In Search of a Program Generator to Implement Generic Transformations for High-performance Computing

Albert Cohen [a] Sébastien Donadio [b] Maria-Jesus Garzaran [c]
Christoph Herrmann [d] Oleg Kiselyov [e] David Padua [c]

[a] *ALCHEMY group, INRIA Futurs, Orsay, France*
[b] *PRiSM, University of Versailles, France*
[c] *DCS, University of Illinois at Urbana-Champaign, IL, USA*
[d] *FMI, University of Passau, Germany*
[e] *FNMOC, Monterey, CA, USA*

Abstract

The quality of compiler-optimized code for high-performance applications is far behind what optimization and domain experts can achieve by hand. Although it may seem surprising at first glance, the performance gap has been widening over time, due to the tremendous complexity increase in microprocessor and memory architectures, and to the rising level of abstraction of popular programming languages and styles. This paper explores in-between solutions, neither fully automatic nor fully manual ways to adapt a compute-intensive application to the target architecture. By mimicking complex sequences of transformations useful to optimize real codes, we show that generative programming is a practical means to implement architecture-aware optimizations for high-performance applications.

This work explores the promises of generative programming languages and techniques for the high-performance computing expert. We show that complex, architecture-specific optimizations *can* be implemented in a type-safe, purely generative framework. Peak performance is achievable through the careful combination of a high-level, multi-stage evaluation language — MetaOCaml — with low-level code generation techniques. Nevertheless, our results also show that generative approaches for high-performance computing do not come without technical caveats and implementation barriers concerning productivity and reuse. We describe these difficulties and identify ways to hide or overcome them, from abstract syntaxes to heterogeneous generators of code generators, combining high-level and type-safe multi-stage programming with a back-end generator of imperative code.

Key words: Multi-stage programming, loop transformations, adaptive libraries, application-specific program generators.

1 Introduction and Motivation

High-performance computing is maturing. Architecture, compiler and language designs have lately been rather incremental. However, compute-intensive application programmers still complain about the lack of efficiency of their machines — the ratio of sustained to peak performance — and the poor performance of optimizing compilers [1]. Indeed, the path from research prototypes to production-quality optimizers has been more difficult than expected, and too many loop-nest and interprocedural optimizations are still performed manually by application programmers. The main reasons are the following:

- driving and selecting profitable optimizations is increasingly difficult, due to the complexity and dynamic behavior of modern processors [2–4];
- domain-specific knowledge unavailable to the compiler can be required to prove optimizations' legality or profitability [5,6];
- hard-to-drive transformations are not available in compilers, including transformations whose profitability is difficult to assess or whose risk of degrading performance is high, e.g., speculative optimizations [7,8];
- complex loop transformations do not compose well, due to syntactic constraints and code size increase [9];
- some optimizations are in fact algorithm replacements, where the selection of the most appropriate code may depend on the architecture and input data [10].

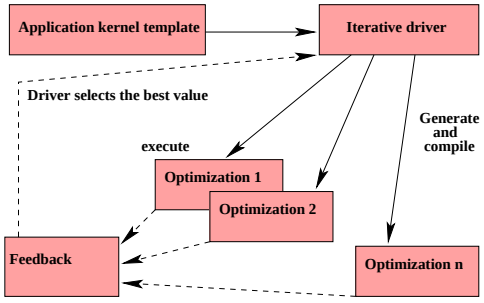

Fig. 1. Iterative search

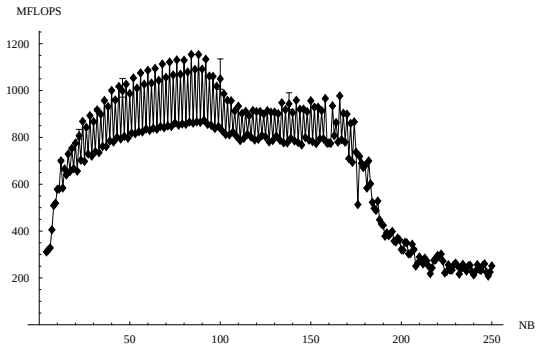

Fig. 2. Influence of parameter selection

Application-specific solutions. It is well known that manual optimizations degrade portability: the performance of a C or Fortran code on a given platform does not say much about its performance on different architectures. Several works have successfully addressed this issue, not by improving the compiler, but through the design of application-specific program generators, a.k.a. active libraries [11]. Such generators often rely on feedback-directed optimization to select the best generation strategy [12], but not exclusively [13]. The most popular examples are ATLAS [1] for dense matrix operations and FFTW [14] for the fast Fourier transform. Such generators follow an iterative optimization scheme, as depicted in Figure 1. In the

case of ATLAS: an external control loop generates multiple versions of the optimized code, varying the optimization parameters such as the tile size of the blocked matrix product, loop unrolling factors, etc. These versions are benchmarked, and the empirical search engine selects the best parameters. Figure 2 shows the influence of the tile size of the blocked matrix product on performance on the AMD Athlon MP processor; as expected from the two-level cache hierarchy of the processor, three main intervals can be identified, corresponding to the temporal reuse of cached array elements; it is harder to predict and statically model the pseudo-periodic variations within each interval, due to alignment and associativity conflicts in caches [13].

Most of these generators use transformations previously proposed for traditional compilers, which fail to apply them for the aforementioned reasons. Conversely, optimizations often involve domain knowledge, from the specialization and inter-procedural optimization of library functions [15,16] to application-specific optimizations such as algorithm selection [10]. Recently, the SPIRAL project [17] investigated a domain-specific extension of such program generators, operating on a domain-specific language of digital signal processing formulas. This project is one step forward to bridge the gap between application-specific generators and generic compiler-based approaches, and to improve the portability of application performance.

Meta-programming optimizations for high-performance. Beyond application specific generators, iterative optimization techniques proved useful in driving complex transformations in traditional compilers. These techniques explore the optimization search space given the feedback (e.g., timing) from the actual executions of the versions of the optimized program, with the help of Operations Research algorithms [2], machine learning [10], and empirical experience [4].

The traditional compiler frameworks might be too low-level however: they operate on program representations such as hierarchical abstract syntax trees [18] or three address SSA form [19] in which the original algorithm is hardly recognizable. They rely on fragile pattern-matching rules, suffer from the lack of natural encoding for semantic equivalence (e.g., algebraic expressions), and resort to heuristic shortcuts in the exploration of (combinatorial) rewrite trees [20].

We thus advocate for a higher-level framework allowing domain experts to design and express their transformations at a level close to that of the original algorithm. The framework should be flexible enough to allow generation of various candidate implementations (to be used later in the empirical search for the best implementation on a particular platform). It should come with a set of built-in optimization strategies (loop tiling, loop fusion, etc.), which can be applied in a platform-independent manner. It must offer some guarantees of correctness of the generated code (e.g., the absence of typing or other compilation errors).

This goal is similar to the one of *telescoping languages* [21,16], a compiler approach to reduce the overhead of calling generic library functions and to enable aggressive interprocedural optimizations, by making the semantic information about these libraries available to the compiler. Beyond libraries, similar ideas have been proposed for domain-specific optimizations [6]. These works highlight the pressing need for researchers and developers in high-performance computing to meta-program their optimizations in a portable fashion.

Alas, the meta-programming community has been slow in responding to these challenges. It has been mostly interested in partial evaluation, specialization and simplification. These transformations are useful, in particular to lower the abstraction penalty; but they are of a different nature than the optimizations (some of which described in this paper) needed to adapt an application to a complex architecture. In fact, research on generative programming and multi-stage evaluation has not greatly influenced the design of high-performance applications and compilers. ATLAS and most of the aforementioned projects are ad-hoc string-based program generators.

Contributions and outline of the paper. This work explores the potential of generative programming languages and techniques for the high-performance computing expert, and, conversely, identifies significant open problems for research in multi-stage programming. We show that complex optimizations can be implemented in a type-safe, purely generative framework. We also show that peak performance is achievable through the careful combination of a high-level, multi-stage language — MetaOCaml [22] — with low-level code generation techniques. Yet, the applicability of generative approaches comes with caveats.

- On the "positive" side, we show that multi-stage programming offers major productivity improvements in the semi-automatic/semi-manual optimization of performance-critical applications. Indeed, large gains *can* be obtained without more general reflective meta-programming (introspection and rewriting).
- On the "negative" side, we rediscover intrinsic limitations of multi-stage evaluation when attempting to mimic a few key transformations in a purely generative style or when composing transformations. We also identify some weaknesses specific to type-safe multi-stage evaluation. In most cases, we show that these weaknesses are no fundamental limitations, but reduce the gains in programmer productivity, or make the approach accessible to functional programming or compilation experts only.
- From these mixed results, we explore possible improvements and motivate further research, using MetaOCaml as a reference multi-stage language. We outline a research direction to eliminate abstraction overhead combining high-level and type-safe multi-stage programming with an embedded back-end generator of imperative code. This approach is similar to offshoring [23,24] and allows to implement a code generator for matrix-matrix product that equals the performance of the best library implementations.

Section 2 introduces typical optimizing transformations in high-performance computing, presents possible MetaOCaml implementations, and discusses their trade-offs. Section 3 takes a larger, extended example — matrix-matrix product — and a more complex set of transformations, typical of the ATLAS library generator. To achieve performance competitive with ATLAS, Section 4 describes a simple set of combinators to eliminate abstraction penalty and generate C code from MetaOCaml with strong safety guarantees.

2 Generative Strategies for Loop Transformations

Program transformations for high-performance principally target regular loop nests operating over arrays. Most of them have been defined for an imperative, intraprocedural setting. Although more and more interprocedural analyses are integrated into modern compilers, few interprocedural extensions of these loop transformations have been proposed [16]. These advanced optimizations often fail to be applied by the compiler, if at all implemented, and some important optimizations would require too much domain specific knowledge or miss important hidden information (in libraries or input data structure) [16].

2.1 Choosing a Purely Generative Approach

From a compiler perspective, there seem to be two means to improve the situation: either the programmer must be given a way to teach the compiler new analyses and optimizations, how to drive them, and possibly when to apply them (overriding static analysis) [25], or the programmer must implement a generator for a class of programs to produce optimized code automatically [6]. The trend in high-performance computing is massively in favor of the second direction, *putting meta-programming techniques at the cornerstone of the design of industrial-strength libraries* [1,14,17].

Very close to our work, the TaskGraph active library [26] provides multi-stage evaluation and loop transformation support for adaptive optimization of high performance codes (numerical and image processing, in particular). We share similar goals with this work, but we impose an additional constraint on the meta-programming that may occur during program generation: *we aim for a purely generative approach, i.e., the generated code is treated as a black box and cannot be examined, let alone optimized further*, whereas the TaskGraph library implements loop transformations of an abstract code representation, embedding a full restructuring compiler in the generator, where we only require back-end code generation. In addition, our purely generative approach avoids pattern mismatches that are likely to occur on the TaskGraph intermediate representation [26].

More fundamentally, static typing is critical for program debugging and verification: it offers correctness guarantees (e.g., the generated code is free from compilation errors), and let the compiler report errors when processing the generator rather than the generated code. Static typing and type inference are available for purely generative languages but seem hard to extend to reflective meta-programming [27]. It is also important for a program manipulation language to support equational reasoning, hence to facilitate the design of advanced analyses and optimizations [27].

For these practical and fundamental reasons, we choose to stick with a purely generative framework throughout the paper. The main goal is to understand the impact of this choice on expressiveness and programmer productivity, in the context of program manipulations for high-performance computing.

2.2 Primitive Transformations

Consider a simple example: loop unrolling. This transformation — also called *partial loop unrolling* — can be decomposed into a bound and stride recomputation step and an iterated body instantiation step. The second step is called *full unrolling* and is a straightforward application of multi-stage evaluation. The code in Figure 3 is a recursive multi-stage implementation of full loop unrolling; `body` is a function whose single argument is the value of the loop counter for the loop being unrolled, and `lb` and `ub` are the loop bounds. To enable substitution with an arbitrary expression, notice argument `i` is a code expression rather than a plain integer. Although quite simple, implementing this transformation has two important caveats.

(1) The right-hand side of the following statement is a code expression *parameterized* by `i` (which is a code value):

```
let body = fun i -> .< a.(.~i) <- .~i >.
```

Such parameterized code expressions are important for specialization (or versioning); [1] partial application allows for several generation steps with multiple specializations.

(2) The arguments of `full_unroll` carry structured information (loop bounds and body), as opposed to its "flat" return value (code expression). Since code values are treated as black boxes, they can only be combined by splicing them — with no adjustments — in a larger code value. Thus it will not be easy to compose `full_unroll` with other generation functions implementing further transformation steps.

The second issue highlights a fundamental problem with program generation, when code expressions produced by a program generator may not evolve beyond the predefined set of arguments of a function or the predefined escape points in the expression. Many optimizations built of sequences of simpler transformations cannot be

[1] No free variable may occur in code expressions.

```
let rec full_unroll lb ub body =
  if lb>ub then .< () >.
  else if lb=ub then body .< lb >.
  else .< begin .~(body .< lb >.); .~(full_unroll (lb+1) ub body) end >.

val full_unroll : int -> int
  -> (('a, int) code -> ('b, unit) code) -> ('b, unit) code = <fun>

let a = Array.make 100 0
let body = fun i -> .< a.(.~i) <- .~i >.
let block = full_unroll 0 3 body

block

.< begin a.(0) <- 0; a.(1) <- 1; a.(2) <- 2; a.(3) <- 3 end >.

.! block
a

- : int array = [|0; 1; 2; 3|]
```

Fig. 3. Full unrolling example. We recall the basic syntax of MetaOCaml: if `expression` is an expression of some type `t`, `.< expression >.` is a *code value* of the type `('a,t) code`; the type parameter `'a` is explained in [28,29]. The notation `.~expression` used within code brackets `.< >.` is an *escape*: the `expression` is evaluated at the generation stage and its result, which must be a code value, is spliced in place of the escape. Code values can be printed by MetaOCaml. We can *run* code values (e.g., execute the corresponding code) using the `.!` notation. The vertical bars on the left mark the output of the MetaOcaml top-level. For clarity, we hide the comment `(* cross-stage persistent value *)` in the MetaOCaml's output; cross-stage persistent values are described in [22].

```
let partial_unroll lb ub factor body =
  let number = (ub-lb+1)/factor in
  let bound = number*factor in
    .< begin
        for ii = 0 to number-1 do
          .~(full_unroll 0 (factor-1) (fun i -> body .< ii*factor+lb + .~i >.))
        done;
        for i = bound+lb to ub do
          .~(body .< i >.)
        done
      end >.

val partial_unroll : int -> int -> int
  -> (('a, int) code -> ('a, unit) code) -> ('a, unit) code = <fun>

let body = fun i -> .< a.(.~i) <- .~i >. in partial_unroll 1 14 4 body

.< begin
  for ii_2 = 0 to (3 - 1) do
    begin
      a.((ii_2*4)+1) + 0 <- ((ii_2*4)+1) + 0; a.((ii_2*4)+1) + 1 <- ((ii_2*4)+1) + 1;
      a.((ii_2*4)+1) + 2 <- ((ii_2*4)+1) + 2; a.((ii_2*4)+1) + 3 <- ((ii_2*4)+1) + 3
    end
  done;
  for i_1 = (12 + 1) to 14 do
    a.((i_1) <- i_1
  done end >.
```

Fig. 4. Partial unrolling

implemented as a composition of generators; in other words changing a transformation sequence may incur much more work than modifying some ordered composition of generator functions. This is a major difficulty since building transformation sequences is key to practical optimization strategies for high performance.

However, this is not the end of the road for our study, since there are alternatives to function composition to reuse the code of primitive generators. For example, to achieve partial unrolling, we only have to convert the original loop bounds into adjusted bounds with a strided interval, then we can call `full_unroll` to iterate body instantiations over this interval; see Figure 4. Section 2.3 will discuss some practical means to reuse code generators despite the difficulty to compose them.

2.3 Sequences of Transformations

Notice the previous example could be a good advocate for multi-stage evaluation. However, unrolling does not cause serious compilation problems today (beyond choosing when and how much to unroll).

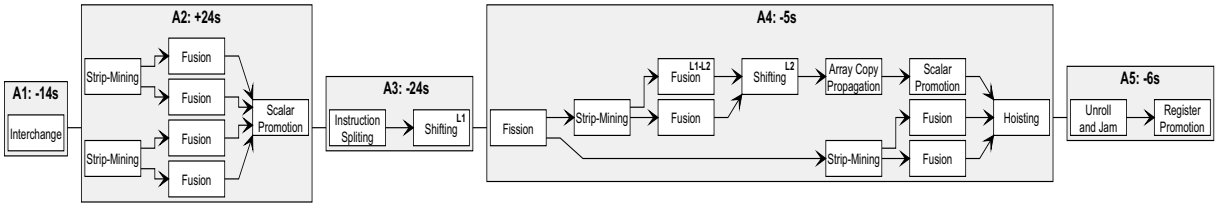

Fig. 5. Optimizing galgel (base 171 s)

Figure 5 describes a real optimization sequence for the galgel SPEC CPU2000 benchmark [30] (borrowed from [4]). This sequence of 23 transformations on the *same program region* (a complex loop nest) was manually applied, following an optimization methodology for feedback-directed optimization [4]. The experimental platform is an HP AlphaServer ES45, 1 GHz Alpha 21264C EV68 with 8 MB L2 cache. Each analysis and transformation phase is depicted as a gray box, showing the time difference when executing the *full benchmark* (in seconds, a negative number is a performance improvement); the base execution time for the benchmark is 171 s. This sequence is out of reach of current compiler technology. Although particularly complex, it is representative of real optimizations performed by some (rare) compilers [31] and some (courageous) programmers [4]. Beyond compositionality, this example also shows how important are extensibility (provisions for implementing new transformations) and debugging support (static and/or generation-time and/or dynamic).

Simpler example. Figure 6 shows a shorter example where complex optimization sequences are necessary: 4 classical loop transformations convert the two simple loop nests at the top of the figure into the bloated code fragment below (partially

Original nests.

```
for j = 1 to n do
  for i = 1 to m do
    a.(i) <- a.(i) + b.(i).(j) * c.(j)
  done
done;
for k = 1 to m do
  for l = 1 to n do
    d.(k) <- d.(k) + e.(l).(k) * c.(l)
  done
done
```

After 4 loop transformations (interchange, fusion, fusion, shifting).

```
let mn = min(m-4, n) in
  for x = 1 to mn do
    a.(1) <- a.(1) + b.(1).(x) * c.(x);
    a.(2) <- a.(2) + b.(2).(x) * c.(x);
    a.(3) <- a.(3) + b.(3).(x) * c.(x);
    a.(4) <- a.(4) + b.(4).(x) * c.(x);
    for y = 1 to mn do
      a.(y+4) <- a.(y+4) + b.(y+4).(x) * c.(x);
      d.(x) <- d.(x) + e.(y).(x) * c.(y)
    done;
    d.(x) <- d.(x) + e.(mn-3).(x) * c.(mn-3);
    d.(x) <- d.(x) + e.(mn-2).(x) * c.(mn-2);
    d.(x) <- d.(x) + e.(mn-1).(x) * c.(mn-1);
    d.(x) <- d.(x) + e.(mn).(x) * c.(mn);
    for y = mn+1 to m-4 do
      a.(y+4) <- a.(y+4) + b.(y+4).(x) * c.(x);
      d.(y) <- d.(y) + e.(y).(x) * c.(y)
    done;
    d.(m-3) <- d.(m-3) + e.(mn-3).(x) * c.(mn-3);
    d.(m-2) <- d.(m-2) + e.(mn-2).(x) * c.(mn-2);
    d.(m-1) <- d.(m-1) + e.(mn-1).(x) * c.(mn-1);
    d.(m) <- d.(m) + e.(mn).(x) * c.(mn)
  done;
  for x = mn+1 to n do
    for y = 1 to m-4 do
      ...
```

Fig. 6. Sequence of loop transformations

shown). These 4 loop transformations are, in application order, loop interchange, loop fusion applied twice, and software pipelining or shifting [32]. Multi-stage evaluation may clearly help the programmer to write a template for the code below, lifting several parameters and generation phases to automatically customize the code for a target architecture. Unfortunately, much of the code in this template turns out too specialized and so cannot be reused in other optimization templates.

Overall, the lack of reusability does limit the applicability of multi-stage evaluation for advanced program transformations. Indeed, generative approaches will only be accepted if they allow programmers to reuse and quickly adapt their generator from one application to another.

There are of course well known solutions to this problem: code reuse and portability can be achieved using an abstract intermediate representation. Higher order skeletons are interesting in that respect: they have been used to define and tune implementation schema for parallel divide and conquer skeletons [33], to build a code

```
let strip_mine lb ub factor body =
  let number = (ub-lb+1)/factor in
    .< begin
        for ii = 0 to number-1 do
          for i = 0 to factor-1 do
            .~(body .< ii*factor+lb+i >.)
          done
        done;
        for i = number*factor+lb to ub do
          .~(body .< i >.)
        done
      end >.

val strip_mine : int -> int -> int
  -> (('a, int) code -> ('a, 'b) code) -> ('a, unit) code = <fun>

let body = fun i -> .< a.(.~i) <- .~i >. in
  strip_mine 1 14 4 body

.< begin
  for ii_8 = 0 to (3 - 1) do
    for i_9 = 0 to (4 - 1) do
      a.(((ii_8 * 4) + 1) + i_9) <- ((ii_8 * 4) + 1) + i_9
    done
  done;
  for i_7 = (12 + 1) to 14 do
    a.(i_7) <- i_7
  done end >.
```

Fig. 7. Strip-mining

```
let g_strip_mine g_loop lb ub factor body =
  let number = (ub-lb+1)/factor in
    .<
      begin
        for ii = 0 to number-1 do
        .~(g_loop 0 (factor-1)
            (fun i-> (body .< ii*factor+lb+ .~i >.)))
        done;
        for i = number*factor+lb to ub do
          .~(body .< i >.)
        done
      end
    >.

let g_loop_gen lw ub body =
  .< for i = lw to ub do .~(body .< i >.) done >.

(* Plain strip-mining *)
let strip_mine = g_strip_mine g_loop_gen
(* Example *)
strip_mine 1 14 4 body

(* Partial unrolling *)
let partial_unroll = g_strip_mine full_unroll
(* Example *)
partial_unroll 1 14 4 body
```

Fig. 8. Factoring strip-mining and unrolling

generator and compiler (HDC) for such schema [34], and recently to build such a code generator using multi-stage programming only [35]. In a more general setting, monadic program composition can simulate operations on an abstract intermediate representation with some cost in readability and performance [36]. Alternatively,

domain-specific languages may combine the advantages of both, see e.g., [37,17] and the survey by Consel in [6].

Composition of generators. The loop unrolling example shows that the design of a complex code generator from primitive ones is not straightforward: code expressions produced by a generator are black boxes that may not be subject to further optimizations, which, as Figure 6 shows, are often necessary. Solutions based on abstract syntaxes and their impact on productivity will be studied later in the paper. Let us first study this limitation on loop unrolling again. We recall that the classical *strip-mining* transformation consists in striping a loop into a number of consecutive blocks of a constant number of iterations, effectively resulting in two nested loops, the outer one iterating on blocks and the inner one iterating within each block. Partial loop unrolling can be decomposed into strip-mining and full unrolling of the inner strip-mined loop; we may thus implement the strip-mine generator shown in Figure 7, hoping that some composition mechanism will allow to define partial unrolling from `strip_mine` and `full_unroll`. Unfortunately, this straightforward approach does not work because `full_unroll` does not operate on a closed code expression but on a pair of integer arguments (the bounds) and a function on code expressions (the body).

Now, the higher-order generators `strip_mine` and `partial_unroll` look very similar (see Figure 7): let aside the recursive implementation of `full_unroll`, partial unrolling seems almost like a *lifted* version of strip-mining where the code expression has been extended from the loop body to the whole inner loop. Based on these similarities, it is indeed possible to *reuse* some code, by factoring the common template of `strip_mine` and `partial_unroll`. This is the purpose of the generalized strip-mining generator in Figure 8. Thanks to the new `g_strip_mine` generator, it is easy to implement *both* the plain strip-mining and partial unrolling, by *composition*; see Figure 8 again. It is even possible to *chain multiple strip-mining and unrolling steps*, with partial applications of `g_strip_mine`.

Beyond composition: code reuse. This brings us to the most important question in our evaluation of generative programming. Function composition is the most natural means to build sequences of loop transformations, yet it is not applicable to most of our generators. But is it the only way to achieve our goal? Since we are primarily interested in productivity, we should consider the wider concept of *generator code reuse*. To overcome the fundamental asymmetry between the structured arguments of generator functions and the flat code expressions they produce, one way is to try to generalize the `g_strip_mine` higher-order generator — also called combinator — approach. Indeed, when the set of transformations to be composed is known for a given application domain, one may aim for a domain-specific set of combinators, where each of those capture a core generative construct: e.g.,

skeleton, control or data structure. In other words, this is the approach of defining generators of code generators and combining them.

This approach requires programming in continuation-passing (or monadic) style, delaying the generation of code expressions to the very last composition of higher-order generators [38,39]. Also, our intuition is that the larger and more expressive the set of transformations, the less semantically rich the associated set of combinators, hence the less productivity gains for the domain developer.

Alternatively, domain-specific generators can be built from primitive code combinators that operate on a suitable *intermediate representation*. That almost looks like the implementation of a domain-specific language compiler, a daunting task with low productivity for the programmer. Yet one may expect many benefits from domain-specific knowledge, from the static checking of MetaOCaml, and from the support of polymorphic and functional code expressions. Furthermore, building an abstract intermediate representation for loop transformations may be as simple as a polyvariant tree of code expressions and loop triplets (bounds and stride).

After carrying this survey, we come to the conclusion that, although theoretically sufficient, a multi-stage programming approach relying on abstract intermediate representations would hardly be satisfactory for most programmers. Indeed, the effort to master the language may not be worth its safety and abstraction. In addition, given the complexity and variety of program transformations for high-performance computing, some implementation errors may not even be avoided by such an approach (including array dependences). The next section develops our argumentation, proposing alternative ways to improve compositionality.

2.4 Alternative Ideas

The polyhedral model has been proposed to represent loop nests and loop transformations in a unified fashion [40–42]. Although constrained by regularity assumptions on control structures, it is applicable to many loop nests [43] in scientific applications and captures loop-specific information as systems of affine inequalities. Recent advances in polyhedral code generation [44] and a new approach to interpret transformation sequences as compositions of affine functions [9] fit very well with our search for a generative and compositional optimization framework. Fortunately, there exists a powerful OCaml interface [45] to the two most effective libraries to operate on polyhedra [46–48]. Together with multi-stage evaluation, it seems very easy and efficient to design a polyhedral code generator and to couple it with lower-level back-end optimizations, including scalar promotion and instruction scheduling. The use of the OCaml language should also facilitate the implementation of the symbolic legality-checking and profitability analyses involved in polyhedral techniques.

Besides the polyhedral model, alternative ways to achieve better compositionality would include some form of reification (to further transform closed code expressions). This may take the form of an extension of OCaml's pattern-matching construct, with type-safe term-rewriting rules (type-invariant substitution, polymorphic type specialization, etc.). We believe it is unlikely that a general-purpose, user-extensible transformation framework can be built with multi-stage evaluation as the only meta-programming primitive. Allowing some user control on the generated code is thus a pragmatic solution to consider. The ability to post-process generated code with pattern-based substitution is a powerful tool, yet transformations operating at an earlier stage are undoubtedly more elegant, especially when they are conducted at an abstract level exposing more algebraic properties of the transformations themselves — like the polyhedral model.

3 Generative Implementation of Complex Optimizations

To experiment with a realistic example, we study the Automatically Tuned Linear Algebra Software (ATLAS) [1], a state-of-the-art, highly optimized library generator for linear algebra operations. It is composed of tens of parameterized code emitters for kernel functions, with an empirical driver to automatically tune the parameters for each target architecture. We focus on the ATLAS code generator for matrix-matrix product [1,13] (see file `emit.c` in `DGEMM`).

3.1 Revisiting a Custom Generator

Figure 9 shows a pseudo-code for the "generative" matrix product in an OCaml-like syntax. In ATLAS, the generator spans more than 3500 lines of C, it is cluttered with `printf`s, obscure index computations, and string substitutions. For the sake of clarity, our pseudo-code and the following experiments operate on integers, whereas `DGEMM` operates on double-precision floats.

The first nest, executed by the generator, computes indirection arrays to build the names of all temporary scalar variables in the innermost computation block. The four nested loops on `ii`, `jj`, `i` and `j` implement the tiled (a.k.a. blocked) matrix product, with a square tile of size `B`. These tiles are further decomposed into smaller blocks of size `MU` and `NU`; the loops iterating on these blocks are fully unrolled, and all temporary results stored into scalar variables. In a first part, values of the result array `c` are loaded in the registers named `c_m_n`. The second part loads one row of `a` and one column of `b` into the set of registers `a_m` and `b_n`, then operates on these registers to compute partial sums for this pair of rows and columns. Loads are decoupled from the actual computations, and additions are decoupled from multiplications, to hide the latency of the L2 cache and of the multiplication. In the third

```
(* executed by the generator *)
h := 0;
for m=0 to MU-1 do
  for n=0 to NU-1 do
    i1.(h) <- m; i2.(h) <- n;
    h := !h+1
  done
done

(* Template of the generated code *)
for ii=0 to N-1 step B do
  for jj=0 to M-1 step B do
    for i=ii to ii+B-NU step NU do
      for j=jj to jj+B-MU step MU do
        (* Two fully unrolled loops to load the values from matrix block *)
        (* c[i..i+MU-1][j..j+NU-1] into promoted scalar variables c_m_n *)
        for m=0 to MU-1 do
          for n=0 to NU-1 do
            c_m_n := c.(i+m).(j+n)
          done
        done;
        for k=0 to B-1 do
          (* All fully unrolled loops to load one pair of rows and columns of matrices a and b *)
          (* then compute their dot product in a pipelined fashion *)
          for m=0 to MU-1 do
            a_m := a.(i+m).(k)
          done;
          for n=0 to NU-1 do
            b_n := b.(k).(j+n)
          done;
          for m=0 to latency-1 do
            t_i1.(m)_i2.(m) := !a_i1.(m) * !b_i2.(m)
          done;
          for m=0 to MU*NU-latency-1 do
            c_i1.(m)_i2.(m) := !c_i1.(m)_i2.(m) + !t_i1.(m)_i2.(m);
            n := m + latency;
            t_i1.(n)_i2.(n) := !a_i1.(n) * !b_i2.(n)
          done;
          for m=MU*NU-latency to MU*NU-1 do
            c_i1.(m)_i2.(m) := !c_i1.(m)_i2.(m) + !t_i1.(m)_i2.(m)
          done
        done;
        (* Two fully unrolled loops to store the values from scalars *)
        (* c_m_n back into matrix block c[i..i+MU-1][j..j+NU-1] *)
        for m=0 to MU-1 do
          for n=0 to NU-1 do
            c.(i+m).(j+n) <- !c_m_n
          done
        done
      done
      (* Postlude... *)
    done
    (* Postlude... *)
  done
  (* Postlude... *)
done
(* Postlude... *)
```

Fig. 9. Simplified matrix product template

part, the results are stored back into array `c`.

Indirection arrays `i1` and `i2` hold the automatically generated names for a large collection of scalar variables; these are working arrays holding the precomputed values for $\lfloor h/\mathtt{NU} \rfloor$ and h mod `NU` for all $0 \leq h < \mathtt{MU} \times \mathtt{NU}$. Figure 9 is a template rather than

a real code: the strange syntax `t_i1.(m)_i2.(m)` represents scalar variables which look like `t_5_7` when the template is expanded with `i1.(m)` $= 5$ and `i2.(m)` $= 7$. The inner loops on `m` and `n` are fully unrolled and operate on these scalar variables.

Starting from the three nested loops of the natural matrix product algorithm, ATLAS applies four transformations to obtain the code in Figure 9. One of those is loop unrolling and is a well known application of multi-stage evaluation. The three other transformations are more original.

Loop tiling. The outer loops of the matrix product are expanded to compute the blocked product [32]. This locality-enhancing transformation is a sequence of strip-mining (making two nested loops out of one) and loop interchange.

None of these transformations makes much use of multi-stage evaluation. It is possible to implement bound and stride operations and loop generation for strip-mining in the same way as for partial unrolling, but this does not compose with loop interchange at the code expression level. Instead, we have to write an ad-hoc generator for the tiled template of the matrix product. We still get the benefits of typed code expressions instead of strings, but we cannot easily reuse the loop tiling code for further applications of the technique on other loop nests.

Scalar promotion. After a second tiling step, the innermost loops are fully unrolled (this is also called unroll and jam [32]) and array accesses in the large resulting blocks are promoted to scalars to enable register reuse (the whole transformation is called register tiling [32]).

This transformation should definitely fit a generative framework since it falls back to straightforward code substitution. However, one may not explicitly craft new variable names in MetaOCaml, since identifiers in `let` or `fun` bindings are not first-class citizens: `let .~name = ...` is not a valid syntax. We know two methods to overcome this limitation.

The first method [38,39] follows a continuation-passing style and assumes the ability to bind intermediate values to fresh variables — in other words, dynamic *single assignment* arrays [49] — defined only once during the execution. Names for these fresh variables can be automatically generated by the MetaOCaml system. This approach has the advantage of directly generating efficient scalar code (unboxed), but it has most of the cons of programming with monads or with explicit continuations, an unnatural style for programmers of high-performance numerical applications. [2]

The second method avoids continuation-passing style and does not require the fresh variable (or single assignment) property, letting an individual scalar-promoted array element element to be overwritten multiple times. [3] It is indeed possible to simulate the mutable array elements of matrices `c` and `t` with a finite

[2] Also syntactic sugar like the monadic `do` notation helps hide conceptual complexity [38,39].

[3] Provided we can assure the legality of the scalar promotion transformation, i.e., the absence of any conflict.

set of boxed variables of type `int ref`. This solution may produce efficient code if the back-end OCaml compiler is able to unbox the scalar references, otherwise it has no impact on the effective number of memory operations performed over the matrix block (i.e., it does not perform any scalar promotion). An example and analysis of this approach will be developed in the next section.

Instruction scheduling. To better hide memory and floating point operation latencies, some instructions in the innermost loops are rescheduled in the loop body and possibly delayed or advanced by a few iterations. This optimization improves the effects of the scheduling heuristic in the back-end compiler.

At first glance, instruction scheduling and software pipelining seem inadequate for generative languages. Yet these optimizations fit in a typical scheduling strategy where instructions are extracted from a priority list and generated in order. It is even possible to write a generic list-scheduler [50] on a dependence graph of code expressions and to extend it to modulo-scheduling [51].

The next section describes the MetaOCaml implementation of a (shorter) type-safe generator for the computational kernel in the tiled, scalar promoted and software-pipelined matrix product.

3.2 Type-Safe Generator for the Matrix Product

The greatest challenge is to use the meta-programming facilities of MetaOCaml to express the compute-intensive kernel in a style in which array accesses are replaced by references to single variables. We expect that an advanced native-code compiler for MetaOCaml will use processor registers for the reference variables.

We define the three combinators in Figure 10.

- `makeCodeRefs` *n* generates a list of *n* code expressions, each one being a distinct value of type `int ref code`, initialized to 0. Each expression will be given a name by the MetaOCaml system at run time.[4] The first implementation of `makeCodeRefs` relies on cross-stage persistence to make each code expression in the list evaluate to a distinct reference at *code generation time*. This is achieved in binding a local variable `ref0` to `ref 0` before staging its value and appending the result to the list; appending `.< ref 0 >.` multiple times would not achieve the expected result: every code expression would evaluate into a unique reference (at run-time). Figure 11 provides an alternative, less intuitive implementation of `makeCodeRefs`: to avoid cross-stage persistence, distinct references in the generated code are explicitly bound within the continuation function `cont` operating on them; this implementation leads to more efficient code due to (current) inefficiencies in the implementation of cross-stage persistence in MetaOCaml.

[4] The names for reference variables are constructed internally by the MetaOCaml system.

```
let rec makeCodeRefs n =
  if n = 0 then []
  else let ref0 = ref 0 in
    (.< ref0 >.) :: makeCodeRefs (n-1)

val makeCodeRefs : int -> ('a, int ref) code list

let dynamicFor low high step f =
  .<
    let i = ref .~low in
      while !i <= .~high do
        (.~f) (!i);
        i := !i + .~step
      done
  >.

val dynamicFor : ('a, int) code -> ('a, int) code
  -> ('a, int) code -> ('a, int -> 'b) code -> ('a, unit) code

let staticFor low high step f =
  let rec gen a =
    if a > high then .< () >.
    else .<begin .~(f a); .~(gen (a+step)) end>.
  in gen low

val staticFor : int -> int -> (int -> ('a, unit) code)
  -> ('a, unit) code
```

Fig. 10. Array of code and loop iteration combinators

```
let makeCodeRefs' n cont =
  let rec loop n acc cont =
    if n = 0
    then cont acc
    else .<let t = ref 0 in .~(loop (n-1) (.<t>. :: acc) cont)>.
  in loop n [] cont
```

Fig. 11. Boxed scalars without cross-stage persistence

- `dynamicFor` creates a `for`-loop environment in which loops can have a stride greater than or equal to 1; it is based on OCaml `while`-loops. `dynamicFor` takes the lower bound, upper bound, stride and a function which abstracts the loop body in the iteration variable. All arguments of `dynamicFor` are code expressions since they depend on dynamic values.
- `staticFor` expresses loops of stride one which are to be fully unrolled in the program. `staticFor` takes a static lower and upper bound and a function for the loop body which maps the (static) index to the code expression of the loop body.

Figure 12 shows the scalar-promoted loop program for the matrix multiplication without the body of the loop nest. The static arguments are the unrolling factors `mu` and `nu`, and the software pipeline `latency`. The dynamic arguments are listed in the lambda-abstraction `fun a b c nn mm nb`. For each of the arrays `a`, `b`, `t` and `c` we apply the `makeCodeRefs` combinator and define functions `aa`, `bb`, `tt` and `cc` which translate each access to an array into the access to the appropriate register.

To use the improved variant of `makeCodeRefs`, it is sufficient to replace the four lines defining `aregs`, `bregs`, `tregs` and `cregs` by the code in Figure 13 (and close the 4 additional parentheses in the end). This does not really complicate the code for

```
let matmult mu nu latency =
  let i1 = Array.make (mu*nu) 0
  and i2 = Array.make (mu*nu) 0
  and h = ref 0 in
    for m=0 to mu-1 do
      for n=0 to nu-1 do
        i1.(!h) <- m;
        i2.(!h) <- n;
        h := !h+1
      done
    done;
    let aregs = makeCodeRefs mu
    and bregs = makeCodeRefs nu
    and tregs = makeCodeRefs (mu*nu)
    and cregs = makeCodeRefs (mu*nu) in
    let aa i = nth aregs i
    and bb i = nth bregs i
    and tt i j = nth tregs (i*nu+j)
    and cc i j = nth cregs (i*nu+j) in
      .< fun a b c nn mm nb ->
        .~(dynamicFor .<0>. .<nn-1>. .<nb>. .<fun ii ->
          .~(dynamicFor .<0>. .<mm-1>. .<nb>. .<fun jj ->
            .~(dynamicFor .<ii>. .<ii+nb-nu>. .<nu>. .<fun i ->
             .~(dynamicFor .<jj>. .<jj+nb-mu>. .<mu>. .<fun j ->
               (* Loop nest body in Figure~14 *)
             >.) >.) >.) >.) >.

val matmult : int -> int -> int ->
  ('a, int array array -> int array array -> int array array ->
   int -> int -> int -> int -> unit) code
```

Fig. 12. Scalar promotion — loop template

a programmer familiar with continuation-passing style, but is not as straightforward as the first, cross-stage persistent version (which was less efficient).

```
(* ... *)
makeCodeRefs' mu (fun aregs ->
makeCodeRefs' nu (fun bregs ->
makeCodeRefs' (mu*nu) (fun tregs ->
makeCodeRefs' (mu*nu) (fun cregs ->
(* ... *)
```

Fig. 13. More efficient variant without cross-stage persistence

The loop nest body is depicted in Figure 14. In a first part, values of the result array `c` are loaded in the register set accessed by `cc`. The second part operates on these registers to compute partial sums for every pair of rows in `a` and columns in `b`. In the third part, the results are stored back into the array. The second part is the most interesting; it consists of a dynamic loop on `k`, whose body consists of sequences of statements, which are expressed here in terms of our combinator `staticFor`. First, elements from the arrays `a` and `b` are loaded into the register sets indexed through functions `aa` and `bb`, respectively. The temporary register set indexed through function `tt` is filled with initial products, then expresses the pipelined dot product of every pair of rows and columns. At last, when all products have been computed, the results are committed into to the register set indexed through function `cc`.

Figure 15 shows a snippet of the output of the `matmult` generator (the optimized version without cross-stage persistence). One easily recognizes initialization code

```
.~(staticFor 0 (mu-1) 1 (fun m ->
    staticFor 0 (nu-1) 1 (fun n ->
      .< .~(cc m n) := c.(i+m).(j+n) >.)));
for k=0 to nb-1 do
  .~(staticFor 0 (mu-1) 1 (fun m ->
      .< .~(aa m) := a.(i+m).(k) >.));
  .~(staticFor 0 (nu-1) 1 (fun n ->
      .< .~(bb n) := b.(k).(j+n) >.));
  .~(staticFor 0 (latency-1) 1 (fun m ->
      .< .~(tt (i1.(m)) (i2.(m)))
        := !(.~(aa (i1.(m))))
          * !(.~(bb (i2.(m)))) >.));
  .~(staticFor 0 (mu*nu-latency-1) 1 (fun m -> let n = m+latency in
      .< begin
        .~(cc (i1.(m)) (i2.(m)))
          := !(.~(cc (i1.(m)) (i2.(m))))
            + !(.~(tt (i1.(m)) (i2.(m))));
        .~(tt (i1.(n)) (i2.(n)))
          := !(.~(aa (i1.(n))))
            * !(.~(bb (i2.(n))))
         end >.));
  .~(staticFor (mu*nu-latency) (mu*nu-1) 1 (fun m ->
      .< .~(cc (i1.(m)) (i2.(m)))
        := !(.~(cc (i1.(m)) (i2.(m))))
          + !(.~(tt (i1.(m)) (i2.(m)))) >.))
done;
.~(staticFor 0 (mu-1) 1 (fun m ->
    staticFor 0 (nu-1) 1 (fun n ->
      .< c.(i+m).(j+n) <- !(.~(cc m n)) >.)))
```

Fig. 14. Scalar promotion — loop body

for temporary variables in the fully unrolled block of the matrix product.

```
(*  ...  *)
  let i_53 = (ref jj_49) in
    while ((! i_53) <= ((jj_49 + mb_47) - 4)) do
      ((fun j_51 ->
          begin
            begin
              t_40 := (c_43.(i_50 + 0)).(j_51 + 0);
              t_39 := (c_43.(i_50 + 0)).(j_51 + 1);
              t_38 := (c_43.(i_50 + 0)).(j_51 + 2);
(*  ...  *)
              t_24 := ((! t_4) * (! t_8));
              t_23 := ((! t_4) * (! t_7));
              t_22 := ((! t_4) * (! t_6));
(*  ...  *)
```

Fig. 15. Scalar promotion — generated code

3.3 Evaluation and Feedback from the Case Study

At the time of the experiments, no stable native code MetaOCaml compiler was available. Therefore we did not try to measure the performance of the code directly, since the interpretation overhead is at least of the same order of magnitude as the architectural phenomena we are trying to optimize for.

However, the generated code expressions “look” very much like the output of the C code emitted by ATLAS. We thus bet that, provided the abstraction penalty of exe-

cuting of MetaOCaml programs can be eliminated, our type-safe, purely generative implementation can match the performance of the custom, string-based generator of ATLAS. The next section will show our early solution to completely eliminate the abstraction penalty and confirm this analysis.

Besides performance, an important issue is code readability and debugging. Static type-checking is a great asset for developing robust meta-programs. Nevertheless, writing a generative template for the tiled, unrolled, scalar promoted and pipelined matrix product can be a trial and error experience. The assembly of parameterized code combinators with complex index expressions is hard to follow for high-performance computing experts used to an imperative flow of algebraic operations.

We did not investigate interprocedural optimizations beyond classical applications of generative languages: specialization, cloning and inlining. More complex transformations may even combine loop and function transformations [16]. Yet addressing the issues raised by matrix-matrix product is a necessary step, and we believe it will contribute to understand how to support interprocedural transformations.

We did not study the legality of the code transformations either. This issue is of course important, although not as critical as in a stand-alone compiler where the decision of considering a program transformation has to be fully automatic. If we were to check for array or scalar dependences [32] or to evaluate the global impact of data layout transformations, the OCaml type system would be insufficient. Coupling a multi-stage generator with a static analysis framework seems like an interesting research direction.

4 Safe Meta-Programming in C

There exist several two-stage evaluation extensions of the C language, from the standard C preprocessor to C++ template meta-programming [52,26], and to the ‘C project [53] based on a fast run-time code generation framework. None of these tools provide the higher order functions on code expressions, cross-stage persistence and static checking features of MetaOCaml.

It seems possible to design a multi-stage C extension with most of these features, probably with some restrictions on the language constructs allowed at the higher stages, but that would be a long-term project. Using MetaOCaml as a generator for C programs seems more pragmatic, at least as a research demonstrator:

(1) this may either involve an OCaml-to-C translator, taking benefit of the imperative features of OCaml to ease the generation of efficient C code;
(2) or one may design a small preprocessor to embed C code fragments into the lower stages of a MetaOCaml program, relying on a *higher level abstract*

syntax built on *code generation combinators* [54] to assemble these fragments.

The first solution is very cumbersome when aiming at the generation of stand-alone "C only" programs. It is unlikely that such an OCaml-to-C translator can ever be designed, without simultaneously lacking robustness and performance. In the context of multi-stage evaluation, *offshoring* is an original, very promising variant whose development is conducted within MetaOCaml itself [23,24]. Instead of a pure translation, offshoring maps code expressions in a subset of OCaml to a subset of C, providing a "homogeneous look" to the heterogeneous combination of OCaml and C execution stages.

We plan to study this offshoring approach in the future, but currently follow on the second direction, for two reasons:

Pure generative programming: using a higher-level abstract syntax avoids transforming generated code; offshoring is also based on the expansion of OCaml constructs into imperative ones, yet this comes with a few restrictions in expressiveness [24];

Expressiveness and control: an abstract syntax can provide full C expressiveness through the explicit combination of higher-order generators of imperative code in MetaOCaml, eliminating all risks of hidden overhead (operating on OCaml values, marshaling and unmarshaling) in the translation of high level constructs.

To avoid hidden overheads, we assume a complete separation of the two languages: the code generation part of is written in MetaOCaml, and does not exchange values with the target application written only in C.

The rest of this section outlines the main technical ideas of this approach and describes its application to the matrix-matrix product example. It is not the main point of the paper, but a technical step to make the MetaOCaml code generators eventually produce machine code without any abstraction penalty. The following abstract combinators support a subset of the C language, and we cannot guarantee safety with respect to all syntactic and semantic requirements of C compilation. Rather, these abstract combinators are useful to translate all imperative constructs needed in our experiments into pure C code.

4.1 Safe Generation of C code in OCaml

Our goal is to design a set of C code generation primitives such that *an OCaml program calling these primitives may only generate syntactically sound and type-safe C code*. This goal is less ambitious than the *offshoring approach* [23], since we do not aim at transparent communication of OCaml values to staged C code and vice-versa (which involves Marshaling in [23]). Instead, we assume complete separation of the OCaml and C worlds: only the code generation part of is written

in MetaOCaml, the target application being a C (or Fortran) only program.

Of course, instead of calling generation primitives explicitly, it would be natural to design a preprocessor converting embedded C syntax into the proper OCaml declarations and calls. This does not bear any fundamental difficulty and is left for future development work.

Our current solution does not aim at generating the whole C syntax (it has very limited support for function declarations and calls), and it is not even guaranteed to be fully safe: it should rather be seen as a partially coercive environment for generating safe code, with the ability for a hacker to circumvent the soundness restrictions, but limiting the risk for bugs to be unwillingly introduced. A safer solution would necessarily include abstract types and heavy usage of the module system [39].

```
type environment = {
  mutable txt:string; (* C text being produced *)
  mutable cnt:int; (* Private counter for alpha-renaming *)
  ind:string (* Pretty printing (indentation) *)
}
(* Some syntactic elements of C *)
type 'a c_rvalue = RValue of string * 'a
type 'a c_lvalue = LValue of string * 'a
type 'a c_loop = Cloop of ('a c_rvalue) * (bool c_rvalue)
                   * ('a c_rvalue) * (environment -> unit)

let get_nam var = match var with LValue (n, _) -> n
let get_def var = match var with LValue (_, d) -> d
let get_txt exp = match exp with RValue (t, _) -> t
let get_val exp = match exp with RValue (_, v) -> v
let get_init var = match var with Cloop (n, _, _, _) -> n
let get_cond var = match var with Cloop (_, c, _, _) -> c
let get_iter var = match var with Cloop (_, _, i, _) -> i
let get_body var = match var with Cloop (_, _, _, b) -> b

(* Support function: turns an lvalue into an rvalue *)
let lvrv var = RValue (get_nam var, get_def var)
(* Support function: build a loop object *)
let create_loop (init : 'a c_rvalue) (cond : bool c_rvalue)
  (iterator : 'a c_rvalue) block =
  Cloop (init, cond, iterator, block)
```

Fig. 16. Some support functions for C generation

Technical overview. Practically, we mirror the C grammar productions as generator functions operating on specific polyvariant types to represent meaningful C fragments. Each C variable is embedded into an OCaml pair of a string (its name) and a dummy value matching the type of the C variable; these variables can only be declared through the explicit usage of OCaml variables. Figures 16 and 17 show the main types, support functions and generation primitives (for intraprocedural constructs only) and Figure 18 shows a code generation example.

By design, the grammar productions guarantee syntactic correctness. It is more challenging to deal with the scope of the C variable declarations and to enforce

```
(* Output the declaration of <nam> to <env> *)
let gen_int_decl nam cont env =
  let var = nam ^ "_" ^ (string_of_int env.cnt) in
  let txt = sprintf "int %s;\n" var in
    env.txt <- env.txt ^ env.ind ^ txt;
    env.cnt <- env.cnt + 1;
    cont (LValue (var, 0))
(* Output an instruction built of expression <exp> to <env> *)
let gen_inst exp env =
  let txt = sprintf "%s;\n" (get_txt exp) in
    env.txt <- env.txt ^ env.ind ^ txt

(* Output a block to <enclosing>
   Notice <block> is a function on environments, this allows
   to defer the evaluation of the generators in the block *)
let gen_block block e =
  let env = {txt=e.ind ^ "{\n"; cnt=0; ind=e.ind ^ "  "} in
    block env;
    env.txt <- env.txt ^ e.ind ^ "}\n\n";
    e.txt <- e.txt ^ env.txt

(* The following productions are self-explanatory *)

let gen_loop loop env =
  env.txt <- env.ind ^ "for (" ^ (get_txt (get_init loop))
    ^ "; " ^ (get_txt (get_cond loop)) ^ "; "
    ^ (get_txt (get_iter loop)) ^ ")\n";
  gen_block (get_body loop) env
let gen_if_then_else (cond : bool c_rvalue) b_then b_else env =
  env.txt <- env.txt ^ "if (" ^ (get_txt cond) ^ ")\n";
  gen_block b_then env;
  env.txt <- env.txt ^ "else\n";
  gen_block b_else env
let gen_int_cst (cst : int) =
  let txt = sprintf "%d" cst in RValue (txt, cst)
let gen_assign (var : 'a c_lvalue) (exp : 'a c_rvalue) =
  let txt = sprintf "%s = %s" (get_nam var) (get_txt exp)
  in RValue (txt, exp)
let gen_lt (op1 : 'a c_rvalue) (op2 : 'a c_rvalue) =
  let txt = sprintf "(%s < %s)" (get_txt op1) (get_txt op2)
  in RValue (txt, true)
let gen_add (op1 : 'a c_rvalue) (op2 : 'a c_rvalue) =
  let txt = sprintf "(%s + %s)" (get_txt op1) (get_txt op2)
  in RValue (txt, (get_val op1) + (get_val op2))
```

Fig. 17. Some generation primitives for C

type-safety. We choose to represent a C block (the only placeholder for variable declarations) as a *function on an environment*. Interestingly, environments both record the code being generated and serve to delay the evaluation of the generation primitives until after the production of the surrounding C block syntax. This continuation passing style is the key to the embedding of C scoping and typing rules into the OCaml ones. For example, an integer variable `x` is simultaneously declared to OCaml and generated to C code when evaluating

`let x = gen_int_decl "x" env in` *`continuation`*.

C code is appended to the environment `env`. Further assignments to `x` may call `gen_assign` directly, and read references to `x` must first turn it into an “rvalue” through the `lvrv` function. Technically, we make use of partial evaluation to η-reduce the environment argument when composing generators. This leads to a syntax closer to the syntax tree these generators reflect, reducing the chances to corrupt

```
let global = {txt=""; cnt=0; ind=""};;

let generated_block =
  let block env =
    let x = gen_int_decl "x" env
    and y = gen_int_decl "y" env in
      gen_loop
        (create_loop
           (gen_assign x (gen_int_cst 1))
           (gen_lt (lvrv x) (gen_int_cst 10))
           (gen_assign x (gen_add (lvrv x) (gen_int_cst 1)))
           (gen_loop
              (create_loop
                 (gen_assign y (gen_int_cst 2))
                 (gen_lt (lvrv y) (gen_int_cst 20))
                 (gen_assign y (gen_add (lvrv y) (gen_int_cst 2)))
                 (fun env ->
                    let z = gen_int_decl "z" env in
                      gen_inst (gen_assign z (gen_int_cst 5)) env))))
        env;
      gen_inst
        (gen_assign y (gen_add (lvrv x) (gen_int_cst 3)))
        env;
      gen_if_then_else (gen_lt (gen_int_cst 2) (lvrv x))
        (gen_inst (gen_assign x (gen_int_cst 5)))
        (gen_inst (gen_assign x (gen_int_cst 3)))
        env
  in gen_block block global

let _ =
  print_string global.txt
```

```
{
  int x_0;
  int y_2;
  for (x_0 = 1; (x_0 < 10); x_0 = (x_0 + 1))
  {
    for (y_2 = 2; (y_2 < 10); y_2 = (y_2 + 2))
    {
      int z_0;
      z_0 = 5;
    }
  }
  y_2 = (x_0 + 3);
  if ((2 < x_0))
  {
    x_0 = 5;
  }
  else
  {
    x_0 = 3;
  }
}
```

Fig. 18. Example of C generation

the C code by mixing multiple environments. A systematic application of this η-reduction approach incurs rewriting all generators as environment to environment transformers; this is not a trivial task since many generators already have a return value whose type is key to the enforcement of safe C code generation [39].

Finally, Figure 20 also shows an interesting use of MetaOCaml to stage the generator templates themselves. Indeed, the use of more than two stages is very convenient for heterogeneous program generation.

It is now easy to support the generation of new C variable names, without explicit usage of monads or continuation-passing style [38], and avoiding the possible overhead of the solution proposed in Section 3.2. Figure 19 shows a class to operate on scalar-promoted arrays with *generation-time* array bound checking. Of course, this cannot be done statically in general (although specific cases could be handled, with the proper encoding in the OCaml type system): scalar promoting an array with out-of-bound accesses yields generation-time out-of-bound exceptions. For example, a scalar-promoted array `a` is simultaneously declared to OCaml and generated as C code when evaluating `let a = new sp_int_array "a" 10`, and it may further be referenced through the method `idx`.

```
class sp_int_array = fun nam siz ini env ->
  let _ = for i = 0 to siz-1 do
            env.txt <- env.txt ^ (sprintf "int %s_%d;\n" nam i)
          done in
object
  val name = nam
  val size = siz
  method idx x =
    if (0 <= x & x < siz)
      let var = sprintf "%s_%d" name x in
        LValue (var, 0)
    else
      failwith "Illegal reference to " ^ nam ^ "[" ^ x ^ "]"
end
```

Fig. 19. Support for scalar-promotion

4.3 Application to ATLAS

Figure 20 highlights the most significant parts of the implementation of a generator template for the matrix-matrix product. In this simple implementation, instruction scheduling is done by hand through an explicit split into three separate unrolled loops. Also, empirical search strategies to drive the optimization (e.g., simulated annealing) are not included.

Our generator produces a C code that closely resemble the C program generated by ATLAS (but operates on integers rather than double-precision floats). Provided with the optimal parameters selected by the ATLAS empirical optimization driver, this C code achieves more than 80% of the peak performance of the target processor. [5] A naive implementation delivers only 8% of the peak performance. These results were obtained for small and medium matrix sizes (less than 192×192), on two different architectures: an AMD Athlon XP 2800+ (Barton) at 2.08GHz with 512KB L2 cache, and a AMD Athlon 64 3400+ (ClawHammer) at 2.2GHz with 1MB L2

[5] For this application, the peak performance is the absolute measure of the maximal number of multiplications per second that the target processor can compute.

```
(*  ...  *)

let q = new sp_int_array "q" 20 env
and r = new sp_int_array "r" 20 env
and t = new sp_int_array_array "t" 20 20 env

(*  ...  *)

in let block_mab1 it env =
  gen_block
    (gen_inst (gen_assign (t#idx i1.(it) i2.(it))
      (gen_mul (lvrv (q#idx i1.(it))) (lvrv (r#idx i2.(it))))))
    env

(*  ...  *)

in let multiply env =
  for m = 0 to mu-1 do
    for n = 0 to nu-1 do
      i1.(!h) <- m;
      i2.(!h) <- n;
      h := !h + 1
    done
  done;
  .! full_unroll 0 (latency - 1)
    (fun i -> .< block_mab1 .~i env >.);
  .! full_unroll 0 (mu * nu - latency - 1)
    (fun i -> .< block_mab2 .~i env >.);
  .! full_unroll (mu * nu - latency) (mu * nu - 1)
    (fun i -> .< block_mab3 .~i env >.)

(*  ...  *)
```

Fig. 20. Revisited matrix product template

cache. Performance on larger matrices is lower (60% of the peak) since our generator does not support the necessary page-copying step when iterating across tiles, an important optimization to reduce the pressure on the Translation Look-aside Buffer (TLB); there is no difficulty implementing this additional step in our framework.

Overall, we reach a level of performance almost identical to highly tuned vendor libraries (90% of the peak). Yet our generator is approximately 15× smaller than the ATLAS one, which has more than 3500 lines of intricate string-manipulation C code. Our generator is also guaranteed against compilation errors, much more amenable to future evolutions and lightweight maintenance, and slightly easier to understand.

4.4 Planned Extensions

Currently, support for function declarations and calls is awkward. Like for scalar types, specific generator functions for each function type must be implemented. Three OCaml functions are needed per C function type: one for the function prototype, one for the declaration and one for the call. These generators operate on a polyvariant `'c_function` type and enforce the proper prototype/declaration/call ordering of the C language (assuming the generation of a single-file program). It is

still unclear whether more polymorphism can be achieved, possibly requiring one triple of generators for each given number of arguments only.

The next step is of course to write the preprocessor for OCaml-embedded C code, to reduce the burden of writing code like the block generator in Figure 20.

5 Conclusion and Perspectives

We revisited some classical optimizations in the domain of high-performance computing, discussing the benefits and caveats of multi-stage evaluation. Since most advanced transformations are currently applied manually by domain and optimization experts, the potential benefit of generative approaches is very high. Indeed, several projects have followed ad-hoc string-based generation strategies to build adaptive libraries with self-tuned computation kernels. We took one of these projects as a running example. We show that MetaOCaml is suitable for the implementation of this kind of application-specific generators, with the benefit of its static checking properties to increase the productivity and portability of manual optimizations. But our results also show that severe reusability and compositionality issues arise for complex optimizations, when multiple transformation steps are applied to a given code fragment. In addition, we show that static type checking in MetaOCaml — in the current state of the system — somewhat restricts the usage of important optimizations like the scalar promotion of arrays. Nevertheless, we also show that on all our examples, this difficulty can be overcome in practice, at the price of additional coding complexity — monadic style — or relying on a smarter back-end compiler — unboxing and offshoring.

Let us summarize our findings. Although multi-stage evaluation can be used in a simple and effective way to implement program generators for portable high-performance computing applications, it does not relieve the programmer from reimplementing the main generator parts for each target application. The only way to improve code reuse in the generator is to base its design on a custom intermediate representation, which may be almost as convoluted for application programmers as designing their own compiler. These academic results are confronted with a concrete example, matrix-matrix product in the ATLAS adaptive library. We show that competitive performance is achievable with a high-level, type-safe program generation approach, provided the domain expert has a good technical practice with functional programming and with optimizations to lower the abstraction penalty. However, we were not able — up to now — to fully eliminate the abstraction penalty with MetaOCaml alone. We had to resort to a heterogeneous multi-stage approach to produce lower level C code. While bringing safe meta-programming to low-level imperative programming, this approach also helps lifting some restrictions of MetaOCaml, but delays some safety checks to the C compiler.

In the future, we plan to use MetaOCaml as an experimental framework to evaluate multi-stage evaluation and possibly extend it to better support our adaptive optimization needs. We also wish to further investigate the potential of the heterogeneous MetaOCaml/C generator, combining it with a preprocessor to embed actual C code in MetaOCaml. Finally, to strengthen the safety guarantees of complex transformations, we are interested in coupling multi-stage evaluation with more expressive static analyses of data dependences and pointer aliasing. These future works may take place in a meta-programming extension for imperative languages, on top of LLVM [55], an advanced compiler infrastructure for lifelong, adaptive program transformation.

Acknowledgments. This work is supported by exchange programs between the French CNRS, the University of Illinois, and the German DAAD. We wish to thank Paul Feautrier, Vikram Adve, Marc Snir, Changhao Jiang, Patrick Meredith, Xiaoming Li, Chris Lengauer and Walid Taha. We are also indebted to the anonymous reviewers for this special issue and for the first MetaOCaml workshop; they pointed out important references and contributed elegant solutions to some issues raised in the original version of this paper.

References

[1] R. C. Whaley, A. Petitet, J. J. Dongarra, Automated empirical optimizations of software and the ATLAS project, Parallel Computing 27 (1–2) (2001), pp. 3–35.

[2] T. Kisuki, P. Knijnenburg, M. O'Boyle, H. Wijshoff, Iterative compilation in program optimization, in: Proc. CPC'10 (Compilers for Parallel Computers), 2000, pp. 35–44.

[3] K. D. Cooper, D. Subramanian, L. Torczon, Adaptive optimizing compilers for the 21st century, J. of Supercomputing 23 (1) (2002), pp. 7–22.

[4] D. Parello, O. Temam, A. Cohen, J.-M. Verdun, Towards a systematic, pragmatic and architecture-aware program optimization process for complex processors, in: ACM Supercomputing'04, Pittsburgh, Pennsylvania, 2004, p. 15,

[5] A. J. C. Bik, M. Girkar, P. M. Grey, X. Tian, Automatic intra-register vectorization for the Intel architecture, Int. J. of Parallel Programming 30 (2) (2002), pp. 65–98.

[6] C. Lengauer, D. Batory, C. Consel, M. Odersky (Eds.), Domain-Specific Program Generation, no. 3016 in LNCS, Springer-Verlag, 2003.

[7] D. I. August, D. A. Connors, S. A. Mahlke, J. W. Sias, K. M. Crozier, B.-C. Cheng, P. R. Eaton, Q. B. Olaniran, W.-M. Hwu, Integrated predicated and speculative execution in the IMPACT EPIC architecture, in: Proceedings of the 25th Intl. Symp. on Computer Architecture, 1998, pp. 227–237.

[8] L. Rauchwerger, D. Padua, The LRPD test: Speculative run–time parallelization of loops with privatization and reduction parallelization, IEEE Transactions on Parallel

and Distributed Systems, Special Issue on Compilers and Languages for Parallel and Distributed Computers 10 (2) (1999), pp. 160–180.

[9] A. Cohen, S. Girbal, O. Temam, A polyhedral approach to ease the composition of program transformations, in: Euro-Par'04, no. 3149 in LNCS, Springer-Verlag, Pisa, Italy, 2004, pp. 292–303,

[10] X. Li, M.-J. Garzaran, D. Padua, A dynamically tuned sorting library, in: ACM Conf. on Code Generation and Optimization (CGO'04), San Jose, CA, 2004, pp. 111–124.

[11] T. Veldhuizen, D. Gannon, Active libraries: Rethinking the roles of compilers and libraries, in: SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing, 1998, pp. 21–23.

[12] M. D. Smith, Overcoming the challenges to feedback-directed optimization, in: ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization, 2000, pp. 1–11, (Keynote Talk).

[13] K. Yotov, X. Li, G. Ren, M. Cibulskis, G. DeJong, M. Garzaran, D. Padua, K. Pingali, P. Stodghill, P. Wu, A comparison of empirical and model-driven optimization, in: ACM Symp. on Programming Language Design and Implementation (PLDI'03), San Diego, CA, 2003, pp. 63–76.

[14] M. Frigo, S. G. Johnson, FFTW: An adaptive software architecture for the FFT, in: Proc. of the ICASSP Conf., Vol. 3, 1998, pp. 1381–1384.

[15] L. De Rose, D. Padua, Techniques for the translation of MATLAB programs into Fortran90, ACM Trans. on Programming Languages and Systems 21 (2) (1999), pp. 286–323.

[16] A. Chauhan, K. Kennedy, Optimizing strategies for telescoping languages: procedure strength reduction and procedure vectorization, in: ACM Intl. Conf. on Supercomputing (ICS'04), 2001, pp. 92–101.

[17] M. Püschel, B. Singer, J. Xiong, J. Moura, J. Johnson, D. Padua, M. Veloso, R. W. Johnson, SPIRAL: A generator for platform-adapted libraries of signal processing algorithms, Journal of High Performance Computing and Applications, special issue on Automatic Performance Tuning 18 (1) (2004), pp. 21–45.

[18] Open research compiler, `http://ipf-orc.sourceforge.net`.

[19] J. Merill, GENERIC and GIMPLE: a new tree representation for entire functions, in: Proceedings of the 2003 GCC Developers Summit, 2003, pp. 171–180, `http://www.gccsummit.org/2003`.

[20] A. Cohen, S. Girbal, D. Parello, M. Sigler, O. Temam, N. Vasilache, Facilitating the search for compositions of program transformations, in: ACM Intl. Conf. on Supercomputing (ICS'05), Boston, Massachusetts, 2005, pp. 151–160,

[21] K. Kennedy, Telescoping languages: A compiler strategy for implementation of high-level domain-specific programming systems, in: Proc. Intl. Parallel and Distributed Processing Symposium (IPIPS'00), 2000, pp. 297–304.

[22] C. Calcagno, W. Taha, L. Huang, X. Leroy, Implementing multi-stage languages using ASTs, Gensym, and reflection, in: ACM SIGPLAN/SIGSOFT Intl. Conf. Generative Programming and Component Engineering (GPCE'03), 2003, pp. 57–76.

[23] J. Ekhardt, R. Kaiabachev, K. Swadi, Offshoring: Representing C and Fortran90 in OCaml, in: 1st MetaOCaml Workshop (informal proceedings), 2004.

[24] J. Ekhardt, R. Kaiabachev, E. Pašalić, K. Swadi, W. Taha, Implicitly heterogeneous multi-stage programming, in: ACM SIGPLAN/SIGSOFT Intl. Conf. Generative Programming and Component Engineering (GPCE'05), no. 3676 in LNCS, 2005.

[25] D. J. Quinlan, M. Schordan, Q. Yi, B. R. de Supinski, Semantic-driven parallelization of loops operating on user-defined containers, in: Workshop on Languages and Compilers for Parallel Computing, Vol. 2958 of LNCS, Springer-Verlag, 2003, pp. 524–538.

[26] O. Beckmann, A. Houghton, P. H. J. Kelly, M. Mellor, Run-time code generation in C++ as a foundation for domain-specific optimisation, in: Proceedings of the 2003 Dagstuhl Workshop on Domain-Specific Program Generation, 2003.

[27] W. Taha, A sound reduction semantics for untyped CBN mutli-stage computation. or, the theory of MetaML is non-trival, in: Proc. of the ACM workshop on Partial Evaluation and semantics-based Program Manipulation (PEPM'00), Boston, Massachusetts, 2000, pp. 34–43.

[28] W. Taha, M. F. Nielsen, Environment classifiers, in: ACM Symp. on Principles of Programming Languages (PoPL'03), 2003, pp. 26–37.

[29] C. Calcagno, E. Moggi, W. Taha, ML-like inference for classifiers, in: D. Schmidt (Ed.), European Symposium on Programming (ESOP), no. 2986 in LNCS, Springer-Verlag, 2004, pp. 79–93.

[30] Standard performance evaluation corporation, `http://www.spec.org`.

[31] KAP C/OpenMP for Tru64 UNIX and KAP DEC Fortran for Digital UNIX, `http://www.hp.com/techsevers/software/kap.html`.

[32] R. Allen, K. Kennedy, Optimizing Compilers for Modern Architectures, Morgan and Kaufman, 2002.

[33] C. A. Herrmann, C. Lengauer, Parallelization of divide-and-conquer by translation to nested loops, J. of Functional Programming 9 (3) (1999), pp. 279–310.

[34] C. A. Herrmann, C. Lengauer, HDC: A higher-order language for divide-and-conquer, Parallel Processing Letters 10 (2/3) (2000), pp. 239–250.

[35] C. A. Herrmann, Generating message-passing programs from abstract specifications by partial evaluation, Parallel Processing Letters 15 (3) (2005), pp. 305–320.

[36] S. Liang, P. Hudak, M. Jones, Monad transformers and modular interpreters, in: ACM Symp. on Principles of Programming Languages (PoPL'95), 1995, pp. 333–343.

[37] A. Van Deursen, P. Klint, J. Visser, Domain-specific languages: An annotated bibliography, ACM SIGPLAN Notices 35 (6) (2000), pp. 26–36.

[38] O. Kiselyov, K. N. Swadi, W. Taha, A methodology for generating verified combinarorial circuits, in: Embedded Software Conf. (EMSOFT'04), Pisa, Italy, 2004, pp. 249–258.

[39] J. Carrette, O. Kiselyov, Multi-stage programming with functors and monads: eliminating abstraction overhead from generic code, in: ACM SIGPLAN/SIGSOFT Intl. Conf. Generative Programming and Component Engineering (GPCE'05), no. 3676 in LNCS, 2005.

[40] P. Feautrier, Some efficient solutions to the affine scheduling problem, part II, multidimensional time, Int. J. of Parallel Programming 21 (6) (1992), pp. 389–420, see also Part I, one dimensional time, 21(5):315–348.

[41] M. E. Wolf, Improving locality and parallelism in nested loops, Ph.D. thesis, Stanford University, published as CSL-TR-92-538 (Aug. 1992).

[42] W. Kelly, Optimization within a unified transformation framework, Tech. Rep. CS-TR-3725, University of Maryland (1996).

[43] C. Bastoul, A. Cohen, S. Girbal, S. Sharma, O. Temam, Putting polyhedral loop transformations to work, in: Workshop on Languages and Compilers for Parallel Computing (LCPC'03), LNCS, College Station, Texas, 2003, pp. 23–30,

[44] C. Bastoul, Code generation in the polyhedral model is easier than you think, in: Parallel Architectures and Compilation Techniques (PACT'04), Juan-les-Pins, 2004, pp. 7–16.

[45] P. Boulet, X. Redon, SPPoC: Symbolic parameterized polyhedral calculator, `http://www.lifl.fr/west/sppoc`.

[46] P. Feautrier, Parametric integer programming, RAIRO Recherche Opérationnelle 22 (1988), pp. 243–268.

[47] V. Loechner, D. Wilde, Parameterized polyhedra and their vertices, Int. J. of Parallel Programming 25 (6), `http://icps.u-strasbg.fr/PolyLib`.

[48] W. Pugh, A practical algorithm for exact array dependence analysis, Communications of the ACM 35 (8) (1992), pp. 27–47.

[49] P. Feautrier, Array expansion, in: ACM Intl. Conf. on Supercomputing, St. Malo, France, 1988, pp. 429–441.

[50] A. Darte, Y. Robert, F. Vivien, Scheduling and Automatic Parallelization, Birkhaüser, Boston, 2000.

[51] J. Llosa, Swing modulo scheduling: A lifetime-sensitive approach, in: Parallel Architectures and Compilation Techniques (PACT'96), 1996, pp. 80–87.

[52] T. Veldhuizen, Using C++ template metaprograms, C++ Report 7 (4) (1995), pp. 36–43.

[53] M. Poletto, W. C. Hsieh, D. R. Engler, M. F. Kaashoek, 'C and tcc: A language and compiler for dynamic code generation, ACM Trans. on Programming Languages and Systems 21 (2) (1999), pp. 324–369.

[54] D. A. Turner, A new implementation technique for applicative languages, Software – Practice and Experience 9 (1) (1979), pp. 31–49.

[55] C. Lattner, V. Adve, LLVM: A compilation framework for lifelong program analysis & transformation, in: ACM Conf. on Code Generation and Optimization (CGO'04), San Jose, CA, 2004, pp. 75–88.