# Meta-Meta-Programming

## Generating C++ Template Metaprograms with Racket Macros

Michael Ballantyne
University of Utah
mballant@cs.utah.edu

Chris Earl
University of Utah
cwearl@cs.utah.edu

Matthew Might
University of Utah
might@cs.utah.edu

## Abstract

Domain specific languages embedded in C++ (EDSLs) often use the techniques of template metaprogramming and expression templates. However, these techniques can require verbose code and introduce maintenance and debugging challenges. This paper presents a tool written in Racket for generating C++ programs, paying particular attention to the challenges of metaprogramming. The code generator uses Racket's macros to provide syntax for defining C++ metafunctions that is more concise and offers more opportunity for error checking than that of native C++.

## 1. Introduction

Embedded domain specific languages (EDSLs) in C++ have proven to be an effective way to introduce new programming abstractions to fields like scientific computing. implementing C++ EDSLs with a popular technique known as expression templates [16] requires many similar function definitions and operator overloads. The code below shows part of the implementation for the operators + and * from one such EDSL.

```
template<typename LHS, typename RHS>
typename BinExprRetType<SumOp, LHS, RHS>::result
operator+(const LHS & lhs, const RHS & rhs) {
    return binExpr<SumOp>(lhs, rhs);
}

template<typename LHS, typename RHS>
typename BinExprRetType<MultOp, LHS, RHS>::result
operator*(const LHS & lhs, const RHS & rhs) {
    return binExpr<MultOp>(lhs, rhs);
}
```

The details of these implementations are beyond the scope of this paper, but we need to produce this kind of function for each operator in our EDSL, and they're all quite similar. In this case, each differs only by the symbol for the operator (+, *) and the name of the class that implements it (SumOp, MultOp). Expression template EDSL implementations often use C preprocessor macros to reduce this duplication [9, 15]. However, as we discuss in Section 3.4, preprocessor macros scale poorly as our code generation needs become more complex.

For the implementation of Nebo, an EDSL we've published on previously [4, 5], we instead chose to implement a code generator for C++ in Racket. It allows us to describe the implementation of the interface functions once and subsequently generate the C++ code for many operators. For example, we write the following to generate both the interface functions and expression template objects for the operators +, *, and others besides:

```
(build-binary-operator 'SumOp '+
  (add-spaces 'operator '+))
(build-binary-operator 'ProdOp '*
  (add-spaces 'operator '*))
(build-binary-logical-operator 'AndOp '&&
  (add-spaces 'operator '&&))
(build-unary-logical-function 'NotOp '!
  (add-spaces 'operator '!))
(build-extremum-function 'MaxFcn '> 'max)
```

Note that our EDSL provides several types of operator, each with different syntactic rules. Our code generator allows these operators to cleanly share much of their implementation.

Given that we use our code generator to eliminate repetion in interface functions, it would be natural to also generate other components of our EDSL implementation for which the C++ code is difficult to understand and maintain. For example, to provide syntax checking for our EDSL we use template metaprogramming [14] to compute functions from types to types, known as metafunctions. The C++ implementation of one such metafunction is shown in Figure 1. Racket's metaprogramming abilities allow us to write the same metafunction through the following code:

```
(define/meta (join-location l1 l2)
  [('SingleValue 'SingleValue) 'SingleValue]
  [('SingleValue l) l]
  [(l 'SingleValue) l]
  [(l l) l])
```

This paper discusses the design and implementation of our code generator. Specifically, our contributions are:

```
template<typename L1, typename L2 >
 struct JoinLocation;

template< >
struct JoinLocation<SingleValue, SingleValue > {
    SingleValue typedef result;
};

template<typename L >
struct JoinLocation<SingleValue, L > {
    L typedef result;
};

template<typename L >
struct JoinLocation<L, SingleValue > {
    L typedef result;
};

template<typename L >
struct JoinLocation<L, L > {
    L typedef result;
};
```

**Figure 1.** C++ metafunction

- A strategy for generating C++ EDSL implementations with a Racket code generator (Section 3). Our code generator is publicly available at `https://github.com/michaelballantyne/fulmar`. It allows EDSL developers to use Racket as an expressive metaprogramming language while EDSL users continue to write in C++. We show that this approach makes iterative development of EDSLs easier (Section 3.3).
- A EDSL in Racket that corresponds to C++ metafunctions, with concise syntax and integration with our code generation approach (Section 4). Syntactic correctness of the definition and use of the metafunctions is checked at Racket runtime as the C++ implementation of the EDSL is generated. (Section 4.5). The syntax of the EDSL in Racket elucidates the relationship between Scheme-style pattern matching and C++ template metaprogramming (Section 4.3).
- Discussion of the tradeoffs of using the Racket-based code generator as opposed to preprocessor macros in the context of expression template based C++ EDSLs (Section 3.4).

## 2. Expression Templates

C++ provides limited means to transform code at compile time through preprocessor macros and the template system. While the template system was originally designed as a generic programming mechanism, C++ programmers have devised ways to use the object system and compile-time spe-

```
rhs <<= divX( interpX(alpha) * gradX(phi) )
      + divY( interpY(alpha) * gradY(phi) );

phi <<= phi + deltaT * rhs;

phi <<= cond( left,         10.0 )
            ( right,         0.0 )
            ( top || bottom, 5.0 )
            ( phi );
```

**Figure 2.** Iteration of the solution to the 2D heat equation with Nebo

cialization of generic code to achieve more general program transformations. [16]. C++ objects can be used to implement the abstract syntax tree of an embedded domain specific language, while functions and overloaded operators that construct those tree elements define its grammar and type system. This technique is referred to as expression templates (ET) for reasons we'll see shortly.

### 2.1 Deforestation

As an example, consider pointwise addition of vectors:

```
std::vector<int> a, b, c, d;
d = a + b + c;
```

A straightforward way to implement such syntax in C++ would be to overload the + operator to loop over the vectors it receives as arguments and construct a new vector with the result. Given more than one instance of such an operator on the right hand side of an assignment, however, this approach allocates memory proportional to the size of the vectors for each operator call.

Instead, each + operator call can construct an object with an `eval(int i)` method that evaluates the operation at a single index. The object is an instance of a templated class parameterized by the types of its arguments, which may be either `std::vector` or themselves represent parts of a computation like the type of `a + b` above. The loop over indices doesn't happen until the = assignment operator is invoked; it calls the `eval` method on the right hand side for each index in turn and updates the vector on the left hand side.

The templated classes for the objects representing a computation are referred to as expression templates. This particular use of the delayed evaluation they offer corresponds to the deforestation optimizations that Haskell compilers use to remove temporaries from composed list functions [8]. Because the C++ compiler lacks such optimizations, C++ programmers pursuing high performance achieve the same effect with expression templates.

### 2.2 Accelerator Portability with Nebo

Another application of expression templates allows compilation of a single code base for multiple architectures, including accelerators like GPUs and Intel's Xeon Phi [2]. Our C++ EDSL, Nebo, is of this variety [4]. Figure 2 shows a simple use of Nebo. Client code using Nebo is compiled with variants for both CPU and GPU execution, with the decision of which to use being delayed until runtime.

To implement accelerator portability, expression template objects have parallel objects implementing the computation on each device. For the deforestation example the EDSL implementation might include `SumOpEvalCPU` and `SumOpEvalGPU` classes. The initial expression template object constructed by operators and representing the abstract operator has methods that construct these parallel trees. Once the assignment operation has selected a device on which to execute the expression, it calls such a method to obtain the appropriate code variant.

Expression templates for accelerator portability form an extension of the technique used for deforestation. Use of an EDSL handling deforestation might be limited to those computations that most benefit from the optimization. When considering accelerator portability, however, the significant cost of data transfer between accelerator devices and the CPU means it is important to run every calculation on the accelerator. Resultantly, the EDSL must be expressive enough to describe all the performance sensitive calculations in an application. Such EDSLS need many syntactic objects and rules to describe their combination. These rules are encoded in the types of the interface functions or overloaded operators. To extend the deforestation example to allow users to add scalar values to vectors, we'd need to add additional overloads of the `+` operator for each ordering of types: `int` and `SumOp`, `SumOp` and `int`, `int` and `vector`, and `vector` and `int`. Combined with the variants we already had we'd need as many as six implementations of the operator.

We also need many similar classes for the expression template objects. Each different type of operator, like binary expressions of numbers, unary expressions of numbers, binary expressions of booleans, or comparisons of numbers, requires objects for the abstract operator that lacks knowledge of its evaluation architecture, CPU evaluation, and GPU evaluation, among others. The objects needed for each category are similar but meaningfully different.

Implementing these variants becomes overwhelming in a EDSL with many operators and many types of subexpressions. Some elements of this repetitious code can be abstracted away with C++ template metaprogramming, but for a general solution we'll turn to code generation in another language with strong metaprogramming support: Racket.

## 3. Code Generation for C++ EDSLs

Generating code with Racket means we can use a full featured functional programming language for parts of our metaprogramming, with first-class functions, pattern matching, variadic functions, and a rich set of data structures we missed when working with C preprocessor macros.

Code generation for our C++ EDSL presents a unique set of requirements. The purpose of the EDSL is to offer programmers new abstractions within C++ by transforming the expressions they provide at C++ compile time, so we can only use the code generator to produce the implementation of the language. Users of the language are delivered C++ header files containing the template metaprograms that operate on expressions written the EDSL. Furthermore, the EDSL integrates with runtime support code for memory management and threading maintained by C++ programmers. The C++ we generate needs to be human readable so those programmers can understand and debug the interaction of the EDSL with the code they maintain.

Because we're generating C++ source code, we're responsible for:

- Source code formatting for each C++ construct we generate. We need the resulting C++ to be readable, so we need to carefully insert whitespace and line breaks to match programmers' expectations for what well-formatted code looks like. We tried several C++ pretty-printers, but found that when generating code from scratch it worked best to implement this ourselves.
- A well-thought-out representation of each piece of C++ syntax we use. We'd like the code we write with our tool to be high level and easy to understand, so we build syntax constructors as a tower of little languages, with specialized constructors for each complex pattern in our C++ code.

To fill these needs, our implementation builds application and language specific constructs on top of a mostly language-agnostic core for pretty printing based on speculative string concatenation.

### 3.1 Pretty Printer

Our pretty printer transforms a tree of structures to a string. We call the structures in the tree "chunks". Our algorithm is based on speculative concatenation: the speculative chunk gives a default way of constructing a string from its subelements along with an alternate that allows added line breaks. If the string resulting from the default concatenation doesn't exceed the maximum line width, it's accepted. Otherwise the concatenator tries the alternate.

The pretty printer also needs to keep track of indention. Because the pretty printer is composed of mutually recursive functions, we use Racket's `parameterize` to keep track of the indention state in a particular subtree via dynamically scoped variables. It is also slightly language specific in order to handle block comments and indention within them, again with dynamic scope tracking the state within a subtree of chunks.

Our algorithm gets the job done, but it isn't ideal. In some cases a long series of default and alternate concatenations are attempted to complete a single line. We'd like to investigate

```
(define (constructor name
         params assigns . chunks)
  (concat
    name
    (paren-list params)
    (if-empty
      assigns
      (immediate space)
      (surround
        new-line
        (constructor-assignment-list assigns)))
    (apply body chunks)))
```

**Figure 3.** Implementation of constructor chunk

using the ideas of Wadler's "A prettier printer" [17] in the future.

### 3.2 Chunk Constructors

Chunk constructors construct trees of chunks for a particular textual or syntactic construct of the target language. For example `sur-paren` surrounds its arguments in parentheses. Next, `paren-list` builds on top of `sur-paren`, comma separating its arguments and placing them within parentheses. Finally `constructor` handles a constructor definition inside a C++ class, and uses `paren-list` to handle the list of constructor arguments. Figure 3 shows the implementation of `constructor` as an example. Each of the functions called in the definition is a more basic chunk constructor.

### 3.3 Iterative Development

The C++ implementing Nebo is generated by the highest level chunk constructors, and they abstract the patterns found throughout the EDSL implementation to make sure we don't repeat ourselves. For example, each type of EDSL syntax requires implementations of the parallel objects discussed earlier: abstract operator, CPU execution, GPU execution, etc. The set of objects required for each ET is centrally defined in one chunk constructor. As a result, adding a new architectural target for all ET objects has a limited impact on the code base.

The impact of changes to the core C++ interfaces is similarly limited. When the arguments to the `eval` methods shared by every ET object need to change, the change can be made once rather than once for each class. This frees us to make larger changes to the structure of our EDSL implementation more quickly as requirements evolve.

### 3.4 Comparison with Preprocessor Macros

Most C++ EDSLs use function-like C preprocessor macros to satisfy some of the same needs our code generator fills. Each choice has tradeoffs.

One feature of our EDSL needed variants of an ET object for each arity of function we support, and we were looking at supporting functions of up to 10 arguments. Our solution was initially implemented with preprocessor macros and we had a function-like macro implementing the basic ET interface that took 35 arguments, each being a code fragment. Crucially, C preprocessor macros lack lambda expressions and scope for macro names. The Boost Preprocessing library [1] offers a more complete language by building an interpreter inside the preprocessor language. However, our requirements didn't tie us to the preprocessor so we're happier with Racket.

Our approach is also in some ways limiting. We're adding an unfamiliar language to learn and a new tool to run for our EDSL developers, which has limited the accessibility of Nebo's codebase for programmers trained only in C++. At the same time, generating well formatted C++ without a maze of preprocessor directives has improved our implementation's readability.

## 4. Embedding Metafunctions in Racket

When we switched from preprocessor macros to Racket our use of the code generator mimicked the approach we'd used with the preprocessor. By taking advantage of Racket's macros, we can do better. Other authors have noted that partial specialization of C++ templates is a form of pattern matching. In this section we introduce syntax for our code generator that looks like pattern matching on structure types but generates C++ metafunctions that use the pattern matching provided by partial specialization.

### 4.1 Metafunctions and Partial Specialization in C++

C++ metafunctions are a use of C++ templates to perform compile-time computation on types [14]. For example, the code in Figure 4 performs addition on Peano numbers embedded in the type system by `struct Zero` and `struct Succ`.

The first definition of `Add` is called the base template. Its template parameters define the number of parameters the metafunction receives. The remaining definitions are partial specializations of the base template, where the types given in angle brackets following the name of the struct specify the combination of template arguments for which this specialization should be used.

A similar form of computation can be implemented with pattern matching on structures in Racket. Figure 5 shows zero, successor, and addition constructs implemented in such a way. When writing metafunctions in our code generator, we'd like to write syntax similar to Racket's structure pattern matching but generate C++ code like that of Figure 4.

### 4.2 define/meta

We extended our code generator with a new syntactic form, `define/meta`. Figure 6 shows Racket code written with `define/meta` that generates the C++ shown in Figure 4.

`define/meta` can be used to define two types of entities: meta-structs and meta-functions. Meta-structs correspond to

```
struct Zero {} ;

template <typename N>
struct Succ {} ;

template <typename N, typename M>
struct Add {} ;

template<typename NMinusOne, typename M>
struct Add<Succ<NMinusOne>, M> {
typename Add<NMinusOne,
             Succ<M> >::result typedef result;
};

template <typename M>
struct Add<Zero, M> {
    M typedef result;
};
```

**Figure 4.** Add metafunction in C++

```
(struct zero () #:transparent)
(struct succ (n) #:transparent)

(define/match (add m n)
  [((succ n-minus-one) m) (add n-minus-one
                               (succ m))]
  [((zero) m) m])
```

**Figure 5.** Add with Racket structure types

```
(definitions
  (define/meta zero)
  (define/meta succ (n))
  (define/meta (add m n)
    [((succ n-minus-one) m) (add n-minus-one
                                 (succ m))]
    [((zero) m) m]))
```

**Figure 6.** Add metafunction with define/meta

C++ structures with only a base template and no definitions in the structure body. These are essentially compile-time named tuples. Meta-functions correspond to C++ structures that act as functions from types to types. Our convention is that such structures indicate their return value by defining the member `result` as a typedef, as `Add` does in Figure 4.

`define/meta` has three usages to produce these types of entities:

- `(define/meta name)`

  This form defines a meta-struct with no fields. The `name` is converted to a generated identifier appropriate for a C++ type by capitalizing the first letter of each hyphen-separated word and removing hyphens.

- `(define/meta name (fields ...))`

  Like the previous form, but for a structure with fields. The names of the fields are transformed like the meta-struct name for the generated C++ code.

-
  ```
  (define/meta (name args ...)
    [(patterns ...) result-expression]
    ...)
  ```

  This form defines a meta-function. Each clause includes a set of patterns to match against the arguments, and the `result-expression` describes the type that will be given by the `result` field of the generated C++ `struct`.

  Pattern variables defined as part of the `patterns ...` in a clause are bound in the of the `result-expression`. Otherwise, the `result-expression` is a normal expression context, so any functions or macros defined by our code generator are available. The next section describes the rules for pattern matching.

These forms don't directly generate C++ code, but rather bind the given `name` to a Racket struct with information about the meta-struct or meta-function that can later be used to generate C++ code for their declaration, definition, or use. The struct is also directly callable using the `procedure` property of Racket structs [6], producing chunks for a reference to the meta-struct or meta-function. Section 4.4 describes the `definitions` syntax used to generate the code for declarations and definitions. Appendix A provides an implementation of a lambda calculus interpreter in C++ templates via `define/meta` and `definitions` as an extended usage example.

### 4.3 Pattern Matching

The format of the patterns used in meta-function definitions is defined by the following grammar, where `structname` is an identifier bound to a meta-struct definition, *identifier* is any Racket identifier, *symbol* is any Racket symbol, and *string* is any Racket string.

$$\begin{aligned} \mathit{pattern} :=\ & (\texttt{structname}\ \mathit{pattern}_1\ \ldots) \\ & \mid \mathit{identifier} \\ & \mid \mathit{symbol} \\ & \mid \mathit{string} \\ & \mid \_ \end{aligned}$$

Symbols and strings indicate literal C++ types and match only a symbol or string with the same string value. Meta-struct patterns allow further matching in the arguments to the meta-struct. Finally, identifiers bind pattern variables. If an identifier appears more than once in the patterns for a clause, each instance of the identifier refers to the same pattern variable. The clause will only match for arguments where the same C++ type would be bound to each use of the identifier. An underscore indicates a pattern that will match anything but that does not bind a pattern variable.

Unlike the semantics of `match` in Racket or other match forms in Scheme dialects, the order of clauses in a meta-function definition doesn't matter. Rather than resolving situations where more than one pattern matches by selecting the first, C++ and thus meta-functions choose the *most specific*. For the limited C++ we allow in our restricted meta-functions, we can understand pattern A to be more specific than pattern B with respect to a particular input if the pattern for B has non-literal values wherever the pattern for A does, but the pattern for A has literal values in at least one place B has non-literal values.

This is only a partial ordering; as such, there may be cases where there are multiple matching templates with no order between them. Such a circumstance constitutes a user error in the definition and use of the template. We don't yet detect that error in our code generator, but we expect to be able to in the future.

### 4.4 Definitions Syntax

As mentioned before, `define/meta` doesn't actually emit C++ declarations and definitions for meta-structs and meta-functions. Meta-functions can reference each other, so we might not have all the information we need to generate their code until a group of them have been defined. The `definitions` syntactic form is responsible for ordering the declarations and definitions of each meta-function and meta-struct defined or referenced as a sub-form. Figure 6 includes a simple example of its use.

`definitions` is implemented as a macro that uses Racket's `local-expand` to macro expand subforms before processing [7]. This design choice allows for later syntactic extension; if an even higher level syntactic form expands to `define/meta` forms, it will work with `definitions`.

### 4.5 Catching Errors

Our meta-language allows us to catch some errors at code generation time that we couldn't previously. Specifically, if we try to reference an invalid meta-struct in the pattern match or result-expression, or an invalid meta-function in the result-expression, we'll receive an error at Racket runtime indicating that the identifier is not bound. If we misspell `succ` as `suc` in the pattern on line 5 of Figure 6, Racket will produce the following error:

```
suc: unbound identifier in module
  in: suc
```

Similarly, we'll receive an error if we refer to a meta-struct or meta-function with the wrong number of arguments. If we replace `(succ m)` with simply `(succ)` on line 6 of the same example, we receive this error:

```
meta-struct Succ: arity mismatch;
    the expected number of arguments does
      not match the given number
    expected: 1
    given: 0
    arguments:
```

If we didn't catch these errors in our code generator they'd be expressed as template expansion errors at C++ compile time.

## 5. Related Work

Template metaprogramming and expression templates are now nearly two decades old, and there have been many previous efforts to make them more useful and easier to work with. The Boost MPL and Proto libraries are of particular note. Boost MPL [10] offers a variety of algorithms and data structures for template metaprogramming. Boost Proto [12] builds on MPL to allow users to specify and use expression template EDSLs based on a grammar, all at compile time.

Porkoláb and Sinkovics [13] developed a compiler for a subset of Haskell that produces C++ template metaprograms. The compiler supports a functional core including lazy evaluation, currying, recursion, and lambda expressions. It also allows the functions written in Haskell to interoperate with metafunctions written directly in C++. While their approach, like ours, substantially reduces the lines of code required to implement metafunctions, their choice of abstractions leads to increased template recursion depth and compilation time compared to native implementations. In contrast our code generator improves upon native template code only in syntax and error checking and not in choice of abstraction, but doesn't damage the performance of metaprograms. It also integrates into the rest of our code generator for expression templates.

There have also been a number of approaches to accelerator portability with expression templates. Wiemann et al. [18] present an approach that uses expression templates but where the ET tree is walked at runtime and the information within is used to generate CUDA C source code that is then compiled by runtime use of the compiler. Their use of runtime code generation was motivated by the limited support the CUDA C++ compiler offered for templates at that time. Chen et al. [3] expanded upon this approach. To our knowledge, Nebo is the first EDSL to use expression templates for portability between accelerators and CPUs without requiring runtime code generation.

## 6. Future Work

Much of Nebo is still generated by code written in the style of the preprocessor macros from which it was ported. Future work centers around further syntactic extension of our code generator to improve Nebo's maintainability and reduce the cost of developing C++ EDSLs for other domains using the same techniques.

Some of Nebo's language features are implemented by translation to simpler features. For example, Nebo includes a pointwise `cond` implemented by transformation to expression template objects with the functionality of `if`. We'd like to be able to express that transformation with syntax akin to Scheme's `syntax-rules`.

We'd also like to further take advantage of our syntax for template metaprogramming to improve error checking at C++ compile time. Boost MPL [10] includes metafunctions to make compile-time assertions and ensure that failure messages, written as type names, are visible in the C++ compiler's error output. It should be possible to automatically add these static assertions to our metafunction implementations based on type annotations in the Racket syntax. Users of our EDSL could receive better error messages when they misuse syntax without adding an undue burden on us as EDSL implementors.

More ambitiously, we'd like to generate template metaprogramming boilerplate for C++ EDSL implementations from a high-level specification of the grammar and type rules of the EDSL.

## 7. Conclusion

We've found that our code generator simplifies the task of maintaining Nebo. The code generation approach avoids the choice between twin pitfalls: swaths of repetitive code or inscrutable preprocessor macros. Whereas preprocessor macros limited our ability to introduce abstractions, Racket allows us to create new syntax for frequently recurring patterns. It also lets us produce well-formatted C++ that is (relatively) easy to debug and that integrates well with supporting library code.

## 8. Acknowledgements

Many thanks to Will Byrd, Michael Adams, and Celeste Hollenbeck for their helpful feedback on our ideas and extensive comments on our presentation in earlier drafts.

This material is based upon work supported by the National Science Foundation under Grant No. 1337145.

## References

[1] Boost preprocessing library. URL `http://www.boost.org/doc/libs/1_56_0/libs/preprocessor/doc/index.html`.

[2] Intel Xeon Phi product family. URL `http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html`.

[3] J. Chen, B. Joo, W. Watson, and R. Edwards. Automatic offloading C++ expression templates to CUDA enabled GPUs. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 2359–2368, May 2012. doi: 10.1109/IPDPSW.2012.293.

[4] C. Earl. *Introspective pushdown analysis and Nebo*. PhD thesis, University of Utah, 2014.

[5] C. Earl and J. Sutherland. SpatialOps documentation, 2014. URL `http://minimac.crsim.utah.edu:8080/job/SpatialOps/doxygen/`.

[6] M. Flatt. Creating languages in Racket. *Queue*, 9(11):21:20–21:34, Nov. 2011. doi: 10.1145/2063166.2068896.

[7] M. Flatt, R. Culpepper, D. Darais, and R. B. Findler. Macros that work together. *Journal of Functional Programming*, 22: 181–216, 3 2012. doi: 10.1017/S0956796812000093.

[8] A. Gill, J. Launchbury, and S. L. Peyton Jones. A short cut to deforestation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, FPCA '93, pages 223–232, New York, NY, USA, 1993. ACM. doi: 10.1145/165180.165214.

[9] G. Guennebaud and B. J. and others. Eigen v3. http://eigen.tuxfamily.org, 2010.

[10] A. Gurtovoy and D. Abrahams. Boost MPL library (2004).

[11] M. Might. C++ templates: Creating a compile-time higher-order meta-programming language. http://matt.might.net/articles/c++-template-meta-programming-with-lambda-calculus/.

[12] E. Niebler. Proto: A compiler construction toolkit for DSELs. In *Proceedings of the 2007 Symposium on Library-Centric Software Design*, pages 42–51. ACM, 2007.

[13] Z. Porkoláb and Á. Sinkovics. C++ template metaprogramming with embedded Haskell. In *Proceedings of the 8th International Conference on Generative Programming & Component Engineering (GPCE 2009), ACM*, pages 99–108, 2009.

[14] T. Veldhuizen. Template metaprograms. *C++ Report*, 7(4): 36–43, 1995.

[15] T. Veldhuizen. Blitz++ users guide, 2006.

[16] T. L. Veldhuizen. Expression templates. *C++ Report*, 7(5): 26–31, June 1995. ISSN 1040-6042. Reprinted in C++ Gems, ed. Stanley Lippman.

[17] P. Wadler. A prettier printer. *The Fun of Programming, Cornerstones of Computing*, pages 223–243, 2003.

[18] P. Wiemann, S. Wenger, and M. Magnor. CUDA expression templates. In *WSCG Communication Papers Proceedings 2011*, pages 185–192, Jan. 2011. ISBN 978-80-86943-82-4.

## A. Lambda Calculus Interpreter with define/meta

As a usage example of our Racket EDSL, we adapt the lambda calculus interpreter implemented in C++ templates from Might [11].

### A.1 With define/meta

```
(definitions
  ; structs
  (define/meta m-lambda (name body))
  (define/meta app (fun arg))
  (define/meta ref (name))
  (define/meta lit (t))
  (define/meta emptyenv)
  (define/meta binding (name value env))
  (define/meta closure (lam env))
  ; functions
  (define/meta (env-lookup name env)
    [(name (binding name value env))  value]
    [(_    (binding name2 value env)) (env-lookup name env)])
  (define/meta (m-eval exp env)
    [((lit t)              _) t]
    [((ref name)           _) (env-lookup name env)]
    [((m-lambda name body) _) (closure (m-lambda name body) env)]
    [((app fun arg)        _) (m-apply (m-eval fun env)
                                       (m-eval arg env))]])
  (define/meta (m-apply proc value)
    [((closure (m-lambda name body) env) _)
     (m-eval body (binding name value env))]])
```

### A.2 Generated C++

```
template<typename Name, typename Body >
 struct MLambda {};

template<typename Fun, typename Arg >
 struct App {};

template<typename Name >
 struct Ref {};

template<typename T >
 struct Lit {};

struct Emptyenv {};

template<typename Name, typename Value, typename Env >
 struct Binding {};

template<typename Lam, typename Env >
 struct Closure {};

template<typename Name, typename Env >
 struct EnvLookup;

template<typename Exp, typename Env >
 struct MEval;
```

```
template<typename Proc, typename Value >
 struct MApply;

template<typename A, typename B >
 struct MEqual;

template<typename Name, typename Value, typename Env >
 struct EnvLookup<Name, Binding<Name, Value, Env > > { Value typedef result; };

template<typename Gensym7, typename Name2, typename Value, typename Env >
 struct EnvLookup<Gensym7, Binding<Name2, Value, Env > > {
   typename EnvLookup<Gensym7, Env >::result typedef result;
};

template<typename T, typename Gensym8 >
 struct MEval<Lit<T >, Gensym8 > { T typedef result; };

template<typename Name, typename Gensym9 >
 struct MEval<Ref<Name >, Gensym9 > {
   typename EnvLookup<Name, Gensym9 >::result typedef result;
};

template<typename Name, typename Body, typename Gensym10 >
 struct MEval<MLambda<Name, Body >, Gensym10 > {
   Closure<MLambda<Name, Body >, Gensym10 > typedef result;
};

template<typename Fun, typename Arg, typename Gensym11 >
 struct MEval<App<Fun, Arg >, Gensym11 > {
   typename MApply<typename MEval<Fun, Gensym11 >::result,
                   typename MEval<Arg, Gensym11 >::result >::result typedef
   result;
};

template<typename Name, typename Body, typename Env, typename Gensym12 >
 struct MApply<Closure<MLambda<Name, Body >, Env >, Gensym12 > {
   typename MEval<Body, Binding<Name, Gensym12, Env > >::result typedef result;
};
```