Defmacro for C: Lightweight, Ad Hoc Code Generation

Kai Selgrad[1] Alexander Lier[1] Markus Wittmann[2] Daniel Lohmann[1] Marc Stamminger[1]

[1] Friedrich-Alexander University Erlangen-Nuremberg

[2] Erlangen Regional Computing Center

{kai.selgrad, alexander.lier, markus.wittmann, daniel.lohmann, marc.stamminger}@fau.de

ABSTRACT

We describe the design and implementation of CGen, a C code generator with support for Common Lisp-style macro expansion. Our code generator supports the simple and efficient management of variants, ad hoc code generation to capture reoccurring patterns, composable abstractions as well as the implementation of embedded domain specific languages by using the Common Lisp macro system. We demonstrate the applicability of our approach by numerous examples from small scale convenience macros over embedded languages to real-world applications in high-performance computing.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*code generation*; D.2.3 [**Software Engineering**]: Coding Tools and Techniques—*pretty printers*; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*evolutionary prototyping*

General Terms

Design, Languages, Experimentation, Management, Performance

Keywords

Code Generation, Common Lisp, Configurability, Maintenance, Macros, Meta Programming

1. INTRODUCTION

Code generation and its application in domain-specific languages is a long-established method to help reduce the amount of code to write, as well as to express solutions much closer to the problem at hand. In Lisp the former is provided by `defmacro`, while the latter is usually accomplished through its application. In this paper we present a formulation of C (and C-like languages in general) that is amenable to transformation by the Lisp macro processor.

With this formulation we strive towards providing more elegant and flexible methods of code configuration and easing investigation of different variants during evaluation (e.g. to satisfy performance requirements) without additional costs at run-time. The significance of this can be seen by the vast amount of work on variant management [26, 30], code generation and domain-specific languages for heterogeneous systems (e.g. [21, 17]) and code optimization in general [23] in the last years. It is, e.g., mandatory in performance critical applications to reevaluate different versions of an algorithm as required by advances in hardware and systems design (see e.g. [9, 18]). We believe that those approaches to algorithm evaluation will become ever more common in an increasing number of computational disciplines. Our contribution is the description and demonstration of a system that leverages the well established Common Lisp macro system to the benefit of the C family of languages. Additionally this extends to lowering the entry barrier for using meta code by providing a system that is much more suited to ad hoc code generation than current large-scale approaches.

In contrast to stand-alone domain-specific languages that generate C code, such as Yacc [12], most general purpose generative programming methods for C can be placed into two categories: string-based approaches, and systems based on completely parsed and type-checked syntax trees (ASTs). The systems of the former category (e.g. [5, 4]) tend to be suitable for ad hoc code generation, and for simple cases tackling combinatoric complexity (e.g. [9]), but lack layering capabilities (i.e. transforming generated code). Furthermore they suffer from using different languages in the same file (a problem described by [18], too), and thereby encompass problems including complicated scoping schemes. AST-based systems, on the other hand, are very large systems which are not suitable for ad hoc code generation. Even though such systems are very powerful, they are mostly suited for highly specialized tasks. Examples of such larger scopes include product line parameterization [26] and DSLs embedded into syntactically challenging languages [28, 21].

With respect to this classification our approach covers a middle-ground between these two extremes. Our formulation of C facilitates the use of Common Lisp macros and thereby light-weight structural and layered meta programming. Yet, we neither provide nor strive for a completely analyzed syntax tree as this would introduce a much larger gap between the actual language used and its meta code.

Based on our reformulation of C, and tight integration into the Common Lisp system we present a framework that is most suitable for describing domain-specific languages in the C family. We therefore adopt the notion of our system being a meta DSL.

This paper focuses on the basic characteristics of CGen, our implementation of the approach described above. Section 3 presents CGen's syntax and shows simple macro examples to illustrate how input Lisp code is mapped to C code. Section 4 discusses the merits and challenges of directly integrating the CGen language into a Common Lisp system and shows various implementation details. A systematic presentation of more advanced applications with our method is given in Section 5, focusing on how our code generator works on different levels of abstraction and in which way they can be composed. Section 6 evaluates two rather complete and relevant examples found in high performance computing applications [3]. We analyze the abstractions achieved and compare the results to hand-crafted code in terms of variant management, extensibility and maintenance overhead. Section 7 concludes our paper by reflecting our results. Throughout this paper we provide numerous examples to illustrate our generator's capabilities as well as the style of programming it enables.

2. RELATED WORK

Since C is the de facto assembly language for higher level abstractions and common ground in programming and since generative programming [8] is as old as programming itself, there is an enormous amount of previous work in code generation targetting C. We therefore limit our scope to describe the context of our work and describe its relation to established approaches.

Code generation in C is most frequently implemented using the C preprocessor (and template meta programming in C++ [1]). These generators are most commonly used because they are ubiquitous and well known. They are, however, neither simple to use nor easily maintained [25, 8].

The traditional compiler tools, Yacc [12] and Lex [20], generate C from a very high level of abstraction while still allowing for embedding arbitrary C code fragments. Using our framework such applications could be remodelled to be embedded in C (similar to [7]), instead of the other way around. For such specialized applications (and established tools) this may, however, not be appropriate.

Our approach is more comparable to general purpose code generators. As detailed in Section 1 we divide this area into two categories: ad hoc string-based generators, very popular in dynamic languages (e.g. the Python based frameworks Cog [4] and Mako [5]); and large-scale systems (e.g. Clang [28], an extensible C++ parser based on LLVM[19]; Aspect-C++ [26], an extension of C++ to support aspect-oriented programming (AOP)[1]; XVCL [30], a language-agnostic XML-based frame processor) which are most appropriate when tackling large-scale problems. An approach that is conceptually similar to ours is "Selective Embedded JIT Specialization" [6] where C code is generated on the fly and in a programmable fashion and Parenscript [24], an S-Expression notation for JavaScript.

Regarding the entry barrier and our system's applicability to implement simple abstractions in a simple manner our system is close to string and scripting language-based methods. Due to homoiconicity we do not, however, suffer from problems arising because of mixed languages. Furthermore our approach readily supports layering abstractions and modifying generated code to the extent of implementing domain-specific languages in a manner only possible using large-scale systems. The key limitation of our approach is that the macro processor does not know about C types and cannot infer complicated type relations. Using the CLOS [13] based representation of the AST generated internally after macro expansion (see Section 4), any application supported by large-scale systems becomes possible; this is, however, not covered in this paper.

[1]The origins of AOP are from the Lisp community, see [16].

3. AN S-EXPRESSION SYNTAX FOR C

The key component facilitating our approach is a straightforward reformulation of C code in the form of S-Expressions. The following two examples, taken from the classic K&R [14], illustrate the basic syntax.

The first example, shown in Figure 1, is a simple line counting program. Even though the syntax is completely S-Expression-based, it still resembles C at a more detailed level. Functions are introduced with their name first, followed by a potentially empty list of parameters and (notationally inspired by the new C++11 [27] syntax) a return type after the parameter list. Local variables are declared by `decl` which is analogous to `let`.

```
(function main () -> int
  (decl ((int c)
         (int nl 0))
    (while (!= (set c (getchar)) EOF)
      (if (== c #\newline)
          ++nl))
    (printf "%d\n" nl))
  (return 0))
```

```
int main(void) {
    int c;
    int nl = 0;
    while ((c = getchar()) != EOF) {
        if (c == '\n')
            ++nl;
    }
    printf("%d\n", nl);
    return 0;
}

```

Figure 1: A simple line counting program, followed by the C program generated from it.

```
(function strcat ((char p[]) (char q[])) -> void
  (decl ((int i 0) (int j 0))
    (while (!= p[i] #\null)
      i++)
    (while (!= (set p[i++] q[j++]) #\null))))
```

```
void strcat(char p[], char q[]) {
    int i = 0;
    int j = 0;
    while (p[i] != '\0')
        i++;
    while ((p[i++] = q[j++]) != '\0');
}

```

Figure 2: Implementation of the standard library's `strcat`, followed by the generated code.

The resemblance of C is even more pronounced in the second example presented in Figure 2, which shows an implementation of the standard `strcat` function. The use of arrays, with possibly simple expressions embedded, is a shorthand syntax that is converted into the more general `(aref <array> <index>)` notation. This more elaborate notation is required for complicated index computations and ready to be analyzed in macro code.

To illustrate the application of simple macros we show how to add a function definition syntax with the same ordering as in C.

```
(defmacro function* (rt name params &body body)
  `(function ,name ,params -> ,rt
     ,@body))
```

With this the following two definitions are equivalent:

```
(function  foo () -> int (return 0))
(function* int foo () (return 0))
```

As another example consider an implementation of `swap` which exchanges two values and can be configured to use an external variable for intermediate storage. This can be implemented by generating a call to an appropriately surrounded internal macro.

```
(defmacro swap (a b &key (tmp (gensym) tmp-set))
  `(macrolet ((swap# (a b tmp)
                `(set ,tmp ,a
                      ,a   ,b
                      ,b   ,tmp)))
     (lisp (if ,tmp-set
               (cgen (swap# ,a ,b ,tmp))
               (cgen (decl ((int ,tmp))
                       (swap# ,a ,b ,tmp)))))))
```

The `lisp` form is outlined in Section 4. The following examples illustrate the two use cases (input code left, corresponding output code right).

```
(decl ((int x)               int x;
       (int y))              int y;
  (swap x y))                int g209;   // gensym
                             g209 = x;
                             x = y;
                             y = g209;

(decl ((int x)               int x;
       (int y)               int y;
       (int z))              int z;
  (swap x y :tmp z))         z = x;
                             x = y;
                             y = z;
```

Note the use of a gensym for the name of the temporary variable to avoid symbol clashes. More advanced applications of the macro system are demonstrated in Section 5 and 6.

4. IMPLEMENTATION DETAILS

Our system is an embedded domain-specific language for generating C code, tightly integrated into the Common Lisp environment. The internal data structure is an AST which is constructed by evaluating the primitive CGen forms. This implies that arbitrary Lisp forms can be evaluated during the AST's construction; consider e.g. further syntactic enhancements implemented by cl-yacc [7]:

```
(function foo ((int a) (int b) (int c)) -> int
  (return (yacc-parse (a + b * a / c))))
```

All CGen top level forms are compiled into a single AST which is, in turn, processed to generate the desired C output. The following listing shows the successive evaluation steps of a simple arithmetic form which is converted into a single branch of the enclosing AST.

```
(* (+ 1 2) x)
(* #<arith :op '+ :lhs 1 :rhs 2>
   #<name :name "x">)
#<arith :op '*
        :lhs #<arith :op '+ :lhs 1 :rhs 2>
        :rhs #<name :name "x">>
```

Naturally, the implementation of this evaluation scheme must carefully handle ambiguous symbols (i.e. symbols used for Lisp and CGen code), including arithmetic operations as shown in the example above, as well as standard Common Lisp symbols such as `function`, `return`, etc. We chose not to use awkward naming schemes and to default to the CGen interpretation for the sake of convenience. If the Lisp interpretation of an overloaded name is to be used, the corresponding form can be evaluated in a `lisp` form. Similarly the `cgen` form can be used to change back to its original context from inside a Lisp context. This scheme is implemented using the package system. CGen initially uses the `cg-user` package which does not include any of the standard Common Lisp symbols but defines separate versions defaulting to the CGen interpretation. Note that while ambiguous names depend on the current context, unique symbols are available in both contexts.

Considering the above example, we see that the symbol `x` is converted into a node containing the string `"x"`. While Lisp systems usually up-case symbols as they are read, this behavior would not be tolerated with C, especially when the generated code is to interact with native C code. To this end we set the reader to use `:invert` mode case conversion (`:preserve` would not be desirable as this would require using upper case symbol names for all of the standard symbols in most Common Lisp implementations). This scheme leaves the symbol names of CGen code in an inverted state which can easily be compensated for by inverting the symbol names again when they are printed out.

The AST itself is represented as a hierarchy of objects for which certain methods, e.g. traversal and printing, are defined. Naturally, this representation is well suited for extensions. To this end we implemented two different languages which we consider able to be classified as part of the family of C languages. The first language is a notation for CUDA [22], a language used for applying graphics processing units (GPUs) to general purpose computing. Support for CUDA was completed by adding a few node types, e.g. to support the syntax for calling a GPU function from the host side. The second extended C language is GLSL [15], a language used to implement GPU shader code for computer graphics applications. Supporting GLSL was a matter of adding a few additional qualifiers to the declaration syntax (to support handling uniform storage). These examples show how easily our method can be used to provide code for heterogeneous platforms, i.e. to support generating code that can run on different hardware where different (C-like) languages are used for program specification.

As noted previously, our system's AST representation is easily extensible to support any operation expected from a compiler. Our focus is, however, the application of the supported macro system and we therefore leave most of the backend operation to the system's C compiler. Since the AST is only available after macro expansion compilation errors are reported in terms of the expanded code.

5. APPLICATION

In this section we demonstrate how our generator can be applied to a number of different problems. We chose to show unrelated examples on different abstraction levels to illustrate its broad spectrum.

5.1 Ad Hoc Code Generation

A key aspect of our method is the support for ad hoc code generation, i.e. the implementation of localized abstractions as they become apparent during programming.

A simple example of this would be unrolling certain loops or collecting series of expressions. This can be accomplished by the following macro (`defcollector`) which generates macros (`unroll`, `collect`) that take as parameters the name of the variable to use for the current iteration counter, the start and end of the range and the loop body which will be inserted repeatedly.

```
(defmacro defcollector (name list-op)
  `(defmacro ,name ((var start end) &body code)
     `(,',list-op
        ,@(loop for i from start to end collect
             `(symbol-macrolet ((,var ,i))
                ,@code)))))

(defcollector unroll progn)
(defcollector collect clist)
```

The above defined `collect` macro can be used, e.g., to generate tables:

```
(decl ((double sin[360]
         (collect (u 0 359)
           (lisp (sin (* pi (/ u 180.0))))))))
```

The resulting code is entirely static and should not require run-time overhead to initialize the table:

```
double sin[360] = {0.00000, 0.01745, 0.03490, ...};
```

Clearly, many more advanced loop transformation methods could be applied, such as 'peeling' as demonstrated in Section 6.2.

5.2 Configuring Variants

The most straight-forward application of variant-selection is using templates. This can be as simple as providing basic type names, e.g. in a matrix function, and as elaborate as redefining key properties of the algorithm at hand, as shown in the following as well as in Section 6.

Figure 3 shows a rather contrived example where the manner in which a graph is traversed is decoupled from the action at each node. This is not an unusual setup. In our approach, however, there is no run-time cost associated with this flexibility. In this example the traversal method used is given to a macro (`find-max`) which embeds its own code into the body of the expansion of this traversal. This kind of expansion is somewhat similar to compile-time `:before` and `:after` methods.

We assert that having this kind of flexibility without any run-time costs at all allows for more experimentation in performance-critical code (which we demonstrate in Section 6.2). This is especially useful as changes to the code automatically propagate to all versions generated from it, which enables the maintenance of multitudinous versions over an extended period of time. Another application of this technique is in embedded systems where the code size has influence on the system performance and where run-time configuration is not an option.

```
(defmacro find-max (graph trav)
  `(decl ((int max (val (root ,graph))))
     (,trav ,graph
        (if (> (val curr) max)
            (set max (val curr))))))

(defmacro push-stack (v)
  `(if ,v (set stack[++sp] ,v)))

(defmacro preorder-traversal (graph &body code)
  `(decl ((node* stack[N])
          (int sp 0))
     (set stack[0] (root ,graph))
     (while (>= sp 0)
       (decl ((node* curr stack[sp--]))
         ,@code
         (push-stack (left curr))
         (push-stack (right curr))))))

(defmacro breath-first-traversal (graph &body code)
  `(decl ((queue* q (make-queue)))
     (enqueue q ,graph)
     (while (not (empty q))
       (decl ((node* curr (dequeue q)))
         ,@code
         (if (left curr)
             (enqueue q (left curr)))
         (if (right curr)
             (enqueue q (right curr)))))))

(function foo ((graph *g)) -> int
  (find-max g
    preorder-traversal))
```

Figure 3: This example illustrates the configuration of an operation (`find-max`) with two different graph traversal algorithms. Note that this setup does not incur run-time overhead.

5.3 Domain-Specific Languages

To illustrate the definition and use of embedded domain-specific languages we present a syntax to embed elegant and concise regular expression handling in CGen code. Figure 4 provides a very simple implementation with the following syntax.

```
(match text
  ("([^.]*)" (printf "proper list.\n"))
  (".*\."  (printf "improper list.\n")))
```

The generated code can be seen in Figure 5. Note how the output code is structured to only compute the regular expression representations that are required.

```
(defmacro match (expression &rest clauses)
  `(macrolet
     ((match-int (expression &rest clauses)
        `(progn
           (set reg_err (regcomp &reg
                                 ,(caar clauses)
                                 REG_EXTENDED))
           (if (regexec &reg ,expression 0 0 0)
               (progn ,@(cdar clauses))
               ,(lisp (if (cdr clauses)
                          `(match-int
                             ,expression
                             ,@(cdr clauses))))))))
     (decl ((regex_t reg)
            (int reg_err))
       (match-int ,expression ,@clauses))))
```

Figure 4: An example of an embedded domain-specific language for providing an elegant syntax for checking a string against a set of regular expressions.

```
regex_t reg;
int reg_err;
reg_err = regcomp(&reg, "([^.]*)", REG_EXTENDED);
if (regexec(&reg, text, 0, 0, 0))
    printf("proper list.\n");
else {
    reg_err = regcomp(&reg, ".*\.", REG_EXTENDED);
    if (regexec(&reg, text, 0, 0, 0))
        printf("improper list.\n");
}
```

Figure 5: Code resulting from application of the syntax defined in Figure 4.

Clearly, more elaborate variants are possible. Consider, e.g., the popular CL-PPCRE [29] library which analyzes the individual regular expressions and, if static, precomputes the representation. This is not directly applicable to the C regular expression library used here but can be understood as selectively removing the call to `regcomp`.

5.4 Layered Abstractions

One of the canonical examples of aspect-oriented programming is the integration of logging into a system. Without language support it is tedious work to integrate consistent logging into all functions that require it.

Figure 6 presents a macro that automatically logs function calls and the names and values of the parameters, simply by defining the function with a different form:

```
(function  foo (...) ...)                ; does not log
(function+ foo (...) ...)                      ; logs
```

With this definition in place the following form

```
(function+ foo ((int n) (float delta)) -> void
  (return (bar n delta)))
```

evaluates to the requested function:

```
(function foo ((int n) (float delta)) -> void
  (progn
    (printf
      "called foo(n = %d, delta = %f)\n" n delta)
    (return (bar n delta))))
```

With this technique it is easily possible to redefine and combine different language features while honoring the separation of concerns principle. The most simple implementation facilitating this kind of combination would be defining a macro that applies all requested extensions to a given primitive. This could be managed by specifying a set of globally requested aspects which are then integrated into each function (overwriting the standard definition).

```
(defmacro function+ (name param arr ret &body body)
  `(function ,name ,param ,arr ,ret
     (progn
       (printf
         ,(format
            nil "called ~a(~{~a = ~a~^, ~})\n" name
            (loop for item in parameter append
              (list (format nil "~a"
                            (first (reverse item)))
                    (map-type-to-printf
                      (second (reverse item))))))
         ,@(loop for item in parameter collect
             (first (reverse item))))
       ,@body)))
```

Figure 6: Implementation of the logging aspect.

6. EVALUATION

It is hard to overestimate the importance of concise notation for common operations.
B. Stroustrup [27]

As already exemplified in Section 5.3, the quoted text is certainly true, and we agree that the language user, not the designer, knows what operations are to be considered 'common' the best.

In the following we will first present a natural notation for SIMD expressions which are very common in high-performance code. This is followed by an application of our system to a classical problem of high-performance computing which demonstrates how redundancy can be avoided with separation of concerns thereby being applied.

6.1 A Natural Notation for SIMD Arithmetic

SIMD (single instruction, multiple data) is a very common approach to data parallelism, applied in modern CPUs by the SSE [10], AVX [11] and Neon [2] instruction sets. These allow applying a single arithmetic or logic operation (e.g. an addition) to multiple (2, 4, 8, or 16) registers in a single cycle. Naturally, such instruction sets are very popular in high-performance applications where they enable the system to do more work in the same amount of time. The examples in this section will make use of so-called intrinsics, which are functions recognized by the compiler to map directly to assembly instructions.

As an example the following code loads two floating point values from consecutive memory locations into an SSE register and adds another register to it.

```
__m128d reg_a = _mm_load_pd(pointer);
reg_a = _mm_add_pd(reg_a, reg_b);
```

Obviously, more complicated expressions soon become unreadable and require disciplined documentation. Consider, e.g., the expression `(x+y+z)*.5` which would be written as:

```
_mm_mul_pd(
    _mm_add_pd(
        x,
        _mm_add_pd(y, z)),
    .5);
```

There are, of course, many approaches to solving this problem. We compare the light-weightedness and quality of abstraction in our method to a hand-crafted DSL implemented in C using the traditional compiler tools, as well as to an ad hoc code generator framework such as Mako [5]. We argue that the scope of this problem (with the exception of the extreme case of auto-vectorization [17]) does not justify the application of large scale-systems such as writing a source to source compiler using the Clang framework [28].

Traditional Solution.

Our first approach to supply a more readable and configurable notation of SIMD instructions employed traditional compiler technology. The `intrinsify` program reads a file and copies it to its output while transforming expressions that are marked for conversion to intrinsics (after generating an AST for the sub expression using [12] and [20]). The marker is a simple comment in the code, e.g. we transform the following code

```
__m128d accum, factor;
for (int i = 0; i < N; i++) {
    __m128d curr = _mm_load_pd(base + i);
    //#INT accum = accum + factor * curr;
}
```

to produce code that contains the appropriate intrinsics:

```
__m128d accum, factor;
for (int i = 0; i < N; i++) {
    __m128d curr = _mm_load_pd(base + i);
    //#INT accum = accum + factor * curr;
    accum = _mm_add_pd(
                accum,
                _mm_mul_pd(
                   factor,
                   curr
                )
            );
}
```

The instruction set (SSE or AVX) to generate code for can be selected at compile-time.

String-Based Approach.

Using Mako [5] we implemented an ad hoc code generator which runs the input data through Python. In this process the input file is simply copied to the output file and embedded Python code is evaluated on the fly. The previous example is now written as:

```
__m128d accum, factor;
for (int i = 0; i < N; i++) {
    __m128d curr = _mm_load_pd(base + i);
    ${with_sse(set_var "accum"
                   (add "accum"
                        (mul "factor" "curr")))};
}
```

Note how all the data handed to the Python function is entirely string based.

Using CGen.

With our system the extended notation is directly embedded in the source language as follows:

```
(decl ((__m128d accum)
       (__m128d factor))
  (for ((int i 0) (< i N) i++)
    (intrinsify
      (decl ((mm curr (load-val (aref base i))))
        (set accum (+ accum (* factor curr)))))))
```

Comparison.

The implementation of the `intrinsify` program is around 1,500 lines of C/Lex/Yacc code. Using those tools the calculator grammar is very manageable and can be extended in numerous ways to provide a number of different features. Our use case is to automatically convert numeric constants into SIMD format, i.e. converting `//#INT x = 0.5 * x;` to

```
_m128d c_0_500 = _mm_set1_pd(0.5);
x = _mm_mul_pd(c_0_500, x);
```

Since keeping track of names that have already been generated is straight-forward, this is a robust approach to further simplify the notation. Note that it is not easily possible to move such temporaries out of loops as this would require the construction of a rather complete AST which was clearly not the intention of writing such a tool. This example demonstrates that once the initial work is completed such a system can be easily extended and maintained.

The string-based version, on the other hand, is very lightweight and only takes up 60 lines of code. Even though this shows that such abstractions can be constructed on demand and the return on the work invested is obtained very quickly, the resulting syntax is not very far from writing the expressions themselves. The extension to extract numeric constants heavily relies on regular expressions and can only be considered maintainable as long as the code does not grow much larger. Further code inspection and moving generated expressions out of loops is not easily integrated.

The implementation of our `intrinsify` macro consists of 45 lines of code, which is comparable to the Python implementation. The notation, however, is very elegant and convenient and the extraction and replacement of constants are simple list operations. As an example, obtaining the list of numbers in an expression is concisely written as:

```
(remove-duplicates
  (remove-if-not #'numberp (flatten body)))
```

6.2 A Configurable Jacobi Solver

In the field of high performance computing a large class of algorithms rely on stencil computations [3]. As a simple example we consider a 2-D Jacobi kernel for solving the heat equation. Hereby a point in the destination grid is updated with the mean value of its direct neighbors from the source grid. After all points have been updated in this way the grids are swapped and the iteration starts over.

Whereas for the chosen example, shown in Figure 7, state-of-the-art compilers can perform vectorization of the code, they fail at more complicated kernels as they appear, e.g. in computational fluid dynamics. This often leads to hand-crafted and hand-tuned variants of such kernels for several architectures and instruction sets, for example with the use of intrinsics. In all further examples we assume that the alignment of the source and destination grid differ by 8-bytes, i.e. the size of a double precision value.

```
#define I(x, y) (((y) * NX) + (x))
double dest[NX * NY], src[NX * NY], * tmp;

void Kernel(double *dst, double *top,
            double *center, double *bottom,
            int len) {
    for (int x = 0; x < len; ++x)
        dst[x] = 0.25 * (top[x] + center[x-1] +
                         center[x+1] + bottom[x]);
}

void Iterate() {
    while (iterate) {
        for (int y = 1; y < NY - 1; ++y)
            Kernel(&dest[I(1,y)], &src[I(1,y-1)],
                   &src[I(1,y)],  &src[I(1,y+1)],
                   NX-2);
        swap(src, dest);
    }
}
```

Figure 7: A simple 2-D Jacobi kernel without any optimizations applied.

Figure 8 shows how a hand-crafted version using intrinsics targetting SSE may look. In this example double precision floating point numbers are used, i.e. the intrinsics work on two values at a time. At the end of the function there is a 'peeled off', non-vectorized version of the stencil operation to support data sets of uneven width. Even for this very simple example the code already becomes rather complex.

```
void KernelSSE(double *d, double *top, double *center,
               double *bottom, int len) {
    const __m128d c_0_25 = _mm_set_pd(0.25);
    __m128d t, cl, cr, b;

    for (int x = 0; x < len - (len % 2); x += 2) {
        t  = _mm_loadu_pd(&top[x]);
        cl = _mm_loadu_pd(&center[x - 1]);
        cr = _mm_loadu_pd(&center[x + 1]);
        b  = _mm_loadu_pd(&bottom[x]);

        _mm_storeu_pd(&dst[x],
            _mm_mul_pd(
                _mm_add_pd(
                    _mm_add_pd(t, cl),
                    _mm_add_pd(cr, b)),
                c_0_25));
    }

    if (len % 2) {
        int x = len - 1;
        dst[x] = 0.25 * (top[x] + center[x - 1] +
                         center[x + 1] + bottom[x]);
    }
}
```

Figure 8: The same operation as shown in Figure 7 but targetting SSE.

A further optimized version could use the non-temporal store instruction (MOVNTPD) to by-pass the cache when writing to memory, which in turn would require a 16-byte alignment of the store address. This would necessitate a manual update of the first element in the front of the loop if its address is incorrectly aligned. Further, for AVX-variants of the kernel the loop increment becomes four since four elements are processed at once. The peeling of elements in front (for non-temporal stores) and after the loop (for left-over elements) would make further loops necessary.

In the following we show how a simple domain-specific approach can implement separation on concerns, i.e. separate the intrinsics optimizations from the actual stencil used. This frees the application programmer from a tedious reimplementation of these optimizations for different stencils and cumbersome maintenance of a number of different versions of each kernel.

We implemented a set of macros to generate the different combinations of aligned/unaligned and scalar/SSE/AVX kernels in 260 lines of code (not further compacted by meta macros). The invocation

```
(defkernel KernelScalar (:arch :scalar)
  (* 0.25 (+ (left) (right) (top) (bottom))))
```

produces the following kernel, virtually identical to Figure 7:

```
void KernelScalar(double *dst, double *top,
                  double *center, double *bottom,
                  int len) {
    for (int x = 0; x < len; x += 1)
        dst[x] = 0.25 * (center[x-1] + center[x+1]
                         + top[x] + bottom[x]);
    return;
}
```

The invocation of

```
(defkernel KernelSSE (:arch :sse)
  (* 0.25 (+ (left) (right) (top) (bottom))))
```

generates code very similar to Figure 8 (not shown again for a more compact representation), and the most elaborate version, an AVX kernel with alignment, can be constructed using

```
(defkernel KernelAlignedAVX (:arch :avx :align t)
  (* 0.25 (+ (left) (right) (top) (bottom))))
```

The resulting code is shown in Figure 9. Note how for each kernel generated exactly the same input routine was specified. The vectorization is implemented analogous to the method described in Section 6.1. In this version, however, we extracted the numeric constants of the complete function and moved them before the loop.

```
void KernelAlignedAVX(double *dst, double *top,
                      double *center, double *bottom,
                      int len) {
    int x = 0;
    const __m256d avx_c_0_25_2713
                   = _mm256_set1_pd(0.25);
    __m256d avx_tmp1590;
    __m256d avx_tmp1437;
    __m256d avx_tmp1131;
    __m256d avx_tmp1284;
    int v_start = 0;
    while (((ulong)dst) % 32 != 0) {
        dst[v_start] = 0.25 * (center[v_start-1]
                               + center[v_start+1]
                               + top[v_start]
                               + bottom[v_start]);
        ++v_start;
    }
    int v_len = len - v_start;
    v_len = (v_len - (v_len % 4)) + v_start;
    for (int x = v_start; x < v_len; x += 4) {
        avx_tmp1590 = _mm256_load_pd(center[x-1]);
        avx_tmp1437 = _mm256_load_pd(center[x+1]);
        avx_tmp1131 = _mm256_load_pd(top[x]);
        avx_tmp1284 = _mm256_load_pd(bottom[x]);
        _mm256_store_pd(
            &dst[x],
            _mm256_mul_pd(
                avx_c_0_25_2713,
                _mm256_add_pd(
                    _mm256_add_pd(
                        _mm256_add_pd(
                            avx_tmp1590,
                            avx_tmp1437),
                        avx_tmp1131),
                    avx_tmp1284)));
    }
    for (int x = v_len; x < len; ++x)
        dst[x] = 0.25 * (center[x-1] + center[x+1]
                         + top[x] + bottom[x]);
    return;
}
```

Figure 9: The same operation as shown in Figure 7 but targetting aligned AVX and generated by CGen.

7. CONCLUSION

In this paper we presented a code generator that enables Common Lisp-style meta programming for C-like platforms and presented numerous examples illustrating its broad applicability. We also showed how it can be applied to real-world high-performance computing applications. We showed how our approach is superior to simple string-based methods and to what extend it reaches towards large-scale systems requiring considerable initial overhead. Furthermore, we showed that our approach is well suited for lowering the entry barrier of using code generation for situations in which taking the large-scale approach can't be justified and simple string-based applications fail to meet the required demands.

8. ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their constructive feedback and suggestions, Christian Heckl for insightful discussions and gratefully acknowledge the generous funding by the German Research Foundation (GRK 1773).

9. REFERENCES

[1] A. Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied.* Addison-Wesley, 2001.

[2] ARM. Introducing NEON, 2009.

[3] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[4] N. Batchelder. Python Success Stories. http://www.python.org/about/success/cog/, 2014.

[5] M. Bayer. Mako Templates for Python. http://www.makotemplates.org/, 2014.

[6] B. Catanzaro, S. A. Kamil, Y. Lee, K. Asanović, J. Demmel, K. Keutzer, J. Shalf, K. A. Yelick, and A. Fox. SEJITS: Getting Productivity and Performance With Selective Embedded JIT Specialization. Technical Report UCB/EECS-2010-23, EECS Department, University of California, Berkeley, Mar 2010.

[7] J. Chroboczek. *The CL-Yacc Manual*, 2008.

[8] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications.* ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[9] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pages 1–12, Nov 2008.

[10] Intel. *SSE4 Programming Reference*, 2007.

[11] Intel. *Intel Advanced Vector Extensions Programming Reference*, January 2014.

[12] S. C. Johnson. YACC—yet another compiler-compiler. Technical Report CS-32, AT&T Bell Laboratories, Murray Hill, N.J., 1975.

[13] S. E. Keene. *Object-oriented programming in COMMON LISP - a programmer's guide to CLOS.* Addison-Wesley, 1989.

[14] B. W. Kernighan. *The C Programming Language.* Prentice Hall Professional Technical Reference, 2nd edition, 1988.

[15] J. Kessenich, D. Baldwin, and R. Randi. *The OpenGL Shading Language*, January 2014.

[16] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. marc Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP*. SpringerVerlag, 1997.

[17] O. Krzikalla, K. Feldhoff, R. Müller-Pfefferkorn, and W. Nagel. Scout: A source-to-source transformator for SIMD-optimizations. In *Euro-Par 2011: Parallel Processing Workshops*, volume 7156 of *Lecture Notes in Computer Science*, pages 137–145. Springer Berlin Heidelberg, 2012.

[18] M. Köster, R. Leißa, S. Hack, R. Membarth, and P. Slusallek. Platform-Specific Optimization and Mapping of Stencil Codes through Refinement. In *In Proceedings of the First International Workshop on High-Performance Stencil Computations (HiStencils)*, pages 1–6, Vienna, Austria.

[19] C. Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. *See* `http://llvm.cs.uiuc.edu`.

[20] M. E. L. Lesk and E. Schmidt. Lex — a lexical analyzer generator. Technical report, AT&T Bell Laboratories, Murray Hill, New Jersey 07974.

[21] R. Membarth, A. Lokhmotov, and J. Teich. Generating GPU Code from a High-level Representation for Image Processing Kernels. In *Proceedings of the 5th Workshop on Highly Parallel Processing on a Chip (HPPC)*, pages 270–280, Bordeaux, France. Springer.

[22] NVIDIA Corporation. *NVIDIA CUDA C Programming Guide*, June 2011.

[23] M. Pharr and W. Mark. ispc: A SPMD compiler for high-performance CPU programming. In *Innovative Parallel Computing (InPar), 2012*, pages 1–13, May 2012.

[24] V. Sedach. Parenscript. http://common-lisp.net/project/parenscript/, 2014.

[25] H. Spencer and G. Collyer. #ifdef considered harmful, or portability experience with C News. In *Proceedings of the 1992 USENIX Annual Technical Conference*, Berkeley, CA, USA, June 1992. USENIX Association.

[26] O. Spinczyk and D. Lohmann. The design and implementation of AspectC++. *Knowledge-Based Systems, Special Issue on Techniques to Produce Intelligent Secure Software*, 20(7):636–651, 2007.

[27] B. Stroustrup. *The C++ Programming Language, 4th Edition.* Addison-Wesley Professional, 4 edition, May 2013.

[28] The Clang Developers. Clang: A C Language Family Frontend for LLVM. http://clang.llvm.org, 2014.

[29] E. Weitz. CL-PPCRE - Portable Perl-compatible regular expressions for Common Lisp. http://www.weitz.de/cl-ppcre/, 2014.

[30] H. Zhang, S. Jarzabek, and S. M. Swe. Xvcl approach to separating concerns in product family assets. In *Proceedings of the Third International Conference on Generative and Component-Based Software Engineering*, GCSE '01, pages 36–47, London, UK, 2001. Springer-Verlag.