Multi-stage programming with functors and monads: eliminating abstraction overhead from generic code

Jacques Carette[1] and Oleg Kiselyov[2]

[1] McMaster University, 1280 Main St. West, Hamilton, Ontario Canada L8S 4K1
[2] FNMOC, Monterey, CA 93943

Abstract. With Gaussian Elimination as a representative family of numerical and symbolic algorithms, we use multi-stage programming, monads and Ocaml's advanced module system to demonstrate the complete elimination of the abstraction overhead while avoiding any inspection of the generated code. We parameterize our Gaussian Elimination code to a great extent (over domain, matrix representations, determinant tracking, pivoting policies, result types, etc) at no run-time cost. Because the resulting code is generated just right and not changed afterwards, we enjoy MetaOCaml's guaranty that the generated code is well-typed. We further demonstrate that various abstraction parameters (aspects) can be made orthogonal and compositional, even in the presence of name-generation for temporaries and other bindings and "interleaving" of aspects. We also show how to encode some domain-specific knowledge so that "clearly wrong" compositions can be statically rejected by the compiler when processing the generator rather than the generated code.

1 Introduction

In high-performance, symbolic, and numeric computing, there is a well-known issue of balancing between maximal performance and the level of abstraction at which code is written. Furthermore, already in linear algebra, there is a wealth of different aspects that *may* need to be addressed. For example, implementations of the widely used Gaussian Elimination (GE) algorithm — the running example of our paper — may need to account for the representation of the matrix, whether to compute and return the detrminant or rank, how and whether search for pivot, etc. Furthermore, current architectures demand more and more frequent tweaks which, in general, cannot be done by the compiler because the tweaking often involves domain knowledge. A survey [3] of Gaussian elimination implementations in an industrial package Maple found 6 different aspects and 35 different implementations of the algorithm, as well as 45 implementations of directly related algorithms. We can manually write each of these implementations optimizing for particular aspects and using cut-and-paste to "share" similar pieces of code. We can write a very generic GE procedure that accounts for all the aspects with appropriate abstractions [13]. The abstraction mechanisms however – be they procedure, method or a function call – have a significant cost, especially for high-performance numerical computing [3].

A more appealing approach is generative programming [7, 30, 22, 26, 14, 33]. The approach is not without problems, e.g., making sure that the generated code is well-formed. This is a challenge in string-based generation systems, which generally do not offer such guarantees and therefore make it very difficult to determine which part of the generator is at fault when the generated code cannot be parsed. Other problems is preventing accidental variable capture (so-called hygiene [18]) and ensuring the generated code is well-typed. Lisp-style macros, Scheme hygienic macros, camlp4 preprocessor [9], C++ template meta-programming, Template Haskell [8] solve some of the above problems. Of the widely available maintainable languages, only MetaOCaml [2, 20] solves all the above problems including the well-typing of the generated code [29, 27].

But more difficult problems remain. Is the generated code optimal? Do we still need post-processing to eliminate common subexpressions and fold constants, remove redundant bindings? Is the generator readable, resembling the original algorithm, and extensible? Are the aspects truly modular? Can we add another aspect to it or another instance of the existing aspect without affecting the existing ones? Finally, can we express domain-specific knowledge, e.g., one should not attempt to use full division when dealing with matrices of exact integers, nor is it worthwhile to use full pivoting on a matrix over $\mathbb{Q}$.

MetaOCaml is *generative*: generated code can only be treated as a black box: it cannot be inspected and it cannot be post-processed (i.e., no intensional analysis). This approach gives a stronger equational theory [28], and avoids the danger of creating unsoundness [27]. Furthermore, intensional code analysis essentially requires one to insert both an optimizing compiler and an automated theorem proving system into the code generating system [25, 15, 4, 31]. While this is potentially extremely powerful and an exciting area of research, it is also extremely complex, which means that it is currently more error-prone and difficult to ascertain the correctness of the resulting code.

Therefore, in MetaOCaml, code must be generated just right (see [27] for many simple examples). For more complex examples, new techniques are necessary, e.g., abstract interpretation [17]. But more problems remain [6]: generating binding statements ("names"), especially when generating loop bodies or conditional branches; making continuation-passing style (CPS) code clear. Many authors understandably shy away from CPS code as it quickly becomes unreadable. But this is needed for proper name generation. The problems of compositionality of code generators, expressing dependencies among them and domain-specific knowledge remain.

In this paper we report on progress of solving these problems using GE as our running example. Specifically, our contributions:

– Extending monad of [17] for generating binding statements when generating control structures such as loops and conditionals. We argue that code generation is an effect and so requires something like a monad for proper serialization.

– Implementation of the mdo-notation (patterned after do-notation of Haskell) to make monadic code readable.
– Use of functors (including higher-order functors) to modularize the generator, express aspects (including results of various types) and *assure composability of aspects* even for aspects that use state and have to be accounted in many places in the generated code.
– Use functor type sharing constraints to encode domain-specific knowledge.

The rest of this paper is structured as follows: The next section introduces code generation in MetaOCaml, the problem of name generation, and continuation-passing style (CPS) as a general solution. We also introduce the monad and the issues of generating control statements. Section 3 describes the use of parametrized modules of OCaml to encode all of the aspects of the Gaussian Elimination algorithm family in completely separate, independent modules. We briefly discuss related work in section 4. We then outline the future work and conclude. Appendices give samples of the generated code (which is available in full at [5]).

We wish to thank Cristiano Calgano for his help in adapting camlp4 for use with MetaOCaml.

2 Generating binding statements, CPS, and monad

We build code generators out of primitive ones using code generation combinators. MetaOcaml, as an instance of a multi-stage programming system [27], provides exactly the needed features: to construct a code expression, to combine them, and to execute them. Figure 2 shows the simplest code generator `one`, as well as more complex generators.

```
let one = .<1>. and plus x y = .<.~x + .~y>.
let simplest_code = let gen x y = plus x (plus y one) in
  .<fun x y -> .~(gen .<x>. .<y>.)>.
⟹.<fun x_1 -> fun y_2 -> (x_1 + (y_2 + 1))>.
let simplest_param_code plus one = let gen x y = plus x (plus y one) in
  .<fun x y -> .~(gen .<x>. .<y>.)>.
let param_code1 plus one =
  let gen x y = plus (plus y one) (plus x (plus y one)) in
  .<fun x y -> .~(gen .<x>. .<y>.)>.
let param_code1' plus one =
  let gen x y = let ce = (plus y one) in  plus ce (plus x ce) in
  .<fun x y -> .~(gen .<x>. .<y>.)>.
param_code1' plus one
⟹.<fun x_1 -> fun y_2 -> ((y_2 + 1) + (x_1 + (y_2 + 1)))>.
```

Fig. 1. Code generation and combinators. $\Longrightarrow$ under an expression shows the result of its evaluation

We use MetaOCaml brackets `.<...>.` to generate code expressions, i.e., to construct future-stage computations. We use escapes `.~` to perform an immediate code generating computation *while* we are building the future-stage computation. The immediate computation in `simplest_code` is the evaluation of the function gen, which in turn applies plus. The function gen receives code expressions .<x>. and .<y>. as arguments. At the generating stage, we can manipulate code expressions as (opaque) values. The function gen returns a code expression, which is inlined in the place of the escape. MetaOCaml can print out code expressions, so we can see the final generated code. It has no traces of gen and plus: their applications are done at the generation stage.

The final MetaOCaml feature, .! (pronounced "run") executes the code expression: .! simplest_code is a function of two integers, which we can apply: (.! simplest_code) 1 2. The original simplest_code is not a function on integers – it is a code expression.

To see the benefit of code generation, we notice that we can easily parameterize our code, simplest_param_code, and use it to generate code that operates on integers, floating point numbers or booleans – in general, any domain that implements plus and one.

The generator param_code1 has two occurrences of plus y one, which may be quite a complex computation and so we would rather not do it twice. We may be tempted to rely on the compiler's common-subexpression elimination optimization. When the generated code is very complex, however, the compiler may overlook common subexpressions. Or the subexpressions may occur in such an imperative context where the compiler might not be able to determine if lifting them is sound. So, being conservative, the optimizer will leave the duplicates as they are. We may attempt to eliminate subexpressions as in param_code1'. However, the result of param_code1' plus one still exhibits duplicate sub-expressions. Our let-insertion optimization saved the computation at the generating stage. We need a combinator that inserts the let expression in the generat*ed* code. We need a combinator letgen to be used as let ce = letgen (plus y one) in plus ce (plus x ce) yielding the code like .<let t = y + 1 in t + (x + t)>. But that seems impossible because letgen exp has to generate the expression .<let t = exp in body>. but letgen does not have the body yet. The body needs a temporary identifier .<t>. that is supposed to be the result of letgen itself. Certainly letgen cannot generate only part of a let-expression, without the body, as all generated expressions in MetaOCaml are well-formed and complete.

The key is to use continuation-passing style (CPS). Its benefits were first pointed out by [1] in the context of partial evaluation, and extensively used by [17] for code generation. Now, param_code2 plus one gives us the desired code.

```
let letgen exp k = .<let t = .~exp in .~(k .<t>.)>.
let param_code2 plus one =
  let gen x y k = letgen (plus y one) (fun ce -> k (plus ce (plus x ce)))
  and k0 x = x
  in .<fun x y -> .~(gen .<x>. .<y>. k0)>.
param_code2 plus one
⟹.<fun x_1 -> fun y_2 -> let t_3 = (y_2 + 1) in (t_3 + (x_1 + t_3))>.
```

Comparison of the code that did let-insertion at the generating stage
let ce = (plus y one) in plus ce (plus x ce)

with the corresponding code inserting let at the generated code stage
`letgen (plus y one) (fun ce -> k (plus ce (plus x ce)))`
clearly shows the difference between direct-style and CPS code. What was
`let ce = init in ...` in direct style became `init' (fun ce -> ...)` in CPS. For one thing, `let` became "inverted". For another, what used to be an expression that yields a value, `init`, became an expression that takes an extra argument, the continuation, and invokes it. The differences look negligible in the above example. In larger expressions with many let-forms, the number of parentheses around `fun` increases, the need to add and then invoke the `k` continuation argument become increasingly annoying. The inconvenience is great enough for some people to explicitly avoid CPS or claim that numerical programmers (our users) cannot or will not program in CPS. Clearly a better notation is needed.

The `do`-notation of Haskell [24] shows that it is possible to write CPS code in a conventional-looking style. The `do`-notation is the notation for monadic code [21]. Not only can monadic code represent CPS [12], it also helps in composability by offering to add different layers of effects (state, exception, non-determinism, etc) to the basic monad [19] in a controlled way.

A monad [21] is an abstract datatype representing computations that yield a value and may have an *effect*. The datatype must have at least two operations, `return` to build trivial effect-less computations and `bind` for combining computations. These operations must satisfy *monadic laws*: `return` being the left and the right unit of `bind` and `bind` being associative. Figure 2 defines the monad used throughout the present paper and shows its implementation.

```
type ('v,'s,'w) monad = 's -> ('s -> 'v -> 'w) -> 'w
let ret a = fun s k -> k s a
let bind a f = fun s k -> a s (fun s' b -> f b s' k)
let fetch s k = k s s  and  store v s k = k (v::s) ()

let k0 s v = v
let runM m = m [] k0

let l1 f = fun x -> mdo { t <-- x; f t}
let l2 f = fun x y -> mdo { tx <-- x; ty <-- y; f tx ty}

let retN a = fun s k -> .<let t = .~a in .~(k s .<t>.)>.

let ifL test th el = ret .< if .~test then .~th else .~el >.
let ifM test th el = fun s k ->
  k s .< if .~(test s k0) then .~(th s k0) else .~(el s k0) >.
```

Fig. 2. Our monad

Our monad represents two kinds of computational effects: reading and writing a computation-wide state, and control effects. The latter are normally associated with exceptions, forking of computations, etc. – in general, whenever a computation ends with something other than invoking its natural continuation in the tail position. In our case the control effects manifest themselves as code generation.

In Figure 2, the monad is implemented as a function of two arguments: the state (of type `s`) and the continuation. The continuation receives the current state, the value (of the type v) and yields the answer of the type w. The monad is polymorphic over the three type parameters. Other implementations are possible. Except for the code in Figure 2, the rest of our code treats the monad as a truly abstract data type. The implementation of the basic monadic operations `ret` and `bind` are conventional. It is easy to see that the monadic laws are satisfied. Other monadic operations construct computations that do have specific effects. Operations `fetch` and `store v` construct computations that read and write the state. In our case the state is a list (of polymorphic variants), which models an open discriminated union, as we shall see later.

The operation `retN a` is the let-insertion operation, whose simpler version we called `letgen` earlier. It is the first computation with a control effect: indeed, the result of `retN a` is *not* the result of invoking its continuation `k`. Rather, its result is a `let` code expression. Such a behavior is symptomatic of control operators (in particular, `abort`).

Finally, `runM` runs our monad, that is, performs the computation of the monad and returns its result, which in our case is the code expression. We run the monad by passing it the initial state and the initial continuation `k0`. We can now re-write our `param_code2` example of the previous section as `param_code3`.

```
let param_code3 plus one =
  let gen x y = bind (retN (plus y one)) (fun ce ->
                ret (plus ce (plus x ce)))
  in .<fun x y -> .~(runM (gen .<x>. .<y>.))>.
let param_code4 plus one =
  let gen x y = mdo { ce <-- retN (plus y one);
                      ret (plus ce (plus x ce)) }
  in .<fun x y -> .~(runM (gen .<x>. .<y>.))>.
let ifM' test th el = mdo {
  testc <-- test; thc <-- th; elc <-- el;
  ifL testc thc elc}
let gen a i = ifM' (ret .<(.~i) >= 0>.)
                   (retN .<Some (.~a).(.~i)>.) (ret .<None>.)
 in .<fun a i -> .~(runM (gen .<a>. .<i>.))>.
⟹.<fun a_1 i_2 ->let t_3 = (Some a_1.(i_2)) in if (i_2 >= 0) then t_3 else None>.
let gen a i = ifM (ret .<(.~i) >= 0>.)
                  (retN .<Some (.~a).(.~i)>.) (ret .<None>.)
 in .<fun a i -> .~(runM (gen .<a>. .<i>.))>.
⟹.<fun a_1 i_2 ->if (i_2 >= 0) then let t_3 = (Some a_1.(i_2)) in t_3 else None>.
```

That does not seem like much of an improvement. With the help of camlp4 pre-processor, we introduce the `mdo`-notation (cite XXX), patterned after the `do`-notation of Haskell. The function `param_code4`, written in the `mdo`-notation, is equivalent to `param_code3` – in fact, the camlp4 preprocessor will convert the former into the latter. And yet, `param_code4` looks far more conventional, as if it were indeed in direct style.

We can write operations that generate code other than let-statements, e.g., conditionals: see `ifL` in Figure 2. The function `ifL`, albeit straightforward, is not as general as we wish: its arguments are already generated pieces of code rather than monadic values. We "lift it", see `ifM'`. We define functions `l1`, l2, l3 (analogues of liftM, liftM2, liftM3 of Haskell) to make such a lifting generic. However we also need another ifM function, with the same interface (see Figure 2). The difference between them is apparent: in the code above with ifM', the let-insertion happened *before* the if-expression, that is, before the test that the index i is positive. If i turned out negative, a.(i) would generate an out-of-bound array access error. On the other hand, the code with ifM accesses the array only when we have verified that the index is non-negative. This example makes it clear that the code generation (such as the one in retN) is truly an effect and we have to be clear about the sequencing of effects when generating control constructions such as conditionals. The form ifM handles such effects correctly. We need similar operators for other Ocaml control forms: for generating case-matching statements and for- and while-loops.

3 Aspects and Functors

The monad represents finer-scale code generation. We need tools for larger-scale modularization; we can use any abstraction mechanisms we want to structure our code generators, as long as none of those abstractions infiltrate the generated code.

While the Object-Oriented Design community has acquired an extensive vocabulary for describing modularity ideas, the guiding principles for modular designs has not changed since they were first articulated by Parnas [23] and Dijkstra [10]: information hiding and separation of concerns. To apply these principles to the study of Gaussian Elimination, we need to understand what are the changes between different implementations, and what concerns need to be addressed. We also need to study the degree to which these concerns are independent. A study of Gaussian Elimination [3] shows that the following variations occur:

1. **Domain**: In which (algebraic) domain do the matrix elements belong to. Sometimes the domains are very specific (e.g., $\mathbb{Z}, \mathbb{Q}, \mathbb{Z}_p$ and floating point numbers), while in other cases the domains were left generic, e.g., multivariate polynomials over a field. In the roughly 85 pieces of code surveyed [3] 20 different domains were encountered.
2. **Container**: Whether the matrix is represented as an array of arrays, a one-dimensional array, a hash table, a sparse matrix, etc., and whether indexing is done in C or Fortran style. Additionally, if a particular representation had a special mechanism for efficient row exchanges.
3. **Output choices**: Whether just the reduced matrix, or additionally the rank, the determinant, and the pivoting matrix are to be returned. In the larger algorithm family, routines like Maple's LinearAlgebra:-LUDecomposition have up to $2^6 + 2^5 + 2^2 = 100$ outputs.
4. **Fraction-free**: Whether the Gaussian Elimination algorithm is allowed to use unrestricted division, or only exact (remainder-free) division.
5. **Pivoting**: Whether to use no pivoting, column-wise pivoting, or full pivoting.

6. **Augmented Matrices**: Whether we are doing GE on a full matrix, or only a restricted number of columns, while doing elimination on the full matrix. We currently do not treat this aspect in our code.

In addition to the above variations, there are two aspects that recur frequently:

1. **Length measure**: For stability reasons (numerical or coefficient growth), if a domain possesses an appropriate length measure, this is sometimes used to choose an "optimal" pivot.
2. **Normalization and zero-equivalence**: Whether the arithmetic operations of the domain at hand gives results in normalized form, and whether a specialized zero-equivalence routine needs to be used.

These are separated out from the others are they are cross-cutting concerns: in the case of the length measure, a property of the domain will influence the method used to do pivoting *if* pivoting is to be performed.

The simplest parametrization is to make the domain abstract. As it turns out, we need the following to exist in our domains: 0, 1, $+$, $*$, (unary and binary) $-$, at least *exact* division, normalization, and potentially a relative size measure. The simplest case of such domain abstraction is `param_code1` in Fig. 2. There, code-generators such as `plus` and `one` were passed as arguments. We need far more than two parameters, so we have to group them. Instead of the grouping offered by regular records, we use Ocaml *structures* (i.e., modules) so we can take advantage of extensibility, type abstraction and constraints, and especially parameterized structures (*functors*). We define the type of the domain, the signature `DOMAIN` which different domains must satisfy:

```
module type DOMAIN = sig
  type v    type 'a vc = ('a,v) code
  type kind (* Field or Ring ? *)
  val zero : 'a vc   val one : 'a vc
  val plus : 'a vc -> 'a vc -> ('a vc, 's, 'w) monad
  (* times, minus, uminus, div elided for brevity *)
  val better_than : ('a vc -> 'a vc ->
      (('a,bool) code, 's, 'w) monad) option
  val normalizerf : (('a,v -> v) code ) option
end
module IntegerDomain : DOMAIN = struct
  type v = int  type kind = domain_is_ring
  type 'a vc = ('a,v) code
  let zero = .< 0 >.  and one = .< 1 >.
  let plus x y = ret .<.~x + .~y>.
  let better_than = Some (fun x y -> ret .<abs .~x > abs .~y >. )
  let normalizerf = None
  ...
end
```

The types above are generally lifted twice: once from the value domain `v` to the code domain `'a vc`, and once more from values to monadic computations `('a vc, 's, 'w) monad`.

One particular domain instance is `IntegerDomain`. The notation `module IntegerDomain : DOMAIN` makes the compiler verify that our `IntegerDomain` is indeed a `DOMAIN`, that is, satisfies the required signature. The constraint `DOMAIN` may be omitted; in that case, the compiler will verify the type when we try to use that structure as a `DOMAIN`. In any case, the errors such as missing "methods" or methods with incorrect types will be caught statically, even *before* any code generation takes place. The abstract type `domain_is_ring` encodes a semantic constraint that the full division is not available. While the `DOMAIN` type may have looked daunting to some, the implementation is quite straightforward. Other domains such as `float` and arbitrary precision exact rational numbers `Num.num` are equally simple.

Parametrizing by the kind of container representing a matrix is almost as straightforward. Our containers are parametric over a `DOMAIN`, i.e., functors from a `DOMAIN` module to the actual implementation of a container. The functor signature `CONTAINER2D` specifies that a container must provide functions `dim1` and `dim2` to extract the dimensions, functions `get` and `set` to generate container getter and setters, the cloning generator `copy` and functions that generate code for row and column swapping. The inclusion of these functions in the signature of all containers makes it simpler to optimize the relevant functions depending on the actual representation of the container while not burdening the users of containers with efficiency details.

The use of a `functor` for making a container parametric is fairly straightforward. More interesting is the aspect of what to return from the GE algorithm. One could create an algebraic data type (as was done in [3]) to encode the various choices: the matrix, the matrix and the rank, the matrix and the determinant, the matrix, rank and determinant, and so on. This is wholly unsatisfying as we know that for any single use, only one of the choices is ever possible, yet any routine which calls the generated code must deal with these unreachable options. Instead we use a module type with an *abstract* type `res` for the result type; different instances of the signature set the result type differently. Given below is this module type and one instantiation, which specifies the output of a GE algorithm as a 3-tuple `contr * Det.outdet * int` of the U-factor, the determinant and the rank.

```
module type OUTPUT = sig
  type contr  type res
  module D : DETERMINANT   module R : RANK   module P : TRACKPIVOT
  val make_result : ('a,contr) code ->
    (('a,res) code,
     [> `TDet of 'a D.lstate | `TRan of 'a R.lstate | `TPivot of 'a P.lstate]
       list, ('a,'w) code) monad
end
module OutDetRank(Dom:DOMAIN)(C: CONTAINER2D)
    (Det : DETERMINANT with type indet = Dom.v and type outdet = Dom.v)
    (Rank : RANK) = struct
  module Ctr = C(Dom)
  type contr = Ctr.contr
```

```
  type res = contr * Det.outdet * int
  module D = Det   module R = Rank   module P = DiscardPivot
  let make_result b = mdo { det  <-- D.fin ();  rank <-- R.fin ();
    ret .< ( .~b, .~det, .~rank ) >. }
end
```

As is apparent from the output choices, several different quantities *may* need to be tracked in a particular GE implementation. We therefore need to be able to conditionally generate variables representing the tracking state, and weave in corresponding tracking code. We may need to (independently) keep track of the rank, the determinant and the permutation list. The tracking state variables then become part of the *state* that is tracked by our monad. To have all this choice when needed, and yet have our code be modular and composable as well as ensuring that the generated code does not contain any abstraction artifacts, it is important to make this state modular. For example,

```
module type DETERMINANT = sig
  type indet  type outdet  type 'a lstate
  type tdet = outdet ref
  val decl : unit ->
    (unit, [> `TDet of 'a lstate ] list, ('a,'b) code) monad
  val upd_sign : unit ->
    (('a,unit) code, [> `TDet of 'a lstate ] list, ('a,'b) code) monad
  ...
end
```

determinant tracking aspects involves being able to generate code that defines variables used for tracking (`decl`), generate code that updates the sign or the absolute value of the determinant, and finally, converting the tracking state to the final determinant value of the type `outdet`. GE of a floating-point matrix with no determinant tracking uses the instantiation of `DETERMINANT` where `outdet` is `unit` and all the functions of that module generate no code. For integer matrices, we have to track some aspects of the determinant, even if we don't output it. The determinant tracking aspect is complex because tracking variables, if any, are to be declared at the beginning of GE; the sign of the determinant has to be updated on each row or column permutation; the value of the determinant should be updated per each pivoting. We use `lstate` to pass the tracking state, e.g., a piece of code for the value of the type `Dom.v ref`, among various determinant-tracking functions. The `lstate` is a part of the overall monadic state. Other aspects, e.g., rank tracking, may use the monadic state for passing of rank tracking variables. To be able to compose determinant and rank tracking functors – each of which may (or may not) use the monadic state for passing its own data – we make extensive use of open records (a list of polymorphic variants appeared to be the easiest way to implement such a union, in a purely functional way). This lets us freely compose determinant-tracking, rank-tracking, and other aspects.

The GE generator functor itself is parameterized by the domain, container, pivoting policy (full, row, nonzero, nopivoting), update policy (that use either "fraction-less' or full division), and by what to yield as the result. Some of the

```
module Gen(Dom: DOMAIN)(C: CONTAINER2D)(PivotF: PIVOT)
          (Update: UPDATE with type baseobj = Dom.v and type ctr = C(Dom).contr)
          (Out: OUTPUT with type contr = C(Dom).contr and type D.indet = Dom.v
                        and type 'a D.lstate = 'a Update.D.lstate) = struct
    module Ctr = C(Dom)
    module Pivot = PivotF(Dom)(C)(Out.D)
    let gen =
      let zerobelow b r c m n brc =
        let innerbody i = mdo {
            bic <-- Ctr.get b i c;
            whenM (l1 LogicCode.not (LogicCode.equal bic Dom.zero ))
                (seqM (retLoopM (Idx.succ c) (Idx.pred m)
                          (fun k -> Update.update b r c i k) )
                      (Ctr.set b i c Dom.zero)) } in
        mdo {
              seqM (retLoopM (Idx.succ r) (Idx.pred n) innerbody)
                   (Update.update_det brc) } in
      let dogen a = mdo {
          r <-- Out.R.decl ();
          c <-- retN (liftRef Idx.zero);
          b <-- retN (Ctr.mapper Dom.normalizerf (Ctr.copy a));
          m <-- retN (Ctr.dim1 a);
          n <-- retN (Ctr.dim2 a);
          () <-- Update.D.decl ();
          () <-- Out.P.decl ();
          seqM
            (retWhileM (LogicCode.and_ (Idx.less (liftGet c) m)
                                       (Idx.less (liftGet r) n) )
               ( mdo {
               rr <-- retN (liftGet r);
               cc <-- retN (liftGet c);
               pivot <-- l1 retN (Pivot.findpivot b rr m cc n);
               seqM (retMatchM pivot (fun pv ->
                         seqM (zerobelow b rr cc m n pv)
                              (Out.R.succ ()) )
                         (Update.D.zero_sign () ))
                     (Code.update c Idx.succ) } ))
            (Out.make_result b) } in
    .<fun a -> .~(runM (dogen .<a>.)) >.
end
```

argument modules such as `PIVOT` are functors themselves (parameterized by the domain, the container, and the determinant functor). The sharing constraints express obvious constraints on the instantiation of `Gen`, for example, pivoting, determinant etc. components all use the same domain. It must be stressed that all structures (i.e., module instances) are stateless, and so we never have to worry that different aspect functors (such as `CONTAINER2D` and `PIVOT`) are instantiated with different but type-compatible instances of `DOMAIN`. That is, we are not concerened at all about the value sharing. Aspects such as determinant tracking may be stateful so that the determinant update code have access to the determinant tracking variables declared previously. But that state is handled via the monadic state. As we have shown, open unions makes the overall monadic state compositional with respect to the state of various aspects.

In addition to the "regular" type sharing constraints shown in the `Gen` functor, there are also "semantic" sharing constraints, shown in the following structure of the `UPDATE` signature:

```
module DivisionUpdate
  (Dom:DOMAIN with type kind = domain_is_field)
  (C:CONTAINER2D)
  (Det:DETERMINANT with type indet=Dom.v) = struct ... end
```

This structure implements an update policy of using `Dom.div` operation without restrictions – which is possible only if the domain has such unrestricted operation. A domain such as the integer domain may still provide `Dom.div` of the same type, but that operation may only be used when we are sure that the division is exact. Our type sharing constraint expresses such domain-specific knowledge: instantiating `DivisionUpdate` with `integerDomain` leads to a compile-time error, when compiling the *generator* code. Thus, in some cases we can use module types for "semantic" constraints that cannot normally be expressed via the types of module members.

```
module GenIV5 = Gen(IntegerDomain)
   (GenericVectorContainer)(FullPivot)
   (FractionFreeUpdate(IntegerDomain)(GenericVectorContainer)(IDet))
   (OutDetRank(IntegerDomain)(GenericVectorContainer)(IDet)(Rank))
module GenFA1 = Gen(FloatDomain)
   (GenericArrayContainer)(RowPivot)
   (DivisionUpdate(FloatDomain)(GenericArrayContainer)(NoDet(FloatDomain)))
   (OutJustMatrix(FloatDomain)(GenericArrayContainer)(NoDet(FloatDomain)))
```

We can instantiate the `Gen` functor as shown above and inspect the generated code, e.g., by printing `GenFA1.gen`. The code can then be "compiled" as `!. GenFA1.gen` or with off-shoring. The code for `GenIV5` (Appendix A) shows full pivoting, determinant and rank tracking. The code for all these aspects is fully inlined; no extra functions are invoked and no tests other than those needed by the GE algorithm itself are performed. The GE function returns a triple `int array * int * int` of the U-factor, determinant and the rank. The code generated by `GenFA1` (Appendix B) shows absolutely no traces of determinant tracking: no declaration of spurious variables, no extra tests, etc. The code appears as if the determinant tracking aspect did not exist at all. The generated code for the above and other instantiations of `Gen` can be examined at [5].

4 Related and future work

Note that the monad is similar to the one used in [17]. However, the latter work used only `retN` of all monadic operations, and used fixpoints (for performing iterations at generation time). In this paper we do not use monadic fixpoints (because the generator is not recursive) but we make extensive use of monadic operations for generating conditional and looping operations.

`Blitz++` [30] and in general template meta-programming in C++ achieve similar results of eliminating levels of abstraction. Using traits and concepts, some domain-specific knowledge can also be encoded. However inlining critically depends on the compiler's fully inlining of all methods, and as has been reported in the literature, this can be challenging to insure. Furthermore, all errors (such as type errors and concept violation errors, that is, composition errors) are detected only when compiling the generated code. It is immensely difficult to correlate errors (e.g., line numbers) to the ones in the generator itself. Furthermore, as templates cannot be local, a lot of code generation is scattered around instead of being modularized.

ATLAS [33] is another successful project in this area. However they use much simpler weaving technology, which leads them to note that *generator complexity tends to go up along with flexibility, so that these routines become almost insurmountable barriers to outside contribution*. Our results show how to surmount this barrier, by building modular, composable generators. SPIRAL [25] is another such even more ambitious project. But SPIRAL does intentional code analysis, relying on a set of code transformation "rules" which make sense, but which are not proved to be either complete or confluent. The strength of both of these project relies on their platform-specific optimizations performed via search techniques, something we have not attempted here.

The highly parametric version of our Gaussian Elimination is directly influenced by the generic implementations available in Axiom [13] and Aldor [32]. Even though the Aldor compiler frequently can optimize away a lot of abstraction overhead, it does not provide any guarantees that it will do so, unlike our approach.

To the best of our knowledge, nobody has yet used functors to abstract code generators, or even mixed functors and multi-stage programming.

We plan to further investigate the connection between delimited continuations and our implementations of code generators like `ifM`. As well, by using some additional syntactic sugar (for `ifM`, `whileM`, etc.), the available notation should be even more direct-style, and potentially clearer.

There are many more aspects which can also be handled Input variations (augmented matrices), error reporting (i.e. asking for the determinant of a non-square matrix), memory hierarchy issues, loop-unrolling [6], warnings when zero-testing is undecidable and a value is only probabilistically non-zero, etc. The larger program family of LU decompositions contains more aspects still.

Finally, we would like to be able to make the `Gen` functor applicable to runnable functors in Ocaml, as well as to code generation functors in MetaOCaml - for ease of debugging. The current use of abstract types should make this straightforward.

5 Conclusion

The combination of stateless modules (functors and structures), and our monad with the compositional state makes aspects freely composable without having to worry about the presence or absense of value aliasing. The only constraints to compositionality are the typing constraints plus the constraints we specifically impose, including semantic constraints (e.g., rings do not have full division).

There is an interesting relation with aspect-oriented code [16]: in AspectJ, aspects are (comparatively) lightly typed, and are post-facto extensions of an existing piece of code. Here aspects are weaved together "from scracth" to make up a piece of code/functionality. One can understand previous work to be more akin to dynamically typed aspect weaving, while we have started investigating statically typed aspect weaving.

References

1. Anders Bondorf. Improving binding times without explicit CPS-conversion. In *1992 ACM Conference on Lisp and Functional Programming. San Francisco, California*, pages 1–10, 1992.
2. Cristiano Calcagno, Walid Taha, Liwen Huang, and Xavier Leroy. Implementing multi-stage languages using asts, gensym, and reflection. In Krzysztof Czarnecki, Frank Pfenning, and Yannis Smaragdakis, editors, *Generative Programming and Component Engineering (GPCE)*, LNCS. Springer-Verlag, 2003.
3. Jacques Carette. Gaussian Elimination: a case study in efficient genericity with MetaOCaml, 2005. submitted.
4. Zizhong Chen, Jack Dongarra, Piotr Luszczek, and Kenneth Rothe. Lapack for clusters project: An example of self adapting numerical software. *Hawaii International Conference on System Sciences HICSS-37*, 2004.
5. Source code. `http://www.cas.mcmaster.ca/~carette/metamonads/`.
6. Albert Cohen, Sebastien Donadio, Mari'a Jesu's Garzara'n, Cristoph Herrmann, and David Padua. In search for a program generator to implement generic transformations for high-performance computing. MetaOCaml Workshop, October 2004.
7. Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., 2000.
8. Krzysztof Czarnecki, John T. O'Donnell, Jörg Striegnitz, and Walid Taha. DSL implementation in MetaOCaml, Template Haskell, and C++. In Christian Lengauer, Don S. Batory, Charles Consel, and Martin Odersky, editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 51–72. Springer, 2003.
9. Daniel de Rauglaudre. Camlp4 reference manual. `http://caml.inria.fr/camlp4/manual/`, Jan. 2002.
10. Edsger W. Dijkstra. On the role of scientific thought. published as [11].
11. Edsger W. Dijkstra. On the role of scientific thought. In *Selected Writings on Computing: A Personal Perspective*, pages 60–66. Springer-Verlag, 1982.
12. Andrzej Filinski. Representing monads. In *POPL*, pages 446–457, 1994.
13. Richard D. Jenks and Robert S. Sutor. *AXIOM: The Scientific Computation System*. Springer Verlag, 1992.
14. III John V.W. Reynders and Julian C. Cummings. The POOMA framework. *Comput. Phys.*, 12(5):453–459, 1998.
15. Ken Kennedy, Bradley Broom, Keith Cooper, Jack Dongarra, Rob Fowler, Dennis Gannon, Lennart Johnsson, John Mellor-Crummey, and Linda Torczon. Telescoping languages: A strategy for automatic generation of scientific problem-solving systems from annotated libraries. *Journal of Parallel and Distributed Computing*, 61(12):1803–1826, December 2001.

16. Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.
17. Oleg Kiselyov, Kedar N. Swadi, and Walid Taha. A methodology for generating verified combinatorial circuits. In *ACM International conference on Embedded Software (EMSOFT)*, 2004.
18. Eugene E. Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce F. Duba. Hygienic macro expansion. In *LISP and Functional Programming*, pages 151–161, 1986.
19. Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *POPL '95: Conference Record of the Annual ACM Symposium on Principles of Programming Languages*, pages 333–343, New York, 1995. ACM Press.
20. MetaOCaml. `http://www.metaocaml.org`.
21. Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.
22. David R. Musser and Alexander A. Stepanov. Algorithm-oriented generic libraries. *Software - Practice and Experience*, 24(7):623–642, 1994.
23. David Lorge Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.
24. Simon Peyton Jones et al. *The Revised Haskell 98 Report*. Cambridge Univ. Press, 2003. Also on http://haskell.org/.
25. Markus Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan W. Singer, Jianxin Xiong, Franz Franchetti, Aca Gačić, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nick Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2), 2005.
26. Jeremy Siek, Lee-Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley, 2002.
27. Walid Taha. *Multi-Stage Programming: its theory and applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, 1999.
28. Walid Taha. A sound reduction semantics for untyped cbn mutli-stage computation. or, the theory of metaml is non-trival. In *PEPM*, pages 34–43, 2000.
29. Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *Proceedings of the Symposium on Partial Evaluation and Semantic-Based Program Manipulation (PEPM)*, pages 203–217, Amsterdam, June 1997. ACM Press.
30. Todd L. Veldhuizen. Arrays in Blitz++. In *Proceedings of the 2nd International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'98)*, Lecture Notes in Computer Science. Springer-Verlag, 1998.
31. Todd L. Veldhuizen. *Active Libraries and Universal Languages*. PhD thesis, Indiana University Computer Science, May 2004.
32. Stephen M. Watt. Aldor. In J. Grabmeier, E. Kaltofen, and V. Weispfennig, editors, *Computer Algebra Handbook: Foundations, Applications, Systems*. Springer Verlag, 2003.
33. R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001.

6 Appendix A

The code generated for GenIV5, fraction-free GE of the integer matrix represented by a flat vector, full pivoting, returning the U-factor, the determinant and the rank.

```
# val resIV5 :
  ('a,
   Funct4.GenIV5.Ctr.contr ->
   Funct4.OutDetRank(Funct4.IntegerDomain)(Funct4.GenericVectorContainer)
                    (Funct4.IDet)(Funct4.Rank).res)
  code =
  .<fun a_405 ->
   let t_406 = (ref 0) in
   let t_407 = (ref 0) in
   let t_408 = arr = (Array.copy a_405.arr) (a_405) in
   let t_409 = a_405.m in
   let t_410 = a_405.n in
   let t_411 = (ref 1) in
   let t_412 = (ref 1) in
   while (((! t_407) < t_409) && ((! t_406) < t_410)) do
    let t_413 = (! t_406) in
    let t_414 = (! t_407) in
    let t_415 = (ref (None)) in
    let t_435 =
     begin
      for j_431 = t_413 to (t_410 - 1) do
       for j_432 = t_414 to (t_409 - 1) do
        let t_433 = (t_408.arr).((j_431 * t_408.m) + j_432) in
        if (not (t_433 = 0)) then
         (match (! t_415) with
          | Some (i_434) ->
             if ((abs (snd i_434)) > (abs t_433)) then
              (t_415 := (Some ((j_431, j_432), t_433)))
             else ()
          | None -> (t_415 := (Some ((j_431, j_432), t_433))))
        else ()
       done
      done;
      (match (! t_415) with
       | Some (i_416) ->
          if ((snd (fst i_416)) <> t_414) then begin
           let a_424 = t_408.arr
           and nm_425 = (t_408.n * t_408.m)
           and m_426 = t_408.m in
           let rec loop_427 =
            fun i1_428 ->
             fun i2_429 ->
              if (i2_429 < nm_425) then
               let t_430 = a_424.(i1_428) in
```

```
            a_424.(i1_428) <- a_424.(i2_429);
            a_424.(i2_429) <- t_430;
            (loop_427 (i1_428 + m_426) (i2_429 + m_426))
           else () in
        (loop_427 t_414 (snd (fst i_416)));
        (t_412 := (~- (! t_412)))
       end else ();
       if ((fst (fst i_416)) <> t_413) then begin
        let a_417 = t_408.arr
        and n_418 = t_408.n
        and m_419 = t_408.m in
        let i1_420 = (t_413 * m_419)
        and i2_421 = ((fst (fst i_416)) * m_419) in
        for i_422 = 0 to (m_419 - 1) do
         let t_423 = a_417.(i1_420 + i_422) in
         a_417.(i1_420 + i_422) <- a_417.(i2_421 + i_422);
         a_417.(i2_421 + i_422) <- t_423
        done;
        (t_412 := (~- (! t_412)))
       end else ();
       (Some (snd i_416))
    | None -> (None))
  end in
 (match t_435 with
  | Some (i_436) ->
     begin
      for j_437 = (t_413 + 1) to (t_410 - 1) do
       if (not ((t_408.arr).((j_437 * t_408.m) + t_414) = 0)) then begin
        for j_438 = (t_414 + 1) to (t_409 - 1) do
         (t_408.arr).((j_437 * t_408.m) + j_438) <-
          ((((t_408.arr).((j_437 * t_408.m) + j_438) *
              (t_408.arr).((t_413 * t_408.m) + t_414)) -
             ((t_408.arr).((t_413 * t_408.m) + j_438) *
               (t_408.arr).((j_437 * t_408.m) + t_413))) / (! t_411))
        done;
        (t_408.arr).((j_437 * t_408.m) + t_414) <- 0
       end else ()
      done;
      (t_411 := i_436)
     end;
     (t_406 := ((! t_406) + 1))
  | None -> (t_412 := 0));
 (t_407 := ((! t_407) + 1))
done;
(t_408,
 if ((! t_412) = 0) then 0
 else if ((! t_412) = 1) then (! t_411)
 else (~- (! t_411)), (! t_406))>.
```

7 Appendix B

The code generated for GenFA1, GE of the floating point matrix represented by a 2D array, row pivoting, returning just the U-factor.

```
#   val resFA1 :
  ('a,
   Funct4.GenFA1.Ctr.contr ->
   Funct4.OutJustMatrix(Funct4.FloatDomain)(Funct4.GenericArrayContainer)
                     (Funct4.NoDet(Funct4.FloatDomain)).res)
  code =
  .<fun a_1 ->
   let t_2 = (ref 0) in
   let t_3 = (ref 0) in
   let t_5 = (Array.map (fun x_4 -> (Array.copy x_4)) (Array.copy a_1)) in
   let t_6 = (Array.length a_1.(0)) in
   let t_7 = (Array.length a_1) in
   while (((! t_3) < t_6) && ((! t_2) < t_7)) do
    let t_8 = (! t_2) in
    let t_9 = (! t_3) in
    let t_10 = (ref (None)) in
    let t_16 =
     begin
      for j_13 = t_8 to (t_7 - 1) do
       let t_14 = (t_5.(j_13)).(t_9) in
       if (not (t_14 = 0.)) then
        (match (! t_10) with
         | Some (i_15) ->
            if ((abs_float (snd i_15)) < (abs_float t_14)) then
             (t_10 := (Some (j_13, t_14)))
            else ()
         | None -> (t_10 := (Some (j_13, t_14))))
       else ()
      done;
      (match (! t_10) with
       | Some (i_11) ->
          if ((fst i_11) <> t_8) then begin
           let t_12 = t_5.(t_8) in
           t_5.(t_8) <- t_5.(fst i_11);
           t_5.(fst i_11) <- t_12;
           ()
          end else ();
          (Some (snd i_11))
       | None -> (None))
     end in
    (match t_16 with
     | Some (i_17) ->
        begin
         for j_18 = (t_8 + 1) to (t_7 - 1) do
          if (not ((t_5.(j_18)).(t_9) = 0.)) then begin
```

```
        for j_19 = (t_9 + 1) to (t_6 - 1) do
         (t_5.(j_18)).(j_19) <-
          ((t_5.(j_18)).(j_19) -.
            (((t_5.(j_18)).(t_9) /. (t_5.(t_8)).(t_9)) *.
              (t_5.(t_8)).(j_19)))
        done;
        (t_5.(j_18)).(t_9) <- 0.
       end else ()
      done;
      ()
     end;
     (t_2 := ((! t_2) + 1))
  | None -> ());
 (t_3 := ((! t_3) + 1))
done;
t_5>.
```