 (Cons 2 Nil)
```

Listing 13: Functional definition of List in C++

```
struct Nil {};
template <int H, class T>
struct Cons {};
typedef Cons<1,Cons<2, Nil> > list;
```

At compile time, the structures *Nil* and *Cons* correspond to Haskell’s *Cons* and *Nil* implementations. At runtime Nil behave just like a class and Cons represent a template class. But during static invocation (i.e. at compile time) they represent data type constructors.

The template meta programming support *de-structuring* of data using pattern matching. The technique facilitates pattern matching is template *specialization*. This also gives us a way to encode Haskell like functions into C++ templates. As an example list length function in Haskell is given in Listing 14 and analogous C++ template implementation is given in Listing 15.

Listing 14: List length function in Haskell

```
len :: List $\longrightarrow$ int;
len Nil = 0
len (Cons h t) = 1 + (len t)
```

Listing 15: C++ template based list length function

```
template <class List>
struct Len;
template <>
struct Len<Nil> {
enum { RET = 0 }; };
template <int H,class T>
struct Len<Cons<H,T> > {
enum { RET = 1 + Len<T>::RET }; };
```

The empty and non-empty cases are handled through template specialization using Len<Nil> and Len<Cons<H,T> >. C++ template's ability to act as functional language at compile time is used to develop shallow embedded DSLs. In following we demonstrate how we use C++ template type driven expressions to implement our tiny DSL.

Listing 16: Use of type driven expressions to implement arithmatic DSL

```
template<int N>
struct Num {
  enum { value = N };
};

template<typename L, typename R>
struct Add {
  enum { value = L::value + R::value };
};

template<typename L, typename R>
struct Mult {
  enum { value = L::value * R::value };
};

template<typename L, typename R>
struct Sub {
  enum { value = L::value - R::value };
};

// Factorial<4>::value == 24
// Factorial<0>::value == 1
int main()
{
  std::cout << Add<Num<2>, Num<3> >::value << std::endl;
  std::cout << Mult< Add<Num<2>, Num<3> >, Num<2> >::
       value << std::endl;
}
```

As depicted in Listing 16, all Add, Mult, Sub data types are encoded as template data structures. The value is updated within each data structure. This implementation does not have a separate function which traverse through a AST like structure. Therefore above implementation can be categorised as shallow embedding.

4. LANGUAGE IMPLEMENTATIONS

In this section we will go through several EDSLs covering each embedding strategy. For deep embedding we will discuss about Accelerate [2] a functional DSL embedded in Haskell for GPU computations, Intel Array Building Blocks (ArBB) [6] , an EDSL for low level parallel operations based on templates and uses dynamic compilation with an Intermediate Representation (IR). For shallow embedding we will discuss Boost Graph Library(BGL) [5] which is a form of DSL and also an API. Feldspar [1] is combined embedded DSL, in Section 4.3 we will discuss Feldspar implementation briefly. For each language, we will briefly discuss about domain applied, how EDSL looks like and the embedding strategy used.

4.1 Accelerate

Domain : Accelerate is an array based EDSL for generating efficient code for GPUs. Primary objective of Accelerate is to ease the GPU programming for user while achieving maximum possible performance.

The DSL : Main data item in Accelerate is an array. We can define multidimensional arrays in Accelerate. In Accelerate an array definition takes following form;

$$< Array >< Shape >< ElementType >$$

The Shape basically says whether defining array is a 1 dimension array, 2 dimension array etc. Type synonyms are used to define commonly used dimensions. E.g :- “Array DIM0 Int” represents a scalar, “Array DIM1 Float” represent an array of floating point number etc.

Further Accelerate have type synonyms for vectors and scalars as follows;
type Array DIM0 e = Scalar e
type Array DIM1 e = Vector e

Accelerate deal with two types of arrays. Accelerate arrays which are used for computations in the GPU device and Haskell language arrays which can be used in the host machine. Accelerate distinguish these 2 array types using “Acc”. Following is an Accelerate program that computes the *dot product* of two vectors.

Listing 17: Dot product in Accelerate

```
dotp :: Vector Float -> Vector Float -> Acc (Scalar Float)
dotp xs ys = let xsn = use xs
                 ysn = use ys
             in
             fold (+) 0 (zipWith (*) xsn ysn)
```

The *Vector Float* represent a normal Haskell array and the output of dotp is a scalar value array in GPU device format. The *use* function converts Haskell array into a GPU compatible array. The *fold* and *zipWith* are from Accelerate library implementation.

The Embedding : Accelerate is a deeply embedded DSL in which it does not execute instructions immediately rather creates an internal representation of the DSL program. Accelerate uses a *Higher Order Abstract Syntax* (HOAS) graph as its intermediate representation. HOAS is specifically useful in implementing *sharing*. Sharing avoids evaluating the same term more than once. Infact [7]shows that HOAS based graphs are much suitable as EDSL representations. After converting to HOAS format Accelerate further convert HOAS representation to a nameless representation using *de Bruijn indices*.

As an example if we have program in 18, Accelerate will generate a HOAS graph as shown in Figure 3.

Listing 18: Example to demonstrate HOAS

```
let inc = (+) 1
in let nine = let three = inc 2
             in
             (*) three three
in
(-) (inc nine) nine
```

In Figure 3, “@” represent an application. HOAS become a graph as we connect shared bindings through edges. <*nine*> binding is used in 2 places, therefore more edges get connected to <*inc*>.

Accelerate does *sharing* and other types of optimizations (which are out of scope of this paper) on the generated graph and finally optimized graph is given to the evaluation function based on the selected backend. Accelerate is designed to support multiple backends, but the active backend that is described in their initial publication is CUDA backend.

4.2 Intel's Array Building Blocks (ArBB)

Domain : ArBB is a DSL that is designed for a similar purpose like Accelerate. While Accelerate more focused towards to GPUs, ArBB is focused on generating SIMD parallelized code and both implementations are based on array data types. Further ArBB DSL is based on C++ templates and it uses a runtime representation

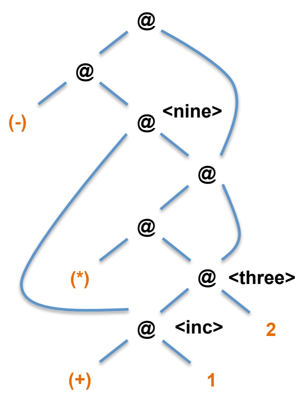


Figure 3: HOAS structure for example in Listing 18

to optimize program and JIT compile code. ArBB comes as a dynamically linkable library (.so OR .dll).

The DSL : ArBB provides set of constructs to perform array operations. Scalar types in ArBB are i32 (32 bit integer), f32 (32 bit floating point number) etc. ArBB also allows users to define multi-dimensional arrays. "dense<f32> a" defines a one dimensional array of 32 bit floating point numbers. Similarly "dense<f32, 2> b" defines a two dimensional array of 32 bit floating point numbers. In addition ArBB provides set of functions related to parallel operations (such as *reductions*). Further ArBB also provides *closures*.

ArBB provides an interesting concept called *Elemental Function*. An elemental function is written as a regular C/C++ function, and the algorithm within describes the operation on one element, using scalar syntax. The function can then be called as a regular C/C++ function to operate on single element or it can be called in a data parallel context, providing many elements to operate on. An example elemental function is given in 19.

Listing 19: An elemental function

```
void sum(dense<i32> a, dense<i32> b, dense<i32>& res) {
  res = a + b;
}
```

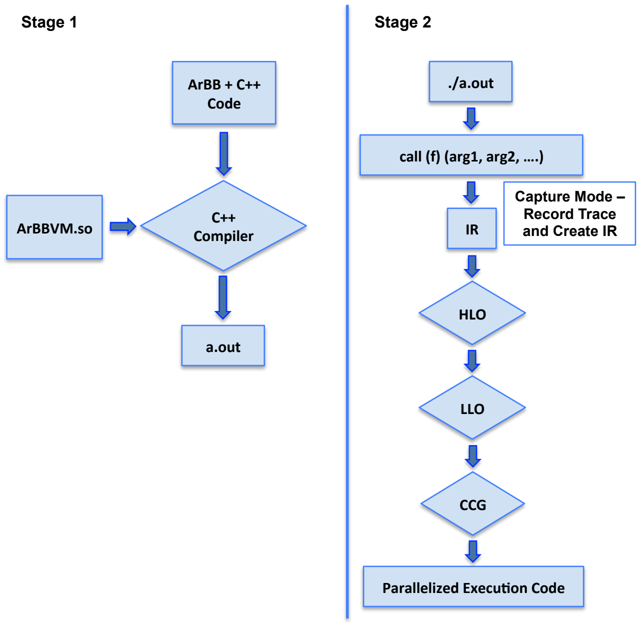
Though we see that "+" acts on scalars it actually work on dense arrays a and b. In addition *bind* and *call* are two important functions provided by ArBB. Example 20 shows how bind and call are used.

Listing 20: An example ArBB program; Referenced from [6]

```
float32_t scale[] = âĂę;
void mandelbrot1_call(int* res_arbb) {
 dense<f32> sR; bind(sR, scale, N_ROWS);
 dense<f32> sC; bind(sC, scale, N_COLS);
 dense<i32,2> dest;
 bind(dest, res_arbb, N_COLS, N_ROWS);
 call(mandelbrot1)(dest, sR, sC);
}
```

Listing 20 shows a portion of ArBB code from reference [6]. The bind operation maps a normal C++ array to a ArBB array type. "call" is a special function which we will discuss under embedding subsection.

The Embedding : The ArBB works in 2 stages. The first stage is fairly straight forward where we write array program by including ArBB include files and compile using C++ compiler and link ArBB library to create an executable. Then the second stage starts when we really execute the program. During execution program goes through several phases as shown in Figure 4.


Stage 1
ArBB + C++ Code
ArBBVM.so
C++ Compiler
a.out
Stage 2
./a.out
call (f) (arg1, arg2, ...)
IR
Capture Mode – Record Trace and Create IR
HLO
LLO
CCG
Parallelized Execution Code
Figure 4: ArBB execution stages.

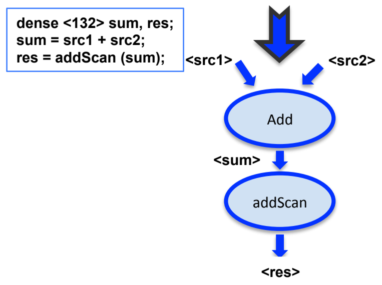


Figure 5: ArBB Intermediate Representation.

When program execution enters call function for the first time it goes to a so called *capture mode*. In capture mode instead of executing actual operations on ArBB types immediately, a trace of the invoked sequence of operations on ArBB types is recorded and creates an internal representation (IR). Then this internal representation goes through several optimization phases (HLO - High Level Optimizer, LLO - Low Level Optimizer) and finally generates parallelized target code (by CCG- Converged Code Generator).

IR is a representation of the execution code similar to an AST. But IR is created at runtime when call on a ArBB data type is first invoked. Caching is used to avoid creating IR and doing transformation to it for subsequent executions. Figure 5 shows the sample ArBB domain code and the created IR.

Another notable aspect is; ArBB uses templates for DSL specification but it does not use template metaprogramming for DSL implementation. Instead ArBB uses a IR generated at runtime. As per authors of ArBB the IR provides more provision to do optimizations compared to compile time transformations.

4.3 Feldspar

Domain : Feldspar is a domain specific language for programming digital signal processing (DSP) algorithms. Feldspar is a special embeded DSL due to its embedding strategy; i.e.it uses combined embedding strategy. Further Feldspar DSL tries to resemble Haskell host language constructs and it generates C code as the target.

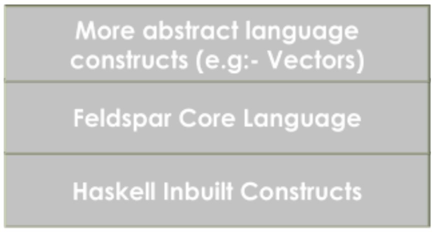

More abstract language constructs (e.g:- Vectors)
Feldspar Core Language
Haskell Inbuilt Constructs

Figure 6: Layering of Feldspar EDSL.

The DSL : The example in Listing 21 is a Feldspar program that closely resembles the corresponding Haskell function. It computes the bitwise and of a mask with each integer in the range 0 to n.

Listing 21: Example Feldspar program that computes bitwise and of a mask

```
mask : : Data Int ->  Data Int ->   DVector Int
mask m n = map (m .&.) ( 0 . . .n)
```

Feldspar DSL consists of 2 parts. The first part, so called *core* language provides basic constructs for DSL algorithm writing and the core language is almost similar to Data library in Haskell in appearance, but with semantics of DSP. Listing 22 show part of Feldspar core language definitions.

Listing 22: Feldspar core language definition; only selected constructors

```
data Expr a where
Value       :: Storable a => a -> Expr a
Function    :: String -> (a -> b) -> Expr (a -> b)
Application :: Expr (a -> b) -> Data a -> Expr b
...
```

Feldspar defines its own set of data types to distinguish DSP types from Haskell languages data types. All the types that start with *Data* are DSP compatible Feldspar types (e.g:- Data a). All DSP operations are performed on Feldspar data types. To enforce this requirement Feldspar uses Haskell type classes. Storable is a type class interface that each Feldspar type needs to adhere to. For example if user is working with a value of a type "Data a" then a must be a Storable type. The value function can be used to convert a Haskell type to a Feldspar type.

Many of the DSP algorithms require collections. Therefore Feldspar build a collection library on top of core language. The implementation of vector is given in reference [1]. We can view Feldspar as a layered system as in Figure 6.

The Embedding : The core language is implemented as deeply embedded DSL with data type definitions. A core language program is mapped to appropriate GADT definitions and an AST is created. Then the core language AST is optimized for DSP execution and interpreted by a function like "eval" (As in Section 2.1.1).

Additional abstract constructs such as vectors are implemented as a shallow embedded DSL; i.e. abstract constructs are defined as functions and function implementations are encoded without creating an intermediate representation. The shallow embedding function definitions use underlying core language; e.gâĂİ- semantics for vector (or for other abstract constructs) operations are defined using core language constructs. Therefore deep embedding must be in place before implementing the shallow embedding features.

4.4 Boost Graph Library (BGL)

Domain : BGL is a shallow embedded DSL for graph processing. More commonly BGL is viewed as a framework or a library with an API. When DSLs are implemented as libraries with an API we usually don't think them as DSLs, but BGL provides all the necessary abstractions, rules, restrictions and operations pertaining to the domain.

The DSL : BGL provides constructs to create different types of graphs; directed, undirected etc. and functions to manipulate those graphs. Further BGL provides set of predefined graph algorithms. BGL is also a generic interface that allows access to a graph's structure, but hides the details of the implementation. BGL heavily uses C++ templates and metaprogramming.

BGL provides 2 types of graph implementations; adjacency list implementation and compressed sparse row (csr) implementation. We can create an adjacency list graph as shown in Listing 23;

Listing 23: BGL example; Creating an adjacency list graph

```
// create a typedef for the Graph type
typedef adjacency_list<vecS, vecS, bidirectionalS> Graph;
...
const int num_vertices = N;
...
// declare a graph object
Graph g(num_vertices);
```

The template argument vecS instructs BGL to use vector data type to store vertices and edges of the graph definition "Graph". After creating graph "g" we can perform operations such adding edges, traversing edges, executing algorithms etc.

Further BGL provides user friendly language constructs to traverse the graph based on preprocessor macros.

```
E.g:-
vertices_size_type visited = 0;
BGL_FORALL_VERTICES(v, boost_graph, Digraph) {
  if(boost::get(distance, v) < std::numeric_limits<weight_type>::max())
    ++visited;
}
```

In above code the BGL_FORALL_VERTICES is a preprocessor macro. Semantically it says that we need to traverse through all the vertices in the directed graph "boost_graph".

The Embedding : BGL is a shallow embedded DSL. Interestingly all BGL implementations comes in C++ header files. Therefore the user does not need to link any library for DSL to work, but needs to include appropriate header files. All the DSL syntax terms are either C++ classes or structures implemented using templates. Therefore the conversion from DSL to host language extremely direct.

In addition Boost enforce certain restrictions using *concept classes*. These concept classes ensures we need necessary abstract features in a particular graph implementation, iterator implementation or an algorithm implementation.

BGL also uses preprocessor macros to abstract certain complex code fragments and provides user an easy interface. As an example we can infer the functionality of BGL_FORALL_VERTICES macro intuitively.

5. WHICH EMBEDDING ?

In this section we will discuss tradeoffs and pros and cons between different embedding strategies.

We discussed about 3 embedding strategies; namely deep, shallow and combined. Many of performance oriented DSLs used deep embedding. Those DSLs do lot of optimizations on the generated intermediate representation or AST like structure. On the other hand many of the DSLs that are frequently changing uses shallow embedding strategies. Combined embedding is useful when the implementing DSL can be layered. That is when we have a core DSL language that does most of the hard work and then we implement additional functionalities with the help of core DSL constructs.

Shallow embedded DSLs are easier to implement than deeply embedded languages. Also maintenance wise shallow embedded DSLs are easier to maintain than deeply embedded DSLs. For example it would be easy to add Div (division) operation to our tiny DSL. We only need to add a Haskell function which calls Haskell division. For deep embedding when we add language features the generated AST changes which implies we also need to change the interpretation function. This is cumbersome when the generated AST is quite large and if we are doing lot of optimizations on the generated AST (we have to change all the places where AST undergoes transformations).

Shallow embedded DSLs are usually concise and elegant as they directly work with the semantics. But a downside of shallow embedded dsls is they are hard to debug due to single interpretation. On the other hand in deeply embedded DSLs we can view the AST or intermediate representation to debug. Another downside in deeply embedded DSLs in Haskell is the difficulty of using *sharing* (sharing avoids duplicate computations) and *recursion*. In most situations deeply embedded dsls manually implement sharing and recovery optimizations.

Combined embedding is *not* well suited when shallow language extensions are not efficiently expressible in the underlying deep embedded implementation (core language). On such situation we need to extend the underlying core (deep embedding) implementation. Therefore combined embedding will be useful when core implementation reaches some level maturity and when adding new features does not require to change the core implementation.

6. SUMMARY

Language embedding allows us to reuse language features such as parsing, optimizations from the host language and its a rapid development strategy to build DSLs. In this paper we discussed about 3 main strategies to embed a DSL within a general purpose programming language. In shallow embedding DSL constructs are directly mapped to appropriate semantics and in deep embedding DSL, we construct an AST or an internal representation which is similar to AST. Combined embedding is a strategy that combines shallow and deep embedding.

There are several techniques to implement each type of embedding. Multi Stage Programming, Templates and Abstract Data Type definitions are few of mostly used techniques for implementing deeply embedded DSLs. API's, Macros and Templates are widely used mechanisms for shallow embedding.

Embedding is not always the best approach to develop DSLs. For example embedding is not the best approach when there is a mismatch in the concrete syntax. A prerequisite for embedding is that the syntax of the new language must be a subset of the syntax of host language. In addition the semantics of the DSL and the host languages coincide for the common subset. As an example if the host language is a call-by-value and if we are trying to implement a call-by-name DSL then we will face unnecessary complications. Therefore when implementing an embedded DSL we need to carefully analyze and decide whether embedded DSL required for the problem.

7. REFERENCES

[1] E. Axelsson, K. Claessen, G. Dévai, Z. Horváth, K. Keijzer, B. Lyckegård, A. Persson, M. Sheeran, J. Svenningsson, and A. Vajda. Feldspar: A domain specific language for digital signal processing algorithms. In *Formal Methods and Models for Codesign (MEMOCODE), 2010 8th IEEE/ACM International Conference on*, pages 169–178. IEEE, 2010.

[2] M. M. Chakravarty, G. Keller, S. Lee, T. L. McDonell, and V. Grover. Accelerating haskell array codes with multicore gpus. In *Proceedings of the sixth workshop on Declarative aspects of multicore programming*, pages 3–14. ACM, 2011.

[3] K. Czarnecki, J. T. OâĂŹDonnell, J. Striegnitz, and W. Taha. Dsl implementation in metaocaml, template haskell, and c++. In *Domain-Specific Program Generation*, pages 51–72. Springer, 2004.

[4] S. Haney and J. Crotinger. Pete: The portable expression template engine. *Dr. Dobb's journal*, 24(10), 1999.

[5] L.-Q. Lee and A. Lumsdaine. *The Boost graph library: user guide and reference manual*. Addison-Wesley Professional, 2002.

[6] C. J. Newburn, B. So, Z. Liu, M. McCool, A. Ghuloum, S. D. Toit, Z. G. Wang, Z. H. Du, Y. Chen, G. Wu, et al. Intel's array building blocks: A retargetable, dynamic compiler and embedded language. In *Code generation and optimization (CGO), 2011 9th annual IEEE/ACM international symposium on*, pages 224–235. IEEE, 2011.

[7] B. C. d. S. Oliveira and A. Löh. Abstract syntax graphs for domain specific languages. In *Proceedings of the ACM SIGPLAN 2013 workshop on Partial evaluation and program manipulation*, pages 87–96. ACM, 2013.

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

[9] W. Taha. A gentle introduction to multi-stage programming. In *Domain-Specific Program Generation*, pages 30–50. Springer, 2004.

[10] A. T. Tan, J. Falcou, D. Etiemble, H. Kaiser, et al. Automatic task-based code generation for high performance domain specific embedded language. In *HLPP 2014*, 2014.

[11] L. Tratt. Domain specific language implementation via compile-time meta-programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(6):31, 2008.

[12] T. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, 1995.

[13] Wikipedia. Domain-specific language — wikipedia, the free encyclopedia, 2014. [Online; accessed 3-December-2014].