

# Language Support for Domain Specific Languages

Krzysztof Czarnecki<sup>1</sup>, John O'Donnell<sup>2</sup>, Jörg Striegnitz<sup>3</sup>, and Walid Taha<sup>4</sup>

<sup>1</sup> University of Waterloo, Canada

<sup>2</sup> University of Glasgow, United Kingdom

<sup>3</sup> Forschungszentrum Jülich, Germany

<sup>4</sup> Rice University, USA

**Abstract.** An attractive way to implement domain specific languages (DSLs) is by writing a library in a host language. Some DSLs, however, do not fit perfectly within the host language and a pure library solution is insufficient. In many cases, metaprogramming offers a solution to the problem. This paper surveys and compares the metaprogramming support offered by three languages—Template Haskell, C++, and MetaOCaml—and discusses the techniques they make available to the implementer of a DSL.

## 1 Introduction

Recently there has been increasing interest in the design and implementation of domain specific languages (DSLs), which need relatively lightweight implementations that can be produced with limited resources. Several general purpose languages have been enhanced with improved capabilities for implementing DSLs. The purpose of this paper is to compare the approaches taken by three of them: C++ templates, MetaOCaml, and Template Haskell.

Suppose that we have an existing host language  $L^H$ , and wish to use it to implement a target language  $L^T$ . A range of approaches is possible.

At one extreme—sometimes called the “closed language approach”—there is no relationship between the two languages, and  $L^T$  is implemented using traditional interpreter or compiler technology. The compiler happens to be written in  $L^H$ , but it could just as easily be written in any other language suitable for writing compilers.

At the other extreme—sometimes called the “open language approach”—we have  $L^T \subset L^H$ , so that any syntax valid in  $L^T$  is also valid in  $L^H$ . Furthermore, the target language inherits the semantics of the host, so that any program  $p \in L^T$  has the same meaning in  $L^T$  as it has in  $L^H$ . As a result, target programs can be compiled using an existing compiler for  $L^H$ , and we can say that  $L^T$  is *perfectly embedded* within  $L^H$ .

A perfect embedding typically consists of a library along with an intended style of usage. A simple example would be a C++ class library for image processing: it provides operations that are not directly present in C++ itself, but

the user is still writing what is obviously C++ code. A more involved example is the perfect embedding used Hydra [13], a DSL for designing digital circuits. Versions of Hydra from the early 1990s used library definitions for combinators, functions, type classes, types, instances, operator syntax, and operator fixity declarations—as well as a style of usage quite different from ordinary Haskell programming. It was still the case, however, that a Hydra specification was a valid Haskell program. The essential point is that for a perfect embedding, a program  $p \in L^T$  does not need to be preprocessed in any way; it can simply be compiled using the  $L^H$  compiler.

The designer of a DSL needs to consider both the language design (from the user’s perspective) and the cost of the implementation. An embedding imposes constraints on the language design of the DSL. Sometimes these constraints are helpful, as they avoid unnecessary proliferation of notations, but they may also be harmful if the host language syntax differs too much from the notations used in the application domain.

Perfect embedding with a library is an attractive technique for implementing DSLs because it reuses much of the design of  $L^H$  as well as its compiler. The DSL implementer just needs to provide the library, rather than a complete compiler. Sometimes, however, a situation arises which is between the extremes of full-scale compiler construction and perfect embedding. This occurs when there is much commonality between  $L^T$  and  $L^H$ , but there is also a slight mismatch. There are many causes of such mismatches, including the following:

- *Syntactic sugar.* Domain specific language notations may be in  $L^T$  but not  $L^H$ .
- *Modified semantics.* The target language may have a more stringent type system, or some constructs may have different meanings.
- *Debugging support.* A debugging tool for a programming language may be viewed as a domain-specific language which can do everything in the host language, and also has access to information available to the compiler (such as variable names and line numbers where constructs appear) and the runtime system (such as the state of the stack or heap).
- *Generating additional code.* Many applications require additional definitions to be included in a program. These could be expressed in the host language, but the application programmer should not be required to write them by hand.

In all of these cases of mismatch, a perfect embedding is impossible. This means that the implementation of the DSL requires something more than just a library. There are also situations where it would be preferable for an  $L^T$  compiler to behave differently than a  $L^H$  compiler would:

- *Restriction to the target.* Even in a perfect embedding, it is useful to report an error message if the domain-specific program steps outside the features of  $L^T$ , and uses something from  $L^H$ .
- *Meaningful error messages.* Error messages that refer to the problem domain will be more helpful to the user. Furthermore, an error message that relates

to parts of the host language that do not lie within the DSL can be quite confusing.

- *Domain specific optimizations.* Domain-specific embedded languages may exploit their richer knowledge of the behavior of the program in order to perform optimizations that are not open to the host compiler.

There are many kinds of tools that can be used to overcome the mismatches, making imperfect embeddings possible. The complexity of the tools required depends largely on the degree of mismatch between  $L^T$  and  $L^H$ . Sometimes a simple syntactic transformation is sufficient, and this may be achieved by the C preprocessor, or with a scripting language like Perl. A more sophisticated approach is to provide metaprogramming built in to the host programming language, giving the programmer the ability to adapt the behavior of the host's compiler in order to address the problems outlined above. Other related tools are macros and partial evaluation.

If a host language provides no support at all for metaprogramming, then the only way it can be used to implement a DSL is with one of the two extreme approaches: either a perfect embedding, or a full compiler with no embedding at all. The great benefit of metaprogramming is that it makes intermediate solutions possible. Some languages (e.g. C++ with templates and Template Haskell) support metaprogramming which is restricted to compile time. Other systems (e.g. MetaML) provide support for metaprogramming at runtime, allowing DSL implementations using multistaged approaches.

If a perfect embedding is not possible, the DSL implementer needs to provide either a full interpreter or compiler, or some form of preprocessor for an existing language. These approaches are often more difficult to implement than a library, and they may also be more difficult for the user to install. A significant advantage of adding metaprogramming support to the host language is that it allows many mismatches to be overcome in a perfect embedding. For example, the Hydra language mentioned above can be embedded imperfectly in Haskell, but there is a large mismatch caused by the need for automatically generated definitions. The mismatch is solved by an automatic program transformation performed by the metaprogramming tools of Template Haskell: thus Hydra can be embedded almost perfectly in Template Haskell, with just a trivial syntax preprocessing.

Since there is a wide range of techniques which have been used for embedding, and they operate in disparate programming languages, it may be impossible to give precise technical definitions for all the approaches. The remainder of this paper offers a perspective by discussing and comparing the metaprogramming support offered by three host languages that have been used successfully for implementing DSLs: Template Haskell, C++ templates, and MetaML.

## 2 Template Haskell

Haskell [4] is a pure, nonstrict functional language. A variety of experimental extensions to Haskell have been implemented, many of which require additional syntax, semantics, or compile-time program transformations.

Template Haskell [14] is such an extension to Haskell, whose purpose is to enable any programmer to define new language features by writing algorithms that generate new Haskell code to be executed. This avoids the need to modify the compilers each time a new feature is added to the language, and it directly supports the use of Haskell for defining domain specific languages. Some applications using Template Haskell are described in [11], [12], [7].

Type safety in Template Haskell is ensured by staged typing. All metacode is typechecked before it is executed; any code which it generates is then typechecked before it is spliced into the final program. The final object code is guaranteed not to fail due to type errors. Template Haskell is quite flexible: for example, a metaprogram can read data from a disk file and use that while generating code.

The main components of Template Haskell are an algebraic datatype for representing Haskell programs; a monad for constructing code while maintaining an evolving state; mechanisms for gaining access to the concrete syntax tree for code and splicing concrete syntax back into the program; and a high level quasiquote notation. Each of these components will be described briefly; for full details the reader is referred to the references cited earlier.

Template Haskell is a *homogeneous two-level language*, which means that terms in a Template Haskell program appear in one of two levels: runtime terms define computations to be performed when the final program is executed, while compile time terms are executed during compilation, and are typically used to generate runtime terms. Template Haskell provides syntactic constructs for controlling which level a term belongs to. Splicing evaluates a compile time term and inserts the result into a runtime expression. Reification takes a runtime term, and makes its abstract syntax tree available to a compile time term. However, there is still a distinct compile time, during which all meta-computations occur and code is generated. The compiler produces object code, which contains no further metacomputation.

## 2.1 Representation of programs

The centerpiece of Template Haskell is an algebraic datatype that represents Haskell programs, and which can be processed by metaprograms. The representation is an abstract syntax; superficial aspects of the appearance of a program, such as parentheses, spacing, and comments, are omitted.

```
data Lit = ... | Integer Integer | ...
data Exp = ... | Var String | Lit Lit
          | App Exp Exp | ... | Tup [Exp] | ...
data Pat = PLit Lit | Pvar String | Ptup [Pat] | ...
data Dec = Fun String [Clause] | Val Pat RightHandSide [Dec] | ...
data Clause = Clause [Pat] RightHandSide [Dec]
data RightHandSide = Guarded [(Exp,Exp)] | Normal Exp
```

To illustrate the usage of these data types, consider the following function definition. The function `circ` is both a valid Haskell function definition, and a circuit specification in the Hydra DSL:

```

circ x y = (a,b)
  where a = inv x
        b = xor2 a y

```

The abstract syntax tree representation of `circ` is given below. The definition consists of one clause corresponding to the single defining equation (if `circ` were defined with several equations selected by pattern matching, there would be several corresponding clauses). Within the clause, there are three pieces of information: the list of patterns, the expression to be returned, and a list of local definitions for `a` and `b`.

```

Fun "circ"
  [Clause
    [Pvar "x", Pvar "y"]
    (Normal (Tup [Var "a", Var "b"]))
    [Val (Pvar "a") (Normal (App (Var "inv") (Var "x"))) [],
      Val (Pvar "b") (Normal (App (App (Var "xor2")
                                     (Var "a"))) (Var "y"))) [] ]]

```

Once a piece of Haskell code has been represented as a tree, it can be analyzed or transformed using standard elementary programming techniques.

## 2.2 The quotation monad

Sometimes it is more convenient to write monadic computations that will produce a piece of abstract syntax, and to combine these with monadic operators, rather than doing everything directly by ordinary recursive tree traversals. The main reason for this is that when a metaprogram generates code, it is in general necessary to create fresh names in order to avoid accidental name capture. This requires the ability to perform “gensym” operations, which in turn requires a state. A second reason for putting the metaprogram into monadic form is that monadic Input/Output operations may be performed. Template Haskell provides a full set of monadic operations for constructing code. For example, instead of

```

exp1 = App (Var "f") (Var "x")
exp2 = App exp1 ...

```

we can write

```

mkexp :: Q Exp
mkexp =
  do exp1 <- app (var "f") (var "x")
     exp2 <- app exp1 ...

```

If the construction of the code is straightforward, there is no particular advantage to the monadic form. Consider, however, the case where fresh variables must be generated. The `Q` monad provides a computation `gensym :: String -> Q Exp` which constructs a new variable, whose name begins with the string argument, but which is guaranteed not already to be in scope. For example, suppose we need to generate code that makes a local definition for a variable `x`, but we want to ensure that this does not clash with any other `x` that might already exist. This is achieved by

```

mkexp :: Q Dec
mkexp =
  do v <- gensym "x"
     val (pat v) ...

```

### 2.3 Reification and splicing

The special syntax  $\$(exp)$ , called a splice, indicates that the  $exp$  is a monadic computation of type  $Q\ Exp$ . While the program is being compiled, Template Haskell will handle the syntax by performing the following steps: (1) it checks the type of  $exp$ , to ensure that it really does have the required type  $Q\ Exp$ ; (2) it executes the code, resulting in a value of type  $Exp$  giving the abstract syntax of a Haskell expression; (3) it inserts this code in place of the entire  $\$(exp)$  expression. The compilation then continues normally; in particular, the code produced by the splice is typechecked along with the hand-written Haskell code.

There are two ways for a metaprogram to gain access to the representation of runtime terms. The reification operator may be applied to an expression, and will return its code representation. Alternatively, the source code may be surrounded by quotation brackets  $[ \dots ]$ ; the value of this syntactic construct is the abstract syntax tree for the quoted term.

There are two opportunities for a type error to be detected. The first is when the monadic operation to produce code is typechecked; if an error is detected here, then no code can be generated, and the compilation is aborted with an error message. Later on, the final code generated by the splice is also typechecked. It is interesting to note, however, that the metaprogram in the splice can perform arbitrarily complex computations; for example it could generate ill-typed code and later throw it away. The only constraint is that the final generated code has to be type-safe, not the intermediate values defined in the course of the program generation.

The quotation monad provides computations that permit Input/Output operations to be performed. This opens a wide variety of possibilities. For example, a program generator could generate C code that implements the semantics of a function more efficiently than the usual graph reduction could [11].

### 2.4 High level syntax: quasi-quote

Previous sections have shown how program representations can be built using constructors or monadic operations. In many cases, quasi-quote provides an easier alternative: the program can contain ordinary Haskell code enclosed within reification brackets, and the entire notation produces a monadic computation which constructs the representation of the code. For example, the notation  $[|x+1|]$  has type  $Q\ Exp$ , and it denotes a monadic computation that, when executed, will construct the representation of the expression  $x+1$ . This value could be spliced immediately, but typically it would be used to help build up a larger monadic operation.

## 2.5 Example: optimizing vector functions

Metaprogramming allows the construction of libraries that examine code and optimize it. Consider, for example, a library for computing with vectors that contains element-wise addition and multiplication functions:

```
add, mul :: [Float] -> [Float] -> [Float]
add xs ys = map2 (+) xs ys
mul xs ys = map2 (*) xs ys
```

Suppose these functions are used strictly. Each application of `add` or `mul` requires a separate recursion, so a function that uses several vector operations will require several recursions, each with its own iteration overhead:

```
f v1 v2 v3 = v1 'add' (v2 'mul' v3)
```

To improve performance, we might want to use metaprogramming to collapse the multiple recursions into one. To do this, we first make the code representation of the user's definitions visible. This can be done by defining `user_defs` to be the tree representing the definitions in the user's module:

```
user_defs = [|
  f v1 v2 v3 = v1 'add' (v2 'mul' v3)
|]
```

The code between the `[|` and `|]` brackets is the user's DSL program. We also define a function `optimize` which takes the representation of user code, searches it for opportunities to combine recursions, and replaces it with the optimized version. For example, the definition of `f` would be optimized to:

```
f v1 v2 v3 = map3 h v1 v2 v3
  where h x y z = x+(y*z)
```

The optimized code can be used in a DSL module by splicing in the transformed code:

```
$(optimize user_defs)
... (f p q r) ...
```

What we've just presented is an example of a simple *compile-time* embedded DSL: the Template Haskell compiler will compile the host program together with the embedded DSL program and produce object code, which contains no further metacomputation.

## 3 C++

C++ supports implementing two kinds of compile-time embedded DSLs: type-based DSLs and expression-based DSLs. Both variants are implemented by means of template metaprogramming and allow for domain-specific code generation, optimization, and error checking.

### 3.1 Template Metaprogramming (TMP)

Originally, templates were intended to support the development of generic containers and algorithms. Driven by the practical needs of library implementers, the C++ template mechanism became very elaborate. By chance rather than by design, it now constitutes a Turing-complete, functional sublanguage of C++ that is interpreted by the C++ compiler [21]. This makes C++ a two-level language (just like Template Haskell): a C++ program may contain both level- $-1$  code, which is evaluated at compile time, and level-0 code, which is compiled and later executed at runtime. However, unlike Template Haskell, C++ is a heterogeneous two-level language because the languages at each level are different: level- $(-1)$  is functional and level-0 is object-oriented.

Template metaprogramming uses templates in three ways: Level-0 code uses templates as generic components. Level- $-1$  code adds two rather unusual views of class templates: class templates as data constructors of an algebraic datatype and class templates as functions. The C++ template instantiation mechanism provides the execution semantics for level- $-1$  code. Here is an example of C++ level- $-1$  code defining a list and the corresponding Haskell code for comparison:

<b>C++:</b>	<b>Haskell:</b>
<pre>struct Nil {}; template &lt;int H, class T&gt; struct Cons {};</pre>	<pre>data List = Nil             Cons Int List</pre>
<pre>typedef Cons&lt;1,Cons&lt;2, Nil&gt; &gt; list;</pre>	<pre>list = Cons 1 (Cons 2 Nil)</pre>

The C++ templates `Nil` and `Cons` correspond to data constructors in Haskell. We can view templates as data constructors of a single imaginary algebraic datatype `ANYTYPE`, which they implicitly extend. Haskell allows us to define different algebraic datatypes, of course, and the `data` declaration makes them explicit.

Next, we compare how functions are implemented at the C++ level- $-1$  and in Haskell:

<b>C++:</b>	<b>Haskell:</b>
<pre>template &lt;class List&gt; struct Len; template &lt;&gt; struct Len&lt;Nil&gt; {     enum { RET = 0 };};</pre>	<pre>len :: List -&gt; int; len Nil      = 0</pre>
<pre>template &lt;int H, class T&gt; struct Len&lt;Cons&lt;H,T&gt; &gt; {     enum { RET = 1 + Len&lt;T&gt;::RET };};</pre>	<pre>len (Cons h t) = 1 + (len t)</pre>

Class template specialization serves as a vehicle to support pattern matching, as common in functional programming languages. In contrast to Haskell or ML, the relative order of specializations in the program text is insignificant. The compiler selects the best match. Instantiating a class template corresponds to

function application and yields a class. In template metaprogramming, static members of the resulting class are used and accessed using the `::` syntax. For example, `Len<list>::RET` will evaluate to 2. The member being accessed could also be a computed type, e.g., `Tail<list>::RET`, or a static member function. Both are useful for generating code.

### 3.2 Type-based DSLs

Templates that produce classes having some interesting function and data members can be viewed as generators. Template metaprogramming enables generators that can perform complex computations on parameters and assemble generic components [1]. Type-based DSLs comprise the type expressions and integer values passed to such generators. A generator for different implementations of matrix types can be used as follows:

```
typedef MatrixGen<ElemType<float>,Optim<space>,Shape<u_triang> >::RET M1;
typedef MatrixGen<Shape<l_triang>,Optim<speed> >::RET M2;
```

`MatrixGen` takes a list of properties, checks their validity, completes the specification by computing defaults for unspecified properties, and computes the appropriate composition of elementary generic components (e.g., containers, shape adapters, bounds checkers, etc.). This generator uses a specific style for specifying properties that simulates keyword parameters: we wrap a parameter in a template to indicate its name, e.g., `ElemType<float>` [2]. Generators can compose mixins (i.e., template classes derived from their class parameters) and thus generate whole class hierarchies.

### 3.3 Expression-based DSLs

A simple way to embed a DSL into a programming language is to define a set of functions and to treat nested function calls as statements of the DSL. Obviously, this approach is limited: not all possible nestings may be correct statements of the DSL, but the compiler does not have the information to filter them out. Moreover, the compiler is lacking domain specific knowledge that might be essential to perform domain specific optimization. C++ templates can be used to overcome this problem by allowing us to generate parse trees of C++ expressions and manipulate them at compile-time. The technique used is called expression templates [20], [8].

As an example, consider a simple DSL of vector expressions such as `(a+b)*c`. This DSL could easily be embedded by overloading the `+` and `*` operators to perform the desired computations. However, assume that our DSL should have parallel execution semantics, e.g., using OpenMP. In order to achieve this, we would like to generate the following code:

```
#pragma OMP parallel for
for (int i=0 ; i<= vectorSize; ++i)
    // traverse the expression tree and compute value at index i
    evalAt( tree, i );
```

This cannot be done just with simple operator overloading (since each operator would contribute an additional loop). However, expression templates and template metaprograms can be used to generate the desired code. The gist of expression templates is to overload operators to return an object that reflects the algebraic structure of the expression in its type. This has basically the effect of reifying the expression as a parse tree at compile time. The leaves of the parse tree will be vectors and we'll use `BNode` to represent non-leaf nodes:

```
struct OpPlus {};
struct OpMult {};

template <class OP,class Left,class Right> struct BNode
{ BNode(const Op& o,const Left& l,const Right& r);
  OP    Op;
  Left  left;
  Right right;
};

template <class T, int size> class Vector
{public:
  T& operator[] (int i);
  //...
  T data[size];
};
```

For example, the type of the expression `(a+b)*c`, where all variables are of type `Vector<int, 10>`, would be:

```
BNode< OpMult,
      BNode< OpPlus,
            Vector<int, 10>,
            Vector<int, 10> >,
      Vector<int, 10> >
```

To create such an object, we properly overload all the operators used in our DSL for each combination of operand types (`Vector/Vector`, `Vector/BNode`, `BNode/Vector`, and `BNode/BNode`):

```
//implementation for Vector+Vector; others analogous
template <class V,int i>
BNode<OpPlus,Vector<V,i>,Vector<V,i> >
operator+(const Vector<V,i>& l,const Vector<V,i>& j) {
  return BNode<OpPlus(),Vector<V,i>,Vector<V,i> >(o, l, r );
}
```

Next, we define `Eval`, which is a metaprogram that takes a parse tree and generates the desired code by function inlining while traversing the tree. While visiting a leaf, we generate a function that evaluates a vector at a certain position:

```

template <class T,int size> struct Eval< Vector<T,size> >
{ static inline T evalAt(const Vector<T,size>& v, int i)
  { return v[i]; }
};

```

For a binary node, we first compute the values of the siblings and then combine the results using the desired operation:

```

template <class L,class R> struct Eval< BNode<OpPlus,L,R> >
{ static inline T evalAt(const BNode<OpPlus,L,R>& b,int i)
  { return Eval<L>::evalAt(b.left,i) + Eval<R>::evalAt(b.right,i);}
};

```

The following function template allows us to invoke `Eval` conveniently without having to pass the expression explicitly (template parameters of function templates can be inferred automatically):

```

template <class T>
T inline eval(const T& expression,int i) {
  return Eval<T>::evalAt(expression, i);
}

```

When we call `eval((a+b)*c)`, the compiler instantiates the function template, which will automatically bind `T` to the type of `(a+b)*c`. The compiler will then expand `Eval<T>::evalAt(e,i)` recursively into efficient code to evaluate `expression` at index `i`.

The final step in integrating the DSL is to trigger evaluation of a program – actually to call the `eval` function. Evaluation should happen automatically if the user assigns a DSL expression to a `Vector`. We thus add an overloaded assignment operator to the `Vector` class:

```

template <class A,class Expression,int size>
Vector<A,size> Vector<A,size>::operator=(const Expression& rhs)
{ #pragama OMP parallel for
  for (int i=0 ; i<size ; ++i)
    { data[i] = eval( rhs, i ); }
};

```

## 4 MetaOCaml

MetaOCaml is a statically typed homogeneous multi-level language based on OCaml. It extends OCaml with three basic constructs called Brackets, Escape, and Run for building, combining, and executing host-language programs at runtime, respectively. MetaML is a similar extension of SML/NJ interpreter.

Brackets and Escape roughly correspond to quasi-quotes and splice in Template Haskell, respectively. Run corresponds to `eval` in Lisp. Brackets increment the level of the enclosed code and Escape decrements the level. In contrast to Template Haskell, MetaOCaml allows brackets to be nested directly. Starting at

level 0, a MetaOCaml program may additionally contain code at levels 1 through  $n$ , where  $n$  is some positive number. This is why MetaOCaml is referred to as multi-level. Template Haskell and C++ programs also start with level 0, but may contain level -1 code (to be executed at compile time). There is no level -1 code in MetaOCaml programs.

MetaOCaml can be used to build *runtime* embedded DSLs, where the embedded DSL program is composed and compiled at the runtime of the host program, or closed DSLs. The basic approach involves first building an interpreter for the DSL, staging this interpreter, and then composing the result of this interpreter with a call to the host language compiler. Implementations derived in this manner can be as simple as an interpretive implementation, and at the same time, have the performance of a compiled implementation.

#### 4.1 QBF Interpreter

Let us consider a simplistic DSL motivated by the logic of quantified boolean formulae (QBF). The syntax of such a language can be represented in OCaml as follows:

```

type bexp = True           (* T *)
          | False          (* F *)
          | And of bexp * bexp (* T ^ F *)
          | Or  of bexp * bexp (* T v T *)
          | Not of bexp      (* not T *)
          | Implies of bexp * bexp (* F => T *)
          | Forall of string * bexp (* forall x. x and not x*)
          | Var of string      (* x *)

```

Formulae such as  $\forall p.T \Rightarrow q$  can be represented using this datatype as follows:

```
Forall ("p", Implies(True, Var "p"))
```

Implementing this DSL would consist of implementing an interpreter that checks the validity of the formula. Such a function is concisely implemented by the following function:

```

exception VarNotFound;;

let env0 x = raise VarNotFound

let ext env x v = fun y -> if y=x then v else env y

let rec eval b env =
  match b with
  | True -> true
  | False -> false
  | And (b1,b2) -> (eval b1 env) && (eval b2 env)
  | Or (b1,b2) -> (eval b1 env) || (eval b2 env)
  | Not b1 -> not (eval b1 env)
  | Implies (b1,b2) -> eval (Or(b2,And(Not(b2),Not(b1)))) env

```

```

| Forall (x,b1) ->
    let trywith bv = (eval b1 (ext env x bv))
    in (trywith true) && (trywith false)
| Var x -> env x

```

The first line declares an exception that may be raised (and caught) by the code to follow. We implement an environment as a function that takes a name and either returns a corresponding value or raises an exception if that value is not found. The initial environment `env0` always raises the exception because it contains no proper bindings. We add proper bindings to an environment using the function `ext`, which takes an environment `env`, a name `x`, and a value `v`. It returns a new environment that is identical to `env`, except that it returns `v` if we look up the name `x`.

The evaluation function itself takes a formula `b` and environment `env` and returns a boolean.

## 4.2 Staged QBF Interpreter

While the interpreter above is easy to write, it has the undesirable property that it must repeatedly traverse the abstract datatype that represents the formula during evaluation. Especially when a formula has many quantifiers, subterms will be evaluated repeatedly, and traversing them multiple times can have a significant effect on the time and space needed to complete the computation. The key benefit of staging is that it allows us to separate the two distinct stages of computation: traversing the term and evaluating it. Staging the above function is simply a matter of rewriting it by adding Brackets `.<...>` and Escapes `.~...>` as follows:

```

let rec eval' b env =
  match b with
  | True -> .<true>.
  | False -> .<false>.
  | And (b1,b2) -> .< .~(eval' b1 env) && .~(eval' b2 env) >.
  | Or (b1,b2) -> .< .~(eval' b1 env) || .~(eval' b2 env) >.
  | Not b1 -> .< not .~(eval' b1 env) >.
  | Implies (b1,b2) -> .< .~(eval' (Or(b2,And(Not(b2),Not(b1)))) env) >.
  | Forall (x,b1) ->
    .< let trywith bv = .~(eval' b1 (ext env x .<bv>..))
    in (trywith true) && (trywith false) >.
  | Var x -> env x

```

Brackets delay a computation, while Escapes force a computation inside the surrounding Brackets. Whereas `eval` interleaves the traversal of the term with performing the computation, `eval'` traverses the term exactly once, and produces a program that does the work required to evaluate it. This is illustrated by the following session in the MetaOCaml interactive bytecode interpreter:

```

# let a = eval' (Forall ("p", Implies(True, Var "p"))) env0;;
a : bool code =
.<let trywith = fun bv -> (bv || ((not bv) && (not true)))
  in ((trywith true) && (trywith false))>.

```

Note that the resulting computation is using only native OCaml operations on booleans, and there are no residual representations of the DSL program that was parsed.

MetaOCaml allows us to execute generated code fragments by invoking the compiler on each code fragment, and then incorporating the result into the runtime environment. This is done concisely as follows:

```
# .! a;;  
- : bool = false
```

The staged interpreter approach is also possible in Template Haskell and C++. A Template Haskell version of the QBF Interpreter example would be very similar to the MetaOCaml version. Staging would be done by adding quasi-quotes and splices. The staged interpreter could implement an embedded or closed DSL. The latter is possible thanks to I/O at the metalevel, e.g.:

```
splice( staged_interpreter (parse (get_input_from_file "foo.dsl") ) )
```

The `evalAt` function in Section 3.3 is an example of a staged interpreter in C++. Staging in C++ is not as straightforward due to its heterogeneity.

In Template Haskell, optimizations can be applied on the source representation and the result of running the staged interpreter. In MetaOCaml and C++, optimizations can be applied only on the source representation.

## 5 Comparison

**Style of DSL implementation supported.** Template Haskell and C++ both support implementing compile-time embedded DSLs. Template Haskell additionally supports implementing closed DSLs as staged interpreters. MetaOCaml supports implementing runtime embedded DSLs and closed DSLs as staged interpreters. Runtime embedded DSLs are particularly useful if runtime optimizations are needed (e.g., when multiplying large matrices, runtime optimizations may offer significant speedups; see [5]), otherwise compile-time embedded DSLs will be more efficient and safe to use.

**Metaprogramming model.** Template Haskell is a homogeneous two-level language with Haskell as its meta and object language. C++ is a heterogeneous two-level language: functional at the metalevel and imperative object-oriented at the object level. Metaprograms are encoded in the type system and use templates both as algebraic data constructors and functions. MetaOCaml is a homogeneous multi-level language (you can nest quasi-quotes directly). Homogeneity has the advantage of enabling reuse of code and programming skills across levels.

**When is code generated?.** In Template Haskell and C++ all code generation is done at compile time. In MetaOCaml all code generation is done at runtime. In MetaOCaml’s “runtime,” such dynamically generated code can also be compiled and executed.

**How is code generated?.** In C++ TMP, code generation involves composing functions, classes, and templates. Functions are selected using a metaprogram and composed by function call or inlining (the latter provides a way to

generate code in one piece). Templates are composed by template inlining. In C++, the smallest fragment to be used in the generated target has to be a valid function, class or template. This is also the reason why C++ TMP does not need an explicit (user-level) gensym. MetaOCaml generates code by composing quasi-quoted expressions. It also does not need gensym, as the implementation always does the necessary renaming. Template Haskell is more fine-grained: it can construct parse trees from syntactic primitives or by quasi-quoting. For this reason, Template Haskell needs to support gensym, as renamings to avoid name capture cannot always be performed automatically.

**What can be generated?** C++ can generate classes (and class hierarchies; see Section 3.2), functions, expressions (using expression templates; see Section 3.3), and templates (as members). Functions can be generated using function templates, e.g., `power<3>(int 4)` could be a call to a function template taking the exponent as a static template parameter and generating code with an unrolled loop (see [3, p. 495]). Template Haskell can generate expressions and declarations of types, functions, values, classes, and instances. The power example could look like this: `$(power 3) 4`. Only expressions and declarations can be spliced in (but not, e.g., patterns). No new (user-defined) identifiers can be generated in C++, but this is possible in Template Haskell, e.g.,

```
splice(powerX 3) -- generate and splice declaration of power3
power3 4        -- apply power3
```

In MetaOCaml, only expressions can be spliced in (but they can contain declarations, modules, new classes, etc).

**Type checking of the generator code.** In the context of program generation, we have three opportunities for performing static typechecking:

1. typechecking the code executed at generation time before generation
2. typechecking the quoted code (or functions and templates representing the code to be composed in C++) before generation
3. typechecking of the generated code before running it

Template Haskell performs (1) and (2), compiles and runs the metacode, splices the results, and performs (3). This is called staged type inference [15]. MetaOCaml performs only (1) and (2) since the generated code is guaranteed to be type-safe (statically typed MSP, see [18]). MetaOCaml uses a stronger type-system than Template Haskell: a quoted integer expression in MetaOCaml has the type of “integer code” whereas in Template Haskell it has the type “expression”. C++ will only do (3). The metalevel in C++ has just two kinds of values: types and integers, and it is dynamically typed, i.e., typing problems are discovered only when the metacode is run. With regard to (2), functions in C++ get typechecked before generation, but templates won’t. In general, the benefits of MetaOCaml style of typing would be most desirable. However, this kind of typing causes some limitations, e.g., inspection of the quoted code is not possible. Template Haskell strikes a balance between type-safety and the expressiveness necessary for a compile-time only meta-system.

**Separate compilation.** MetaOCaml and Template Haskell both support separate compilation of generator code. C++ does not. Consider importing generator code from a library. In MetaOCaml, it is guaranteed that the generator code will not fail due to a typing error and it will produce type-safe code, i.e., no typing errors due to the generator will be discovered at the site of its use. In Template Haskell, the generator will not fail due to a typing error at the site of use, but it may generate ill-typed code. In C++, both kinds of typing error will surface at the site of generator use. Also, libraries including C++ template metaprograms can be distributed in source form only.

**Concrete syntax of DSLs.** Type-based DSLs in C++ are represented by static integral constants and type expressions, e.g., `ElemType<float>, Optim<space>`. Expression-based DSLs in C++ are represented as C++ expressions with overloaded functions and operators, e.g., `v1+v2*v3`. C++ supports no user-defined operators. Template Haskell supports user-defined operators, but it has a more restricted form of operator overloading than C++. In Template Haskell, any Haskell syntax can be used to represent a DSL. Currently, the quoted code must appear in a different module than the splice, although this requirement may be relaxed in the future. Thus one can write:

```
$(optimize code)$
code = [d|
v1 = [1 2 3]
v2 = [1 2 3]
v3 = [1 2 3]
r = v1 'add' (v2 'mul' v3)
|]
```

In Template Haskell, any syntax can be used if placed in a string:

```
r = $( evaluate (parse "integral (d(x), x^2, 0.0--5.0)" ) )
```

Closed DSLs in Template Haskell or MetaOCaml can use any syntax.

**Syntax for invoking the metalevel.** Template Haskell uses splice as an explicit notation to invoke computations at the metalevel. In C++, the metalevel is invoked by instantiating a template. This can be done explicitly for type-based DSLs and implicitly for expression DSLs. The implicit invocation is due to automatic type inference for function templates.

**How are transformations performed.** Template Haskell transforms parse trees represented by algebraic data-structures and it has standard data-structures for representing Haskell programs. In C++, transformations are performed by manipulating algebraic data-structures representing the structure of an expression in expression DSLs or parameters in type-based DSLs. MetaOCaml also transforms algebraic data-structures.

**What is the scope of transformations.** In C++, the scope of transformations in the expression-based DSLs is limited to a single C++ expression, i.e., no global optimizations are possible. However, a sequence of DSL statements can be represented as a single C++ expression by using the overloaded comma operator as a separator; this allows transforming several DSL statements at once (see [9], [16]). The scope of a transformation in Template Haskell is limited to a

splice. If desired, a whole program could be quoted and passed to an optimization function in a splice. In MetaOCaml, the entire input to a staged interpreter can be transformed.

**Reification.** Compile-time reification allows discovering properties of code written by the user (including quoted code). Template Haskell has extensive reification capabilities. First, quoted code can be inspected as a parse tree. Furthermore, there are built-in operators to access the parse tree of a user-defined declaration or the current source position (useful for error reporting). It is possible, in effect, to gain access to an entire module by enclosing its entire contents inside quotation brackets in importing the resulting code into another module. Compile-time reification in C++ is quite limited. A few properties of types can be discovered automatically, e.g., whether a type is a pointer type or if type A is derived from B (see [6]), but the vast majority (e.g., names of variables, functions and classes; number of parameters of a function; member functions of a class; etc.), if needed, must be provided manually. C++ has some idioms for doing that, e.g., traits members, classes, and templates [6]. Expression templates can also be viewed as an elegant reification technique: normally, we do not have access to the structure of an expression at compile-time. But expression templates reify the structure of an expression as a parse tree at compile time (see Section 3.3). This reification is programmed manually for different type and function combinations, but it can be included in a library and the user can benefit automatically and does not need to write any reification code. Unfortunately, expression templates only reify the algebraic structure of an expression, e.g., we cannot query the name of a variable (unless we encode it in the type), which could be useful to optimize  $\mathbf{a+b-a}$ . This allows the metaprogram to discover everything defined within the module. There is no special support for reification in MetaOCaml.

**Analysis and error reporting.** Analysis and error reporting need special attention in compile-time embedded DSLs. Both Template Haskell and C++ allow writing code that will check the structure a DSL program for some domain-specific properties, e.g., whether different units of some measurements system (e.g., the SI system) are properly combined and used. The latter usually requires embedding some domain-specific type inference and can be done using static metaprogramming. C++ does not allow any I/O at the metalevel; errors can be reported only using a rather crude programming trick by enforcing a compiler error reporting an identifier with the error cause as its name (see [3, p. 669]). Template Haskell supports I/O at the metalevel. Besides error reporting, I/O also has other applications such as inspecting the code being generated for debugging purposes (which is not possible in C++), reading additional specification from some file (e.g., an XML configuration file), or generating code in other languages (e.g., C for interfacing to some native libraries).

**Practical applications and limitations.** There is extensive experience with implementing DSLs using C++ TMP. The first application to vector computations in 1996 [19] was followed by an avalanche of others: DSLs for matrix computations, statistical algorithms, network protocols, signal processing, mea-

surement units, database persistence, image processing, and more. C++ TMP suffers from a number of limitations, including portability problems due to compiler limitations (although this has significantly improved in the last few years), no debugging support or I/O at level -1, long compilation times, long and incomprehensible errors, poor readability of the code, and poor error reporting. Nevertheless, these limitations did not prevent people from creating useful embedded DSLs that would not have been possible otherwise. Collections of library primitives to make metaprogramming easier are now available [10, p. 433-451], [10].

MetaOCaml and Template Haskell are very young: their capabilities are currently being explored and the languages as well as implementations are still evolving. A collection of small-scale student projects in MetaOCaml (including implementations of SSA, parsers, a subset of XSMML, and small programming languages) is available online [17]. Initial experience with Template Haskell was reported in [11], [12], [7], and the implementation of several DSLs in Template Haskell is currently under way, including Hydra'2003 and a parallel skeleton project by Kevin Hammond and colleagues. Both MetaOCaml and Template Haskell still lack proper tool support to debug generator code.

## 6 Conclusions

Perfect embedding in a programming language is the easiest way to implement a DSL. Also, since many applications need to be written using more than one DSL, embedding is a natural setting for DSL composition. Unfortunately, some DSLs do not fit perfectly within available programming languages. There may be mismatches in areas such as domain-specific syntax, optimizations, debugging, error reporting, and code generation. Such mismatches prevent the traditional library solution, but two-level languages such as Template Haskell and C++ provide a convenient and lightweight approach to resolving many kinds of mismatch at compile time. Furthermore, homogeneous multilevel languages make it possible to provide useful runtime optimizations. If the mismatch is very significant, using such languages to implement a closed DSL as a staged interpreter might be more attractive than building a full compiler.

The large number of practical DSL implementations using metaprogramming in C++ shows that metaprogramming support is useful—and sometimes necessary—in a mainstream language. This support is especially important for library programmers. The success of metaprogramming in C++ is remarkable, especially considering the fact that C++ was never designed for metaprogramming and suffers from many limitations. Languages such as MetaOCaml and Template Haskell approach the problem by design. Template Haskell offers an interesting combination of type safety and flexibility that should be appealing to researchers experimenting with new language features and building their own DSLs. MetaOCaml offers an operationally more general framework and a static type system that ensures type safety for all dynamically generated and executed code.

These languages incorporate significant advances in providing support for DSL implementers, and experience with them is currently revealing new challenges where such support could be improved. For example, providing proper domain-specific error reporting is both a challenge and an opportunity: if a library told you in an understandable way that you are using it ineffectively or incorrectly, it could be much for friendly to end users. However, all three of the languages compared here lack comprehensive facilities for debugging implementations of embedded DSLs. Providing proper debugging support for the user of a DSL is largely unexplored. There are subtle problems with providing proper support for reification which need further study. In general, more foundational research in the area of program generation is needed, including more expressive type systems, type safe ways for doing intensional analysis, and techniques for minimizing dynamic, generation and compilation times. The current generation of metaprogramming languages, such as the ones compared in this paper, will make it possible to gain more practical experience with building real applications, and that experience will feed back to improve our understanding of the best ways to provide language support for metaprogramming.

## References

1. K. Czarnecki and U. W. Eisenecker. Synthesizing objects. In *Proceedings of ECOOP'99*, number 1628 in LNCS, pages 18–42. Springer-Verlag, 1999.
2. K. Czarnecki and U. W. Eisenecker. Named parameters for configuration generators. 2000.
3. K. Czarnecki and U.W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.
4. Simon Peyton Jones (ed.). Haskell 98 language and libraries. *Journal of Functional Programming*, 13(1):1–255, January 2003.
5. Dawson R. Engler, Wilson C. Hsieh, and M. Frans Kaashoek. ‘C: A language for high-level, efficient, and machine-independent dynamic code generation. In *POPL'96*, pages 131–144, 1996.
6. Aleksey Gurtovoy. Boost type traits library.
7. K. Hammond, R. Loogen, and J. Berhold. Automatic Skeletons in Template Haskell. In *Proceedings of 2003 Workshop on High Level Parallel Programming, Paris, France*, June 2003.
8. Scott Haney, James Crotinger, Steve Karmesin, and Stephen Smith. Pete: the poertable expression template engine. *Dr. Dobbs Journal*, October 1999.
9. Jaakko Järvi and Gary Powell. The lambda library: Lambda abstraction in c++. In *Second Workshop on C++ Template Programming, Tampa Bay, Florida, USA*, 10 2001.
10. et al. John Maddock, Steve Cleary. Boost mpl library (template metaprogramming framework).
11. Ian Lynagh. Template Haskell: A report from the field. May 2003.
12. Ian Lynagh. Unrolling and simplifying expressions with Template Haskell. May 2003.
13. John O'Donnell. Overview of Hydra: A concurrent language for synchronous digital circuit design. In *Proceedings 16th International Parallel & Distributed Processing Symposium*, page 234 (abstract). IEEE Computer Society, April 2002. Workshop on

Parallel and Distributed Scientific and Engineering Computing with Applications—  
PDSECA.

14. Tim Sheard and Simon Peyton Jones. Template metaprogramming for Haskell. In Manuel M. T. Chakravarty, editor, *ACM SIGPLAN Haskell Workshop 02*, pages 1–16. ACM Press, October 2002.
15. Mark Shields, Tim Sheard, and Simon L. Peyton Jones. Dynamic typing as staged type inference. In *Symposium on Principles of Programming Languages*, pages 289–302, 1998.
16. Jörg Striegnitz and Stephen Smith. An expression template aware lambda function. In *First Workshop on C++ Template Programming, Erfurt, Germany*, 10 2000.
17. W. Taha. Multi-stage programming course projects. 2000.
18. Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *Proceedings of the Symposium on Partial Evaluation and Semantic-Based Program Manipulation (PEPM)*, pages 203–217, Amsterdam, 1997. ACM Press.
19. Todd Veldhuizen and Kumaraswamy Ponnambalam. Linear algebra with C++ template metaprograms. *Dr. Dobb's Journal of Software Tools*, 21(8):38–44, August 1996.
20. Todd L. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, 1995.
21. Todd L. Veldhuizen. Template metaprograms. *C++ Report*, 7(4):36–43, 1995.