Gaussian Elimination: a case study in efficient genericity with MetaOCaml

Jacques Carette [1]

Abstract

The Gaussian Elimination algorithm is in fact an algorithm family - common implementations contain at least 6 (mostly independent) “design choices”. A generic implementation can easily be parametrized by all these design choices, but this usually leads to slow and bloated code. Using MetaOCaml’s staging facilities, we show how we can produce a natural and type-safe implementation of Gaussian Elimination which exposes its design choices at code-generation time, so that these choices can effectively be specialized away, and where the resulting code is quite efficient.

Key words: MetaOcaml, linear algebra, genericity, generative, staging
PACS: code

1 Introduction

Gaussian Elimination is a fundamental algorithm in computational mathematics. As it finds uses in many areas of mathematics, quite a number of variants on the basic algorithm have been invented over the years, but the basic algorithm persists through all variants. An analysis of the source code [2] for the library of Maple [3], a large commercial computer algebra system, found 35 different implementations of Gaussian Elimination (GE), and at least as many more implementations of highly related algorithms (LU factorization, Cholesky factorization, reduction to Smith Normal Form, linear equation solving, etc). There was little actual code duplication between these implementations, as they all embodied a certain set of design

Email address: `carette@mcmaster.ca` (Jacques Carette).
URL: `http://www.cas.mcmaster.ca/~carette` (Jacques Carette).

[1] Work supported in part by NSERC Discovery Grant XXX

[2] Note to reviewers: for completeness, this paper includes source code samples in Maple and the complete MetaOCaml implementation in appendices. The code which is deemed inessential for a final print version will instead be made available on the Web

[3] Maple is a registered trademark of Waterloo Maple Inc.

decisions that were hard-coded. As these design decisions have a measurable effect on the actual shape of the code, there was little obvious opportunity for abstraction. However, since the authors of the code knew that the underlying algorithm was all the same, it would have been possible to use higher-order functions to parametrize these choices. On the other hand, a number of these variants are rather esoteric, or depend on quite subtle properties of the underlying domain at hand - certainly an average user who knew they needed to use Gaussian Elimination could not be expected to understand all these choices, and furnish the appropriate functions. Thus it still necessary to expose this algorithm via different specialized interfaces. But even when there were parametrization opportunities, the ensuing loss of efficiency between the generic code versus the specialized code (up to a factor of 30, in part due to the fact that Maple is a dynamically typed interpreted language, but even in a statically typed language, the overhead of an (indirect) function call, a likely pipeline stall and cache miss(es) would produce similar overhead) naturally reinforced the idea that all these versions need to be coded seperately. This clearly leads to a maintenance nightmare. Thus we seek to answer the question:

Is it possible to write a generic yet efficient Gaussian Elimination algorithm?

In Maple itself, this question can be answered positively: Maple has very good reflection and reification capabilities (see [1] for definitions), with full introspection of all programs being quite easy; in fact, these capabilities compare favorably to those of LISP/Scheme [2]. Maple's reflection/reification are somewhat more type-safe than Scheme's; like in Scheme it is not possible to reify procedures which are syntactic nonsense, but also some mild type invariants are also guaranteed to hold, although the result may still be obvious semantic nonsense. We are more interested in the following more precise question:

Is it possible to write a generic, type-safe yet efficient Gaussian Elimination algorithm?

To ensure type safety, it is necessary to move to a typed language; to allow for efficiency, it is necessary to remove all the overhead introduced when making the algorithm generic. There are 3 obvious choices of languages to try [3]: MetaOcaml, Template Haskell, and C++. Because we wanted our Gaussian Elimination algorithm to "look like" the original algorithms in Maple as much as possible while at the same time enforced as much type-safety as is currently possible, the choice of MetaOCaml was quite natural.

1.1 Algorithm families

The concept of a *program family* [4] is now an integral part of software engineering, at least at the design level. This involves abstracting all the common parts of a family of (related) programs, and designing both the common parts and the variation points right at the outset. Knowing the variation points makes a modularization based on the *design for change* [5] principle that much easier.

This concept of program family can apply to very small programs – even programs made up of just one function, as in our case. This principle tells us that we should isolate each *aspect* [4] into a separate entity (a *module* in the case of program families), and then to somehow paste these aspects together to form a whole.

This is exactly the route we will follow. However, instead of presenting the results as a *fait accompli*, we will in part show how to perform stepwise abstractions (i.e. the inverse steps of stepwise refinement [8]) to go from a particular implementation of Gaussian Elimination to a generic one. We believe this to be more instructive. This presentation method allows us to show that, at every step, the resulting source code is quite readable, is still recognizable as Gaussian Elimination, and yet still specializes to the original version.

1.2 Generic and generative programming

While one can find many different reasons for generic programming [9,10,11], there is one fundamentally pragmatic reason: all decent programmers intensely dislike code duplication. Code duplication is sometimes tolerated for two reasons: a poor programming language which does not have proper abstraction mechanisms to unify certains language idioms, and efficiency. These issues are not independent: many abstraction mechanisms (higher-order functions, late-binding, ad hoc polymorphism, type classes, etc) generally incur a run-time cost. In the case of scientific computation applications, this run-time cost is frequently unacceptable, at least in the inner-loop of intensive computations.

Macros and pre-processors were common pseudo-solutions for both of these issues (and are still heavily used – many models in physical oceanography are written in `Fortran 77` with `cpp` directives peppered throughout for configuration choices! [12]). More recently, generative programming [13] has become an increasingly popular method in scientific computation - as witnessed by (amongst others) Blitz++ [14], Active Libraries [15], GMCL [13], POOMA [16], BOOST [17], and ATLAS [18]. These have also been applied to product families [19]. These solutions are mostly `C++`-centric, which brings problems of its own [3].

In a typeful setting, what we want to do is to *expose* the configuration parameters of our programs in such a way that the type system will help rather than hinder, in other words, the type system will insure that we are configuring our programs consistently. In particular, we are not interested in being *too* generic: being able to

[4] as in separation of concerns, terminology coined by Dijkstra [6] published as [7], the “other side of the coin” of modularization according to Parnas.

use any kind of connector to connect arbitrary components together is pointless, and will merely result in delaying problems to (at worst) run-time.

For example, if we have a stateful code type `code`, the type of the sequential composition operator of statements `;` is not

```
(;) : code × code -> code
```

but rather

```
(;) : (state -> unit) code × (state -> unit) code
      -> (state -> unit) code
```

where `state` is a specific representation for a (closed) state type. This ensures that only statements operating on the same state can be composed, thereby increasing robustness and safety.

1.3 Problem

The Gaussian Elimination algorithm has many different incarnations (see section 2), which are frequently coded seperately (for efficiency or sometimes source code clarity reasons). This is clearly unsatisfactory, as well as potentially leading to a maintenance nightmare. On the other hand, even when it is possible to use a language's abstraction facility to natively express all the design points of GE, this usually has too high an efficiency cost.

We want to be able to express a generic version of GE which abstracts out, in the context of symbolic computation, the most common design points: the domain of the matrix elements ($\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{Z}_p$, $\mathbb{Q}(\alpha)\,[x]$, or even $\mathbb{Z}_p(\alpha_1,\ldots,\alpha_n)\,[[x_1,\ldots,x_m]]$ and floating point numbers), the container type used (arrays of arrays, flattened arrays, implicit representations, as well as variants for storage orientation) for the matrix, the exact outputs (matrix, rank, determinant), as well as some domain-dependent aspects, like whether the elements of the domain are stored in a normalized manner, or whether there exists a natural (and useful) size function on the elements of the domain. Not only should this be done in a type-safe manner, the resulting code should also be recognizable as being Gaussian Elimination. As well, it should be clear that the technique could scale to the complete family of algorithms related to GE, like LU and Cholesky factorization, back-elimination, as well as more general matrix solving algorithms.

We want our solution to still contain a recognizable Gaussian Elimination algorithm. While it is possible to write GE functionally, an imperative solution is what is generally expected. The best compromise would be to write this in something like Haskell's State Monad which, via the do notation, would *look* right, yet still be functional. As we did not want to add additional complexity by creating some syntactic sugar (via campl4), we bit the bullet and dove in headlong into an imperative solution.

1.4 Contributions

We show that a number of techniques, namely combinators, abstract interpretation [20], state-passing style, and dictionary-passing at staging time can be successfuly combined to achieve our goals of writing a generic, efficient, type-safe implementation of the Gaussian Elimination family of algorithms. These techniques have to be very carefully combined, so that whenever a new design point of Gaussian Elimination is exposed, the generated (source) code does not include new unecessary artifacts, but in fact stays constant when this design point is specialized back to its "default" value.

The principal features and contributions of this article are

- a thorough analysis of a real situation,
- the use of staging for code configuration and clone management,
- the analysis of imperative-style code instead of functional, as is more customary,
- cleanly documenting and exposing all design-time choices at staging time,
- the use of state-passing style (for staging),
- and the use of dictionary passing at staging-time.

1.5 Organization

In section 2, we quickly review the Gaussian Elimination algorithm and examine 3 different variants, to get an idea of the design points involved. We then proceed with a more thorough analysis of the design points that we will tackle in this paper, followed by an analysis of the relations between these design points. Section 3 quickly introduces staging, as applied to an abstract Gaussian Elimination implementation. We carefully show how turning this algorithm into a sequence of code generators can be done in such a way as to not introduce any staging artifacts. We then show how to insert a particular design choice in to the staged version, and preserve this "no artifacts" property. Section 4 then introduces the various techniques used in the complete staging of GE. Only short examples are provided to illustrate the concepts, but the complete source code is included in an appendix. Related work is reviewed in section 6, and conclusions are drawn in section 7. There are 3 appendices, one for a fraction-free implementation of GE (in Maple), one for a feature-rich implementation of GE over algebraic extensions of finite fields (in Maple), and finally a complete listing of our final implementation of a staged GE in metaocaml.

2 Gaussian Elimination

After a quick review of the Gaussian Elimination algorithm, this section reviews some examples of GE implementations from Maple's library, to get an idea of some of the design points involved. This is followed by a more thorough analysis of the design points that were encountered, and which of those will be explicitly treated in this paper. We draw particular attention to those design points which are not fully orthogonal to other design points, as these need special handling to avoid producing inconsistent or meaningless code.

2.1 Review of GE algorithm

Gaussian Elimination is generally understood to be an algorithm which reduces (square) Matrices to upper-triangular form via a sequence of elementary Matrix operations, namely row/column interchanges and addition of a row multiple to another row. The basic requirements on the the Matrix elements are that they can be added, multiplied, and zero elements can be recognized. Many versions also require that division for non-zero Matrix elements also exists. The requirements on the Matrix are that all elements are accessible, and it is simple to do elementary row operations (swapping and adding a multiple).

Denote the elements of Matrix A by a_{ij} as usual. This algorithm proceeds as follows: in the first column, a non-zero element is chosen, say in row k. If this element is not in the first row, exchange row k and the first row. For every row j below the first one, add $-\frac{a_{j1}}{a_{11}}$ times the first row to row j; this ensures that in the resulting row, $a_{j1} = 0$. This is then repeated recursively on the submatrix obtained by deleting the first row and column. The algorithm can be modified to deal with columns with all zeroes, as well as dealing with non-square matrices. Details from a numerical analysis point of view can be found in Chapter 6 of [21], and for exact computations in Chapter 9 of [22]. The latter also contains an exposition of the fraction-free version of Gaussian Elimination. One can then show that this version of the algorithm works (ie reduces an $n \times n$ Matrix of rank n to an upper triangular matrix with a non-zero diagonal) whenever the elements of the Matrix are from an arbitrary *integral domain*. There are extensions to rings which contain zero-divisors as well, but these are beyond the scope of the current paper.

A frequent variation point is that for reasons of either numerical stability or coefficient growth, instead of choosing a random row k with a non-zero element, pivoting is performed by choosing an element that optimizes a particular metric. In the case of numerical stability, the largest (in absolute value) is chosen, whereas for coefficient growth cases, it is the smallest (according to some given size metric) that is chosen. For numerical stability, full pivoting is also frequently used, although we will not deal with this here.

2.2 Some examples

First, we discuss 3 different implementations of Gaussian Elimination from Maple's library. The first one (shown in figure 1) is the most straightforward implementation of Gaussian Elimination. However, it does include some explicit parametrization, namely the use of `Normalizer` and `Testzero` to allow the use of this algorithm with any arithmetic domain (in other words one where the globally defined, polymorphic operations and constants $+, *, /, -$, and 0 are defined, as well as having a notion of `length`). This implementation uses Maple's old `matrix` datatype which is in fact implemented via hash tables; while this makes element access very slow, it does allow very efficient storage of sparse matrices, as well as allowing *indexing functions* to be used for structured matrices. See the Maple documentation [23] for more details on these features. It is important to note that this implementation would **not** be appropriate for floating point numbers, as it would be numerically quite unstable.

The second version (see appendix A) is significantly more parametrized: it is implemented via Maple's `Domains` package which implements generic programming as originally found in Axiom [24]. These facilities are modelled on the structures of classical Algebra like Rings, Fields, Monoids, Groups, etc. While the resulting program is very generic, and in Axiom (at least when compiled via Aldor [25]) is quite efficient, in an interpreted language like Maple, the results are unacceptably slow. In any case, this program implements the fraction-free version of Gaussian Elimination, which works over any Division Ring. If the Division Ring is also a Euclidean Domain, then the underlying Euclidean Norm is used as the notion of *size*, otherwise the underlying universally polymorphic `length` function, which works for any Maple structure, is used. Again, one should note that floating point numbers do not form a Division Ring (as addition is not associative and multiplication does not distribute over addition). This implementation works for any container A (storing the matrix) which allows 2-dimensional indexing for element retrieval and storage. Unlike the previous version, neither `Normalizer` nor `Testzero` are used, which implies that the underlying rings are assumed to store their elements in normalized form, that arithmetic operations return results in normalized form, and thus zero-testing is straightforward.

The third variant (see Appendix B) is also quite parametrized, but along different dimensions: in this case the input domain is fixed to be an arbitrary algebraic extension (possibly empty) of a finite field $\mathbb{Z}_p$. This particular implementation has potentially multiple outputs: the actual reduced Matrix (always), the rank of the Matrix (sometimes), and the determinant of the Matrix (sometimes). It works for non-square matrices, as well as for augmented matrices, where one can pass in an

```
gausselim := proc(A::matrix)
local B,n,m,i,j,c,t,r;
n := rowdim(A); m := coldim(A);
B := map(Normalizer, A); r := 1;
for c to m while r <= n do
    # Search for a provably non-zero pivot element
    i := 0;
    for j from r to n do
        if not Testzero(B[j,c]) then
            if i = 0 then i := j;
            elif length(B[j,c]) < length(B[i,c]) then
                i := j
            end if; # otherwise B[i,c] is the best
        end if;
    end do;
    if i <> 0 then
        # interchange row i with row r is necessary
        if i <> r then
          for j from c to m do
            t := B[i,j]; B[i,j] := B[r,j]; B[r,j] := t
          end do
        end if;
        for i from r+1 to n do
          if B[i,c] <> 0 then
              t := Normalizer(B[i,c]/B[r,c]);
              for j from c+1 to m do
                B[i,j] := Normalizer(B[i,j]-t*B[r,j])
              end do;
              B[i,c] := 0
          end if;
        end do;
        r := r + 1          # go to next row
    end if
end do;              # go to next column
end proc:
```

Fig. 1. Generic Gaussian Elimination in Maple

integer `right_margin` parameter to specify this. The determinant can only be computed in the case where the principal matrix is square. Elements are not assumed normalized, and for efficiency reasons, the normalization procedure used for a pure finite field is different than the one used for algebraic extensions. There is also explicit code for keeping track of the determinant. This version is optimized for the newer `rtable` data-structure representation of Maple's newer `Matrix` constructor, which allows a one-line expression

```
B[i,1..-1], B[row,1..-1] := B[row,1..-1], B[i,1..-1];
```

```
let ge_float a =
  let b = copy(a) in
  let n = Array.length b and m = Array.length b.(0) in
  let r = ref 0 and c = ref 0 and i = ref 0 in
  while !c < m && !r < n do
    i := (-1);
    for j = !r to n-1 do
      if not ( b.(j).(!c) = 0.) then
        if !i = (-1) ||
           abs_float b.(j).(!c) > abs_float( b.(!i).(!c) ) then
             i := j;
    done;
    if !i != (-1) then begin
      if !i <> !r then
        begin
          for j = !c to m-1 do
              let t = b.(!i).(j) in
              b.(!i).(j) <- b.(!r).(j);
              b.(!r).(j) <- t;
          done;
        end;
      for ii = !r+1 to n-1 do
        if not (b.(ii).(!c) = 0.)  then begin
          let t = b.(ii).(!c) /. b.(!r).(!c) in
          for j = !c+1 to m-1 do
              b.(ii).(j) <-  b.(ii).(j) -. t*.b.(!r).(j)
          done;
          b.(ii).(!c) <- 0.;
        end;
      done;
      r := !r + 1;
    end;
    c := !c + 1;
  done ;
  b
```

Fig. 2. Floating-point GE in Ocaml

to be used for exchanging 2 rows.

For reference, the straightforward implementation of Gaussian Elimination for matrices of floating point numbers, in OCaml is shown in figure 2. It has been coded to resemble the first Maple version in figure 1 as much as possible, where this makes sense; the most significant difference is that pivoting is done on the largest number instead of the smallest.

A careful review of the rougly 35 implementations of GE we found, as well as roughly 50 additional implementations of related algorithms, a number of implementation patterns became readily apparent. The main variations found were:

(1) **Domain**: In which (algebraic) domain do the matrix elements belong to. Some implementations were very specific ($\mathbb{Z}, \mathbb{Q}, \mathbb{Z}_{|}, \mathbb{Z}_p\left[\alpha_1, \ldots, \alpha_n\right], \mathbb{Z}\left[x\right]$, $\mathbb{Q}\left(x\right)$, $\mathbb{Q}\left[\alpha\right]$, and floating point numbers for example), while others were generic for elements of a field, multivariate polynomials over a field, elements of a formal Ring with possibly undecidable zero-equivalence, or elements of a Division Ring. In the roughly 85 pieces of code we surveyed, 20 different domains were encountered.
(2) **Fraction-free**: Whether the Gaussian Elimination algorithm is allowed to use unrestricted division, or only exact (remainder-free) division.
(3) **Representation of the matrix**: Whether the matrix was represented as an array of arrays, a one-dimensional array, a hash table, whether indexing was done in C or Fortran style. Additionally, if a particular representation had a special mechanism for efficient row exchanges, this was sometimes used. Other representations were also used.
(4) **Length measure (for pivoting)**: For stability reasons (whether numerical or coefficient growth), if a domain possesses an appropriate length measure, this was sometimes used to choose an "optimal" pivot for row exchanges.
(5) **Output choices**: Whether just the reduced matrix, or additional the rank and the determinant were to be returned. It is worthwhile noting that in the larger algorithm family, routines like `LinearAlgebra:-LUDecomposition` have up to $2^6 + 2^5 + 2^2 = 100$ possible outputs.
(6) **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 choices are not completely independent:

- While it is always possible to use full Gaussian Elimination for matrices with elements in a Division Ring, the result will in general lie in the Field of fractions associated with this Ring. On the other hand, fraction-free GE will always give results in the same domain as the input. There are subtle coefficient growth reasons that come into play when deciding, for each domain, the "best" version of GE to use.
- Some domains do not have meaningful length measures. In particular $\mathbb{Z}_p$ is such a field; but by the same token, there is no coefficient growth problem with $\mathbb{Z}_p$. On the other hand, even though floating point numbers are of fixed syntactic size, numerical stability implies that the absolute value gives a meaningful length measure.

- While zero-testing is often done via normalization, there are times when full normalization is very expensive, and heuristic-driven zero-testing can be fruitfully used as a filter for “easy” zero equivalence, before more expensive techniques are used.

A few additional design points were encountered in some cases:

- If zero equivalence was known to be undecidable, and a case was encountered where a (provably) non-zero pivot could not be found, either a warning or an error were issued.
- Some implementations had embedded `userinfo` statements, which is Maple lingo for user-controlled directives that give the user information about the execution state of particular algorithms.
- Specification that the Matrix was in fact an augmented matrix, and that it possessed a natural “right margin”. Elimination is then performed on the full matrix, but column choices stop at this given right margin. Curiously, while this was frequently implemented, this feature was very rarely documented, even though this functionality is frequently used in textbooks which outline how Gaussian Elimination can be used on augmented matrices to implement solving of linear equations and matrix inversion.

3 Staging Gaussian Elimination

Using an abstract version of Gaussian Elimination, and assuming that the reader has some knowledge of staging in MetaOCaml (as expounded in the other papers in this volume), we show a first naïve staging of Gaussian Elimination, based on an abstract version of the algorithm.

3.1 Abstract Gaussian Elimination

Looking at all the Gaussian Elimination algorithms seen so far, one can extract a staging-free high-level Ocaml version as follows:

```
let ge_high findpivot swap zerobelow a m n =
    let b = copy(a) in
    let r = ref 0 and c = ref 0 in
    while !c < m && !r < n do
        match findpivot b c r n with
        Some i -> begin
            if i <> !r then
                for j = !c to m-1 do
                    swap b i r j;
                done;
            zerobelow b r c n m;
            r := !r + 1;
            end;
        | None -> () ;
        c := !c + 1;
    done ;
    b
```

For a version of GE that works with arrays of arrays of floating point numbers, the 3 required routines can be given as

```
let findpivot b c r n =
  let i = ref (-1) in begin
    for j = !r to n-1 do
      if not ( b.(j).(!c) = 0.) then
        if !i = (-1) ||
          abs_float b.(j).(!c) < abs_float b.(!i).(!c) then
            i := j;
    done;
    if !i== -1 then None else Some !i;
  end;

let swap b i r j =
  let t = b.(i).(j) in begin
    b.(i).(j) <- b.(!r).(j);
    b.(!r).(j) <- t;
  end;

let zerobelow b r c n m =
  for ii = !r+1 to n-1 do
    if not (b.(ii).(!c) = 0.)  then
      begin
        let t = b.(ii).(!c) /. b.(!r).(!c) in
        for j = !c+1 to m-1 do
          b.(ii).(j) <-  b.(ii).(j) -. t*.b.(!r).(j)
        done;
        b.(ii).(!c) <- 0.;
      end;
    done;
```

We can recover the previous version `ge_float` as

```
let ge_float a =
    let n = Array.length a and m = Array.length a.(0) in
    ge_high findpivot swap zerobelow a m n
```

It is however worthwhile to examine closely the type inferred for `ge_high`:

```
# val ge_high :
  ('a -> int ref -> int ref -> int -> int option) ->
  ('a -> int -> int ref -> int -> 'b) ->
  ('a -> int ref -> int ref -> int -> int -> 'c) ->
  'a -> int -> int -> 'a = <fun>
```

The “matrix” `a` is completely abstracted out, and we have a fully abstract `'a` as its type. Only when `findpivot` and friends are instantiated do we get this particular type refined. As well, since the matrix type is abstract, this also abstracts out the type of elements. So this abstraction exercise has already gained us (some) representation independence.

3.2 First staging

Of course, if one were either greedy for every single microsecond of efficiency or, as in our case, we are preparing for a thorough staging of the above routine, we would like to modify the above code to completely remove all (run-time) traces of these abstractions. This can be done as follows:

```
let ge_gen findpivot swap zerobelow =
    .< fun a m n ->
    let b = copy(a) in
    let r = ref 0 and c = ref 0 in
    while !c < m && !r < n do
        match .~(findpivot_gen .<b>. .<c>. .<r>. .<n>.) with
        Some i -> begin
            if i <> !r then
                for j = !c to m-1 do
                    .~(swap_gen .<b>. .<i>. (lr .<r>.) .<j>.);
                done;
            .~(zerobelow_gen .<b>. .<r>. .<c>. .<n>. .<m>.);
            r := !r + 1;
            end;
        | None -> () ;
        c := !c + 1;
    done ;
    b
```

While the source code above is not particularly attractive, the modifications necessary are certainly quite routine, and the resulting code is quite readable. To get routine `swap_gen` from routine `swap` is also straightforward:

```
let lr x = .< ! .~ x >.
let swap_gen b i r j =
    .< let b = .~b and i = .~i and j = .~j and r = .~r in
        let t = b.(i).(j) in  begin
        b.(i).(j) <- b.(r).(j);
        b.(r).(j) <- t;
    end >.
```

However, this produces some extra let bindings in the resulting code (not shown here). To get back to the original source code, as intended, the modification is still simple, although the resulting code (below) now looks quite ugly. We will assume that these let bindings are harmless (in other words completely removed by the compiler), and use them in our presentation, since we know we could remove them if needed.

```
let swap_gen2 b i r j =
    .< let t = (.~b).(.~i).(.~j) in  begin
        (.~b).(.~i).(.~j) <- (.~b).(.~r).(.~j);
        (.~b).(.~r).(.~j) <- t;
    end >.
```

Similar modifications to `zerobelow` and `findpivot` are necessary as well. Once performed, we can obtain `ge_high` verbatim via specialization:

```
let spec_ge = (ge_gen findpivot_gen swap_gen2 zerobelow_gen);
let ge_float2 a =
    let n = Array.length a and m = Array.length a.(0) in
    let ge = .! spec_ge in
    ge a m n ;
```

Close inspection of the program `spec_ge` shows that, up to variable renaming, the resulting code is essentially that of `spec_float`. The only real change (and it is not strictly necessary) is the use of the `option` type to pass information out from `findpivot` instead of using a flag value of -1 for the integer reference variable `i`.

3.3 Optimized pivoting

The first "design point" that we will stage is whether we simple use the first non-zero element we encounter as a pivot, or whether we search in the complete column for the "best" pivot. We will thus assume that we are (potentially) given a function `better_pivot` which, given two values, will return `true` if the second value would be a better pivot. By using an option type, we can also choose whether we want to use this function. We parametrize `findpivot_gen` by this new value, giving:

```
let findpivot_gen better_pivot b c r n =
    let cond i x y = match better_pivot with
    | Some f -> .< !(.~i) = (-1) || .~(f x y) >.
    | None ->   .< !(.~i) = (-1) >.
    in
    .< let i = ref (-1) in begin
        for j = !(.~r) to .~n-1 do
            if not ( (.~b).(j).(!(.~c)) = 0.) then
                if .~(cond .<i>. .< (.~b).(j).(!(.~c)) >.
                           .< (.~b).(!i).(! .~c) >. ) then
                    i := j
        done;
        if !i == -1 then None else Some !i;
    end >. ;;
```

where we have added a new (local) code-generating function `cond` which generates either single condition `!i = -1` or splices in the code of our passed-in `pivot_len` function. Since this is a code-generator, the body of `findpivot_gen` has to be modified accordingly to pass in the *code* representations necessary for properly constructing the conditional.

Naturally, we need to make a corresponding change to the instantiation call. To reproduce the previous program, this would be done as

```
let spec_ge =
  let fp = findpivot_gen (Some
    (fun x y -> .<abs_float .~x < abs_float .~y >. )) in
  ge_gen fp swap_gen2 zerobelow_gen;
```

4 Staging techniques

While it is possible to continue staging the code in the naïve manner described in the previous section, eventually the resulting source code becomes almost impossible to read. In other words, this ad hoc method of staging, as we found out first-hand, does not scale very well, nor is it always straightforward. Also, this suggest that maintenance would be very challenging – yet one of the explicit goals of this exercise to help maintainability. We need to find smarter ways to combine and stage code, to simultaneously obtain readable and efficient programs. This section presents various techniques that we have used to successfully stage all of the principal design points described in subsection 2.3.

We will use a combination of four techniques: the use of basic combinators for pieces of code, state passing, abstract interpretation, and dictionary passing. It is important to note that we are staging (deeply!) imperative code, to obtain imperative code. Thus we are forced to develop some techniques which differ from previous work [26,27] where they were staging purely functional code, sometimes with the aim of obtaining functional residual code, and sometimes aiming for imperative residual code. Of course, some of the techniques developped below have similar semantic underpinnings as the monadic approach of [27], although this is probably not readily apparent when syntactically comparing our results with theirs.

4.1 Preliminaries

Probably the most important point to remember when staging code, especially in a higher-order language like Ocaml, is that the usual abstraction technique of using higher-order functions does not quite work as expected anymore. The only object that can be effectively inlined has to be of `code` type; in particular it cannot be of a functional type. More subtly, we cannot just "lift" functions in the hopes of still being able to apply them before run-time, we instead have to be careful to deal with (lifted) function bodies. For example, compare

```
let lift x = .< x >.
let star1 = lift ( * ) in
let f1 x y = .< .~star1 (x+1) (y+3) >. in
f1 2 3 ;;

let star2 x y = .< .~x * .~y >. in
  let f2 x y = star2 .<x+1>. .<y+3>.
  and f3 x y = let z=x+1 and zz=y+3 in
      star2 (lift z) (lift zz)
  and f4 x y = let z=x+1 and zz=y+3 in
      star2 .<z>. .<zz>. in
[ f2 2 3 ; f3 2 4 ; f4 2 3] ;;
```

which gives as results

```
# - : ('a, int) code =
.<(((* compiled code omitted (as id: x) *))
    (2 + 1) (3 + 3))>.
# - : ('a, int) code list =
[.<((2 + 1) * (3 + 3))>.;
 .<((* compiled code omitted (as id: x) *) *
    (* compiled code omitted (as id: x) *))>.;
 .<(3 * 6)>.]
```

All of these are operationally equivalent! However, only for `f4` do we get what we were really after. Careful reading of [26] explains all of these, but they are still a bit disconcerting to the novice MetaOCaml programmer. While the difference between `f2` and `f4` is easy to explain in terms of binding-time, the difference between `f1` and `f4` takes more getting used to for the seasoned functional programmer, who is used to passing around functions. It is very important to pass around code instead - which is the main difference between `star1` and `star2`: the first is completely closed code which can only applied at run-time, while the latter can still be applied at stage 1. This may be obvious to meta programming gurus, but is not emphasized enough in practical introductions to staging in MetaOCaml.

4.2 Combinators

It is very natural to try to develop *code combinators*, to attempt to do for code what worked so well for parsers [28]. While this works to a certain extent, the next technique in part destroys the applicability of pure code combinators to many cases of interest, as we are here mainly dealing with combining imperative program fragments.

While we experimented with many pure code combinators, in the end the only one we actually used was a simple lift of sequencing:

```
let seq a b = .< begin .~a ; .~b end >.
```

It is worthwhile noting that the begin/end are needed, otherwise MetaOCaml complains that this is syntactically invalid. Another useful combinator (used in development) was composition:

```
let compose a b = fun x -> .< .~a ( .~b x ) >.
```

While it is possible to create many more combinators, like

```
let _if_ cond body = .<if .~cond then .~body >. ;;
```

in actual use we have found that this ends up making the code much less readable than staging larger pieces of code. But the reader should remember that our aims are quite different than that of other authors, so these conclusions a priori might only apply to our particular setting.

4.3 State passing

The first really new technique (in the context of staging) is that of *state passing style*. This is taken simultaneously from two different sources: the state-transformer paradigm of denotational semantics for programming languages [29], the State-Monad and ST monads of Haskell, and Sumii's work [30] on hybrid online-offline partial evaluation, where a state-passing style was used over a continuation-passing style, to great effect.

While in these general paradigms, the state space is usually open [5], we want much

[5] in other words is potentially infinite

tighter control on the state. So we use an explicit record type to encode our state configuration, with the finite set of state variables that our complete program will use. We can then rely on the type-checker to make sure that this is the only state we use. For example, for our needs we define

```
type 'a state =
    {b:'a array array; r:int ref; c:int ref; m:int; n:int}
```

which then forces us to modify all our generators to refer to this state configuration explicitly, and we have to add record dereferences as appropriate.

```
let swap_gen s i j =
    .< let s = .~s and i= .~i and j= .~j in
       let t = s.b.(i).(j) in  begin
        s.b.(i).(j) <- s.b.(!(s.r)).(j);
        s.b.(!(s.r)).(j) <- t;
    end >.
```

While this does make some code a bit harder to read, this has (essentially) no run-time cost, and definitely cuts down on the number of parameters needed for each generator. The main generator then needs to be modified to create this state configuration

```
.< fun a ->
   let s = {b=a; r=ref 0; c=ref 0; m=Array.length a.(0);
         n=Array.length a} in
  ... >.
```

Note that it is possible to remove even this (potential) overhead of record selection, by defining our state configuration to contain code generators instead of values. See subsection 4.5 for an example of this.

4.4 Abstract interpretation

Programmers coming from systems that permit unbounded manipulation of white-box code (Scheme, Maple, Mathematica, etc) need to remember that white-box code cannot exist in MetaOCaml. Thus any information about a piece of code that is necessary to be re-used later must be explicitly exposed, since such information cannot be seen once it is encapsulated in a code type.

Suppose we know a priori that the generic code contains `if !i = -1 then`, but that sometimes we would like to modify this to
`if !i = -1 ||`
`abs_float b.(j).(!c) > abs_float( b.(!i).(!c) ) then`, if we had white-box code, this could be done by actually finding that code, and explicitly

modifying it. In metaocaml, this technique is not available. There are reasons for this (like voiding equational reasoning principles on computations inside brackets [26], as this essentially reduces the brackets to syntactic questions [31]). Instead, one must explicitly insert a code generator
`if .~(cond_gen .<i>. etc) then` which will conditionally generate the correct code. The question then becomes, how do we indeed conditionally generate the correct code for each particular instance?

We use a simple case of the general technique of *abstract interpretation* [20] to solve this problem; see [27] for a different use of the same technique. The main idea is to take the fully abstract code type, and to add some additional information to it, information which can be used at staging time to generate more precise code. This additional information is an approximation of the complete information contained in the white-box code. In our case, this translates to knowing whether our domain contains the functions necessary to optimize the choice of a pivot. Thus we define a type

```
type pivoting_code_option =
  (('a,'b) code -> ('a,'b) code -> (('a,bool) code)) option;
```

which says that we *might* have a "smaller than" boolean code generator available. This is used with a conditional code generator

```
let orcond_gen main_cond = function
    | Some alt_cond -> .< .~main_cond || .~alt_cond >.
    | None -> main_cond ;;
```

to generate the required code depending on the situation. A more complex use of the same technique is when a domain requires a normalization step; in that case, we have to insert the normalization code into the abstract code for mapping over our generic two-dimensional container. We end up using code that looks like

```
let mdapply1 mapper g c1 = match g with
    | Some f -> .< .~(mapper f c1) >.
    | None   -> c1 ;

.~(mdapply1 mapper normalizerf .<a>. )
```

where `mapper` is a code generator for mapping a function `f` (also given as code) over the code of an abstract container. Also see section 7 for a more substantial application of abstract interpretation.

4.5 Dictionaries

While it is possible to use just the techniques of the previous sections to stage a fairly generic GE, the resulting source code would be hideous. In particular, most generators would have between 15 to 20 parameters, since a typical domain has 10 different components, and a generic container has 5. Instead, it makes a lot more sense to gather these parameters together into a type.

To be more specific, let us look at domains. Since they are essentially Division Rings, we know that they must contain the constants zero and one, as well as the operations of addition, subtraction, multiplication and (a restricted kind of) division. It may optionally come with a "smaller than" relation, a normalizer, and a zero-tester. For simplicity, we can also assume that we are given an explicit -1 value as well. So we could define a type corresponding to a domain over some arbitrary base set:

```
type 'b domain = {zero:'b; one:'b; minusone:'b;
    plus:'b -> 'b -> 'b; times:'b -> 'b -> 'b;
    minus:'b -> 'b -> 'b; div:'b -> 'b ->'b;
    smaller_than:('b -> 'b -> bool) option;
    normalizer:('b -> 'b ) option;
} ;;
```

It should be remarked that what we are essentially building there is a type for a (static) dictionary for domains. This is akin to (simple) type classes in Haskell, or indeed of defining a module in OCaml over an abstract type.

If we lift this domain up one stage, then via partial application and inlining, we could hope to completely eliminate the overhead of this level of genericity. But doing things this way would run into the problems already outlined in subsection 4.1. While we would get a program which is operationally correct, and which Ocaml's underlying optimizer *might* resolve to what we want, we certainly have no guarantees of that. To be quite sure of this, we need to provide not the values and functions that correspond to a Division Ring, but rather the code (and code generators) for each of these. In other words, the type we really need is

```
type ('a,'b) domain = {zero:('a,'b) code; one:('a,'b) code;
  minusone:('a,'b) code;
  plus:('a,'b) code -> ('a,'b) code -> ('a,'b) code;
  times:('a,'b) code -> ('a,'b) code -> ('a,'b) code;
  minus:('a,'b) code -> ('a,'b) code -> ('a,'b) code;
  div:('a,'b) code -> ('a,'b) code ->('a,'b) code;
  smaller_than:(('a,'b) code -> ('a,'b) code ->
             (('a,bool) code)) option;
  normalizerf:(('a,'b -> 'b) code) option;
  normalizerg:(('a,'b) code -> ('a,'b) code) option;
} ;;
```

We need two different versions of the normalizer function because we might use the normalizer in two different ways: either in an imperative setting, like in the body of a `for` loop, or in a functional setting, like as an argument to `Array.map`.

For example, one can define the domain of integers $\mathbb{Z}$ as

```
let dom_int = { zero = .< 0 >. ; one = .< 1 >. ;
   minusone = .< -1 >. ;
   plus = (fun x y -> .<.~x + .~y>.) ;
   times = (fun x y -> .<.~x * .~y>.) ;
   minus = (fun x y -> .<.~x - .~y>.) ;
   div = (fun x y -> .<.~x / .~y>.) ;
   smaller_than = Some(fun x y -> .< abs .~x < abs .~y >.) ;
   normalizerf = None ;
   normalizerg = None }
```

where we have been careful to define code-generating functions rather than code-of-functions. We have also completely eschewed the use of `lift`, to ensure we get complete inlining. This might be ultimately be unecessary, but we would have to trust the MetaOCaml `run` function `.!` to systematically do this for us, which[6] we cannot rely on.

Using similar techniques, we can also completely abstract out the matrix container as follows:

```
type ('a,'c,'d) container2d = {
  get: ('a,'c) code -> ('a,int) code ->
       ('a,int) code -> ('a,'d) code ;
  set: ('a,'c) code -> ('a,int) code ->
       ('a,int) code -> ('a,'d) code -> ('a,unit) code ;
  dim1: ('a,'c) code -> ('a,int) code;
  dim2: ('a,'c) code -> ('a,int) code;
  mapper: ('a, 'd->'d) code -> ('a,'c) code -> ('a,'c) code
}
```

where `'a` is the extra type variable required by the abstract code type, `'c` is the abstract container type, and `'d` is the abstract element type. Even though from `get` and `set`, we could derive the code for `mapper`, letting the user give an explicit `mapper` functions allows for extra efficiency. This could certainly be considered to be an additional design point.

It is worthwhile noting that these 2 types are completely independent, and only when are they actually used together will the type system unify the 3rd type variable of `container2d` with the 2nd type variable from `domain`. This is quite pleasing, as it reassures us that we have an appropriate separation of concerns between the domain of elements and the abstract container.

These 2 types have values that are fully available at staging time. Thus any overhead that could be associated with these abstractions is in fact completely eliminated,

[6] for theoretically sound reasons!

and the generated code is free of these abstractions. In other words, we were able to use staging instead of defunctionalisation [32] to remove a complete layer of abstraction.

5 Staging Gaussian Elimination, continued

The techniques of the previous section allow us to take care of all of the major design points that were outlined in subsection 2.3. For example, by using a combination of abstract interpretation and a code combinator, we can track which information is needed in the final code, allowing the final code generator for the swap routine to look like

```
let swapr_gen dom contr track s i j =
  let swap =
  .< let b = (.~s).b and r = !((.~s).r)
       and i= .~i and j= .~j in
     let t = .~(contr.get .<b>. .<i>. .<j>. ) in  begin
       .~(seq
            (contr.set .<b>. .<i>. .<j>.
              (contr.get .<b>. .<r>. .<j>. ))
            (contr.set .<b>. .<r>. .<j>. .<t>. ) );
  end >. in
  let tds =
     .< (.~s).detsign :=
   .~(dom.times .<!((.~s).detsign)>. dom.minusone) >. in
  match track with
  | TrackNothing -> swap
  | TrackDet     -> seq swap tds
```

where we see the use of the abstract container `cont`, the abstract domain `dom`, the tracking information `track`, and the combinator `seq`.

The one area that was slightly less satisfying was in staging the choice of outputs. While any given instantiation of GE will produce a specific output, the family of algorithms exposes a choice. Since typing in MetaOCaml is done once, this choice persists in the *type* of the final program, even though it should be readily apparent that this choice does not in fact persist in the actual generated code. More precisely, we use a type

```
type ('b,'c) outputs =
      Matrix of 'b
    | MatrixRank of 'b * int
    | MatrixDet of 'b * 'c
    | MatrixRankDet of 'b * int * 'c
```

for the output of GE. Clearly any generated code will only use one of these choices, but this will not be apparent in the type of the generated code.

In the end, the type of our function `specializer` is

```
val specializer : ('a, 'b) domain ->
  ('a, 'c, 'b) container2d -> fracfree:bool ->
  outputs:outchoice -> ('a, 'c -> ('c, 'b) outputs) code
```

which says that it outputs code for a function of type `'c -> ('c, 'b) outputs` with the constraints that `'c` be a container type and `'b` be a domain type. A complete listing, with a specialization, is given in Appendix reffinalcode.

6 Related work

There is now quite an established line of work in scientific (numerical) programming of using generative techniques (mainly in C++, see [13,14,18] and the work they cite). However, their worries are somewhat different than ours: they mainly aim for efficiency - they are able to generate code which is usually faster than the handwritten Fortran programs they replace. While they do not overlook maintainability issues, that is often a secondary concern. This drive for efficiency usually means worrying about many low-level details (cache sizes, cache misses, etc), whereas we are much more interested in writing very generic programs which can hopefully still be quite efficient. While clearly very successful, with FFTW [33] being the most famous this work cannot claim to be type-safe. In the case of FFTW, the work of Kiselyov, Swadi, and Taha [27] is probably the first that can claim type safety.

Our work is much more closely related to the work on generic programming coming out of Symbolic Computation and Computer Algebra [9,24]. Computer Algebra systems, starting with ScratchPad [34], on through Axiom [24] and Aldor [25] worked very hard at modelling the structures encountered in mathematics (Groups, Rings, Fields, Modules, etc) through abstract data types in various programming languages. This allowed them to code many algorithms very generically, allowing extremely general mathematical structures to be defined and computed with. Unfortunately, this very genericity came at an extremely high efficiency cost; part of this is due to the fact that many mathematical structures (with Fields and Matricies being good examples) are not inductive data types. In fact, any reasonable type system for properly reflecting mathematics requires dependent types (as was already known to the AUTOMATH people in 1968 [35], but has been re-discovered many times). In Maple itself, the *Domains* package (based on an older package called Gauss [36]) successfully implements this strategy, but at a whopping factor of 30 efficiency cost over base Maple, which is already a factor of 60 slower than comparable C code. This is why generative techiques must be used to (at least) regain that last factor of 30. What we show here is that (in the context of MetaOCaml), this indeed works.

Others have used partial evaluation techniques to solve the "configuration" problem, closely related to what we have done here. In particular, the work of Consel and his colleagues [37,38] is relevant. They use a declarative language to declare the specialization points of components; the components are written in a mainstream programming language (here C), without the need for annotations. While very pragmatic, this inherits all of the defects of the host language - and in this case this means that type-safety is completely lost. Additionally, it is clear that such an approach could not be used to express nearly as many design points as we have been able to express in quite compact code. Nevertheless, some of the underlying ideas of this work is very close to ours.

Lastly, it should be noted that the programs we end up writing are very reminiscent of hand-written language interpreters in cogen style [39,40]. In hindsight, this is not really surprising, as this is the style that MetaOCaml forces upon us, by not having white-box code. Thus we have to be writing code generating functions.

7 Conclusion

We have shown how to combine several techniques namely combinators, abstract interpretation, state-passing style, and dictionary passing (at staging time) can be carefully and successfuly combined to achieve our goals of writing a generic, efficient, type-safe implementation of the Gaussian Elimination family of algorithms. Furthermore, the generated code does not include unecessary artifacts introduced by generalizing the code over several design points.

It should be easy to see this these techniques generalize immediately to the larger family of algorithms containing LU decomposition, Cholesky factorization, forward as well as backward elimination, and of course linear system solvers.

7.1 Design points revisited

Reviewing the design points listed in subsection 2.3, we have shown how to abstract out domains, whether the algorithm should be fraction-free or not, matrix representations, the use of a size measure (when available) for pivoting, output choices, and the potential need for normalization.

On the other hand, we have not shown how to deal with domains where zero-testing is undecidable, nor in fact on how to use a special zero-testing routine when it is given; both of these are straightforward using the techniques shown here.

Something else which has not been adequately addressed is whether a particular domain requires that a fraction-free algorithm be used or not – in effect whether the input domain is in fact a Field or just a Division Ring. We could simply add a new boolean field `is_field` to our domain type, but this is not very elegant. It would be much more pleasant if the compiler had some Theorem Proving capabilities, and we could instead let the compiler decide this property.

Although we showed how to use a domain's length function, we are hard-coding the `smaller_than` function. Akin to the decision to use a fraction-free algorithm, it would be much better to let the compiler somehow prove that we are either in a numerical domain that suffers from instability issues, or in a symbolic domain which suffers from coefficient growth issues, and conclude how the length function should be used.

While we have shown how to deal with output choices, our approach clearly does not scale: given something like the current implementations for LU Decomposition in Maple, we would have to use an algebraic type with 100 constructors. This is clearly not a realistic solution. However, in these cases, a more sophisticated type system could perhaps deal with the issue: the output type is quite simple at stage 0. Thus if there could be *staged types* as well as staged code, this problem could be easily handled. Another intriguing possibility would be to use GADTs. Naturally, this could also be handled via dependent types, but in this instance, this seems like overkill.

Lastly, the container type could be abstracted out even more. Our algorithm right now is only really well-suited to dense rectangular matrices. Any kind of structured matrix, especially a sparse matrix, could benefit from a differently structured traversal strategy. Simple issues like faster methods for row exchange would be quite straightforward to handle, but handling sparsity is more challenging. We believe that the techniques we show here are sufficient to handle this as well, but would require a more significant restructuring of the code generators. Note that we have also completely ignored some issues of frequent concern in implementations of Gaussian Elimination for large matrices of floating point numbers, like full pivoting, blocking, etc. Full pivoting would be quite straightforward to implement, whereas blocking would be more challenging.

7.2 Future work

We intend to deal with structured matrices next, as well as actually implementing larger portions of the algorithm family centered around Gaussian Elimination. We then hope to continue generalizing this work – as it is known that Gröbner bases degenerate to Gaussian Elimination in the case of linear systems, it would be interesting to see if we could reproduce this via staging as well. This is definitely considerably more difficult, as linearity is a *property* of the input system. Dealing with the complete family of algorithms related to Gaussian Elimination, and linear algebra algorithms in general, should be relatively straightforward, and we intend to complete this work to ensure this is indeed the case.

We also want to pursue more aggressively the issue of abstract interpretation. One item that intrigues us is how straightforward it is in our setting to generalize the code shown in section 2 of [27]:

```
type ('b,'c) abstract = Elem of 'b | Any of 'c
type ('a,'b) monoid = {one: 'b; op:'b->'b->'b;
    op_gen:('a,'b) code->('a,'b) code->('a,'b) code}

let concretize = function
    | Elem x -> .<x>.
    | Any  x -> x
and generalize x = Elem x

let special_op dom a y =
    if a=dom.one then
        y
    else
        Any (dom.op_gen .<a>. .< .~(concretize y) >.)

let gen_op dom = fun x y ->
    match (x,y) with
    | (Elem a, Elem b) -> Elem (dom.op a b)
    | (Elem a, y)      -> special_op dom a y
    | (x, Elem b)      -> special_op dom b x
    | (x, y)           -> Any (dom.op_gen
      .< .~(concretize x) >. .< .~(concretize y) >.)
```

This code generates an abstract operator for any monoid. We need to specify the (actual) identity for the monoid, as well as the monoid operation and a code generator for the monoid information. Then we use abstract interpretation to evaluate the monoid operations as early as possible. Note that here we are able to evaluate any monoid operation, and not just operations involving the identity. We can easily specialize this to the case covered in [27] via

```
let ( ** ) x y = gen {
    one = 1 ; op = (fun x y -> x * y);
    op_gen = fun x y -> .< .~x * .~y >. } x y
let rec power n x =
    if n=0 then generalize 1 else x ** (power (n-1) x)
let power3 =
    .< fun x -> .~(concretize (power 3 (Any .<x>.))) >.
```

and we get the expected result of

```
val power3 = .< fun x -> x*x*x >.
```

Better still, the version of power above, apart from the use of `generalize`, looks very much like the power code that one would naïvely write (modifying it to do binary powering is straightforward). The implementation above assumes a commutative monoid, which is frequently the case (and is the case here), but it can easily be generalized to the non-commutative case. On the other hand, the above version is easily generalized to do normalization of multivariate monomials over an arbitrary base monoid.

While we have used records here to encode all our abstract types, we really should have used Ocaml's modules, as that is the proper way to encode dictionaries in Ocaml. If we then use Functors, for both input and output configuration [**?**], we can make our implementation even clearer, as well as offering a solution to the problem of configurable output types.

Lastly, we want to investigate the additional power that using a combination of monadic and continuation passing style may bring. It would appear that staging certain complex aspects (like determinant tracking) can be made considerably simpler by using this style.

7.3 Acknowledgments

This work started with a visit by the author to Walid Taha while the latter was still at Yale. Thanks to his gentle prodding, this paper finally got written. This paper benefited from several useful conversations with Bill Farmer on the topics of genericity, generative software, and compilers as theorem provers. Thanks go to David Linder for helping the author catalogue all the implementations of Gaussian-Elimination related algorithms in Maple, and Frédéric Chyzak for drawing the authors' attention (years ago) to this issue. Not least, Oleg Kiselyov made some very very fruitful comments on a draft of this paper, for which author is grateful.

References

[1] N. Jones, C. Gomard, P. Sestoft, Partial Evaluation and Automatic Program Generation, Prentice-Hall, 1993. 3

[2] H. Abelson, R. K. Dybvig, C. T. Haynes, G. J. Rozas, N. I. A. IV, D. P. Friedman, E. Kohlbecker, G. L. Steele Jr., D. H. Bartley, R. Halstead, D. Oxley, G. J. Sussman, G. Brooks, C. Hanson, K. M. Pitman, M. Wand, Revised report on the algorithmic

language Scheme, Higher-Order and Symbolic Computation 11 (1) (1998) 7–105.
URL http://www.wkap.nl/oasis.htm/168705 3

[3] K. Czarnecki, J. T. O'Donnell, J. Striegnitz, W. Taha, DSL implementation in MetaOCaml, Template Haskell, and C++., in: C. Lengauer, D. S. Batory, C. Consel, M. Odersky (Eds.), Domain-Specific Program Generation, Vol. 3016 of Lecture Notes in Computer Science, Springer, 2003, pp. 51–72. 3, 1.2

[4] D. L. Parnas, On the design and development of program families., IEEE Trans. Software Eng. 2 (1) (1976) 1–9. 1.1

[5] D. L. Parnas, On the criteria to be used in decomposing systems into modules., Commun. ACM 15 (12) (1972) 1053–1058. 1.1

[6] E. W. Dijkstra, On the role of scientific thought, published as [7] (Aug. 1974).
URL http://www.cs.utexas.edu/users/EWD/ewd04xx/EWD447.PDF
4

[7] E. W. Dijkstra, On the role of scientific thought, in: Selected Writings on Computing: A Personal Perspective, Springer-Verlag, 1982, pp. 60–66. 4, 6

[8] N. Wirth, Program development by stepwise refinement., Commun. ACM 14 (4) (1971) 221–227. 4

[9] D. R. Musser, A. A. Stepanov, Generic programming, in: ISSAC 1988: Proceedings of the International Symposium on Symbolic and Algebraic Computation, Vol. 358 of Lecture Notes in Computer Science, Springer-Verlag, 1989, pp. 13–25.
URL http://citeseer.lcs.mit.edu/musser88generic.html 1.2, 6

[10] D. R. Musser, A. A. Stepanov, Algorithm-oriented generic libraries, Software - Practice and Experience 24 (7) (1994) 623–642.
URL
http://citeseer.lcs.mit.edu/musser94algorithmoriented.html
1.2

[11] R. C. Backhouse, J. Gibbons (Eds.), Generic Programming - Advanced Lectures, Vol. 2793 of Lecture Notes in Computer Science, Springer, 2003. 1.2

[12] P. G. Myers, SPOM: a regional model of the sub-polar North Atlantic, Atmostphere-Ocean 40 (4) (2002) 445–463. 1.2

[13] K. Czarnecki, U. W. Eisenecker, Generative programming: methods, tools, and applications, ACM Press/Addison-Wesley Publishing Co., 2000. 1.2, 6

[14] T. 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. 1.2, 6

[15] T. L. Veldhuizen, D. Gannon, Active Libraries: Rethinking the roles of compilers and libraries, in: Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing (OO'98), SIAM Press, 1998. 1.2

[16] I. John V.W. Reynders, J. C. Cummings, The POOMA framework, Comput. Phys. 12 (5) (1998) 453–459. 1.2

[17] J. Siek, L.-Q. Lee, A. Lumsdaine, The Boost Graph Library: User Guide and Reference Manual, Addison-Wesley, 2002. 1.2

[18] R. C. Whaley, A. Petitet, J. J. Dongarra, Automated empirical optimization of software and the ATLAS project, Parallel Computing 27 (1–2) (2001) 3–35, also available as University of Tennessee LAPACK Working Note #147, UT-CS-00-448, 2000 (`www.netlib.org/lapack/lawns/lawn147.ps`). 1.2, 6

[19] G. Butler, K. Czarnecki, D. Batory, U. Eisenecker, Generative techniques for product lines, in: ICSE '01: Proceedings of the 23rd International Conference on Software Engineering, IEEE Computer Society, 2001, pp. 760–761. 1.2

[20] P. Cousot, R. Cousot, Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints., in: POPL, 1977, pp. 238–252. 1.4, 4.4

[21] R. L. Burden, J. D. Faires, Numerical analysis: 4th ed, PWS Publishing Co., 1989. 2.1

[22] K. O. Geddes, S. R. Czapor, G. Labahn, Algorithms for Computer Algebra, Kluwer Academic Publishers Group, Norwell, MA, USA, and Dordrecht, The Netherlands, 1992. 2.1

[23] M. B. Monagan, K. O. Geddes, K. M. Heal, G. Labahn, S. M. Vorkoetter, J. McCarron, P. DeMarco, Maple 7 Programming Guide, Waterloo Maple Inc., 2001. 2.2

[24] R. D. Jenks, R. S. Sutor, AXIOM: The Scientific Computation System, Springer Verlag, 1992. 2.2, 6

[25] S. M. Watt, Aldor, in: J. Grabmeier, E. Kaltofen, V. Weispfennig (Eds.), Computer Algebra Handbook: Foundations, Applications, Systems, Springer Verlag, 2003. 2.2, 6

[26] W. Taha, Multi-stage programming: its theory and applications, Ph.D. thesis, Oregon Graduate Institute of Science and Technology (1999). 4, 4.1, 4.4

[27] O. Kiselyov, K. N. Swadi, W. Taha, A methodology for generating verified combinatorial circuits, in: ACM International conference on Embedded Software (EMSOFT), 2004. 4, 4.4, 6, 7.2

[28] G. Hutton, E. Meijer, Monadic parser combinators, JFP 8 (4) (1996) 437–444. 4.2

[29] J. E. Stoy, Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory, MIT Press, 1981. 4.3

[30] E. Sumii, N. Kobayashi, A hybrid approach to online and offline partial evaluation, Higher-Order and Symbolic Computation 14 (2-3) (2001) 101–142.
URL
`http://citeseer.ist.psu.edu/article/sumii00hybrid.html` 4.3

[31] H. Abelson, G. J. Sussman, J. Sussman, Structure and Interpretation of Computer Programs, MIT Press, Cambridge, MA, 1996. 4.4

[32] J. C. Reynolds, Definitional interpreters for higher-order programming languages, in: ACM '72: Proceedings of the ACM annual conference, ACM Press, 1972, pp. 717–740. 6

[33] M. Frigo, S. G. Johnson, FFTW: An adaptive software architecture for the FFT, in: Proc. 1998 IEEE Intl. Conf. Acoustics Speech and Signal Processing, Vol. 3, IEEE, 1998, pp. 1381–1384. 6

[34] J. Davenport, P. Gianni, R. Jenks, V. Miller, S. Morrison, M. Rothstein, C. Sundaresan, R. Sutor, B. Trager, Scratchpad, Mathematical Sciences Department, IBM Thomas Watson Research Center (1984). 6

[35] N. G. de Bruijn, AUTOMATH, a language for mathematics, in: J. Siekmann, G. Wrightson (Eds.), Automation of Reasoning 2: Classical Papers on Computational Logic 1967-1970, Springer, Berlin, Heidelberg, 1983, pp. 159–200. 6

[36] D. Gruntz, M. Monagan, Introduction to Gauss, SIGSAM BULLETIN: Communications on Computer Algebra 28 (2) (1994) 3–19. 6

[37] A.-F. Le Meur, J. Lawall, C. Consel, Specialization scenarios: A pragmatic approach to declaring program specialization, Higher-Order and Symbolic Computation 17 (1) (2004) 47–92. 6

[38] R. Marlet, S. Thibault, C. Consel, Efficient implementations of software architectures via partial evaluation, Journal of Automated Software Engineering 6 (4) (1999) 411–440. 6

[39] A. Bondorf, D. Dussart, Improving CPS-based partial evaluation: Writing cogen by hand., in: PEPM, 1994, pp. 1–9. 6

[40] P. Thiemann, Cogen in six lines., in: ICFP, 1996, pp. 180–189. 6

A Maple code for `FractionFreeElimination`

```
FractionFreeElimination := proc(C,M,A,m,n)
  local d,i,j,k,s,t,size;
  size := `if`(hasCategory(C,EuclideanDomain),
               C[EuclideanNorm], length);
  s := 1; d := C[1];
  for k to m-1 do
      for i from k to m while A[i,k] = C[0] do end do;
      if i > m then return C[0] fi;
      for j from i+1 to m do
          if A[j,k] <> C[0] and
              size(A[j,k]) < size(A[i,k]) then
              i := j
          end if
      end do;
      #  Pivot is A[i,k], interchange if necessary
      if i <> k then
        s := - s;
        for j from k to n do
          t := A[k,j]; A[k,j] := A[i,j]; A[i,j] := t;
        end do;
      end if;
      #  Fraction-free elimination
      for i from k+1 to m do
          for j from k+1 to n do
              t := C[`-`]( C[`*`](A[i,j],A[k,k]),
                           C[`*`](A[k,j],A[i,k]));
              A[i,j] := C[Div](t, d)
          end do;
      end do;
     d := A[k,k]
  end do;
  s
end proc:
```

B Maple code for `mod/rtable/Gausselim`

```
`mod/rtable/Gausselim` := proc(A::Matrix,rank::{integer,name},
                               det::{integer,name})
  local    B, d, nrows, ncols, right_margin, i,j,
      col, row, s, t, p, Q;

  Q := type( A, 'Matrix( rational )' );
  if not Q and not type( A, 'Matrix( ratpoly( algnum ) )' ) then
      error "first argument must be a matrix over a finite field"
  end if;
  ( nrows, ncols ) := LinearAlgebra:-Dimensions( A );

  p := args[-1];
  right_margin := ncols;
  if type( args[-2], 'integer' ) then
      right_margin := args[-2]
  end if;

  if nargs > 3 and type( det, 'name' )
               and nrows <> right_margin then
    error "cannot compute determinant of a non-square matrix"
  end if;
```

```
if Q then
    B := map( s -> s mod p,  A );
else
    B := map( s -> 'Normal'( s ) mod p, A );
end if;

row := 1; # initialise the row
d := 1; # initialise the determinant

for col to right_margin while row <= nrows do
    # Find the first nonzero entry in this column
    for i from row to nrows while B[ i, col ] = 0 do
        # empty
    end do;
    if i > nrows then
        # found a zero column so determinant is zero
        d := 0;
        next
    end if;
    if i <> row then
        # interchange row i with row row
        B[ i, 1..-1 ], B[ row, 1..-1 ] :=
            B[ row, 1..-1 ], B[ i, 1 .. -1 ];
        d := -d mod p; # keep track of determinant
    end if;

    if nargs = 4 then
        d := 'Normal'( d * B[ row, col ] ) mod p
    end if;

    # Make all the nonzero entries below this one
    # are equal to 1
    if Q then
        s := 1/B[ row, col ] mod p;
        for i from row + 1 to nrows do
            # do i-th row
            if B[ i, col ] = 0 then
                next
            end if;
            t := s * B[ i, col ] mod p;
            for j from col + 1 to ncols do
                B[ i, j ] := B[ i, j ] -
                             t * B[ row, j ] mod p
            end do;
            B[ i, col ] := 0
        end do
    else
```

```
            s := 'Normal'( 1 / B[ row, col ] ) mod p;
            for i from row + 1 to nrows do
                if B[ i, col ] = 0 then
                    next
                end if;
                t := 'Normal'( s * B[ i, col ] ) mod p;
                for j from col + 1 to ncols do
                  B[i,j] := 'Normal'( B[ i, j ] -
                                      t * B[ row, j ] ) mod p
                end do;
                B[ i, col ] := 0
            end do
        end if;

        row := 1 + row # go to next row
    end do; # go to next column

    if nargs > 2 and type( rank, 'name' ) then
        rank := row-1
    end if;
    if nargs > 3 and type( det, 'name' ) then
        det := d
    end if;
    B
end proc:
```

C Final MetaOCaml code

```
type ('a,'c,'d) container2d = {
  get: ('a,'c) code -> ('a,int) code ->
       ('a,int) code -> ('a,'d) code ;
  set: ('a,'c) code -> ('a,int) code ->
       ('a,int) code -> ('a,'d) code -> ('a,unit) code;
  dim1: ('a,'c) code -> ('a,int) code;
  dim2: ('a,'c) code -> ('a,int) code;
  mapper: ('a, 'd->'d) code -> ('a,'c) code ->
          ('a,'c) code;
  copy: ('a, 'c) code -> ('a,'c) code
} ;;

type ('a,'c,'d) state = {b:('a,'c) code;
  r:('a,int ref) code; c:('a,int ref) code;
  m:('a,int) code; n:('a,int) code;
  det:('a,'d ref) code; detsign:('a,'d ref) code};;
```

```
type ('a,'b) domain = {
  zero:('a,'b) code; one:('a,'b) code;
  minusone:('a,'b) code;
  plus:('a,'b) code -> ('a,'b) code -> ('a,'b) code;
  times:('a,'b) code -> ('a,'b) code -> ('a,'b) code;
  minus:('a,'b) code -> ('a,'b) code -> ('a,'b) code;
  div:('a,'b) code -> ('a,'b) code ->('a,'b) code;
  smaller_than:(('a,'b) code -> ('a,'b) code ->
                ('a,bool) code) option;
  normalizerf:(('a,'b -> 'b) code ) option;
  normalizerg:(('a,'b) code -> ('a,'b) code ) option;
} ;;

type ('b,'c) outputs =
    Matrix of 'b
  | MatrixRank of 'b * int
  | MatrixDet of 'b * 'c
  | MatrixRankDet of 'b * int * 'c ;;

type outchoice = JustMatrix | Rank | Det | RankDet;;
type dettrack = TrackNothing | TrackDet ;;

type ge_choices =
  {fracfree:bool; track:dettrack; outputs:outchoice} ;;

let orcond_gen main_cond = function
  | Some alt_cond -> .< .~main_cond || .~alt_cond >.
  | None -> main_cond ;;

let seq a b = .< begin .~a ; .~b end >. ;;

let sapply2 g c1 c2 = match g with
  | Some f -> Some (f c1 c2)
  | None   -> None ;;

let dapply1 g c1 = match g with
  | Some f -> f c1
  | None   -> c1 ;;

let mdapply1 mapper g c1 = match g with
  | Some f -> .< .~(mapper f c1) >.
  | None   -> c1 ;;

let choose_output dom s = function
  | JustMatrix -> .< Matrix (.~s.b) >.
  | Rank       -> .< MatrixRank((.~s.b), ! .~(s.r)) >.
  | Det        -> .< MatrixDet((.~s.b),
      .~(dom.times .<! .~(s.det)>. .<! .~(s.detsign)>. ))>.
  | RankDet    -> .<MatrixRankDet((.~s.b),
      ! .~(s.r), .~(dom.times .<! .~(s.det)>.
      .< ! .~(s.detsign) >. )) >.

let ge_state_gen dom contr findpivot swap zerobelow choice=
  let main_loop s =
  .< while !(.~s.c) < .~s.m && !(.~s.r) < .~s.n do
    begin
    match .~(findpivot s) with
    Some i -> begin
        if i <> !(.~s.r) then
            for j = !(.~s.c) to .~s.m-1 do
                .~(swap s .<i>. .<j>.);
            done;
        .~(zerobelow s) ;
        (.~s.r) := !(.~s.r) + 1;
        end;
    | None -> .~(match choice.track with
        | TrackNothing -> .< () >. ;
        | TrackDet     -> .< .~s.detsign := .~dom.zero >.;
        ) ;
    end;
    .~s.c := !(.~s.c) + 1;
  done >. in

  .< fun a ->
  let b= .~(mdapply1 contr.mapper dom.normalizerf
                    (contr.copy .<a>. )) and
      r=ref 0 and c=ref 0 and
      m= .~(contr.dim1 .<a>. ) and
      n= .~(contr.dim2 .<a>. ) and
      det=ref .~(dom.one) and detsign=ref .~(dom.one) in
  .~(let st = {b= .<b>.; r= .<r>.; c= .<c>.; m= .<m>.;
              n= .<n>.; det= .<det>.;
              detsign= .<detsign>.} in
    seq
      (main_loop st )
      (choose_output dom st choice.outputs ) )
  >. ;;

let swapr_gen dom contr track s i j =
  let swap =
  .< let b = .~(s.b) and r = !(.~s.r) and i= .~i
     and j= .~j in
      let t = .~(contr.get .<b>. .<i>. .<j>. ) in  begin
        .~(seq
            (contr.set .<b>. .<i>. .<j>.
               (contr.get .<b>. .<r>. .<j>. ))
            (contr.set .<b>. .<r>. .<j>. .<t>. ) );
  end >. in
  let tds = .< .~s.detsign :=
        .~(dom.times .<! .~s.detsign>. dom.minusone) >. in
  match track with
  | TrackNothing -> swap
  | TrackDet     -> seq swap tds ;;

let findpivot_gen dom contr orcond_gen = fun s ->
  .< let i = ref (-1) in begin
    for j = ! .~s.r to .~s.n-1 do
      if not ( .~(contr.get .< .~s.b>. .<j>.
                 .<!( .~s.c)>. ) = .~(dom.zero)) then
        if (.~(orcond_gen .< !i = (-1) >.
          (sapply2 dom.smaller_than
            .< .~(contr.get .< .~s.b>. .<j>. .<! .~s.c>. )>.
            .< .~(contr.get .< .~s.b>. .<!i>. .<! .~s.c>. )>.
            ))) then
          i := j;
      done;
      if !i == -1 then None else Some !i;
  end >. ;;

let zerobelow_gen dom contr choice s =
  let inner_loop =
  if not choice.fracfree then fun i s ->
    .< let t = .~(dom.div
      (contr.get .<.~s.b>. .< .~i>. .<!(.~s.c)>. )
      (contr.get .<.~s.b>. .<!(.~s.r)>. .<!(.~s.c)>. ) ) in
    for j = !(.~s.c)+1 to .~s.m-1 do
      .~(contr.set .<.~s.b>. .<.~i>. .<j>.
          (dapply1 dom.normalizerg
          (dom.minus (contr.get .<.~s.b>. .<.~i>. .<j>.)
          (dom.times .<t>.
              (contr.get .<.~s.b>. .<!(.~s.r)>. .<j>. )))))
    done; >.
  else fun i s ->
    .< for j = !(.~s.c)+1 to .~s.m-1 do
      let t = .~(dapply1 dom.normalizerg (dom.minus
        (dom.times
          (contr.get .<.~s.b>. .<.~i>. .<j>. )
          (contr.get .<.~s.b>. .<! .~s.r>. .<! .~s.c>. ))
        (dom.times
          (contr.get .<.~s.b>. .<!(.~s.r)>. .<j>. )
          (contr.get .<.~s.b>. .<.~i>. .<! .~s.r>. )))) in
      .~(contr.set .<.~s.b>. .<.~i>. .<j>.
            (dom.div .<t>. .<! .~s.det>. ))
    done >. in
  let outer_loop =
  .< for i = !(.~s.r)+1 to .~s.n-1 do
    if not ( .~(contr.get .<.~s.b>. .<i>. .<! .~s.c>. )
             = .~dom.zero) then
      begin
        .~(inner_loop .<i>. s );
        .~(contr.set .<.~s.b>. .<i>. .<! .~s.c>. dom.zero)
      end;
  done >. in
  match choice.track with
  | TrackNothing -> outer_loop
  | TrackDet     -> seq outer_loop
    (if choice.fracfree then
      .< .~s.det :=
        .~(contr.get .<.~s.b>. .<! .~s.r>. .<! .~s.c>. )>.
    else
      .< .~s.det := .~(dom.times .<! .~s.det>.
          (contr.get .<.~s.b>.
              .<! .~s.r>. .<! .~s.c>. ))>. );;

let specializer dom container ~fracfree ~outputs =
  let t = (if fracfree || outputs == Det
                       || outputs == RankDet then
      TrackDet else TrackNothing ) in
  let choice =
      {fracfree=fracfree; track=t; outputs=outputs} in
  let fp = findpivot_gen dom container orcond_gen in
  let swap = swapr_gen dom container choice.track in
  let zb = zerobelow_gen dom container choice in
  ge_state_gen dom container fp swap zb choice;;

let dom_float = {
   zero = .< 0. >. ;
   one = .< 1. >. ;
   minusone = .< -1. >. ;
   plus = (fun x y -> .<.~x +. .~y>.) ;
   times = (fun x y -> .<.~x *. .~y>.) ;
   minus = (fun x y -> .<.~x -. .~y>.) ;
   div = (fun x y -> .<.~x /. .~y>.) ;
   smaller_than =
       Some(fun x y -> .<abs_float .~x < abs_float .~y >.);
   normalizerf = None ;
   normalizerg = None };;
```

```
let array_container = {
  get = (fun x n m -> .< (.~x).(.~n).(.~m) >. );
  set = (fun x n m y -> .< (.~x).(.~n).(.~m) <- .~y >. );
  dim2 = (fun x -> .< Array.length .~x >.) ;
  dim1 = (fun x -> .< Array.length (.~x).(0) >. ) ;
  mapper = (fun f a -> .< Array.map
      (fun x -> Array.map .~f x) .~a >.);
  copy = (fun a -> .<Array.map (fun x -> Array.copy x)
                         (Array.copy .~a) >. )
};;

let spec_ge_float =
    specializer dom_float array_container
                ~fracfree:false ~outputs:RankDet ;;

let ge_float3 = .! spec_ge_float ;;
```