DSL Implementation in MetaOCaml, Template Haskell, and C++

Krzysztof Czarnecki[1], John O'Donnell[2], Jörg Striegnitz[3], and Walid Taha[4]

[1] University of Waterloo, Canada
[2] University of Glasgow, United Kingdom
[3] Research Centre Jülich, Germany
[4] Rice University, USA

Abstract. A wide range of domain-specific languages (DSLs) has been implemented successfully by embedding them in general purpose languages. This paper reviews embedding, and summarizes how two alternative techniques—staged interpreters and templates—can be used to overcome the limitations of embedding. Both techniques involve a form of generative programming. The paper reviews and compares three programming languages that have special support for generative programming. Two of these languages (MetaOCaml and Template Haskell) are research languages, while the third (C++) is already in wide industrial use. The paper identifies several dimensions that can serve as a basis for comparing generative languages.

1 Introduction

A basic design choice when implementing a new programming language is whether to build an interpreter or a compiler. An interpreter realizes the actions of the program by stepping through the source program. A compiler consists of a translator and a runtime system. The translator maps the source program to a program in an existing language, while the runtime environment provides the primitives needed for the resulting program to be executable. Translation can be to a lower-level language—as is the case in traditional compilers—or to a high-level language for which we already have an implementation.

An interesting special case occurs when no translation is needed. This means that the new language is both syntactically and semantically a subset of an existing host language. In this case, all we need is to implement the runtime system as a library in the host language. This approach of *embedded languages* has recently gained significant popularity in the functional programming community [19]. Functional languages make it possible to implement DSLs that are more sophisticated than is possible with traditional languages. For example, lambda abstractions can be used conveniently to allow DSLs with binding constructs, and nontrivial type systems for the DSLs can be encoded within the sophisticated type systems provided by functional languages. From the DSL implementer's point of view, the benefits of this approach include reusing the parser, type checker, and the compiler of an existing language to implement a

new one. Examples of embedded DSLs include parsing [32, 22, 37], pretty printing [21], graphics [12, 11], functional reactive programming [20], computer music [18], robotics [40], graphical user interfaces [4], and hardware description languages [1, 31, 36]. While this approach works surprisingly well for a wide range of applications, embedding may not be appropriate if there is

- *Mismatch in concrete syntax.* A prerequisite for embedding is that the syntax for the new language be a subset of the syntax for the host language. This excludes many stylistic design choices for the new language, including potentially useful forms of syntactic sugar and notational conventions. Furthermore, as DSLs become easier to implement, we can expect an increase in the use of graphical notations and languages, which often do not use the same syntax as potential host languages.
- *Mismatch in semantics.* The embedding approach requires that the semantics for the the DSL and host languages coincide for the common subset. For example, two languages may share the same syntax but not the same semantics if one is call-by-value and the other is call-by-name. A more domain-specific example arises in the Hydra language [36], which uses the same syntax as Haskell, but which has a different semantics.

Even when these problems can be avoided, the resulting implementation may be lacking in a number of ways:

- *Domain-specific type-systems.* Often DSLs are designed to have limited expressiveness. This is generally hard to achieve with embedding because the DSL inherits the type system of the host language. For instance, consider a host language with subtyping and dynamic dispatch and a DSL for which we want just static method binding.
- *Error messages.* Similarly, when an embedded program fails to type check in the host language, the error messages are expressed in terms that are often incomprehensible (and arguably irrelevant) to the DSL user.
- *Debugging support.* Trying to use the debugger for the existing language is not always useful, as it often exposes details of the runtime system for the host language which are of no interest to the DSL user.
- *Domain-specific optimizations.* Embedding does not lend itself naturally to realizing optimizations that are valid only for the domain, or that simply do not happen to be provided by the existing language implementation. This situation is further aggravated by the fact that the official definitions of many popular host languages do not specify what optimizations the compiler must perform.

1.1 Contributions and Organization of the Rest of this Paper

While there have been a number of promising proposals for addressing each of these points (c.f. [39, 53] for typing, [10, 8, 58] for optimizations), all these approaches use program manipulation and generation, which can involve substantially more effort, care, and expertise on the part of the implementer of the DSLs

[47]. Because of the potential for specialized programming language support to help in controlling the complexity of generative programs, we compare three languages in terms of what they offer to support these approaches. The first two are the research languages MetaOCaml [2, 33] and Template Haskell [44], presented in Sections 2 and 3 respectively. The third is the C++ language (Section 4), which is already in industrial use for generative programming, although it was not designed with that purpose in mind. Throughout the paper, an attempt is made to introduce only the essential terminology for each of the languages, and to introduce terminology useful for comparing and contrasting these languages. Section 5 presents a comparative analysis of the three languages. The goal of this analysis is to identify and highlight the key differences between the three languages and the design philosophies they represent. Section 6 concludes.

2 MetaOCaml

MetaOCaml is a multi-stage extension of the OCaml programming language [2, 33]. Multi-stage programming languages [51, 47, 52] provide a small set of constructs for the construction, combination, and execution of program fragments. The key novelty in multi-stage languages is that they can have static type systems that guarantee *a priori* that all programs generated using these constructs will be well-typed. The basics of programming in MetaOCaml can be illustrated with the following declarations:

```
let rec power n x = (* int -> .<int>. -> .<int>. *)
  if n=0 then .<1>. else .<.~x * .~(power (n-1) x)>.
let power3 = (* int -> int *)
  .! .<fun x -> .~(power 3 .<x>.)>.
```

Ignoring the code type constructor `.<`t`>.` and the three staging annotations brackets `.<`e`>.`, escapes `.~`e and run `.!`, the above code is a standard definition of a function that computes x^n, which is then used to define the specialized function x^3. Without staging, the last step just produces a function that invokes the power function every time it gets a value for x. The effect of staging is best understood by starting at the end of the example. Whereas a term `fun x ->` e `x` is a value, an annotated term `.<fun x -> .~(`e `.<x>.)>.` is not. Brackets indicate that we are constructing a future stage computation, and an escape indicates that we must perform an immediate computation *while* building the bracketed computation. The application e `.<x>.` has to be performed even though x is still an uninstantiated *symbol*. In the `power` example, `power 3 .<x>.` is performed immediately, once and for all, and not repeated every time we have a new value for `x`. In the body of the definition of the `power` function, the recursive application of `power` is also escaped to make sure that they are performed immediately. The run `.!` on the last line invokes the compiler on the generated code fragment, and incorporates the result of compilation into the runtime system.

The basic approach to implementing DSLs in MetaOCaml is the *staged interpreter approach* [13, 26, 25, 43]. First, an interpreter is implemented and tested

for the DSL. Then, the three staging constructs are used to produce an implementation that performs the traversal of the DSL program in a stage earlier than the execution of the essence of the program. 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.

2.1 A QBF Interpreter

We 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 $\forall p.\mathtt{T} \Rightarrow q$ can be represented using this datatype by the value `Forall ("p", Implies(True, Var "p"))`. Implementing this DSL would involve implementing an interpreter that checks the validity of the formula. Such a function is implemented concisely by the following MetaOCaml code:

```
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 that represents the truth of the formula.

2.2 A Staged QBF Interpreter

An interpreter such as the one above is easy to write, but has the undesirable property that it must repeatedly traverse the syntax of the DSL program while evaluating it. For the interpreter above, when a formula contains quantifiers, sub-terms will be evaluated repeatedly, and traversing them multiple times can have a significant effect on the time and space needed to complete the computation. Staging can allow us to separate the two distinct stages of computation: traversing the term and evaluating it. The above interpreter can be staged 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
```

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 using the run construct `.!`:

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

2.3 Domain-specific Analysis and Optimizations

Because functional programming language have concise notation for representing parse trees (algebraic datatypes), domain-specific optimizations can be defined concisely in MetaOCaml. Pattern matching is particularly convenient when we want to implement such optimizations:

```
let rec optimize e =
 match e with
    ...
  | Or  (e, False) -> optimize e
  | And (e, True)  -> optimize e
    ...
```

Note that here we are pattern matching on the datatype used to represent the DSL programs, not on the MetaOCaml code type (which is the return type of the staged interpreter above). Pattern matching on the code type is not encouraged[5], primarily because it is currently not known how to allow it without either allowing ill-typed code to be generated or to make substantial changes to the MetaOCaml type system [49]. This way the programmer is sure that once a particular code fragment is generated, the meaning of this code fragment will not change during the course of a computation.

2.4 The User's View

Staged interpreters can be provided as libraries that have the same interface as a traditional interpreter. For example, we can redefine `eval` as:

```
# let eval a env = .! (eval' a env);;
```

This function has the same signature as the one above, but different performance characteristics. Because MetaOCaml is an extension of a general purpose programming language (OCaml), implementing parsers and loaders is not problematic, and it is easy to see how we can directly execute small DSL programs written in concrete syntax, and load bigger ones from a file:

```
# eval (parse "forall x. x and not x");;
# eval (parse (load "myTheorem.dsl");;
```

3 Template Haskell

Template Haskell is an extension of Haskell that supports compile-time preprocessing of Haskell source programs [9, 44]. The goal of Template Haskell is to enable the Haskell programmer to define new language features without having to modify the compilers [28, 29, 16].

[5] Using coercions it is possible to expose the underlying representation of the MetaOCaml code type, which is the same as the datatype for parse trees used in the OCaml compiler.

To illustrate some of the constructs of Template Haskell, we present an analog of the power example above. The `expand_power` function builds an expression that multiplies `n` copies of the expression `x` together, and `mk_power` yields a function that computes x^n for any integer x.

```
module A where
import Language.Haskell.THSyntax

expand_power :: Int -> Q Exp -> Q Exp
expand_power n x =
  if n==0
    then [| 1 |]
    else [| $x * $(expand_power (n-1) x ) |]

mk_power :: Int -> Q Exp
mk_power n  = [| \x -> $(expand_power n [| x |]) |]
```

A specialized function `power3` is now defined using `$` to splice in the code produced at compile time by `mk_power 3`.

```
module Main where
import Language.Haskell.THSyntax
import A

power3 :: Int -> Int
power3 = $(mk_power 3)
```

In this example, quotations `[|`e`|]` and splice `$`e are analogous to the brackets and escape of MetaOCaml. However, there are three key differences. First, in place of MetaOCaml's parameterized type constructor `.<`t`>.` the type for quoted values in Template Haskell is always `Q Exp`. Second, in the definition of `module Main`, the splice occurs without any explicit surrounding quotations. To compare this with MetaOCaml, the module (apart from the `import` directive) could be interpreted as having implicit quotations.[6] Third, there is no explicit run construct.

The staged interpreter approach can be used in Template Haskell. A Template Haskell version of the QBF Interpreter example would be similar to the MetaOCaml version. Programs can be staged using quasi-quotes and splices. But the primary approach to implementing DSLs in Template Haskell is a variation of the embedding approach. In particular, Template Haskell allows the programming to alter the semantics of a program by transforming it into a different program before it reaches the compiler. This is possible because Template Haskell allows the inspection of quoted values.

3.1 Building and Inspecting Abstract Syntax Trees

Template Haskell provides an algebraic data type for representing Haskell code as an abstract syntax tree. A part of the datatype is the following:

[6] This analogy has in fact been formalized for macro languages [14, 50].

```
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
```

In addition to quotations, trees can be built explicitly using the constructors shown above. Template Haskell also provides a set of monadic operations[7] to construct syntax trees, which offer better support for variable renaming and input/output.

Alternatively, the abstract syntax tree may be defined using a quoted expression, which is more readable than explicit construction. The entire contents of a module may be quoted, in order to produce a list of abstract syntax trees comprising the entire contents of the module. An example of a quoted definition (which represents a program in the Hydra DSL) is:

```
[| circ x y = (a,b)
    where a = inv x
          b = xor2 a y |]
```

The value of the quoted expression is a monadic computation that, when performed, produces the abstract syntax tree:

```
Fun "circ"
  [Clause
    [Pvar "x'0", Pvar "y'1"]
    (Normal (Tup [Var "a'2", Var "b'3"]))
    [Val (Pvar "a'2") (Normal (App (Var "Signal:inv") (Var "x'0"))) [],
     Val (Pvar "b'3") (Normal (App (App (Var "Signal:xor2")
                          (Var "a'2")) (Var "y'1"))) []]]
```

This abstract syntax tree consists of one clause corresponding to the single defining equation. 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`. The variables have been renamed automatically in order to prevent accidental name capture; for example `x` in the original source has become `x'0` in the abstract syntax tree. The numbers attached to variable names are maintained in a monad, and the DSL programmer can create new names using a monadic `gensym` operation.

In general, it is easier to work with code inside `[| |]` brackets; this is more concise and the automatic renaming of bound variables helps to maintain correct lexical scoping. However, the ability to define abstract syntax trees directly offers more flexibility, at the cost of added effort by the DSL implementor.

[7] In Haskell, computations are implemented using an abstract type constructor called a monad [34, 41].

Yet another way to produce an abstract syntax tree is to reify an ordinary Haskell definition. For example, suppose that an ordinary Haskell definition `f x = ...` appears in a program. Although this user code contains no quotations or splices, the DSL implementation can obtain its abstract syntax tree: `ftree = reifyDecl f`. Values and types can be reified, as well as operator fixities (needed for compile time code to understand expressions that contain infix operators) and code locations (which enables the compile time code to produce error messages that specify where in the source file a DSL error occurs).

Once a piece of Haskell code has been represented as an abstract syntax tree, it can be analyzed or transformed using standard pattern matching in Haskell. This makes it possible to transform the code into any Haskell program that we wish to use as the DSL semantics for this expression.

3.2 The User's View

As with MetaOCaml, the staged interpreter can be made available to the user as a library function, along with a parser and a loader. Such functions can only be used at compile time in Template Haskell. It is worth noting, however, that Template Haskell does allow compile-time IO, which is necessary if we want to allow loading DSL programs from files.

Additionally, Template Haskell's quotations make it possible to use Haskell syntax to represent a DSL program. Template Haskell also makes it possible to quote declarations (and not just expressions) using the `[d| ... |]` operator:

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

Then, another module could apply a staged interpreter to this code in order to transform the original code into an optimized version and splice in the result.

Template Haskell also allows templates to construct declarations that introduce new names. For example, suppose we wish to provide a set of specialized power functions, `power2`, `power3`, up to some maximum such as `power8`. We can write a Template Haskell function that generates these functions, using the `mk_power` defined above, along with the names, and then splices them into the program.

```
$(generate_power_functions 8)
```

4 C++ Templates

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 sub-language of C++ that is interpreted by the C++ compiler [54, 57]. Consequently, a C++ program may contain both static code, which is evaluated 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 Metaprogramming [57].

Before giving a short overview of what static code looks like, we recapitulate some C++ template basics.

4.1 Template Basics

C++ supports two kinds of templates: class templates and function templates. The basic use of class templates in C++ is to define generic components. For example, we can define a generic vector as a class template with element type and size as parameters:

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

A template is *instantiated* in a `typedef` statement as follows:

```
typedef Vector<float,10> MyVector;
```

Here, the type parameter `T` in the original definition is simply replaced by `float` and `size` by `10`. Now we can use `MyVector` as a regular C++ class:

```
MyVector v;
v[3] = 2.2;
```

A template can be *specialized* by providing a definition for a specialized implementation as follows:

```
template <int size>
class Vector<bool, size>
{ //...
   // Use just enough memory for 'size' bits
   char data[(sizeof(char)+size-1)/sizeof(char)];
};
```

Such specializations are useful because one general implementation of a template might not be efficient enough for some argument values. Now, whenever we instantiate `Vector` with `bool` as its first parameter, the latter *specialized* implementation will be used.

Function templates are parameterized functions. A function template to swap two values may be implemented like this:

```
template <class T>
void swap (T& a, T& b)
{ const T c = a; a = b; b = c; }
```

The function template `swap` can be called as if it were an ordinary C++ function. The C++ compiler will automatically infer and substitute the appropriate parameter type for `T`:

```
int a = 5, b = 9;
swap(a,b);
```

4.2 Template Metaprogramming

Template Metaprogramming adds two rather unusual views of class templates: class templates as data constructors of a datatype and class templates as functions. To introduce these views, we will compare them with the corresponding Haskell definitions.

Here is an example of C++ template program defining a list, followed by the corresponding Haskell code.

C++:

```
struct Nil {};
template <int H, class T>
struct Cons {};

typedef
Cons<1,Cons<2, Nil> > list;
```

Haskell:

```
data List = Nil | Cons Int List


list = Cons 1 (Cons 2 Nil)
```

The C++ definitions for `Nil` and `Cons` have different dynamic and static semantics. Dynamically, `Nil` is the name of a class, and `Cons` is the name of a class template. Statically, both `Nil` and `Cons` can be viewed as data constructors corresponding to the Haskell declaration on the right. All such constructors can be viewed as belonging to a single "extensible datatype", which such definitions extend.

The C++ template instantiation mechanism [23] provides the semantics for compile-time computations. Instantiating a class template corresponds to applying a function that computes a class. In contrast, class template *specialization* allows us to provide different template implementations for different argument values, which serves as a vehicle to support pattern matching, as common in functional programming languages. Next, we compare how functions are implemented at the C++ compile time and in Haskell:

C++:

```
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 }; };
```

Haskell:

```
len :: List -> int;

len Nil        = 0


len (Cons h t) = 1 + (len t)
```

The template specializations `Len<Nil>` and `Len<Cons<H,T> >` handle the empty and non-empty list case, respectively. In contrast to Haskell or ML, the relative order of specializations in the program text is insignificant. The compiler selects the "best match" (which is precisely defined in the C++ standard [23]).

This calling scheme makes static functions very extensible. Having an extensible datatype and extensible functions, C++ Template Metaprogramming doesn't suffer from the extensibility problem [27].

The `::` operator is used to access the name space of the class. For example, `Len<list>::RET` will evaluate to 2. The member being accessed could also be a computed type or a static member function. Template Metaprogramming also supports higher-order functions through so-called template template parameters.

In relation to DSL implementation, C++ templates can be used in two distinct ways: at the type level, and at the expression level. The first is comparable to Template Haskell's mechanism to analyze and build new declarations. The second is comparable to the staged interpreters approach, except that no parser has to be implemented.

4.3 Type-driven DSLs

Template Metaprogramming allows us to write *configuration generators* [5]. Typically, such generators must check parameters, compute default values, and perform any computations necessary for assembling generic components. Syntactically, type-driven DSLs are subsets of C++ type expressions. To give a concrete example of such a type-driven DSL, consider a generator for different implementations of matrix types. The DSL program (which results in invoking the generator) can be written 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 list of properties by computing defaults for unspecified properties, and computes the appropriate composition of elementary generic components such as containers, shape adapters and bounds checkers. This configuration generator uses a specific style for specifying properties that simulates keyword parameters: for example, when we write `ElemType<float>` we have a parameter `float` wrapped in the

template `ElemType` to indicate its name [7]. Configuration generators can compose mixins (that is, class template whose superclasses are still parameters) and thus generate whole class hierarchies [5]. They can also compose mixin layers [45], which corresponds to generating object-oriented frameworks.

4.4 Expression-driven DSLs

A popular technique for implementing expression-level embedded DSLs is expression templates [56, 17]. To illustrate this approach, we 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., by using OpenMP[3]. In order to achieve this, we need to transform the expression into code like the following:

```
// Tell the OpenMP compiler to parallelize the following loop.
#pragma OMP parallel for
for (int i=0 ; i<= vectorSize; ++i)
  // perform componentwise evaluation of a vector
  // expression, e.g., (a[i] + b[i]) * c[i])
```

This code cannot be produced by 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 structure of the expression in its type. This has the effect of turning the expression into a parse tree at compile-time. Here is how we can reflect the parse tree of the expression: the leaves of the parse tree will be vectors and we will 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;
};
```

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 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 size>
BNode<OpPlus,Vector<V,size>,Vector<V,size> >
  operator+(const Vector<V,size>& l,const Vector<V,size>& r) {
  return BNode<OpPlus,Vector<V,size>,Vector<V,size> >(OpPlus(), l, r );
}
```

Note that code inspection in C++ is limited. For a type-driven DSL all information is embedded in the type. For an expression-driven DSL, this is not the case. For instance, we can only inspect the type of a vector-argument; its address in memory remains unknown at the static level. Inspecting arbitrary C++ code is possible to a very limited extend. For instance, different programming idioms (such as traits members, classes, and templates [35, 30]) have been developed to allow testing a wide range of properties of types. Examples of such properties include whether a type is a pointer type or if type `A` is derived from `B`. Many other properties (such as the names of variables, functions and classes; and member functions of a class) must be hardcoded as traits.

Next, we define `Eval`, which is a transformation that takes a parse tree and generates code to evaluate it. For instance, while visiting a leaf, we generate a function that evaluates a vector at a certain position `i`:

```
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 generate two functions to evaluate the values of the siblings. Once these functions have been generated (i.e, the corresponding templates were instantiated), we generate calls to those functions:

```
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  // generate function
           (b.left,i)       // generate call to generated code
           +  // Operation to perform
           Eval<R>::evalAt  // generate function
           (b.right,i);}    // generate call to generated code
};
```

Since all generated functions are eligible candidates for inlining and tagged with the `inline` keyword, a compiler should be able to remove any call overhead such that resulting effect is the same as if we spliced in code immediately.

Invoking the `Eval` transformation is somewhat tedious because we need to pass the complete parse tree to it. Since type parameters to function templates are inferred automatically, a function template solves this problem quite conveniently:

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

When we call `eval((a+b)*c,1)`, the compiler instantiates this 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 machinery just discussed delays execution of the expression (staging), gives access to the structure of programs (intensional analysis) and provides a means to generate code that evaluates the expression (code splicing). In fact, we can view `evalAt` function as an example of a staged interpreter in C++.

The remaining task is to make the `Vector` class aware of the new evaluation scheme. That is, we have to define how to initialize a vector from an expression and how to assign an expression to it:

```
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 ); }
  return *this;
};
```

The code generated for the assignment statement `d=((a+b)*c);` will be the same as the desired code given at the beginning of this section with the inner comment replaced by `d[i]=(a[i]+b[i])*c[i]`, and it will be executed in parallel.

5 Comparison

In this section, we present some basic dimensions for variation among the three different languages. The following table summarizes where each of the languages falls along each of these dimensions:

Dimension	MetaOCaml	Template Haskell	C++ Templates
1. Approach	Staged interp.	Templates	Templates
2. How	Quotes	Quotes & abs. syn.	Templates
3. When	Runtime	Compile-time	Compile-time
4. Reuse	Compiler	Compiler (& parser)	Compiler & parser
5. Guarantee	Well-typed	Syntax valid	Syntax valid
6. Code inspection	No	Yes	Limited
7. IO	Always	Always	Runtime
8. Homogeneous	Yes	Yes	No
9. CSP	Yes	Yes	No
10. Encapsulation	Yes	No	Yes
11. Modularity	Yes	Yes	No

The definition (and explanation) of each of these dimensions is as follows:

1. Of the DSL implementation approaches discussed in the introduction, what is the primary approach supported? To varying extents, both Template Haskell and C++ can also support the staged interpreter approach (but with a different notion of static typing). MetaOCaml staged interpreters can be translated almost mechanically into Template Haskell. Staging in C++ is not as straightforward due to its heterogeneity. Also, in C++, staged interpreters must use C++ concrete syntax (see Reuse and IO dimensions below).
2. How is code generated? MetaOCaml and Template Haskell generate code through quotes and splices. Additionally, Template Haskell allows splicing explicitly constructed ASTs. C++ Template Metaprogramming generates code through code selection (expressed using template specialization) and inlining. MetaOCaml and C++ perform all necessary renaming automatically.[8] Because Template Haskell allows access to the constructors for building Haskell parse trees, it must also provide an explicit `gensym` function.
3. When is code generated? MetaOCaml allows any subcomputation to involve generation of new code. Template Haskell and C++ limit code generation to compile time. MetaOCaml is called a *multi-stage language* because it provides constructs for both the construction and execution of new code during runtime [47].
4. What part of the implementation of the host language can be reused? Each of the languages considered in this paper can be viewed as providing a means for the programmer to reuse a particular aspect of the implementation of an existing language. All three languages reuse the compiler, because they allow the compiler for the host language to be used in compiling the DSL. The staged interpreter approach in MetaOCaml requires implementing a parser for the DSL. Template Haskell gives the choice of implementing a parser or reusing the Haskell parser in the context of implementing embedded languages. C++ Template Metaprogramming requires reusing the C++ parser.
5. What does the type system guarantee about *generated programs*? When we represent programs as datatypes in a statically typed language, the only runtime guarantee that the static type system provides is that a value of this datatype is a syntactically well-formed program. In MetaOCaml, a stronger guarantee is possible, namely, that any generated program is well-typed. But it is useful to note that if at generation-time, code-to-code coercions are used to circumvent the MetaOCaml type system, this guarantee is lost *but type safety is not lost*. In particular, because MetaOCaml also type checks all code before compiling it (as do the other two languages), as long as the generated code does not contain coercions, the system is still safe. In addition, when the MetaOCaml type-checker rejects a dynamically generated program (for example because coercions were used when generating it) an OCaml exception is raised, and the DSL implementer can take appropriate provisions to catch and handle such exceptions. In the other languages, any such failure

[8] For a subset of MetaOCaml, it has been proven that all necessary renaming is done [2].

is reported directly to the user of the DSL. As an aside, note that C++ does no type checking on the templates before they are instantiated. Thus, compilation-time runtime errors can occur. Still, the result of instantiation is always syntactically valid.

6. Can code be inspected (intensional analysis)? MetaOCaml discourages code inspection, Template Haskell allows all code inspection, and code inspection in C++ lies between these extremes. First, only a limited form of code inspection can be achieved through expression templates and traits. Second, the scope of transformations in the expression-driven 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, allowing the transformation of several DSL statements at once [24, 46]. For staged interpreters, the ability to inspect generated code is not necessary in order to implement domain-specific optimizations, since those optimizations can be defined directly on the datatype representing the syntax of DSL programs.
7. When can IO be performed? For the embedded DSL approach, IO is essential for reporting errors to the user during analysis and type checking. Template Haskell supports IO at expansion time. Besides error reporting, IO also has other applications such as inspecting the code being generated for debugging purposes (which is not possible in C++), reading additional specifications from some file (e.g., an XML configuration file), or generating code in other languages (e.g., C for interfacing to some native libraries). C++ does not allow any IO during template expansion; 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 [6, p. 669]. Note that a staged interpreter in MetaOCaml is just a program, and so it has access to all IO available in the OCaml language.
8. Are both generator and generated languages the same? A generative language is said to be *homogeneous* when both are identical, and *heterogeneous* otherwise [47]. Homogeneity makes the process of staging a program (including an interpreter) a matter of adding staging annotations. It also makes it easy to cut and paste terms in and out of quotations, and only one set of programming skills is required for learning both the generating and the generated language. In MetaOCaml, only expressions can be quoted. But expressions in MetaOCaml can contain declarations, modules, new classes, etc. Template Haskell can generate expressions and declarations of types, functions, values, classes, and instances. Only expressions and declarations can be spliced in (but not, e.g., patterns). Note, however, that Template Haskell does not allow generating code containing quotes or splices (unlike MetaOCaml). C++ can generate classes (and class hierarchies; see Section 4.3), functions, expressions (using expression templates; see Section 4.4), 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 [6, p. 495].

9. Is cross-stage persistence (CSP) [51] supported? CSP makes it possible to embed values that are outside quotations directly into quoted terms (for example, we can write `.< reverse (append l_1 l_2) >.` even when `reverse` and `append` refer to actual functions defined outside quotations). This means only that the generated code can refer dynamically to the generator environment, and not necessarily that the languages used for both are the same (homogeneity).
10. Is generativity encapsulated? In MetaOCaml, the run construct makes it possible for a library internally to generate and execute code. It is also the case that if all generated code is well-typed, then compilation cannot fail, and the user of the library never needs to deal with obscure error messages. Thus, the user of the library does not need to know that generativity is being used, and we can therefore say that it is well *encapsulated*. In C++, templates can be used to provide libraries that appear to the user as being no different from a non-generative implementation. In Template Haskell, the use of templates in libraries can be encapsulated if these templates do not use parameters that the library user needs to provide. Otherwise, an explicit splice must be used in the application code.
11. Is there support for separate compilation (Modularity)?

6 Discussion and Concluding Remarks

When applicable, embedding can be the easiest way to implement a DSL. Because we may wish to write different applications in different DSLs, embedding may also help interoperability. But this approach may not be applicable if there mismatch in syntax, semantics, optimizations, debugging, or error reporting. Languages such as MetaOCaml, Template Haskell and C++ provide alternative, more flexible approaches to implementing DSLs. This paper has presented a brief introduction to these three languages, along with a comparative analysis of them.

The large number of practical DSL implementations using generative programming in C++ shows that generative programming support is useful—and sometimes necessary—in a mainstream language. This support is especially important for library programmers. The success of generative programming in C++ is remarkable, given that C++ was not designed for DSL implementation. C++ Template Metaprogramming suffers from a number of limitations, including portability problems due to compiler limitations (although this has significantly improved in the last few years), lack of debugging support or IO during template instantiation, long compilation times, long and incomprehensible errors, poor readability of the code, and poor error reporting. Nevertheless, these limitations have not prevented people from creating useful embedded DSLs that would have been impossible otherwise (such as ones for vector computations [55] as well as numerous others that followed). Collections of library primitives to make generative programming in C++ easier are now available [7, p. 433-451], [15].

MetaOCaml and Template Haskell are designed specifically to support generative programming: their capabilities are currently being explored and the languages as well as implementations are still evolving. A collection of small-scale projects developed by Rice University students (including implementations of SSH, parsers, a subset of XSML, and small programming languages) is available online [42]. Experience with Template Haskell is also currently being documented [28, 29, 16], and the implementations of several DSLs in Template Haskell are currently under way, including Hydra'2003 and a parallel skeleton project by Kevin Hammond and colleagues.

Experience with these languages is revealing new challenges. 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 more 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, too. Foundational problems that deserve further exploration in program generation include more expressive type systems, type safe ways for inspecting quoted expressions, and techniques for minimizing generation and compilation times. First generation of generative languages such as the ones discussed in this paper will help us gain practical experience with building real applications. No doubt, this experience will feed back to improve our understanding of how a programming language can provide better support for generative programming.

Acknowledgments. The anonymous reviewers, Simon Helsen, Roumen Kaiabachev, and Kedar Swadi provided valuable comments on drafts of this paper.

References

1. Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: Hardware design in Haskell. *ACM SIGPLAN Notices*, 34(1):174–184, January 1999.
2. Cristiano Calcagno, Walid Taha, Liwen Huang, and Xavier Leroy. Implementing multi-stage languages using asts, gensym, and reflection. In Frank Pfenning and Yannis Smaragdakis, editors, *Generative Programming and Component Engineering (GPCE)*, Lecture Notes in Computer Science. Springer-Verlag, 2003.
3. Rohit Chandra, Leonardo Dagum, and Dave Kohr. *Parallel Programming in OpenMP++*. Morgan Kaufmann, 2000.
4. Antony Courtney. Frappé: Functional reactive programming in Java. In *Proceedings of Symposium on Practical Aspects of Declarative Languages*. ACM, 2001.
5. K. Czarnecki and U. W. Eisenecker. Synthesizing objects. In *Proceedings of ECOOP'99*, LNCS 1628, pages 18–42. Springer-Verlag, 1999.
6. K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.
7. K. Czarnecki and U. W. Eisenecker. Named parameters for configuration generators. http://www.generative-programming.org/namedparams/, 2000.
8. K. Czarnecki, U. W. Eisenecker, R. Glück, D. Vandevoorde, and T. Veldhuizen. Generative programming and active libraries (extended abstract). In M. Jazayeri, D. Musser, and R. Loos, editors, *Generic Programming. Proceedings*, volume 1766 of *LNCS*, pages 25–39. Springer-Verlag, 2000.

9. Simon Peyton Jones (ed.). Haskell 98 language and libraries. *Journal of Functional Programming*, 13(1):1–255, January 2003.
10. Conal Elliott, Sigbjørn Finne, and Oege de Moore. Compiling embedded languages. In *[48]*, pages 9–27, 2000.
11. Conal Elliott and Paul Hudak. Functional reactive animation. In *International Conference on Functional Programming*, pages 163–173, June 1997.
12. Sigbjorn Finne and Simon L. Peyton Jones. Pictures: A simple structured graphics model. In *Proceedings of Glasgow Functional Programming Workshop*, July 1995.
13. Yhoshihiko Futamura. Partial evaluation of computation: An approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.
14. Steven Ganz, Amr Sabry, and Walid Taha. Macros as multi-stage computations: Type-safe, generative, binding macros in MacroML. In *the International Conference on Functional Programming (ICFP '01)*, Florence, Italy, September 2001. ACM.
15. Aleksey Gurtovoy. Boost MPL Library (Template metaprogramming framework). http://www.boost.org/libs/mpl/doc/.
16. 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.
17. Scott Haney, James Crotinger, Steve Karmesin, and Stephen Smith. Pete: The portable expression template engine. *Dr. Dobbs Journal*, October 1999.
18. P. Hudak, T. Makucevich, S. Gadde, and B. Whong. Haskore music notation – an algebra of music. *Journal of Functional Programming*, 6(3):465–483, May 1996.
19. Paul Hudak. Building domain specific embedded languages. *ACM Computing Surveys*, 28A:(electronic), December 1996.
20. Paul Hudak. *The Haskell School of Expression – Learning Functional Programming through Multimedia*. Cambridge University Press, New York, 2000.
21. J. Hughes. Pretty-printing: an exercise in functional programming. In R. S. Bird, C. C. Morgan, and J. C. P. Woodcock, editors, *Mathematics of Program Construction; Second International Conference; Proceedings*, pages 11–13, Berlin, Germany, 1993. Springer-Verlag.
22. G. Hutton. Combinator parsing. *Journal of Functional Programming*, 1993.
23. ISO/IEC. Programming languages – C++. ISO/IEC 14882 Standard, October 2003.
24. Jaakko Järvi and Gary Powell. The lambda library: Lambda abstraction in c++. In *Second Workshop on C++ Template Programming, Tampa Bay, Florida, USA*, October 2001.
25. Neil D. Jones. What not to do when writing an interpreter for specialisation. In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, *Partial Evaluation*, volume 1110 of *Lecture Notes in Computer Science*, pages 216–237. Springer-Verlag, 1996.
26. Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.
27. Shiram Krishnamurti, Matthias Felleisen, and Daniel P. Friedman. Synthesizing object-oriented and functional design to promote re-use. In Eric Jul, editor, *European Conference in Object-Oriented Programming*, volume 1445 of *Lecture Notes in Computer Science*, pages 91–113. Springer Verlag, 1998.
28. Ian Lynagh. Template Haskell: A report from the field. http://web.comlab.ox.ac.uk /oucl/work/ian.lynagh/papers/, May 2003.
29. Ian Lynagh. Unrolling and simplifying expressions with Template Haskell. http://web.comlab.ox.ac.uk /oucl/work/ian.lynagh/papers/, May 2003.

30. John Maddock and Steve Cleary et al. Boost type traits library. http://www.boost.org/libs/type_traits/.
31. John Matthews, Byron Cook, and John Launchbury. Microprocessor specification in Hawk. In *Proceedings of the 1998 International Conference on Computer Languages*, pages 90–101. IEEE Computer Society Press, 1998.
32. M. Mauny. Parsers and printers as stream destructors embedded in functional languages. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pages 360–370. ACM/IFIP, 1989.
33. MetaOCaml: A compiled, type-safe multi-stage programming language. Available online from `http://www.metaocaml.org/`, 2003.
34. Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1), 1991.
35. N. C. Myers. Traits: a new and useful template technique. *C++ Report*, 7(5), June 1995.
36. 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 Distribued Scientific and Engineering Computing with Applications—PDSECA.
37. Chris Okasaki. Even higher-order functions for parsing or why would anyone ever want to use a sixth-order function? *Journal of Functional Programming*, 8(2):195–199, March 1998.
38. Oregon Graduate Institute Technical Reports. P.O. Box 91000, Portland, OR 97291-1000,USA. Available online from `ftp://cse.ogi.edu/pub/tech-reports/README.html`.
39. Emir Pašalić, Walid Taha, and Tim Sheard. Tagless staged interpreters for typed languages. In *the International Conference on Functional Programming (ICFP '02)*, Pittsburgh, USA, October 2002. ACM.
40. J. Peterson, G. Hager, and P. Hudak. A language for declarative robotic programming. In *Proceedings of IEEE Conf. on Robotics and Automation*, 1999.
41. Simon Peyton Jones and Philip Wadler. Imperative functional programming. In *the Symposium on Principles of Programming Languages (POPL '93)*. ACM, January 1993. 71–84.
42. Rice Students. Multi-stage programming course projects. http://www.cs.rice.edu/~ taha/teaching/, 2000.
43. Tim Sheard, Zine El-Abidine Benaissa, and Emir Pašalić. DSL implementation using staging and monads. In *Second Conference on Domain-Specific Languages (DSL'99)*, Austin, Texas, 1999. USENIX.
44. 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.
45. Yannis Smaragdakis and Don Batory. Implementing layered designs with mixin layers. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 550–570. Springer-Verlag LNCS 1445, 1998.
46. Jörg Striegnitz and Stephen Smith. An expression template aware lambda function. In *First Workshop on C++ Template Programming, Erfurt, Germany*, October 2000.
47. Walid Taha. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, 1999. Available from [38].

48. Walid Taha, editor. *Semantics, Applications, and Implementation of Program Generation*, volume 1924 of *Lecture Notes in Computer Science*, Montréal, 2000. Springer-Verlag.
49. Walid Taha. A sound reduction semantics for untyped CBN multi-stage computation. Or, the theory of MetaML is non-trivial. In *Proceedings of the Workshop on Partial Evaluation and Semantics-Based Program Maniplation (PEPM)*, Boston, 2000. ACM Press.
50. Walid Taha and Patricia Johann. Staged notational definitions. In Frank Pfenning and Yannis Smaragdakis, editors, *Generative Programming and Component Engineering (GPCE)*, Lecture Notes in Computer Science. Springer-Verlag, 2003.
51. 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.
52. Walid Taha and Tim Sheard. MetaML: Multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1-2), 2000.
53. Peter Thiemann. Programmable Type Systems for Domain Specific Languages. In Marco Comini and Moreno Falaschi, editors, *Electronic Notes in Theoretical Computer Science*, volume 76. Elsevier, 2002.
54. Ervin Unruh. Prime number computation. Internal document, ANSI X3J16-94-0075/ISO WG21-462, 1994.
55. Todd Veldhuizen and Kumaraswamy Ponnambalam. Linear algebra with C++ template metaprograms. *Dr. Dobb's Journal of Software Tools*, 21(8):38–44, August 1996.
56. Todd L. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, 1995.
57. Todd L. Veldhuizen. Template metaprograms. *C++ Report*, 7(4):36–43, 1995.
58. D. Wile. Popart: Producer of parsers and related tools. system builders' manual. Technical report, USC Information Sciences Institute, 1981.