ARTICLE IN PRESS

Science of Computer Programming xx (xxxx) xxx–xxx

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

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

Jacques Carette [a,*], Oleg Kiselyov [b]

[a] McMaster University, 1280 Main Street West, Hamilton, Ontario, Canada L8S 4K1
[b] FNMOC, Monterey, CA 93943, United States

ARTICLE INFO

Article history:
Received 14 July 2007
Received in revised form 23 June 2008
Accepted 5 September 2008
Available online xxxx

Keywords:
MetaOCaml
Linear algebra
Genericity
Generative
Staging
Functor
Symbolic

ABSTRACT

We use multi-stage programming, monads and Ocaml's advanced module system to demonstrate how to eliminate all abstraction overhead from generic programs, while avoiding any inspection of the resulting code. We demonstrate this clearly with Gaussian Elimination as a representative family of symbolic and numeric algorithms. We parameterize our code to a great extent – over domain, input and permutation matrix representations, determinant and rank tracking, pivoting policies, result types, etc. – at no run-time cost. Because the resulting code is generated just right and not changed afterward, MetaOCaml guarantees 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 "interleaving" of aspects. We also show how to encode some domain-specific knowledge so that "clearly wrong" compositions can be rejected at or before generation time, rather than during the compilation or running of 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. Widely used Gaussian Elimination (GE) – the running example of our paper – is typically presented in textbooks as a closely related family of algorithms for solving simultaneous linear equations, LU matrix decomposition, and computing the determinant and the rank of a matrix. All members of the family share the same pattern of applying elementary row operations to rows of the matrix in a particular order. The individual algorithms differ in their output, in application of pivoting, in algebraic domain and the use of full division. Modern architectures demand further divisions of the family for particular matrix layouts, e.g., sparse or tiled.

A survey [1] of Gaussian Elimination implementations in the industrial package Maple [2] found 6 clearly identifiable aspects and 35 different implementations of the algorithm, as well as 45 implementations of directly related algorithms, such as LU decomposition, Cholesky decomposition, and so on. We could manually write each of these implementations, optimizing for particular aspects and using cut-and-paste to "share" similar pieces of code. Or we can write a very generic procedure that accounts for all the aspects with appropriate abstractions [3,4]. The abstraction mechanisms however – be they procedure, method or a function call – have a significant cost, especially for high-performance numerical computing [1].

* Corresponding address: McMaster University, Department of Computing and Software, 1280 Main Street West, L8S 4K1 Hamilton, Ontario, Canada. Tel.: +1 905 525 9140.
E-mail addresses: carette@mcmaster.ca (J. Carette), oleg@pobox.com (O. Kiselyov).
URLs: http://www.cas.mcmaster.ca/~carette (J. Carette), http://okmij.org/ftp/ (O. Kiselyov).

0167-6423/$ – see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2008.09.008

Eliminating this abstraction overhead involves either complex analyses or domain-specific knowledge (or both!) [5–7], and so we cannot rely on a general purpose compiler to assuredly perform such optimizations.

A more appealing approach is generative programming [8–14]. 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 any 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 are preventing accidental variable capture (so-called hygiene [15]), and ensuring that the generated code is well-typed. Lisp-style macros, Scheme hygienic macros, the camlp4 preprocessor [16], C++ template meta-programming, and Template Haskell [17] solve some of the above problems. Of the widely available maintainable languages, only MetaOCaml [18,19] solves all of the above problems, including well-typing of both the generator and the generated code [20,21].

But more difficult problems remain. Is the generated code optimal? Do we still need post-processing to eliminate common subexpressions, fold constants, and remove redundant bindings? Is the generator readable? Does it bear resemblance to the original algorithm? Is the generator extensible? Are the aspects truly modular? Can we add another aspect or another instance of the existing aspect without affecting the existing ones? Finally, can we express domain-specific knowledge (for instance 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 purely *generative*: generated code can only be treated as a black box — in other words, it cannot be inspected nor can it be post-processed (i.e., no intensional analysis). This approach gives a stronger equational theory [22], and avoids the danger of creating unsoundness [21]. Furthermore, intensional code analysis essentially requires one to insert both an optimizing compiler and an automated theorem proving system into the code generating system [23,5,24,6]. 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 [21] for many simple examples). For more complex examples, new techniques are necessary, for example abstract interpretation [25]. But more problems remain [7]: generating binding forms ("names") when generating loop bodies or conditional branches, and 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. To be able to build modular code generators, three important problems remain: compositionality, expressing dependencies, and integration of domain-specific knowledge.

In this paper, we report on our continued progress [26] [1] in using code generation for scientific (both numeric and symbolic) software. We will use the algorithm family of Gaussian Elimination, applied to perform LU decomposition and linear system solving, as our running examples to demonstrate our techniques. Specifically, our contributions are:

- Extending a let-insertion, memorizing monad of [25,27] for generating control structures such as loops and conditionals. The extension is non-trivial because of control dependencies and because let-insertion, as we argue, is a control effect on its own: for example `let x = exp in ...` has a different *effect* within a conditional branch.
- Implementation of the `perform`-notation (patterned after the `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 *insure composability of aspects* even for aspects that use state and have to be accounted for in many places in the generated code.
- Encode domain-specific knowledge in the generators so as to catch domain-specific instantiation errors at generation time.
- Provide a thorough classification of the family of Gaussian Elimination algorithms.

We also used the same technology to implement a Runge–Kutta solver for ordinary differential equations, as well as a reimplementation of the FFT algorithm from [25]. The technology presented here was amply sufficient for these implementations. Since our current implementations of these algorithms are rather straightforward, compared with our versions of LU decomposition, we will not mention them further (the code is available at [28]).

The rest of this paper is structured as follows: the next section gives an overview of the design space of Gaussian Elimination algorithms (and their application to LU and linear system solving). Section 3 introduces code generation in MetaOCaml, the problem of name generation, and the continuation-passing style (CPS) as a general solution. We also present the key monad and the issues of generating control statements. For the sake of reference, in Section 4 we present a particular Gaussian Elimination algorithm, a hand-written implementation of the standard textbook pseudo-code. Section 5 describes the use of the parametrized modules of OCaml to encode all of the aspects of our algorithm family as separate modules. We discuss related work in Section 6, and outline future work. In our conclusion (Section 7), we comment on programming with aspects and sum up our guiding methodology. Appendices give samples of the generated code, available in full at [28].

[1] We describe here a new version of our generator dealing with the complete LU decomposition algorithm, as well as linear solving. We worked out previously missing aspects of in-place updates, representing permutation matrices, dealing with augmented input matrix, and back-propagation. We have changed the representation of domain-specific knowledge about permissible compositions of aspects. Also included, is a careful description of all the aspects involved, as well as documenting our development methodology for highly parametric scientific software.

Please cite this article in press as: J. Carette, O. Kiselyov, Multi-stage programming with functors and monads: Eliminating abstraction overhead from generic code, Science of Computer Programming (2008), doi:10.1016/j.scico.2008.09.008

2. The design space

Before investigating implementation approaches, it is worthwhile to carefully study the design space involved. A preliminary study [1] revealed a number of aspects of the family of Gaussian Elimination algorithms. In the present work, we outline a number of additional aspects involved in the (related) family of LU decomposition algorithms. These will first be presented in a somewhat ad hoc manner, roughly corresponding to the order in which they were "discovered". We then reorganize them into groups of semantically related aspects to form the basis of our design.

Throughout, we assume that the reader is familiar with the basic LU decomposition algorithm, which factors an invertible matrix A into a unit lower triangular matrix L and (usually) an upper triangular matrix U, such that $A = LU$. Pivoting adds a unitary matrix P such that the factorization is now $A = PLU$. The case of numeric matrices is well covered in [29]. When A is singular, one can still get a PLU decomposition with L remaining unit lower-triangular. However, U is no longer upper triangular but rather "staggered" in the upper triangle.

2.1. Aspects

We reuse the English word "aspect" for the various facets of the family of Gaussian Elimination algorithms. While our use shares the spirit of aspect-oriented programming (AOP) [30], our implementation methodology is radically different.[2] We firmly believe that our typed generative methodology is better suited to functional programming, compared with attempts to graft the program-trace-based methodology of object-oriented versions of AOP.

At this point in time, it is better to think of aspects as purely design-time entities. Here, we are firmly influenced by Parnas' original view of modules and information hiding [33], as well as his view of product families [34], and by Dijkstra's ideas on separation of concerns [35]. To apply these principles, 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.

The various aspects listed below all come from variations found in actual implementations (in various languages and settings).

(1) **Domain**: The (algebraic) domain of matrix elements. Some implementations were very specific ($\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{Z}_i$, $\mathbb{Z}_p[\alpha_1, \ldots, \alpha_n]$, $\mathbb{Z}[x]$, $\mathbb{Q}(x)$, $\mathbb{Q}[\alpha]$, and floating point numbers ($\mathbb{F}$) for example), while others were generic for elements of a field, multivariate polynomials over a field, or elements of a division ring with possibly undecidable zero-equivalence. In the roughly 85 pieces of code we surveyed, 20 different domains were encountered.
(2) **Representation of the matrix**: Whether the matrix was represented as an array of arrays, a one-dimensional array with C or Fortran indexing styles, a hash table, etc. Efficient row exchanges, if available for a particular representation, were sometimes used.
(3) **Fraction-free**: Whether the algorithm is allowed to use unrestricted division, or only exact (remainder-free) division.
(4) **Length measure (for pivoting)**: For stability reasons (whether numerical or coefficient growth), if a domain possesses an appropriate length measure, it was sometimes used to choose an "optimal" pivot. Not all domains have such a measure.
(5) **Full division**: Whether the input domain supports full division (i.e. is a *field* or pretends to be ($\mathbb{F}$)) or only exact division (i.e. a *division ring*).
(6) **Domain normalization**: Whether the arithmetic operations of the base domain keep the results in normal form, or whether an extra normalization step is required. For example, some representations of polynomials require an extra step for zero-testing.
(7) **Output choices**: Just the reduced matrix (the 'U' factor) or both L and U factors. The output choices also include the rank, the determinant, and the sequence of pivots. For example, Maple's `LinearAlgebra:- LUDecomposition` routine has $2^6 + 2^5 + 2^2 = 100$ possible outputs, depending on whether one chooses a *PLU*, *PLUR* or *Cholesky* decomposition. We chose to only consider *PLU* for now.
(8) **Rank**: Whether to explicitly track the rank of the matrix as the algorithm proceeds.
(9) **Determinant**: Whether to explicitly track the determinant of the matrix as the algorithm proceeds.
(10) **Code representation**: The form and the language for the generated code (OCaml, C, Fortran, etc.). A degenerate case, useful for testing, is for the generator to run the algorithm directly (albeit with great abstraction overhead).
(11) **Zero-equivalence**: Whether the arithmetic operations require a specialized zero-equivalence routine. For certain classes of *expressions*, it turns out to be convenient to use a zero-equivalence test that is separate from the domain normalization. This is usually the case when zero-equivalence is formally undecidable but semi-algorithms or probabilistic algorithms do exist. See [36] for an example.
(12) **Pivoting**: Whether to use no, column-wise, or full pivoting.
(13) **Augmented Matrices**: Whether all, or only some, columns of the matrix participate in elimination.

[2] However it seems that we are closer to the original ideas of AOP [31,32], which were also concerned with scientific software.

Please cite this article in press as: J. Carette, O. Kiselyov, Multi-stage programming with functors and monads: Eliminating abstraction overhead from generic code, Science of Computer Programming (2008), doi:10.1016/j.scico.2008.09.008

(14) **Pivot representation**: Whether the pivot is represented as a list of row and column exchanges, as a unitary matrix, or as a permutation vector.
(15) **Lower matrix**: Whether the matrix L should be tracked as the algorithm proceeds, reconstructed at the end of the algorithm, or not tracked at all.
(16) **Input choices**: Grouping all the potential choices of inputs — currently only augmented matrices require an extra input.
(17) **Packed**: Whether the output matrices L and U are packed into a single matrix for output.

The aspects in the following group have also been observed in practice, but we have not yet implemented them:

(18) **Logging**: A classical cross-cutting concern.
(19) **Sparsity**: If a matrix is known to be sparse, at least the traversal should be sparse. Maximal preservation of the sparsity is desirable.
(20) **Other structure**: If a matrix is known in advance to be real symmetric tri-diagonal, LU decomposition can be done in $O(n^2)$ rather than $O(n^3)$ time, at an additional $O(n)$ storage cost.
(21) **Warnings**: In a domain with only heuristic zero testing, it is customary to issue a warning (or otherwise log) when a potentially zero pivot is chosen.
(22) **In-place**: Offering an option of in-place decomposition, re-using the input matrix as the storage for the output.
(23) **Error-on-singular**: Raise an exception when the input matrix is (near) singular.

Most of these aspects are inter-dependent. For example, if the determinant is part of the output, the determinant should be tracked during the decomposition. Determinants should also be tracked if the fraction-free aspect is chosen. The availability of the length measure in the domain influences pivoting, *if* pivoting is to be performed. One could therefore select aspects that turn out to be incompatible, and we have to prevent this. More precisely, our goal is to detect the selection of incompatible aspects long before the generated code is run.

2.2. Organizing aspects

Further investigation revealed the following grouping of aspects[3]:

(1) **Abstract domain**. This group includes 'mathematical' aspects: domain of matrix elements, length measure, full division, zero testing, symmetry and other matrix structure.
(2) **Concrete representation**: choosing data structures for domains, containers, permutation matrices. This group also includes packing, in-place decomposition, normalization, and the representation of sparse matrices.
(3) **Interface** of each generated function, including the input and output choices, logging and error reporting.
(4) **Algorithmic Strategy**, such as fraction-free updates, pivoting strategies, augmented matrices.
(5) **Tracking**, of determinant, rank, or pivot, etc.
(6) **Interpretation**: whether the result is the program or a generator that will produce the program.

The groupings are not entirely orthogonal (for example, in-place decomposition is possible only for specific domains), yet are useful as guidance in creating the modular generator discussed in Section 5.

3. Techniques for typed code generation

This section outlines various necessary techniques for typed code generation in MetaOCaml. We start with an introduction to code generation with MetaOCaml, where our examples are chosen to illustrate issues of direct concern to generic programs. Next we introduce monads and our monadic notation, as a means to make writing programs in continuation passing style (CPS) more palatable. Generation of control statements can lead to various subtle issues, and solutions are covered in Section 3.3. Finally, during generation we need to keep track of various aspects, and we use an extensible state for this purpose, described in Section 3.4.

3.1. MetaOCaml and basic abstraction

We wish to build large code generators out of primitive generators, using combinators. MetaOCaml, as an instance of a multi-stage programming system [21], provides exactly the necessary features: to construct a code expression, to combine them, and to execute them. The following shows a simple code generator `one`, and a simple code combinator[4]:

```
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))>.
```

[3] This grouping showed that some of our implementation did not separate distinct concerns well. In particular, our implementation of "containers" mixed representation (i.e. data-structure) issues with abstraction domain issues. We hope to rectify this in the future.
[4] ⟹ under an expression shows the result of its evaluation

Please cite this article in press as: J. Carette, O. Kiselyov, Multi-stage programming with functors and monads: Eliminating abstraction overhead from generic code, Science of Computer Programming (2008), doi:10.1016/j.scico.2008.09.008

We use MetaOCaml brackets `.<...>.` to generate code expressions, i.e. to construct future-stage computations. MetaOCaml provides only one mechanism for combining code expressions, by splicing one piece of code into another. The power of that operation, called escape and denoted `.~`, comes from the fact that the expression to be spliced in (inlined) can be computed: escape lets us perform an arbitrary *code-generating* computation *while* we are building a 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 location of the escape. MetaOCaml conveniently can print out code expressions, so we can examine the final generated code. It has no traces of `gen` or `plus`: those are purely generation stage computations.

The final MetaOCaml feature, `.!` (pronounced "run") executes a 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 which *represents* (or encodes) a function.

By parameterizing our code, we can make the benefits of code generation evident:

```
let simplest_param_code plus one =
  let gen x y = plus x (plus y one) in
  .<fun x y -> .~(gen .<x>. .<y>.)>.
```

and use it to generate code that operates on integers, floating point numbers or booleans — in general, any domain that implements `plus` and `one`:

```
let plus x y = .<.~x +. .~y>. and one = .<1.0>. in
  simplest_param_code plus one
let plus x y = .<.~x || .~y>. and one = .<true>. in
  simplest_param_code plus one
```

Running the former expression yields a function on `floats`, whereas the latter expression is a code expression for a boolean function. This simple technique clearly shows how we can abstract over domain operations, and yet still generate efficient domain-specific code, thus achieving a proper separation of concerns.

Let us consider a more complex expression:

```
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>.)>.
```

with two occurrences of `plus y one`, which may be a rather complex computation which we would rather not do twice. We might 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 an imperative context where the compiler might not be able to determine whether lifting them is sound. So, being conservative, the optimizer will leave the duplicates as they are. We may attempt to eliminate subexpressions as follows:

```
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)))>.
```

The result of `param_code1' plus one` still exhibits duplicate sub-expressions. This is because our `let`-insertion optimization only saved the computation at the generating stage. We need a combinator that inserts the `let` expression in the generat*ed* code, in other words a combinator `letgen` to be used as

```
let ce = letgen (plus y one) in plus ce (plus x ce)
```

yielding 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 yet have the `body`. 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 solution to this problem is to use continuation-passing style (CPS). Its benefits were first pointed out by [37] in the context of partial evaluation, and extensively used by [27,25] for code generation. Like [38], we use this in the context of writing a cogen by hand. 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 =
```


Please cite this article in press as: J. Carette, O. Kiselyov, Multi-stage programming with functors and monads: Eliminating abstraction overhead from generic code, Science of Computer Programming (2008), doi:10.1016/j.scico.2008.09.008

```
type ('p,'v) monad = 's -> ('s -> 'v -> 'w) -> 'w
    constraint 'p = <state : 's; answer : 'w; ..>

let ret (a :'v) : ('p,'v) monad = fun s k -> k s a
let bind (m : ('p,'v) monad) (f : 'v -> ('p,'u) monad) : ('p,'u) monad
  = fun s k -> m s (fun s' b -> f b s' k)

let fetch s k = k s s  and  store v _ k = k v ()

let k0 _ v = v
let runM m = fun s0 -> m s0 k0

let retN (a : ('c,'v) code) :
 (<classif: 'c; answer: ('c,'w) code; ..>,('c,'v) code) monad
   = 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 then .~(th s k0) else .~(el s k0) >.
```

Fig. 1. Our monad, helper functions and uses.

```
  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))>.
```

3.2. Monadic notation, making CPS code clear

Comparing the let-insertion in the generator

```
let ce = (plus y one) in  plus ce (plus x ce)
```

with the corresponding code generating let-insertion for a future 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, `let` became "inverted". Secondly, 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 considerably, 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 programmers of scientific software (our users) cannot or will not program in CPS. Clearly a better notation is needed.

The `do`-notation of Haskell [39] shows that it is possible to write CPS code in a conventional-looking style. The `do`-notation is the notation for monadic code [40]. Not only can monadic code represent CPS [41], it also helps with composability by giving complete control over how different effects are layered (state, exception, non-determinism, etc.) on top of the basic monad [42].

A monad [40] is an abstract data type representing computations that yield a value and may have an *effect*. The data type 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. Fig. 1 defines the monad used throughout the present paper and shows its implementation. Our monad encapsulates 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 Fig. 1, the monad (yielding values of type `v`) is implemented as a function of two arguments: the state (of type `s`) and the continuation. The continuation receives the current state and a value, and yields an answer of type `w`. The monad is polymorphic over the three type parameters, which would require `monad` to be a type constructor with three arguments. When we use this monad for code generation, we will need yet another type variable for the environment classifiers [43] (such as the type variable `'c` in the type of `retN` in Fig. 1). With type constructors taking more and more arguments, it becomes increasingly difficult to read and write types — which we will be doing extensively when writing module signatures in Section 5. The fact that OCaml renames all type variables when printing out types confuses matters further. An elegant

Please cite this article in press as: J. Carette, O. Kiselyov, Multi-stage programming with functors and monads: Eliminating abstraction overhead from generic code, Science of Computer Programming (2008), doi:10.1016/j.scico.2008.09.008

solution to these sorts of problems has been suggested by Jacques Garrigue on the Caml mailing list. We use a single type parameter `'p` to represent all parameters of our monad, more precisely all parameters but the type of the monadic value `'v`. The type variable `'p` is constrained to be the type of an object with methods (fields) `state` and `answer`. The object may include more fields, represented by `...` Values of that type are not part of our computations and need not exist. We merely use the object type as an convenient way to specify extensible *type-level* records in OCaml.

Our monad could be implemented in other ways. Except for the code in Fig. 1, the rest of our code treats the monad as a truly abstract data type. The implementation of the basic monadic operations `ret` and `bind` is conventional and clearly satisfies the monadic laws. Other monadic operations construct computations that do have specific effects. Operations `fetch` and `store v` construct computations that read and write the state.

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 behavior is symptomatic of control operators (in particular, `abort`). The name can be taken to mean *return a Named computation*; the name allows for proper sharing, but otherwise `retN` is used in writing generators in the same way as `ret`. The type of `retN` is a specialization of the type of `ret`: the computation `retN` deals specifically with code values. The types of code values must include the environment classifier (such as `'c`), which denotes the scope of free variables that may occur in the code value. The argument type of `retN` and the answer type of `retN` computation must have the same classifier — which, informally, means that all free variables in the argument of `retN` are preserved in the answer of `retN`'s computation. When writing, for clarity, the type annotations of `retN` we see the benefits of type-level records: we introduce a new component of the record to specify the environment classifier `'c`, and we update the `answer` component of the record to specialize the answer type to be a code type. The `state` component of the record is not affected, and so remains hidden in the ellipsis `...`; the number of parameters to the type constructor `monad` remains the same.

Finally, `runM` runs our monad, that is, given the initial state, it performs the computation of the monad and returns its result, which in our case is a 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>.) ())>.
```

That may not seem like much of an improvement, but with the help of the camlp4 pre-processor, we can introduce the `perform`-notation [28], patterned after the `do`-notation of Haskell (see Appendix A).

```
let param_code4 plus one =
  let gen x y = perform ce <-- retN (plus y one);
                        ret (plus ce (plus x ce))
  in .<fun x y -> .~(runM (gen .<x>. .<y>.) ())>.
```

The function `param_code4`, written using the `perform`-notation, is equivalent to `param_code3` — in fact, the camlp4 preprocessor converts the former into the latter. And yet, `param_code4` looks far more conventional, as if it were indeed in direct style.

3.3. Generating control statements

We can write operations that generate code other than let-statements, e.g., conditionals: see `ifL` in Fig. 1. The function `ifL`, albeit straightforward, is not as general as we wish: its arguments are pieces of code rather than monadic values. We can "lift it":

```
let ifM' test th el = perform
  testc <-- test; thc <-- th; elc <-- el;
  ifL testc thc elc
```

However we also need another `ifM` function, with the same interface (see Fig. 1). The difference between them is apparent from the following example:

```
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>.) ())>.
```


Please cite this article in press as: J. Carette, O. Kiselyov, Multi-stage programming with functors and monads: Eliminating abstraction overhead from generic code, Science of Computer Programming (2008), doi:10.1016/j.scico.2008.09.008

```
⟹.<fun a_1 i_2 ->
     if (i_2 >= 0) then let t_3 = (Some a_1.(i_2)) in t_3 else None>.
```

If we use `ifM'` to generate guarded array access code, the let-insertion happens *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 after we have verified that the index is non-negative. This example demonstrates that code generation (such as the one in `retN`) is truly an effect, and that 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 sequencing, case-matching statements and `for`- and `while`-loops.

```
let seqM a b = fun s k ->
  k s .< begin .~(a s k0) ; .~(b s k0) end >.

let whileM cond body = fun s k ->
  k s .< while .~(cond) do .~(body s k0) done >.

let matchM x som non = fun s k -> k s .< match .~x with
          | Some i -> .~(som .<i>. s k0)
          | None   -> .~(non s k0) >.

let genrecloop gen rtarg = fun s k ->
  k s .<let rec loop j = .~(gen .<loop>. .<j>. s k0) in loop .~rtarg>.
```

One can think of this particular use of continuation as delimiting the "current scope" of a block of code. When constructing blocks with a new scope, we use a fresh continuation; this allows us to generate all named computations at the "top" of the current scope but no farther.

3.4. Maintaining an extensible state

Various aspects of our generator need to keep a state during code generation (for example the name of the variable for the sign of the determinant, Section 5.4). The simplest method of keeping such state is by using mutable variables, private to each module (aspect). That would, however, make our aspects stateful. Although we are generating imperative code, we would like to keep our generators stateless and purely functional, for ease of comprehension and reasoning. Our main program may include several generators referring to one particular aspect — which may be present in one shared instance or in several. That is of no concern if the module is stateless, but with stateful modules, the issues of aliasing or separate instantiation are a source of very subtle problems.

We therefore chose a different way of maintaining generator state, using a monad. We already saw that for let-insertions, we could use a continuation monad; we now demonstrate the state component of our monad (Fig. 1). The monadic actions `fetch` and `store` are used to access that monadic state, which is threaded throughout the entire code-generation computation.

This monadic state has to accommodate several distinct pieces of state, for various aspects. We should be able to add a new aspect – which may need to keep its own state as part of the overall monadic state – without modifying or even recompiling the rest of the code. Thus our monadic state should be extensible. We could use an extensible record: an OCaml object. Each aspect would have its own field; record subtyping would insure modularity. Alas, this approach makes it difficult to create an initial state, to pass to the monad's `runM` method. We would be required to know the names of all fields, and should know the proper initial value for these fields, which breaks modularity.

The MLton team suggests a better approach: property lists [44]. A property list also represents an extensible object but via its dual, namely a list of fields. The initial object is just the empty list. Unfortunately, we cannot apply MLton's approach literally to our case. The MLton approach uses generativity of exceptions or reference cells to generate property names, i.e. the fields of an extensible record. This technique would make our aspects stateful modules, which we described earlier as undesirable. MLton's approach to generating field names also makes it difficult to store code values in those fields, as a code value has a type which contains a generic type variable, the environment classifier. Fortunately, OCaml lets us build open unions with polymorphic variants. Each variant is identified by a manifest tag, e.g. `` `Tdet ``, and may include a value. The tags are not generative and provide a form of manifest naming, similar to symbols in Lisp and Scheme.

Below is the implementation of our open records with manifest naming: functions `orec_store` to add a new field to an open record and `orec_find` to obtain the value associated with a particular field name. Each "field" is characterized by a triple: an injection function, a projection function and the string name. The latter is (only) used for printing error messages. For example, for the determinant tracking aspect (Section 5.4), this triple has the form

```
let ip =
  (fun x -> `TDet x), (function `TDet x -> Some x | _ -> None), "Det"
```

We combine these functions with monadic actions to access monadic state, and so obtain `mo_extend` and `mo_lookup` to store and retrieve one component of the monadic state.

Please cite this article in press as: J. Carette, O. Kiselyov, Multi-stage programming with functors and monads: Eliminating abstraction overhead from generic code, Science of Computer Programming (2008), doi:10.1016/j.scico.2008.09.008

```
type ('a,'b) open_rec = ('a -> 'b) * ('b -> 'a option)  * string

let rec lookup ((_,prj,_) as ip:(('a,'b) open_rec)) : 'b list -> 'a =
   function [] -> raise Not_found
   | (h::t) -> (match prj h with Some x -> x | _ -> lookup ip t)

let orec_store ((inj,_,name) as ip:(('a,'b) open_rec)) (v:'a) (s:'b list)
   : 'b list =
  let () =
    try let _ = lookup ip s in
        failwith ("Field "^name^" of an open record is already present.")
    with Not_found -> () in
  (inj v)::s

let orec_find ((_,_,name) as ip:(('a,'b) open_rec)) (s:'b list) : 'a =
  try lookup ip s
  with Not_found -> failwith ("Failed to locate orec field: " ^ name)

let mo_extend (ip:('a,'b) open_rec) (v:'a) : ('c, 'd) monad =
  perform s <-- fetch; store (orec_store ip v s)

let mo_lookup (ip:('a,'b) open_rec) : ('c, 'd) monad =
  perform s <-- fetch; ret (orec_find ip s)
```

Currently, we check at generation-time that one should not add an already existing field to an open record, nor should one attempt to look up a field that does not exist. It is possible to make these checks static. Our approach has the advantage of generating *much* clearer error messages.

4. Gaussian elimination

For detailed reference, we present one particular Gaussian Elimination [5] algorithm, for an in-place LU decomposition of an integer matrix with full pivoting, returning the U-factor, determinant and rank. The $n \times m$-matrix is represented as a flat vector, with 0-indexed elements laid out in row-major format (C-style). The code in Fig. 2 is a hand-written implementation of the typical pseudo-code in Numerical Analysis textbooks (see for example [45]).

In OCaml, the matrix is represented by a value of the following type:

```
type 'a container2dfromvector = {arr:('a array); n:int; m:int}

let swap a i j =
  let t = a.(i) in begin a.(i) <- a.(j); a.(j) <- t; end

let swap_rows a (n,m) (r,c) i =
  let row_r = r*m in                (* Beginning of row r *)
  let row_i = i*m in                (* Beginning of row i *)
  for k = c to m-1 do
    swap a (row_r + k) (row_i + k)
  done

let swap_cols a (n,m) c j =
  let end_vector = n * m in
  let rec loop col_c col_j =
    if col_j < end_vector then
      begin
        swap a col_c col_j;
        loop (col_c + m) (col_j + m)
      end
  in loop c j
```

We refer to the a_{ij}—the element of the matrix as `a.(i*m+j)`, where `a` is the array component of the record `container2dfromvector`. The code above defines three auxiliary functions, typically used in textbook pseudo-code: `swap a i j` swaps two elements; `swap_cols a (n,m) c j` swaps column `c` with column `j` in the `(n,m)`-matrix `a`. The function `swap_rows a (n,m) (r,c) i` swaps row `r` with row `i` in the non-yet examined portion of the matrix, rectangular block `(r,c)-(n,m)`. We do not touch the elements to the left of the column `c` because they are all zeros. Since we know the layout of the matrix, we avoid 2D index computations.

[5] For the purposes of this paper, we will use Gaussian Elimination (GE) and LU decomposition (LU) as quasi-synonyms, even though we are well-aware that GE can be used for other purposes, and LU can be performed by other means than GE.

Please cite this article in press as: J. Carette, O. Kiselyov, Multi-stage programming with functors and monads: Eliminating abstraction overhead from generic code, Science of Computer Programming (2008), doi:10.1016/j.scico.2008.09.008

```
let find_pivot a (n,m) r c =
  let pivot = ref None in               (* ((i,j), pivot_val) option *)
  begin
  for i = r to n-1 do
    for j = c to m-1 do
      let cur = a.(i*m+j) in
      if not (cur == 0) then
        match !pivot with
        | Some (_,oldpivot) ->
            if abs oldpivot > abs cur then
               pivot := Some ((i,j), cur)
        | None -> pivot := Some ((i,j),cur)
  done; done;
  !pivot
  end
let ge = fun a_orig ->
  let r = ref 0 in                    (* current row index, 0-based *)
  let c = ref 0 in                    (* current column index, 0-based *)
  let a = Array.copy (a_orig.arr) in  (* to avoid clobbering A, save it *)
  let m = a_orig.m in                 (* the number of columns *)
  let n = a_orig.n in                 (* the number of rows *)
  let det_sign = ref 1 in             (* Accumulate sign and magnitude *)
  let det_magn = ref 1 in             (*   of the determinant *)
  while !c < m && !r < n do
    (* Look for a pivot *)
    let pivot = find_pivot a (n,m) !r !c in
    let piv_val = (match  pivot with
    | Some ((piv_r, piv_c),piv_val) ->
        if piv_c <> !c then
          begin
            swap_cols a (n,m) !c piv_c;
            det_sign := - !det_sign (* flip the sign of the det *)
          end;
        if piv_r <> !r then
          begin
            swap_rows a (n,m) (!r,!c) piv_r;
            det_sign := - !det_sign (* flip the sign of the det *)
          end;
        Some piv_val
    | None -> None) in
    (* now do the row-reduction over the (r,c)-(n,m) block *)
    (match piv_val with
    | Some a_rc -> begin
        for ii = !r+1 to n-1 do
          let cur = a.(ii*m + !c) in
          if not (cur == 0)  then
            begin
              for j = !c+1 to m-1 do
                (* fraction-free elimination *)
                a.(ii*m+j) <- (a.(ii*m+j) * a_rc - a.(!r*m+j) * cur) / ! det_magn
              done;
              a.(ii*m+ !c) <- 0
            end;
        done;
        det_magn := a_rc;
        r := !r + 1                 (* advance the rank only if pivot > 0*)
      end
    | None -> det_sign := 0);
    c := !c + 1
    done;
    (* Final result *)
    ({arr=a; n=n; m=m},                 (* The matrix now has the U factor*)
    (if      !det_sign = 0 then  0      (* Compute the signed det *)
     else if !det_sign = 1 then  !det_magn
     else                        (- !det_magn)),
    !r)                                 (* Rank *)
```

Fig. 2. Fraction-free in-place Gaussian Elimination over an integer matrix.

As is typical of textbook presentations, the main algorithm depends on a separately-defined function `pivot` to find a pivot. Here we use full-pivoting, i.e. searching the complete as yet unexamined portion of the matrix, the rectangular block `(r,c)-(n,m)`, for the element with the ~~maximum~~ absolute value. The function returns the value of the pivot thus found,

Please cite this article in press as: J. Carette, O. Kiselyov, Multi-stage programming with functors and monads: Eliminating abstraction overhead from generic code, Science of Computer Programming (2008), doi:10.1016/j.scico.2008.09.008

and its location; the return value is an option type with `None` indicating that all examined elements are zeroes. In the main algorithm, after we have found the pivot, we swap the current column with the pivot column, and swap the current row with the pivot row (if necessary). After the swap, the a_{rc} element of the matrix is the pivot. Swapping two rows or two columns changes the sign of the determinant. After the swaps, the algorithm performs so-called row-reduction over the `(r,c)-(n,m)` block of the matrix. We implement the *fraction-free* version of the algorithm[6] [46], where the division operation in line 50 assuredly divides two integers with no remainder, which requires the accumulation of the determinant.

5. Aspects and functors

Our monad gives us the tools to implement fine-scale code generation. We need tools for larger-scale modularization; conveniently, we can use whatever abstraction mechanisms we want to structure our code generators, as long as these abstractions do not infiltrate the generated code. For our purposes, the ML module system turns out to be most convenient.

In this section, we use the the sample code in Fig. 2 to identify (some of the) aspects discussed in Section 2.1. We abstract these aspects and describe their implementation as modules of the code generator. We then present in Section 5.7 the generic Gaussian Elimination algorithm — what is left after all the aspects are abstracted away. Finally, we show an instantiation of the generic generator, whose execution yields (exactly) the code for the algorithm of Fig. 2, this time automatically generated rather than manually written.

We will describe the following aspects, generally in order of increasing complexity: domain (of the group 'Abstract domain', see Section 2.2), code generation combinators (of the group 'Interpretation'), matrix representation (of the group 'Concrete representation'), determinant (the group 'Tracking'), and output (the group 'Interface').

5.1. Domains

Clearly the basic structure of the algorithm, Fig. 2, remains the same for integer, float, polynomial, etc. matrices and so can be abstracted over the domain. We have already seen the simplest case of domain abstraction in `param_code1` (Section 3.1), which took code-generators such as `plus` and `one` as arguments. We need far more than two parameters: our domains should include 0, 1, $+$, $*$, (unary and binary) $-$, at least *exact* division, normalization, and potentially a relative size measure. We could group these parameters in tuples or records. It is instead more convenient to use OCaml *structures* (i.e., modules) so that we can take advantage of extensibility, type abstraction and constraints, and especially parameterized structures (*functors*). We define a type, the signature `DOMAIN`, which different domains must satisfy:

```
type domain_kind = Domain_is_Ring | Domain_is_Field

module type DOMAIN = sig
  type v
  val kind   : domain_kind
  val zero   : v
  val one    : v
  val plus   : v -> v -> v
  val times  : v -> v -> v
  val minus  : v -> v -> v
  val uminus : v -> v
  val div    : v -> v -> v
  val better_than : (v -> v -> bool) option
  val normalizer  : (v -> v) option
end

module IntegerDomain : DOMAIN with type v = int  = struct
  type v = int
  let kind      = Domain_is_Ring
  let zero      = 0    and one       = 1
  let plus x y  = x + y and minus x y = x - y
  let times x y = x * y and div x y   = x / y
  let uminus x  = -x
  let normalizer = None
  let better_than = Some (fun x y -> abs x > abs y)
end
```

One particular domain instance is `IntegerDomain`. The type annotation `DOMAIN` in the definition of `IntegerDomain` makes the compiler verify that the defined structure is indeed of a type `DOMAIN`. The annotation may be omitted (see `ZpMake` below), in which case the compiler will verify the type when we try to use that structure as a `DOMAIN` (typically

[6] Sometimes also called the Gauss–Bareiss algorithm.

Please cite this article in press as: J. Carette, O. Kiselyov, Multi-stage programming with functors and monads: Eliminating abstraction overhead from generic code, Science of Computer Programming (2008), doi:10.1016/j.scico.2008.09.008

in a functor instantiation). In any case, the errors such as missing "methods" or methods with incorrect types will be caught statically, *before* any code generation takes place. The variant `Domain_is_Ring` of `IntegerDomain.domain_kind` encodes a semantic constraint: that 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.

A more complex domain is `Zp`, the field of integers in prime characteristic:

```
module ZpMake(P:sig val p:int end) = struct
    type v = int
    let kind      = Domain_is_Field
    let zero      = 0 and one = 1
    let plus x y  = (x + y) mod P.p
    let times x y = (x * y) mod P.p
    ...
    let normalizer = None and better_than = None
    let () = assert (is_prime P.p)
end
```

This domain is parametrized by an integer `p`. To be more precise, the structure `ZpMake` is parameterized over another structure of the type described by the signature `P`, which has one field, the `int` value `p`. Such a parameterized structure (or, a function from structures to structures) is a *functor*. The result of `ZpMake` is a domain which is a field with no defined order. Hence `normalizer` and `better_than` are set to `None`. `Zp` forms a field only when `p` is prime.[7] Since we intend to make a field of prime characteristic, we must check this, which is done in the last line of the above code. That line differs from the other bindings in `ZpMake` in that it neither defines a function, such as `plus`, nor binds a value, such as `zero`. This non-value expression `assert (is_prime P.p)`, which we will call an initializing expression, will be evaluated when the corresponding module is instantiated.

```
module Z19 = ZpMake(struct let p = 19 end)
```

If we replace `p = 19` with `p = 9` above, we receive a "run-time" error. However, it is raised as we instantiate and combine modules that will *eventually* make the generator. Although the error is reported at "run-time" rather than during compilation, as one might have hoped, the error is raised when *generating the generator* — well before the generation of the target code could begin. In our code we make extensive use of these "preflight checks" which are performed as part of module initialization. These checks seem to offer a good compromise: they are dynamic and so do not require a complicated type system; on the other hand, the checks are run quite early, when building code generators, and so ensure that no code violating the corresponding *semantic* constraints will be generated. Although some may frown on the use of module initializing expressions, as in general, this requires careful attention to sharing and multiple instantiations of a module, these concerns do not apply in our case: our preflight checks are all idempotent and maintain no state.

5.2. Abstracting interpretations

In our sample GE code, Fig. 2, the operation `not (curr == 0)`, line 46, compares two integers (or, generally, two domain elements); the operation `det_magn := a_rc`, line 55, assigns the domain element to the corresponding reference cell; the operation `r := !r + 1` on the next line increments the rank, the cardinal number. One can easily imagine a different interpretation of the same program, where `not (curr == 0)` *generates* code to compare two domain elements, `det_magn := a_rc` and `r := !r + 1` generate code for the assignment and the in-place increment. The structure of the GE algorithm is clearly invariant upon this change in interpretation. This lets us abstract the algorithm over the interpretation of basic operations, so that the same GE code, given different concrete interpretations, can LU factorize a given matrix, can generate LU-factorization programs in OCaml as well as C or Fortran, or can pretty-print the factorization procedure.

The interpretation aspect is also implemented as an OCaml module. This aspect is quite large – there are many basic operations to abstract over – and so the module is structured into several sub-modules. First we introduce an abstract type `('a,'b) rep` which describes what sort of objects the interpretation may produce (e.g., ASTs, strings with C code, etc). The type has two parameters: the second specifies the type of the object, and the first is the 'placeholder' for all other information that may need to be tracked about the object in a particular interpretation. The interpretation as MetaOCaml code values uses the first parameter of `rep` to track the environment classifier.

The first sub-module of the interpretation aspect is the base domain. The signature `DOMAIN` of Section 5.1 defined the set of operations on base objects of some type `v`. We now generalize, or 'lift', `DOMAIN` into `DOMAINL` so we can likewise operate on other interpretations of these objects, of type `('a,v) rep`:

```
module type DOMAINL = sig
  include DOMAIN
```

[7] or a prime power, a case we do not treat here.

Please cite this article in press as: J. Carette, O. Kiselyov, Multi-stage programming with functors and monads: Eliminating abstraction overhead from generic code, Science of Computer Programming (2008), doi:10.1016/j.scico.2008.09.008

```
  type 'a vc = ('a,v) rep
  val zeroL : 'a vc
  val oneL  : 'a vc
  val ( +^ ) : 'a vc -> 'a vc -> 'a vc
  val ( *^ ) : 'a vc -> 'a vc -> 'a vc
  val ( -^ ) : 'a vc -> 'a vc -> 'a vc
  val uminusL : 'a vc -> 'a vc
  val divL    : 'a vc -> 'a vc -> 'a vc
  val better_thanL : ('a vc -> 'a vc -> ('a,bool) rep) option
  val normalizerL  : ('a vc -> 'a vc) option
end
```

The line `include DOMAIN` says that lifted domains include all members of non-lifted domains, specifically including the initializing expressions with preflight checks.

An interpretation also needs to specify how to compare objects, and to manipulate objects representing integers (typically used as indices of matrix elements). We group the operations into the following two structures:

```
module Logic : sig
  val notL      : ('a, bool) rep -> ('a, bool) rep
  val equalL    : ('a, 'b) rep   -> ('a, 'b) rep   -> ('a, bool) rep
  val notequalL : ('a, 'b) rep   -> ('a, 'b) rep   -> ('a, bool) rep
  val andL      : ('a, bool) rep -> ('a, bool) rep -> ('a, bool) rep
end
module Idx : sig
  val zero : ('a, int) rep
  val one  : ('a, int) rep
  val minusone : ('a, int) rep
  val succ : ('a, int) rep -> ('a, int) rep
  val pred : ('a, int) rep -> ('a, int) rep
  val less : ('a, 'b) rep  -> ('a, 'b) rep -> ('a, bool) rep
  val uminus : ('a, int) rep -> ('a, int) rep
  val add : ('a, int) rep   -> ('a, int) rep -> ('a, int) rep
  val minusoneL : 'a -> ('a -> ('b, int) rep -> 'c) -> 'c
end
```

As we argued in Section 3, we will be writing our generic GE algorithm in a monadic style. For convenience we define the following type synonyms for our `('p,'v) monad` of Fig. 1: The first (`cmonad`) is used for a monadic action that always produces a useful value; the `omonad` synonym describes a monadic action that may produce an interpretation object. The helper type synonym `('pc,'p,'a) cmonad_constraint` is effectively the abbreviation for a set of constraints imposed on its arguments.

```
type ('pc,'p,'a) cmonad_constraint = unit
  constraint 'p = <state : 's list; answer : ('a,'w) rep>
  constraint 'pc = <classif : 'a; answer : 'w; state : 's; ..>

type ('pc,'v) cmonad = ('p,('a,'v) rep) monad
  constraint _ = ('pc,'p,'a) cmonad_constraint

type ('pc,'v) omonad = ('p,('a,'v) rep option) monad
  constraint _ = ('pc,'p,'a) cmonad_constraint
```

Finally, the interpretation aspect must specify the following very basic operations: injection of literal values, function applications, statement sequencing, conditional, loops, creating, dereferencing and assigning to reference cells, etc:

```
val lift  : 'b -> ('a, 'b) rep
val unitL : ('pc,unit) cmonad

val apply  : ('a, 'b -> 'c) rep -> ('a, 'b) rep -> ('a, 'c) rep
val applyM : ('a, 'b -> 'c) rep -> ('a, 'b) rep ->
             (<classif: 'a; ..>,'c) monad

val seqM :
   (<classif: 'a; state: 's; answer: 'b; ..>,'b) cmonad ->
   (<classif: 'a; state: 's; answer: 'c; ..>,'c) cmonad ->
   (<classif: 'a; state: 's; ..>,'c) cmonad
val optSeqM :
   (<classif: 'a; state: 's; answer: 'b; ..>,'b) cmonad ->
   (<classif: 'a; state: 's; answer: 'b; ..>,'b) cmonad option ->
   (<classif: 'a; state: 's; ..>,'c) cmonad
```


Please cite this article in press as: J. Carette, O. Kiselyov, Multi-stage programming with functors and monads: Eliminating abstraction overhead from generic code, Science of Computer Programming (2008), doi:10.1016/j.scico.2008.09.008

```
val ifM : ('a, bool) rep ->
   (<classif: 'a; state: 's; answer: 'b; ..>,'b) cmonad ->
   (<classif: 'a; state: 's; answer: 'b; ..>,'b) cmonad ->
   (<classif: 'a; state: 's; ..>,'b) cmonad

val liftRef : ('a, 'b) rep -> ('a, 'b ref) rep
val liftGet : ('a, 'b ref) rep -> ('a, 'b) rep
val assign  : ('a, 'b ref) rep -> ('a, 'b) rep -> ('a, unit) rep
val assignM : ('a, 'b ref) rep -> ('a, 'b) rep ->
              (<classif: 'a; ..>,unit) cmonad
```

The 'pure' operations of the interpretation produce interpretation objects (of type `('a,'b) rep`) as their result. We can trivially 'lift' these operations into the monad, by composing them with `ret` from the monad. Some other operations, like `seqM` and `ifM` are present only in the monadic form: these are *control* operations.

We provide two concrete instances of the interpretation aspect: one uses thunks (for benchmarking and regression tests purposes) and the other uses MetaOCaml's code values `('a,'v) code` as the realization of `('a,'v) rep`. We could also use interpretations producing C or Fortran code. The following is a sample implementation of the interpretation aspect for MetaOCaml code values:

```
let lift x = .< x >.   and  unitL = fun s k -> k s .< () >.

let liftRef x = .< ref .~x >.  and   liftGet x = .< ! .~x >.
let liftPair x = (.< fst .~x >., .< snd .~x >.)

module Logic = struct
  let notL a        = .< not .~a >.
  let equalL a b    = .< .~a = .~ b >.
  let notequalL a b = .< .~a <> .~ b >.
  let andL a b      = .< .~a && .~b >.
end

module Idx = struct
  let zero = .< 0 >. and one = .< 1 >. and minusone = .< -1 >.
  let succ a = .< .~a + 1 >.  and  pred a = .< .~a - 1 >.
  let less a b = .< .~a < .~b >.
  let uminus a = .< - .~a >.  and  add a b = .< .~a + .~b >.
end

let update  a f = let b = f (liftGet a) in .< .~a := .~b >.
let assign  a b = .< .~a := .~b >.
let apply   f x = .< .~f .~x >.
let updateM a f = ret (update a f)
let assignM a b = ret (assign a b)
let applyM  f x = ret (apply f x)
```

The following is a particular instance of `DOMAINL`, the lifted version of `IntegerDomain` of the previous section, again for the MetaOCaml code value interpretation.

```
module IntegerDomainL = struct
    include IntegerDomain
    type 'a vc = ('a,v) code
    let zeroL      = .< 0 >.       and oneL       = .< 1 >.
    let (+^) x y   = .<.~x + .~y>. and ( -^ ) x y = .<.~x - .~y>.
    let ( *^ ) x y = .<.~x * .~y>. and divL x y   = .<.~x / .~y>.
    let uminusL x  = .<- .~x>.
    let normalizerL  = None
    let better_thanL = Some (fun x y -> .<abs .~x > abs .~y >. )
end
```

Such lifting is completely straightforward. However, it is a program-text to program-text transformation over modules, and as such, could only be automated (currently) by further use of camlp4.

We consider the interpretation aspect "open" and so we can refer to all operations such as `apply`, `seqM`, etc, without further qualification.

5.3. Containers

For our purposes, a container is an abstraction of an n-dimensional vector space, which we specialize here for $n = 1, 2$. The 2-dimensional case is our main interest, and its signature contains many functions particular to $n = 2$. For examples, we

Please cite this article in press as: J. Carette, O. Kiselyov, Multi-stage programming with functors and monads: Eliminating abstraction overhead from generic code, Science of Computer Programming (2008), doi:10.1016/j.scico.2008.09.008

have rows and columns and operations specialized for them. A container explicitly abstracts the underlying representation of the data-structure, while offering an interface which is better-suited to linear algebra. In particular, a container is an abstraction of the flat vector `container2dfromvector` of our sample algorithm in Section 4.

The signature `CONTAINER2D` below specifies that a container must provide functions `dim1` and `dim2` to extract the dimensions, functions `getL` to generate container getters, 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 (see Section 4 for an example of such an optimization).

```
module type CONTAINER2D = sig
  module Dom:DOMAINL
  type contr
  type 'a vc = ('a,contr) rep
  type 'a vo = ('a,Dom.v) rep
  val getL : 'a vc -> ('a,int) rep -> ('a,int) rep -> 'a vo
  val dim1 : 'a vc -> ('a,int) rep
  val dim2 : 'a vc -> ('a,int) rep
  val mapper : ('a vo -> 'a vo) option -> 'a vc -> 'a vc
  val copy : 'a vc -> 'a vc
  val init : ('a,int) rep -> ('a, int) rep -> 'a vc
  val augment : 'a vc -> ('a,int) rep -> ('a, int) rep -> 'a vc ->
                ('a, int) rep -> 'a vc
  val identity : ('a,int) rep -> ('a, int) rep -> 'a vc
  val swap_rows_stmt : 'a vc -> ('a, int) rep -> ('a, int) rep ->
                       ('a,unit) rep
  val swap_cols_stmt : 'a vc -> ('a, int) rep -> ('a, int) rep ->
                       ('a,unit) rep
  val row_head : 'a vc -> ('a, int) rep -> ('a, int) rep -> 'a vo
  val col_head_set : 'a vc -> ('a,int) rep -> ('a,int) rep -> 'a vo ->
            ('a,unit) rep
end
```

The type of our containers includes the lifted domain `Dom` as one of the components. This is quite convenient, since operations on containers are usually accompanied by operations on retrieved values, which are subsequently stored again. The particular instances of the containers are parametric over a `DOMAINL`, i.e. functors from a `DOMAINL` module to the actual implementation of a container. For example, the following functor defines a matrix container as a single array, with elements stored in row-major order — the container used in Fig. 2.

```
module GenericVectorContainer(Dom:DOMAINL) =
  struct
  module Dom = Dom
  type contr = Dom.v container2dfromvector
  type 'a vc = ('a,contr) code
  type 'a vo = ('a,Dom.v) code
  let getL x i j = .< ((.~x).arr).(.~i* (.~x).m + .~j) >.
  let dim2 x = .< (.~x).n >.      (* number of rows *)
  let dim1 x = .< (.~x).m >.      (* number of cols *)
  . . .
```

The accompanying code [28], includes an implementation with elements stored in a 1D array in a column-wise (Fortran-like) mode, and another for a matrix represented as an array of rows.

We could have defined the type `CONTAINER2D` to be a functor with `DOMAINL` as an argument. The type `CONTAINER2D` is used in the signatures of other functors such as `GenLA` of Section 5.7. If the type `CONTAINER2D` were a functor, `GenLA` would have been a higher-order functor, and we would have to pass to `GenLA` two arguments: the container functor and a `DOMAINL` to apply the container functor to. In the current design, the user first builds a particular container instance by applying the functor such as `GenericVectorContainer` to the desired domain. The user then passes this container instance to `GenLA` as a single argument. The current design simplifies module signatures at the expense of making the instantiation of the GE algorithm "multi-stage". The stepwise instantiation seems more intuitive however; it is certainly faster, since currently OCaml is quite slow when instantiating functors with complex signatures.

As mentioned above, the generic GE algorithm `GenLA` is parametrized over `CONTAINER2D`. Given a particular container instance, the functor `GenLA` yields a module containing various algorithmic aspects for the user to choose, as well as the main GE driver. The given container instance is available to all these aspects under the name of `C` (so that the type of the container can be referred to as `C.contr` and the dimensions can be obtained using `C.dim1` and `C.dim2`). The domain, which is part of the container, can be referred to as `C.Dom` and the type of the domain elements is `C.Dom.v`. We shall see many such references as we describe particular algorithmic aspects below.

Please cite this article in press as: J. Carette, O. Kiselyov, Multi-stage programming with functors and monads: Eliminating abstraction overhead from generic code, Science of Computer Programming (2008), doi:10.1016/j.scico.2008.09.008

5.4. Determinant aspect

The determinant aspect is one of several tracking and strategic aspects. The main GE procedure invokes various functions of these aspects at some interesting points, for example when the pivot is required, when two rows have to be permuted, or when the final answer to the user has to be built. An aspect may do something at each one of these times, for example, find a pivot according to a pivoting strategy, update the current value of the determinant, etc.

Our sample Gaussian Elimination algorithm (Fig. 2), demonstrates that tracking the determinant is quite complex: first we define the variables `det_sign` and `det_magn` used for tracking (lines 22 and 23), then we have to change the sign when swapping two rows or two columns, or when the matrix found to be singular (lines 32, 37, 58). The value of the determinant should be updated for each pivoting (line 55). Finally, we convert the tracking state to the resulting determinant value (lines 63–65). Most importantly, we observe that these lines are not contiguous: the determinant aspect is consulted at several separate places in the GE algorithm, and the aspect is supposed to keep state. As with other aspects, the determinant aspect is a module, and has the following signature:

```
module type DETERMINANT = sig
  type tdet = C.Dom.v ref
  type 'a lstate
  type 'pc pc_constraint = unit
    constraint 'pc = <state : [> `TDet of 'a lstate ]; classif : 'a; ..>
  type ('pc,'v) lm = ('pc,'v) cmonad
    constraint _  = 'pc pc_constraint
  type ('pc,'v) om = ('pc,'v) omonad
    constraint _  = 'pc pc_constraint
  type 'pc nm = ('p,unit) monad
    constraint _ = ('pc,'p,_) cmonad_constraint
    constraint _ = 'pc pc_constraint
  val decl : unit -> 'b nm (* no code is generated *)
  val upd_sign  : unit -> ('b,unit) om
  val zero_sign : unit -> ('b,unit) lm
  val acc_magn  : ('a,C.Dom.v) rep -> (<classif : 'a; ..>,unit) lm
  val get_magn  : unit -> ('b,tdet) lm
  val set_magn  : ('a,C.Dom.v) rep -> (<classif : 'a; ..>,unit) lm
  val fin       : unit -> ('b,C.Dom.v) lm
end
```

The type `'a lstate` denotes the state being kept as part of the overall monadic state (tagged as `` `TDet ``). The signature specifies the operations of the aspect, such as `decl` for initializing the tracking state, `zero_sign` to record matrix singularity, `set_magn` to set the magnitude and `fin` to convert the tracking state to the resulting value, of the type `C.Dom.v`, the type of the container elements. Other aspects follow a similar outline. It is instructive to examine the difference in the return types of the `decl`, `upd_sign` and `zero_sign` functions. The first says that `decl` is an action which is executed only for its side-effect. It yields no interpretation object (e.g., produces no code value). The function `zero_sign` always produces a code value — an expression such as assignment that has type unit. The function `upd_sign` is a generator that may produce code, or may not. The option type lets us avoid generating code such as `a := b; ()` with a pointless `()`.

We have two instances of `DETERMINANT`. The first corresponds to no determinant tracking, and so all functions are dummy.

```
module NoDet = struct
  type tdet = C.Dom.v ref
  type 'a lstate = unit
  let decl ()     = ret ()
  let upd_sign () = ret None
  let zero_sign () = unitL
  let acc_magn _   = unitL
  let get_magn ()  = ret (liftRef C.Dom.zeroL)
  let set_magn _   = unitL
  let fin () = failwith "Determinant is needed but not computed"
end
```

The second instance does track the determinant. For integer matrices, and in general whenever the matrix elements do not form a field, the fraction-free update requires tracking some facets of the determinant, even if we do not output it.

```
module AbstractDet = struct
  open C.Dom
  type tdet = v ref
  type 'a lstate = ('a,int ref) rep * ('a,tdet) rep
  let decl () = perform
```

```
      magn <-- retN (liftRef oneL);     (* track magnitude *)
      sign <-- retN (liftRef Idx.one); (* track the sign: +1, 0, -1 *)
      mo_extend ip (sign,magn)
  let upd_sign () = perform             (* flip sign *)
      (sign,_) <-- mo_lookup ip;
      ret (Some (assign sign (Idx.uminus (liftGet sign))))
  let fin = fun () -> perform           (* reconstruct det and *)
      (sign,magn) <-- mo_lookup ip;     (* generate code       *)
      ifM (Logic.equalL (liftGet sign) Idx.zero) (ret zeroL)
      (ifM (Logic.equalL (liftGet sign) Idx.one) (ret (liftGet magn))
          (ret (uminusL (liftGet magn))))
  ...
end
```

The `decl` method generates two let-bindings for mutable variables tracking the magnitude and the sign of the determinant, and places the names of these variables in the monadic state. The `upd_sign` method, invoked from row- or column-swap generators, retrieves the name of the sign-accumulating variable and generates the code to update the sign. The `fin` method retrieves the names of both accumulators and generates code to compute the final, signed determinant value. The generated code closely corresponds to the lines dealing with `det_magn` and `det_sign` in Fig. 2.

5.5. Other aspects

For completeness, we briefly describe and show the signatures of the other aspects. `RANK` tracks rank; this aspect is a simpler version of determinant tracking. `PIVOTKIND` defines the representation of the permutation matrix accumulating pivoting permutations; `TRACKPIVOT` specifies if these permutations are tracked at all. `LOWER` tracks the `L` factor when the GE algorithm is used for in-place LU-decomposition. `PIVOT` is a higher-order functor abstracting over the pivoting algorithm (full pivoting, row pivoting, no pivoting, etc). If a pivot is found, the aspect should swap rows and columns appropriately. The swapping may need to be tracked in a permutation matrix (or some other representation). If we track the determinant, swapping must update its sign. The swapping also affects the `L` factor being accumulated. The `UPDATE` aspect abstracts over the algorithm for updating matrix elements during row-reductions: full division update or fraction-free update. Finally, `INPUT` is an interface aspect letting us handle both ordinary and augmented matrices.

```
module type RANK = sig
  type 'a tag_lstate
  val decl  : unit -> ('b, int ref) lm
  val succ  : unit -> ('b, unit) lm
  val fin   : unit -> ('b, int) lm
end

module type PIVOTKIND = sig
  type perm_rep
  type 'a ira = ('a, int) rep
  type 'a fra
  type 'a pra = ('a, perm_rep) rep
  val add   : 'a fra -> 'a pra -> 'a pra
  val empty : 'a ira -> 'a pra
  val rowrep : 'a ira -> 'a ira -> 'a fra
  val colrep : 'a ira -> 'a ira -> 'a fra
end

module type TRACKPIVOT = sig
  type perm_rep
  type 'a ira = ('a, int) rep
  type 'a fra  type 'a pra  type 'a lstate
  type 'pc pc_constraint = unit
    constraint 'pc = <state : [> `TPivot of 'a lstate ]; classif : 'a; ..>
  type ('pc,'v) lm = ('pc,'v) cmonad constraint _  = 'pc pc_constraint
  type ('pc,'a) nm = ('p,unit) monad
    constraint _ = ('pc,'p,'a) cmonad_constraint
    constraint _ = 'pc pc_constraint
  val rowrep : 'a ira -> 'a ira -> 'a fra
  val colrep : 'a ira -> 'a ira -> 'a fra
  val decl  : ('a, int) rep -> ('pc,'a) nm
  val add : 'a fra -> (<classif : 'a; state : [> `TPivot of 'a lstate ]; ..>, unit) omonad
  val fin   : unit -> ('b,perm_rep) lm
end
```

```
module type LOWER = sig
  type 'a lstate = ('a, C.contr) rep
  type ('pc,'v) lm = ('pc,'v) cmonad
    constraint 'pc = <state : [> `TLower of 'a lstate ]; classif : 'a; ..>
  val decl   : ('a, C.contr) rep -> (<classif : 'a; ..>, C.contr) lm
  val updt   : 'a C.vc -> ('a,int) rep -> ('a,int) rep -> 'a C.vo ->
           'a C.Dom.vc -> (<classif : 'a;..>, unit) lm option
  val fin    : unit -> ('a,  C.contr) lm
  val wants_pack : bool
end

module type PIVOT = functor (D: DETERMINANT) -> functor (P: TRACKPIVOT) ->
         functor (L: LOWER) -> sig
 val findpivot : 'a wmatrix -> 'a curpos ->
   (<classif : 'a;  state : [> `TDet of 'a D.lstate | `TPivot of 'a P.lstate ]; ..>,
    C.Dom.v option) cmonad
end

type update_kind = FractionFree | DivisionBased
module type UPDATE = functor(D:DETERMINANT) -> sig
  type 'a in_val = 'a C.Dom.vc
  val update :  'a in_val -> 'a in_val -> 'a in_val -> 'a in_val ->
      ('a in_val -> ('a, unit) rep) -> ('a, C.Dom.v ref) rep ->
      (<classif : 'a; ..>, unit) cmonad
  val update_det : 'a in_val -> (<classif : 'a; ..>,unit) D.lm
  val upd_kind : update_kind
end

module type INPUT = sig
  type inp
  val get_input : ('a, inp) rep ->
        (<classif : 'a; ..>, ('a, C.contr) rep * ('a, int) rep * bool) monad
end
```

5.6. Output

More interesting, is the aspect of what to return from the GE algorithm. One could create an algebraic data type (as was done in [1]) 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. Below, we show this module type and one instantiation, `OutDetRank`, 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. That choice of output corresponds to our sample algorithm in Fig. 2.

```
module type OUTPUTDEP = sig
    module PivotRep : PIVOTKIND
    module Det      : DETERMINANT
end
module type INTERNAL_FEATURES = sig
  module R      : TrackRank.RANK
  module P      : TRACKPIVOT
  module L      : LOWER
end
module type OUTPUT = functor(OD : OUTPUTDEP) -> sig
  module IF : INTERNAL_FEATURES
  type res
  val make_result : 'a wmatrix -> (<classif: 'a;...>,res) cmonad
end
module OutDetRank(OD : OUTPUTDEP) = struct
  module IF = struct
    module R   = Rank
    module P   = DiscardPivot
    module L   = NoLower end
  type res = C.contr * C.Dom.v * int
  let make_result m = perform
    det  <-- OD.Det.fin ();
    rank <-- IF.R.fin ();
    ret (Tuple.tup3 m.matrix det rank)
  let _ = OD.Det.fin ()
  let _ = IF.R.fin ()
end
```

The initialization expressions `OD.Det.fin ()` and `IF.R.fin ()` are preflight checks. As we saw in the previous section, both instances of `DETERMINANT` contain a `fin ()` function to generate code representing the computed determinant. The instance `NoDet` however does no tracking, and so `fin ()` raises an error. The code `let _ = OD.Det.fin ()` in `OutDetRank` invokes this `fin` function, which will produce the monadic code generating action, or raise an error. We do not run the action at that time — we only make sure there is an action to run. This is another preflight check, to rule out the semantic error where a user specifies that the determinant should be computed and returned, and yet specifies the `NoDet` aspect.

The type `wmatrix` denotes the `(0,0)-(numrow-1,numcol-1)` rectangular block of `matrix`:

```
type 'a wmatrix = {matrix: 'a C.vc; numrow: ('a,int) rep;
                   numcol: ('a,int) rep}
```

The type is used in the `INPUT`, `OUTPUT` and `PIVOT` aspects to refer to the non-augmented part of the matrix.

The module of signature `INTERNAL_FEATURES` bundles information tracking aspects. The latter are not directly selectable by the user. Rather, they are *functions* of other user choices. The implementations of the tracking aspects such as `Rank`, `NoRank`, `PackedLower` are quite similar (and simpler) than the implementation of the `AbstractDet` and `NoDet` aspects in Section 5.4. The choice of a particular tracking aspect may depend on all other choices (e.g. it is not possible to extract the `L` factor if the domain is not a field). Currently, we use preflight checks to ensure consistency of tracking aspects. Previously [26] we tried to implement all our preflight tests at the (module) type level, using sharing constraints and module computations. That lead to obscure code, long impenetrable type error messages, and very slow compilation. Furthermore, type-level computations in OCaml are not powerful enough for the tasks such as verifying that an integer is prime (see Section 5.1).

5.7. Main generation

This section presents the core GE algorithm `GenGE`, after all aspects have been factored out. We also describe how to instantiate the core algorithm with sample aspects, to obtain particular GE procedures, such as our running example (Fig. 2). The instantiation process is multi-step, mainly for the sake of speed of OCaml compilation. The benefit of the multi-step process, simple module signatures, may also help make the instantiation process more comprehensible.

The core GE algorithm and all the aspects are part of one large functor, `Ge.LAMake`, parameterized by the interpretation aspect, Section 5.2. If we select as an interpretation generating code in the form of MetaOCaml code values, we write

```
module GEF = Ge.LAMake(Code)
open GEF
open Domains_code
```

Here, `Code` is the interpretation with `('a,'b) rep` being `('a,b) code`; the module `Domains_code` contains various instances of `DOMAINL` (such as `FloatDomainL`, `IntegerDomainL`) for that particular choice of `('a,'b) rep`. Opening `GEF` makes available various container functors and the functor `GenLA`. Combining the container functors with domain structures gives containers with particular elements and particular representation. For example, in

```
module GVC_I = GenericVectorContainer(IntegerDomainL)
module G_GVC_I = GenLA(GVC_I)
open G_GVC_I
open G_GVC_I.GE
```

we define `GVC_I` to represent a matrix with integer elements arranged in a flat vector in row-major order. Instantiating the functor `GenLA` with this container makes available all algorithmic and tracking aspects of GE (such as `AbstractDet`, `NoDet`, `FullPivot`, etc) as well as the module `GE` with the core GE generator functor `GenGE` (besides `GE`, `GenLA` contains modules for GE-based solvers). All these components already incorporate the choices for the container, domain and interpretation aspects we have made earlier. We may now select particular features of the desired GE algorithm and instantiate and run `GenGE`.

We combine all user-selectable aspects in a "record", which serves as a "keyword argument" list to the `GenGE` functor, shown later.

```
module type FEATURES = sig
  module Det       : DETERMINANT
  module PivotF    : PIVOT
  module PivotRep  : PIVOTKIND
  module Update    : UPDATE
  module Input     : INPUT
  module Output    : OUTPUT
end
```

We instantiate GenGE by passing to the functor the "record" of various aspects; the order is irrelevant, but all aspects such as Det must be specified. In the following code, we request GE generation with full pivot, fraction-free update, operating on non-augmented matrix and returning the U factor, determinant and the rank. This choice corresponds to our sample algorithm of Fig. 2.

```
module GenIV5 = GenGE(struct
    module Det      = AbstractDet
    module PivotF   = FullPivot
    module PivotRep = PermList
    module Update   = FractionFreeUpdate
    module Input    = InpJustMatrix
    module Output   = OutDetRank end)
let instantiate gen =
    .<fun a -> .~(runM (gen .<a>.) []) >.;;
let resIV5 = instantiate GenIV5.gen ;;
```

We run the monad, passing in the initial state [] and thus obtain code, which we can see by printing resIV5. This code can then be "compiled" as !. resIV5 or with offshoring [47]. The code for resIV5 (Appendix B) 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 resulting function returns a triple int array * int * int of the U-factor, determinant and the rank. It is instructive to compare the generated code with the corresponding code in Fig. 2, the hand-written implementation of the textbook pseudo-code; the only difference is the naming of variables and the inlining of pivoting and swapping functions.

The following is another instantiation of the GE generator, with a different set of aspects.

```
module GAC_F = GenericArrayContainer(FloatDomainL)
module G_GAC_F = GenLA(GAC_F)
open G_GAC_F
open G_GAC_F.GE
module GenFA9 = GenGE(struct
    module Det      = NoDet
    module PivotF   = RowPivot
    module PivotRep = PermList
    module Update   = DivisionUpdate
    module Input    = InpJustMatrix
    module Output   = Out_LU_Packed end)
```

The code generated by GenFA9 (Appendix C) shows no traces of determinant tracking whatsoever: 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 [28]. The website also contains benchmark code and timing comparisons.

The core GE algorithm GenGE, as part of Ge.LAMake and GenLA, is already parameterized by the domain, container and interpretation. The algorithm is further parameterized by FEATURES, i.e., pivoting policy (full, row, nonzero, no pivoting), update policy (with either 'fraction-less' or full division), determinant, permutation matrix, input and output specifications. Some of the argument modules, such as PIVOT are functors themselves (parameterized by the domain, the container, and the determinant). We rely on module subtyping: For example, F.Output of the type OUTPUT is a functor requiring an argument of the signature OUTPUTDEP. The fact that the signature FEATURES contains all the fields of OUTPUTDEP and then some lets us pass F (of the type FEATURES) as an argument to instantiate F.Output.

```
module GenGE(F : FEATURES) = struct
  module O = F.Output(F)

  let wants_pack = O.IF.L.wants_pack
  let can_pack   =
    let module U = F.Update(F.Det) in
    (U.upd_kind = DivisionBased)
  (* some more preflight tests *)
  let _ = ensure ((not wants_pack) || can_pack)
        "Cannot return a packed L in this case"

  let zerobelow mat pos =
    let module IF = O.IF in
    let module U = F.Update(F.Det) in
    let innerbody j bjc = perform
      whenM (Logic.notequalL bjc C.Dom.zeroL ) (perform
        det <-- F.Det.get_magn ();
        optSeqM (Iters.col_iter mat.matrix j (Idx.succ pos.p.colpos)
         (Idx.pred mat.numcol) C.getL
              (fun k bjk -> perform
              brk <-- ret (C.getL mat.matrix pos.p.rowpos k);
```


Please cite this article in press as: J. Carette, O. Kiselyov, Multi-stage programming with functors and monads: Eliminating abstraction overhead from generic code, Science of Computer Programming (2008), doi:10.1016/j.scico.2008.09.008

```
            U.update bjc pos.curval brk bjk
              (fun ov -> C.col_head_set mat.matrix j k ov) det) UP )
            (IF.L.updt mat.matrix j pos.p.colpos C.Dom.zeroL
              (* this makes no sense outside a field! *)
              (C.Dom.divL bjc pos.curval))) in
      perform
        seqM (Iters.row_iter mat.matrix pos.p.colpos
               (Idx.succ pos.p.rowpos)
               (Idx.pred mat.numrow) C.getL innerbody UP)
             (U.update_det pos.curval)

  let init input = perform
     let module IF = O.IF in
       (a,rmar,augmented) <-- F.Input.get_input input;
       r <-- IF.R.decl ();
       c <-- retN (liftRef Idx.zero);
       b <-- retN (C.mapper C.Dom.normalizerL (C.copy a));
       m <-- retN (C.dim1 a);
       rmar <-- retN rmar;
       n <-- if augmented then retN (C.dim2 a) else ret rmar;
       F.Det.decl ();
       IF.P.decl rmar;
       _ <-- IF.L.decl (if wants_pack then b else C.identity rmar m);
       let mat = {matrix=b; numrow=n; numcol=m} in
       ret (mat, r, c, rmar)

  let forward_elim (mat, r, c, rmar) = perform
     let module IF = O.IF in
       whileM (Logic.andL (Idx.less (liftGet c) mat.numcol)
                          (Idx.less (liftGet r) rmar) )
        ( perform
        rr <-- retN (liftGet r);
        cc <-- retN (liftGet c);
        let cp  = {rowpos=rr; colpos=cc} in
        let module Pivot = F.PivotF(F.Det)(IF.P) in
        pivot <-- bind (Pivot.findpivot mat cp) retN;
        seqM (matchM pivot (fun pv ->
               seqM (zerobelow mat {p=cp; curval=pv} )
                    (IF.R.succ ()) )
              (F.Det.zero_sign () ))
             (updateM c Idx.succ) )

  let gen input = perform
     (mat, r, c, rmar) <-- init input;
     seqM
       (forward_elim (mat, r, c, rmar))
       (O.make_result mat)
end
```

A careful reading of this code will reveal that the core of the Gaussian Elimination algorithm (Fig. 2) is still visible in this code generator: for example, the `forward_elim` function iterates over the columns and rows of the matrix, finding a pivot, and zeroing the appropriate entries. With sufficient added syntactic sugar, we could indeed make the generator look like the algorithm. There are more preflight checks for various "semantic" constraints, shown in the following structure of the UPDATE signature:

```
module DivisionUpdate(Det:DETERMINANT) = struct
  open C.Dom
  type 'a in_val = 'a vc
  let update bic brc brk bik setter _ = perform
      y <-- ret (bik -^ ((divL bic brc) *^ brk));
      ret (setter (applyMaybe normalizerL y))
  let update_det v = Det.acc_magn v
  let upd_kind = DivisionBased
  let _ = assert (C.Dom.kind = Domains_sig.Domain_is_Field)
end
```

This structure implements an update policy relying on unrestricted `Dom.divL`. Many domains provide `divL`, for example, the integer domain. The latter however assumes that division is applied only if the dividend is an exact multiple of the divisor. Thus if we specified `module Update = DivisionUpdate` when instantiating `GenIV5` above, we would have received an error because `IntegerDomainL` is not a field. That error occurs before any code is produced (i.e. before `resIV5` is computed).

Please cite this article in press as: J. Carette, O. Kiselyov, Multi-stage programming with functors and monads: Eliminating abstraction overhead from generic code, Science of Computer Programming (2008), doi:10.1016/j.scico.2008.09.008

6. Related and future work

The monad in this paper is similar to the one described in [27,25]. However, those papers used only `retN` and fixpoints (for generation-time iterations). Our work does not involve monadic fixpoints, because the generator is not recursive, but heavily relies on monadic operations for generating conditionals and loops.

`Blitz++` [9], and C++ template meta-programming in general, similarly eliminate levels of abstraction. With traits and concepts, some domain-specific knowledge can also be encoded. However overhead elimination critically depends on full inlining of all methods by the compiler, which has been reported to be challenging to insure. Furthermore, all errors (such as type errors and concept violation errors, i.e., 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.

ATLAS [14] 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. A significant part of ATLAS' complexity is that the generator is extremely error-prone and difficult to debug. Indeed, when generating C code in C using `printf`, nothing prevents producing code that is missing semicolons, open or close parentheses or variable bindings. MetaOCaml gives us assurance that these errors, and more subtle type errors, shall never occur in the generated code. SPIRAL [23] is another 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 proven 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 [4] and Aldor [48]. Even though the Aldor compiler can frequently optimize away a lot of abstraction overhead, it does not provide any guarantees that it will do so, unlike our approach.

We should also mention early work [49] on automatic specialization of mathematical algorithms. Although it can eliminate some overhead from a very generic implementation (e.g. by inlining aspects implemented as higher-order functions), specialization cannot change the type of the function and cannot efficiently handle aspects that communicate via a private shared state.

The paper [50] describes early simple experiments in *automatic* and manual staging, and the multi-level language based on an annotated subset of Scheme (which is untyped and has no imperative features). The generated code requires post-processing to attain efficiency.

Our code was initially motivated by trying to unify the various implementations found in Maple. Interestingly, when we compare our end result with the options available from Maple's `LUDecomposition` algorithm, we notice a great deal of similarity. The biggest difference is that in Maple, all the choices are done dynamically (and are dynamically typed), while ours choices are done statically, in a statically typed environment. To us, this shows that the design space along the dynamic–static dimension is quite large and versatile.

Unlike traditional approaches [33], the interfaces of our generated routines vary depending on the choices of input and output aspects. Dynamic approaches in Object-oriented languages (late binding and dynamic dispatch) or functional languages (Haskell's dictionary-based *type classes*) also offer flexibility of interfaces. In our approach, however, it is the generator that produces code whose interfaces depend on the arguments of the generator. The interfaces of the generated routines are all fixed and hence efficient.

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

It would be interesting to implement a camlp4 extension that automates the lifting of (simple) modules as done in Section 5.2, first to the code level, and then to monadic values. Even more interesting, would be a (typed) extension to MetaOCaml that would allow us to write such code with the same guarantees that the rest of MetaOCaml already affords us. Unfortunately, as modules are not first-class objects in OCaml, this currently seems out of reach.

We plan to further investigate the connection between delimited continuations and our implementations of code generators such as `ifM`. The ultimate (and plausible) goal is to write an algorithm in (almost) regular OCaml once, and be able to either run it as a regular OCaml program, or turn it into a code generating aspect.

There are many more aspects which can also be handled: error reporting (i.e. asking for the determinant of a non-square matrix), memory hierarchy issues, loop-unrolling [7], warnings when zero-testing is undecidable and a value is only probabilistically non-zero, etc.

7. Conclusion

In this paper, we have demonstrated code extensively parameterized by complex aspects at no run-time overhead. The combination of stateless functors and structures, and our monad with compositional state makes aspects composable without having to worry about value aliasing. The only constraints to compositionality are the typing ones, plus the constraints we specifically impose, including semantic constraints.

Please cite this article in press as: J. Carette, O. Kiselyov, Multi-stage programming with functors and monads: Eliminating abstraction overhead from generic code, Science of Computer Programming (2008), doi:10.1016/j.scico.2008.09.008

7.1. On aspects

There is an interesting relation with aspect-oriented code [30]: in AspectJ, aspects are (comparatively) lightly typed, and are post-facto extensions of potential program traces, specified in a particular language; these tend to be created to follow the operational behavior of existing code, but are not restricted to such a setting. In our work, aspects are weaved together "from scratch" to make up a piece of code. One can understand previous work to be more akin to dynamically typed and dynamically specified aspect weaving, while we have started investigating statically typed and statically specified aspect weaving.

While the first two families of aspects (abstract domain and concrete representation) are the most obvious, it is quite difficult to separate them out cleanly. Attempts at such a separation in a non-staged setting have lead to fantastically inefficient code, unacceptable for scientific computation. Staging permitted us, perhaps for the first time, to think in terms of an ideal design without worrying about abstraction penalties. Interestingly, in conversations of the first author with D. Parnas, we discovered this is apparently what [33] was advocating. We believe that we are taking the first steps towards a *typed yet efficient* realization of these ideas, where the various design-time entities can be directly encoded in machine-checkable form.

7.2. Methodology

Our overall approach can best be described as a combination of two approaches: hand-writing a code generator (cogen) suitable for multi-level specialization [49,50] and creating an embedded domain-specific language (EDSL) [51,52]. Our approach, using staging, monads and functors, seems to permit an extensible set of aspects and appears flexible enough for iterative improvement in the design.

As we wanted to ensure that we were indeed firmly in a *cogen* setting, we abstracted out the underlying programming language completely. This allowed us to both generate (efficient) code and to write a directly runnable (but highly inefficient) version of the algorithm from the same generator. But this essentially abstracted out all of the syntactic sugar of the underlying programming language, and all we were left with was function application and monadic composition. This means the code for our generator looks mostly like Scheme with added syntax for monads!

More specifically, we start from a *known* set of implementations of an algorithm, and extract commonalities and variation points. This is unlike [34] and most subsequent approaches to product families, as we do not over-engineer our design by imagining variations that are unlikely to come up in realistic situations, but only create variations when we notice them in actual use. This approach is quite well-suited to the development of scientific software, which has a rich history and where most useful variations have already appeared in some form.

Given a set of commonalities and variation points, the first task is to find *semantic* reasons for these. The underlying reason then forms the basis for the abstraction — a (potentially higher-order) module is created to encapsulate the various concepts, and these are implemented as generators. It is very important at this stage to make sure to reify all available *static* information, so that all of it is available to the generation process. While we will see that the technical solutions exist to take advantage of such information, it is still a difficult design problem to properly encode this information. One important item, is to try to keep the various pieces of generation-time information as orthogonal as possible. This is unlike ordinary encodings of run-time information, where compression and elision frequently lead to increased efficiency. Whenever dependency between various bits of information is inevitable, then higher-order encodings should be sought. When choices need to be made based on some (static) information, it is important to encode this information by using semantic concepts.

Uncited references

Q2

[53], [54]

Acknowledgments

We wish to thank Cristiano Calgano for his help in adapting camlp4 for use with MetaOCaml. Many helpful discussions with Walid Taha are very appreciated. The implementation of the monadic notation, `perform`, was joint work with Lydia van Dijk. We gratefully acknowledge numerous, helpful suggestions by the anonymous reviewers. The first author is supported in part by NSERC Discovery Grant RPG262084-03.

Appendix A. The `perform` monad notation

We support four different constructs to introduce a monadic expression:

```
perform exp
perform exp1; exp2
perform x <-- exp1; exp2
perform let x = foo in exp
```

which is almost literally the grammar of the Haskell's "do"-notation, with the differences that Haskell uses `do` and `<-` where we use `perform` and `<--`. We support not only `let x = foo in ...` expressions but arbitrarily complex let-expressions, including `let rec` and `let module`.

The actual bind function of the monad defaults to `bind` and the pattern-match–failure function to `failwith` (only used for refutable patterns). Extended forms of `perform` let us override these defaults. For example, to use the function named `bind` from module `Mod`, we write

```
perform with module Mod in exp2
```

Appendix B. Code of the `GenIV5` algorithm

The code generated for `GenIV5`, fraction-free LU of the integer matrix represented by a flat vector, full pivoting, returning the U-factor, the determinant and the rank. The comments, however, were inserted by hand.

```
val resIV5 : ('a, GVC_I.contr -> GenIV5.O.res) code =
  .<fun a_1 ->
   let t_2 = (ref 0) in
   let t_3 = (ref 0) in
   let t_4 = (a_1) {arr = (Array.copy a_1.arr)} in
   let t_5 = a_1.m in  (* magnitude of det *)
   let t_6 = a_1.n in  (* sign of the det *)
   let t_7 = (ref 1) in
   let t_8 = (ref 1) in
   while ((((! t_3) < t_5) && ((! t_2) < t_6)) do
    let t_13 = (! t_2) in
    let t_14 = (! t_3) in
    let t_15 = (ref (None)) in
    let t_34 =
     begin (* full pivoting, search for the pivot *)
      for j_30 = t_13 to (t_6 - 1) do
       for j_31 = t_14 to (t_5 - 1) do
        let t_32 = (t_4.arr).((j_30 * t_4.m) + j_31) in
        if (t_32 <> 0) then
         (match (! t_15) with
          | Some (i_33) ->
             if ((abs (snd i_33)) > (abs t_32)) then
              (t_15 := (Some ((j_30, j_31), t_32)))
             else ()
          | None -> (t_15 := (Some ((j_30, j_31), t_32))))
        else ()
       done
      done;
      (match (! t_15) with
       | Some (i_16) -> (* swapping of columns *)
          if ((snd (fst i_16)) <> t_14) then begin
           let a_23 = t_4.arr
           and nm_24 = (t_4.n * t_4.m)
           and m_25 = t_4.m in
           let rec loop_26 =
            fun i1_27 ->
             fun i2_28 ->
              if (i2_28 < nm_24) then
               let t_29 = a_23.(i1_27) in
               a_23.(i1_27) <- a_23.(i2_28);
               a_23.(i2_28) <- t_29;
               (loop_26 (i1_27 + m_25) (i2_28 + m_25))
              else () in
           (loop_26 t_14 (snd (fst i_16)));
           (t_8 := (~- (! t_8))) (* adjust the sign of det *)
          end else ();
          if ((fst (fst i_16)) <> t_13) then begin (* swapping of rows *)
           let a_17 = t_4.arr and m_18 = t_4.m in
           let i1_19 = (t_13 * m_18) and i2_20 = ((snd (fst i_16)) * m_18) in
           for i_21 = 0 to (m_18 - 1) do
            let t_22 = a_17.(i1_19 + i_21) in
            a_17.(i1_19 + i_21) <- a_17.(i2_20 + i_21);
            a_17.(i2_20 + i_21) <- t_22
           done;
           (t_8 := (~- (! t_8)))
          end else ();
          (Some (snd i_16))
        | None -> (None))
     end in
    (match t_34 with
     | Some (i_35) ->
        begin (* elimination loop *)
         for j_36 = (t_13 + 1) to (t_6 - 1) do
```

```
        let t_37 = (t_4.arr).((j_36 * t_4.m) + t_14) in
        if (t_37 <> 0) then begin
         for j_38 = (t_14 + 1) to (t_5 - 1) do
          (t_4.arr).((j_36 * t_4.m) + j_38) <-
           ((((t_4.arr).((j_36 * t_4.m) + j_38) * i_35) -
              ((t_4.arr).((t_13 * t_4.m) + j_38) * t_37)) / (! t_7))
         done;
         (t_4.arr).((j_36 * t_4.m) + t_14) <- 0
        end else ()
       done;
       (t_7 := i_35)
      end;
      (t_2 := ((! t_2) + 1)) (* advance the rank *)
   | None -> (t_8 := 0));
  (t_3 := ((! t_3) + 1))
 done;
 (t_4, (* matrix with the U factor *)
  if ((! t_8) = 0) then 0 (* adjust the sign of the determinant *)
  else if ((! t_8) = 1) then (! t_7)
  else (~- (! t_7)), (! t_2))>.
```

Appendix C. Code of the GenFA9 algorithm

The code generated for GenFA9, LU of the floating point non-augmented matrix represented by a 2D array, row pivoting, returning the complete factorization: L and U factors packed in a single matrix and the permutation matrix represented as the list of row number exchanges.

```
val resFA9 : ('a, GAC_F.contr -> GenFA9.0.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
   let t_8 = (ref ([])) in  (* accumulate permutations in a list *)
   while (((! t_3) < t_6) && ((! t_2) < t_7)) do
    let t_9 = (! t_2) in
    let t_10 = (! t_3) in
    let t_11 = (ref (None)) in
    let t_17 =
     begin   (* row pivoting *)
      for j_14 = t_9 to (t_7 - 1) do
       let t_15 = (t_5.(j_14)).(t_10) in
       if (t_15 <> 0.) then
        (match (! t_11) with
         | Some (i_16) ->
            if ((abs_float (snd i_16)) < (abs_float t_15)) then
             (t_11 := (Some (j_14, t_15)))
            else ()
         | None -> (t_11 := (Some (j_14, t_15))))
       else ()
      done;
      (match (! t_11) with  (* swapping of rows *)
       | Some (i_12) ->
          if ((fst i_12) <> t_9) then begin
           let t_13 = t_5.(t_9) in
           t_5.(t_9) <- t_5.(fst i_12);
           t_5.(fst i_12) <- t_13;  (* and accumulate permutations *)
           (t_8 := ((RowSwap ((fst i_12), t_9)) :: (! t_8)))
          end else ();
          (Some (snd i_12))
       | None -> (None))
    end in
    (match t_17 with  (* elimination loop *)
     | Some (i_18) ->
        begin
         for j_19 = (t_9 + 1) to (t_7 - 1) do
          let t_20 = (t_5.(j_19)).(t_10) in
          if (t_20 <> 0.) then
           for j_21 = (t_10 + 1) to (t_6 - 1) do
            (t_5.(j_19)).(j_21) <-
             ((t_5.(j_19)).(j_21) -. ((t_20 /. i_18) *. (t_5.(t_9)).(j_21)))
           done
          else ()
         done;
         ()
        end;
        (t_2 := ((! t_2) + 1))
     | None -> ());
    (t_3 := ((! t_3) + 1))
   done;
   (t_5, (! t_8))>. (* return both L and U factors, list permutations *)
```


Please cite this article in press as: J. Carette, O. Kiselyov, Multi-stage programming with functors and monads: Eliminating abstraction overhead from generic code, Science of Computer Programming (2008), doi:10.1016/j.scico.2008.09.008

References

[1] J. Carette, Gaussian elimination: A case study in efficient genericity with MetaOCaml, in: First MetaOCaml Workshop 2004, Sci. Comput. Programming 62 (1) (2006) 3–24 (special Issue).
[2] 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.
[3] D. Gruntz, M. Monagan, Introduction to Gauss, SIGSAM Bull. 28 (2) (1994) 3–19.
[4] R.D. Jenks, R.S. Sutor, AXIOM: The Scientific Computation System, Springer Verlag, 1992.
[5] K. Kennedy, B. Broom, K. Cooper, J. Dongarra, R. Fowler, D. Gannon, L. Johnsson, J. Mellor-Crummey, L. Torczon, Telescoping languages: A strategy for automatic generation of scientific problem-solving systems from annotated libraries, J. Parallel Distrib. Comput. 61 (12) (2001) 1803–1826.
[6] T.L. Veldhuizen, Active libraries and universal languages, Ph.D. Thesis, Indiana University Computer Science, May 2004. http://osl.iu.edu/~tveldhui/papers/2004/dissertation.pdf.
[7] A. Cohen, S. Donadio, M.J. Garzarán, C.A. Herrmann, O. Kiselyov, D.A. Padua, In search of a program generator to implement generic transformations for high-performance computing, Sci. Comput. Programming 62 (1) (2006) 25–46.
[8] K. Czarnecki, U.W. Eisenecker, Generative Programming: Methods, Tools, and Applications, ACM Press/Addison-Wesley Publishing Co., 2000.
[9] T.L. Veldhuizen, Arrays in Blitz++, in: Proceedings of the 2nd International Scientific Computing in Object-Oriented Parallel Environments, ISCOPE'98, in: Lecture Notes in Computer Science, Springer-Verlag, 1998, pp. 223–230.
[10] D.R. Musser, A.A. Stepanov, Generic programming, in: Proceedings of the International Symposium on Symbolic and Algebraic Computation, ISSAC 1988, in: Lecture Notes in Computer Science, vol. 358, Springer-Verlag, 1989, pp. 13–25.
[11] D.R. Musser, A.A. Stepanov, Algorithm-oriented generic libraries, Softw. - Pract. Exp. 24 (7) (1994) 623–642.
[12] J. Siek, L.-Q. Lee, A. Lumsdaine, The Boost Graph Library: User Guide and Reference Manual, Addison-Wesley, 2002.
[13] I. John, V.W. Reynders, J.C. Cummings, The POOMA framework, Comput. Phys. 12 (5) (1998) 453–459.
[14] R.C. Whaley, A. Petitet, J.J. Dongarra, Automated empirical optimization of software and the ATLAS project, Parallel Comput. 27 (1–2) (2001) 3–35.
[15] E.E. Kohlbecker, D.P. Friedman, M. Felleisen, B.F. Duba, Hygienic macro expansion, in: LISP and Functional Programming, 1986, pp. 151–161.
[16] D. de Rauglaudre, Camlp4 reference manual, January 2002. http://caml.inria.fr/camlp4/manual/.
[17] 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, in: Lecture Notes in Computer Science, vol. 3016, Springer, 2003, pp. 51–72.
[18] C. Calcagno, W. Taha, L. Huang, X. Leroy, Implementing multi-stage languages using asts, gensym, and reflection, in: F. Pfenning, Y. Smaragdakis (Eds.), GPCE, in: Lecture Notes in Computer Science, vol. 2830, Springer, 2003, pp. 57–76.
[19] MetaOCaml. http://www.metaocaml.org.
[20] W. Taha, T. Sheard, Multi-stage programming with explicit annotations, in: Proceedings of the Symposium on Partial Evaluation and Semantic-Based Program Manipulation, PEPM, ACM, Amsterdam, 1997, pp. 203–217.
[21] W. Taha, Multi-stage programming: Its theory and applications, Ph.D. Thesis, Oregon Graduate Institute of Science and Technology, 1999.
[22] W. Taha, A sound reduction semantics for untyped CBN multi-stage computation. Or, the theory of MetaML is non-trival, in: PEPM, 2000, pp. 34–43.
[23] M. Püschel, J.M.F. Moura, J. Johnson, D. Padua, M. Veloso, B.W. Singer, J. Xiong, F. Franchetti, A. Gačić, Y. Voronenko, K. Chen, R.W. Johnson, N. Rizzolo, SPIRAL: Code generation for DSP transforms, in: Program Generation, Optimization, and Adaptation, Proc. IEEE 93 (2) (special issue).
[24] Z. Chen, J. Dongarra, P. Luszczek, K. Rothe, Lapack for clusters project: An example of self adapting numerical software, in: Hawaii International Conference on System Sciences, HICSS-37.
[25] O. Kiselyov, K.N. Swadi, W. Taha, A methodology for generating verified combinatorial circuits, in: Proceedings of the Fourth ACM International Conference on Embedded Software, EMSOFT'04, ACM, 2004, pp. 249–258.
[26] J. Carette, O. Kiselyov, Multi-stage programming with functors and monads: Eliminating abstraction overhead from generic code, in: Glück, Lowry [53], pp. 256–274.
[27] K.N. Swadi, W. Taha, O. Kiselyov, E. Pasalic, A monadic approach for avoiding code duplication when staging memoized functions, in: J. Hatcliff, F. Tip (Eds.), PEPM, ACM, 2006, pp. 160–169.
[28] Source code. http://www.cas.mcmaster.ca/~carette/metamonads/.
[29] G.H. Golub, C.F. Van Loan, Matrix Computations, 3rd ed., Johns Hopkins University Press, Baltimore, MD, 1996.
[30] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, J. Irwin, Aspect-oriented programming, in: M. Akşit, S. Matsuoka (Eds.), in: Proceedings European Conference on Object-Oriented Programming, vol. 1241, Springer-Verlag, 1997, pp. 220–242.
[31] J. Irwin, J.-M. Loingtier, J.R. Gilbert, G. Kiczales, J. Lamping, A. Mendhekar, T. Shpeisman, Aspect-oriented programming of sparse matrix code, in: Proceedings of the Scientific Computing in Object-Oriented Parallel Environments, ISCOPE'97, Springer-Verlag, London, UK, 1997, pp. 249–256.
[32] A. Mendhekar, G. Kiczales, J. Lamping, RG: A case-study for aspect-oriented programming, Tech. Rep. SPL97-009 P9710044, Xerox PARC, Palo Alto, CA, USA, February 1997.
[33] D.L. Parnas, On the criteria to be used in decomposing systems into modules, Commun. ACM 15 (12) (1972) 1053–1058.
[34] D.L. Parnas, On the design and development of program families, IEEE Trans. Softw. Eng. 2 (1) (1976) 1–9.
[35] E.W. Dijkstra, On the role of scientific thought, published as [54]. http://www.cs.utexas.edu/users/EWD/ewd04xx/EWD447.PDF.
[36] W. Zhou, J. Carette, D.J. Jeffrey, M.B. Monagan, Hierarchical representations with signatures for large expression management, in: Proceedings of Artificial Intelligence and Symbolic Computation, in: Lecture Notes in Computer Science, vol. 4120, 2006, pp. 254–268.
[37] A. Bondorf, Improving binding times without explicit CPS-conversion, in: 1992 ACM Conference on Lisp and Functional Programming, San Francisco, CA, 1992, pp. 1–10.
[38] A. Bondorf, D. Dussart, Improving CPS-based partial evaluation: Writing cogen by hand, in: ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, Orlando, FL, 1994, pp. 1–9.
[39] S. Peyton Jones, et al., The Revised Haskell 98 Report, Cambridge Univ. Press, 2003. Also on http://haskell.org/.
[40] E. Moggi, Notions of computation and monads, Inform. Comp. 93 (1) (1991) 55–92.
[41] A. Filinski, Representing monads, in: POPL, 1994, pp. 446–457.
[42] S. Liang, P. Hudak, M. Jones, Monad transformers and modular interpreters, in: Conference Record of the Annual ACM Symposium on Principles of Programming Languages, POPL'95, ACM Press, New York, 1995, pp. 333–343.
[43] W. Taha, M.F. Nielsen, Environment classifiers, in: Conference Record of the Annual ACM Symposium on Principles of Programming Languages, POPL'03, ACM Press, New York, 2003, pp. 26–37.
[44] MLton team, Property list. http://mlton.org/PropertyList.
[45] R.L. Burden, J.D. Faires, Numerical analysis, 4th ed., PWS Publishing Co., Boston, MA, USA, 1989.
[46] E.H. Bareiss, Sylvester's identity and multistep Gaussian Elimination, Math. Comput. 22 (103) (1968) 565–579.
[47] J. Eckhardt, R. Kaiabachev, E. Pasalic, K.N. Swadi, W. Taha, Implicitly heterogeneous multi-stage programming, in: Glück, Lowry [53], pp. 275–292.
[48] S.M. Watt, Aldor, in: J. Grabmeier, E. Kaltofen, V. Weispfennig (Eds.), Computer Algebra Handbook: Foundations, Applications, Systems, Springer Verlag, 2003, pp. 265–270.
[49] R. Glück, R. Nakashige, R. Zöchling, Binding-time analysis applied to mathematical algorithms, in: System Modelling and Optimization, Chapman and Hall, 1995, pp. 137–146.
[50] R. Glück, J. Jørgensen, An automatic program generator for multi-level specialization, LISP Symb. Comput. 10 (2) (1997) 113–158.
[51] P. Hudak, Building domain-specific embedded languages, ACM Comput. Surv. 28 (4es) (1996) 196.
[52] M. Mernik, J. Heering, A.M. Sloane, When and how to develop domain-specific languages, ACM Comput. Surv. 37 (4) (2005) 316–344.
[53] R. Glück, M.R. Lowry (Eds.), Generative Programming and Component Engineering, 4th Internation Conference, GPCE 2005, Tallinn, Estonia, September 29–October 1, 2005, Proceedings, in: Lecture Notes in Computer Science, vol. 3676, Springer, 2005.
[54] E.W. Dijkstra, On the role of scientific thought, in: Selected Writings on Computing: A Personal Perspective, Springer-Verlag, 1982, pp. 60–66.

Please cite this article in press as: J. Carette, O. Kiselyov, Multi-stage programming with functors and monads: Eliminating abstraction overhead from generic code, Science of Computer Programming (2008), doi:10.1016/j.scico.2008.09.008