Partial Evaluation of Maple

Jacques Carette[*]
Department of Computing and Software
McMaster University
1280 Main Street West
Hamilton, Ontario L8S 4K1
Canada
carette@mcmaster.ca

Michael Kucera
mikekucera@gmail.com

ABSTRACT

Having been convinced of the potential benefits of partial evaluation, we wanted to conduct some experiments in our favourite Computer Algebra System, Maple. Maple is a large language, with a few non-standard features. When we tried to implement a partial evaluator for it, we ran into a number of difficulties for which we could find no solution in the literature. Undaunted, we persevered and ultimately implemented a working partial evaluator with which we were able to very successfully [10] conduct our experiments. In this paper, we document the techniques we had to either invent or adapt to achieve these results.

Categories and Subject Descriptors

F.3 [**Theory of Computation**]: Logics and Meaning of Programs; F.3.2 [**Logics and Meaning of Programs**]: Partial Evaluation

General Terms

Languages,Design,Theory

Keywords

Maple, symbolic computation, partial evaluation, residual theorems

1. INTRODUCTION

While symbolic computation is a mainstay of partial evaluation, partial evaluation is not a common technique in symbolic computation, even less so in computer algebra. The authors were convinced that partial evaluation (and metaprogramming in general [4, 6]), should be a very powerful tool when combined with a Computer Algebra System. Eventually, our high hopes were proven correct, as we were able to report at Calculemus 2006 [10]. But what that paper does not really say is how difficult this turned out to be. The basic theory and practice of partial evaluation is lucidly explained in [9], and various papers [7, 16, 17, 18, 20] provided welcome additional techniques. And yet, we needed to invent a number of techniques to be able to handle interesting Maple programs. Here our goal is mainly to explore those techniques that were needed to make an effective partial evaluator for Maple, our Computer Algebra System (CAS) of choice. Note that since doing any kind of static analysis of Maple programs seems outlandishly difficult [5], we had no choice but to write *online* partial evaluator.

First, a few words on what motivated us to write this partial evaluator for Maple. This can be summed up by the slogans "efficient genericity" and "residual theorems".

Generic programming is not a new idea in computer algebra, where it was used long before its current resurgence in the C++ and functional programming communities, as reading Musser and Stepanov's classic paper [12] attests. But in a dynamically typed, interpreted language (such as Maple), the interpretation overhead of such abstractions is so prohibitive that otherwise successful projects in generic programming [8] did not become standard practice. We wanted to keep that programming style, but without the efficiency cost. While [4] shows that typed metaprogramming (in MetaOCaml) can deal with this, we wanted to accomplish the same in Maple. More importantly, we wanted to have a pleasant programming experience while writing generic programs, which is (currently) not the case for C++ template programming nor, unfortunately, for MetaOCaml programming. It is important to note that we are not using partial evaluation as a method to get our programs to run *faster*, but rather to be able to conveniently write generic programs that run *no slower* than previous code. We believe this is an important shift in perspective that should increase the areas of applicability of partial evaluation and program transformation.

"Residual theorems" is the term we coined to refer to expressing the result of symbolic computations on symbolic input as programs with (potentially many) *residual* conditions on the validity of the result. The underlying motivation is that there is a huge amount of information embedded in Computer Algebra libraries, a lot of which encodes special cases for the validity of computations in analysis. However, these special cases are only triggered when the coefficients of the problems at hand are exact constants, and not when they are parametric. As we show in [10], it is not necessary to invent parametric algorithms to deal with this, as the current algorithms and partial evaluation are sufficient.

We assume that the average PEPM attendee is not very familiar with CASes, and even if they have some knowledge of them, it is probably restricted to their use as a glorified calculator rather than as a full-fledged programming language. As such, we give a programming language oriented introduction to Maple in the next section. We focus on those areas of the language which are not standard (i.e. for which clear similarities cannot be found as well-

[*]Supported in part by NSERC grant RPG262084-03.

known features in any of Scheme, Java, C, Ocaml or Haskell). One particuliarity of CASes is that they are designed to deal with *open terms* (which they simply call expressions) as a fundamental data type. What this means is that in Maple, the over-used *power* example is doubly irrelevant: First, because the powering operator is built-in, and second because

```
stagebin := proc(n::posint,f)
  local res, g, x;
  g := proc(x, n) local y;
    if n=0 then 1 elif n=1 then x
    elif n mod 2 = 0 then
      y := g(x, n/2); f(y,y);
    else f(x,g(x,n-1)); end if;
  end proc;
  unapply( g(n,z), z);
end;
```

is a routine which given a positive integer n and a binary associative multiplication-like operator f will return a new procedure that computes f applied n times via binary splitting. The "trick" is to manipulate open terms directly and use `unapply` to get back a procedure. And yet it is hard to fool Maple, as stagebin (5, ‘∗‘) will simply return z −> z^5, as will stagebin (5, **proc**(a,b) a ∗ b **end**). To obtain the desired result it is necessary to resort to using an *inert* multiplication; stagebin (5, ‘&∗‘) will then return the more familiar `proc(z) ‘&*‘(z,‘&*‘(‘&*‘(z,z),‘&*‘(z,z))) end proc`, without doing any real metaprogramming or program transformation.

We are not aware of any previous work on trying to do partial evaluation of a Computer Algebra language. The closest work in this area is on the purely numerical language Matlab, where [7] also reports having to work rather hard to get their results. Of course all the work on partial evaluation for (full) Scheme (like [19]) is quite relevant, as Maple is also a higher-order functional/imperative language with good reification and reflection capabilities.

The main contributions of this paper are: 1) several new techniques in *online* partial evaluation and 2) the demonstration that partial evaluation is an effective tool when applied to a Computer Algebra language. We have named our partial evaluator MapleMIX. The need to support a non-trivial language (62 AST types) led to MapleMIX being divided into several distinct modules with well defined boundaries. The techniques that we developed are the result of the modularization requirement and by the need to support common Maple language features.

Our approach to modularizing the partial evaluator is to have certain modules communicate via a powerful abstract syntax that was designed specifically with partial evaluation in mind. We have discovered that adding new constructs to the language representation, instead of just simplifying the input language, can actually make the specialization module more compact. (Maple has one syntactic form for assignment whereas our intermidate representation has four.) We have even taken the idea of syntax-directed partial evaluation to the next level by having the specializer perform on-the-fly syntax transformations that further drive the specialization process. These on-the-fly transformations are very effective for handling dynamic conditionals while performing static loop unrolling.

We have developed several techniques within the paradigm of online partial evaluation that show the power and accuracy that the online approach is capable of. In particular, our design for the variable binding environment works well with a new algorithm for handling if-statements. Furthermore the environment allows for a completely online syntax-directed approach to handling partially static data structures.

First we describe the context of our work by providing a description of the Maple language in Section 2. An overview of MapleMIX is given in Section 3, and Section 4 goes into detail about the various designs and techniques used in the implementation. A short demonstration of our results is given in Section 5.

2. MAPLE

Although Maple is principally used as a mathematical assistant, in other words as an interactive calculator that allows one to do both routine computations (albeit at a high level of mathematical sophistication) and as an aid to mathematical exploration, at its heart still lies a sophisticated programming language [11].

That programming language is a mixed imperative/functional dynamically typed, eager evaluation language, with higher order functions, first-class modules, first-class (dynamic) types, proper closures and lexical scoping, error handling primitives, arbitrary precision arithmetic (integer, rational and floating point), and a full IEEE-754 compliant implementation of hardware floating point arithmetic. Its fundamental structured data types are the array, the set, the "expression sequence" and the hash table; and since Maple 6, the so-called "rectangular table". It also has extensive I/O libraries, a solid Foreign Function Interface (which includes not just C but also Java, and Fortran), fancy parameter processing primitives (somewhat akin to Python's) and, naturally, a very extensive library of mathematical types and operations.

In Figure 1 is a grammar describing Maple's abstract syntax (**not** its concrete syntax, but influenced by it). Most of it is quite straightforward, so we will outline only those non-standard features. We denote by c the literal constants, e the expressions and s the statements; n is an auxilliary production which denotes expression sequences of specific terms (identifiers or type-decorated identifiers in this case). We use the $+(e)$ to denote an n-ary operator (in this case $+$, $*$, function application and the expression sequence constructor ,). $\mathbb{F}$ denotes the floating point numbers, and `string` denotes string literals delimited by ". The single biggest difference between Maple and other languages is that some of its fundamental operations (like $+$ and $*$) can return *unevalated.* In other words, while $1 + 3$ naturally evaluates to 4, $x + 5 + y + 3$ evaluates to $x+y+8$ if x and y are *symbols*. In Maple, *symbols* are simply identifiers with no assigned value! The language even has a construct for this, called *uneval quotes*; for example while $\sin(\pi/2)$ evaluates to 1, $'\sin'(\pi/2)$ evaluates to the expression $\sin(\pi/2)$ (which, if further evaluated, will give 1). These first-class expressions really are models of *open terms*, which few languages possess.

Figure 2 gives a few of the unusual rules for the operational semantics for Maple. The first two rules say that unassigned identifiers are fine in Maple, they stand for *themselves*. The next 3 rules are the usual ones for $+$, but the last one on the first line says that adding an identifier to anything will simply return an unevaluated $+$. We then have a rule saying that the unevalation quotes do just that, they prevent further evaluation. The next rule expresses that $=$ is just a data-constructor, so that when a boolean is needed (for example by `if`), an implicit call to the built-in function `evalb` is performed, and this function will either return a boolean or throw an exception.

On top of these unusual aspects to the semantics, many of the built-in functions (there are 217 in Maple 10) have unusual semantics

$$
\begin{array}{lll}
c & ::= & \mathbb{Z} \mid \mathbb{Q} \mid \mathbb{F} \mid \text{string} \mid \text{identifier} \\
e & ::= & c \mid +(e) \mid *(e) \mid -e \mid e^{e} \mid e \wedge e \mid e \vee e \mid \neg e \mid e \,\mathtt{xor}\, e \mid e[e] \mid \ 'e' \mid \ `e` \mid e :: e \mid e = e \mid e \neq e \mid e < e \mid e \leq e \mid \{e\} \mid [e] \mid \\
 & & e..e \mid \ e||e \ \mid e(e) \mid e\mathtt{:-}e \mid \mathtt{args} \mid \mathtt{nargs} \mid \ \mathtt{hashtab}(e) \mid \ ,(e) \\
n & ::= & ,(\mathtt{identifier}) \ \mid \ ,(\mathtt{identifier} :: e) \\
s & ::= & e \mid s; s \mid e\mathtt{:=}e \ \mid \mathtt{for}\, e\, \mathtt{from}\, e\, \mathtt{to}\, e\, \mathtt{by}\, e\, \mathtt{while}\, e\, \mathtt{do}\, s \mid \mathtt{for}\, e\, \mathtt{in}\, ee\, \mathtt{do}\, s \mid \mathtt{try}\, s\, (\mathtt{catch}\, e\mathtt{:}\ s)^{*} \mathtt{finally}\, s \mid \mathtt{break} \mid \\
 & & \mathtt{next} \mid \mathtt{error}\, e \ \mid \mathtt{return}\, e \mid \mathtt{if}\, r\, \mathtt{else}\, s\, (\mathtt{elif}\, e\, \mathtt{then}\, s\,)^{*} (\mathtt{else}\, d)? \\
 & & \mathtt{proc}(n) \mathtt{local}\, n\, \mathtt{global}\, n\, \mathtt{description}\, e\, \mathtt{option}\, e\, \mathtt{returntype}\, e\, ;\ s \\
 & & \mathtt{module}() \mathtt{local}\, n\, \mathtt{global}\, n\, \mathtt{export}\, n\, \mathtt{description}\, e\, ;\ s
\end{array}
$$

Figure 1: Simplified Maple Abstract Grammar

$$
\frac{x \in \sigma}{\sigma x} \quad \frac{x \notin \sigma}{x} \quad \frac{e_1 \Rightarrow e_3 \quad e_2 \Rightarrow e_4}{e_1 + e_2 \Rightarrow e_3 + e_4} \quad \frac{}{i_1 + i_2 \Rightarrow i_1 + i_2} i_1, i_2 \in \mathbb{Z} \quad \frac{}{z_1 + z_2 \Rightarrow z_1 + z_2} z_1, z_2 \in \mathbb{F} \quad \frac{e_1 \Rightarrow e_2}{x + e_1 \Rightarrow x + e_2} x \in \mathtt{identifier}
$$

$$
\frac{}{'e' \Rightarrow^{*} e} \quad \frac{e_1 \Rightarrow e_3 \quad e_2 \Rightarrow e_4}{e_1 = e_2 \Rightarrow e_3 = e_4} \quad \frac{\mathtt{evalb}(b) \Rightarrow \mathtt{true}}{\mathtt{if}\ b\ \mathtt{then}\ s_1\ \mathtt{else}\ s_2 \Rightarrow s_1}
$$

Figure 2: Operational Semantics Fragment, expressions

as well. For example the built-in `assigned` is call-by-name even though Maple is generally call-by-value (this function tests if an identifier is assigned), the function `op` is a polymorphic deconstructor which works over any value, `map` is polymorphic over all values, `unapply` will take an expression and will abstract out identifiers and return a procedure, `subs` will perform pure syntactic substitution even if that implies name-capture, etc.

3. A PARTIAL EVALUATOR FOR MAPLE

This section presents an overview of MapleMIX. Abstract syntax plays a major role in the design of our partial evaluator. The system is structured as a specific sequence of program transformations, with a special emphasis on special transformations occuring before and after the specialization phase, inspired in part by the design of some compilers [2]. We believe this approach leads to a highly modular design for practical online partial evaluators of complex languages. We often think of MapleMIX as an interpreter that has the additional functionality of generating residual code for deferred computations. MapleMIX contains an expression reducer that was influenced by the *cogen* approach [18] to partial evaluation. Furthermore we have implemented a novel online approach to handling partially static data-structures such as lists and polynomials.

3.1 Characteristics

MapleMIX is a fully online[1], syntax-directed, function-point polyvariant partial evaluator, itself written in Maple. It was designed with the goal to exploit as much static information as possible in order to achieve good specialization. Maple has very good run-time reification and reflection functions `FromInert` and `ToInert`. We rely on access to the interpreter to ensure adherence to the operational semantics. Maple's automatic simplification feature, instead of hindering us[2], sometimes helps to slightly clean up residual code.

As we are more concerned with with manipulating Maple code than with producing generating extensions, MapleMIX is explicitly not self-applicable. Thus we focus on offering the largest amount of features and supporting the largest subset of Maple as possible. This is made much easier by not placing restrictions on what language features may be used when writing the partial evaluator. Also, since we are not trying to produce generating extensions, we take the standard approach to partial evaluation (in other words mirroring an interpreter); we believe this has contributed to our PE being easier to refine and extend. Having said that, the expression reducer was definitely inspired by an online cogen approach.

[1]No pre-analysis is performed

[2]The computer algebra literature is replete with examples of so-called premature simplification which lead to incorrect results [15]

3.2 Input and Output

Traditionally the input to a partial evaluator is a complete program. In contrast, MapleMIX has access to the source of the full Maple library, as well as any other definitions in the current session. The specialization process must therefore be initiated in a controlled manner. Input to MapleMIX is a single function, called the *goal function*, which will be treated as the starting point of specialization. The parameter list of the goal function will be the dynamic inputs to the resulting specialized program. MapleMIX may generate several residual functions, which are packaged together with the specialized goal function and returned as a Maple module. The specialized goal function will become the main entry point of the returned module. Other than preparing a goal function, there is no need to perform any annotations, language transformations, etc, MapleMIX works on normal Maple programs.

3.3 M-form

The Maple reification function `ToInert` will return the abstract syntax tree of any Maple term, referred to as its *inert form*, which essentially corresponds to the abstract grammar of Figure 1. As the AST produced by `ToInert` was not designed with partial evaluation in mind, we use syntactic transformations to *M-form*, designed to be much more convenient for specialization. Traditionally many existing partial evaluators first transform their input into a simpler core language (for example C-mix to Core C [1]). This approach reduces the syntactic forms that the specializer must support, at the cost of losing certain invariants inherent with certain syntactic forms. For example some languages (like Fortran 77) have for loops that are guaranteed to terminate.

M-form both simplifies and adds to the inert form. Adding syntactic forms does not make the specializer more complex (or longer) but in fact makes it more compact. This seems to be because the removal of certain redundant syntactic forms may actually add complexity since the specializer will have to infer the information that was removed by such a "simplifying" transformation. In the end, while the inert form has 62 cases, M-form has 74.

Since MapleMIX is syntax-directed it is natural to use syntax transformations and new syntactic forms to direct the specializer. Some syntactic constructs in M-form are in fact only introduced by the specializer, which performs on-the-fly insertion of these constructs to proceed (see section 4.6). The design goals of the M-form is to keep all static information available in the inert form intact, while keeping the translation between these forms straightforward, and to help with specialization. Below, we detail the main differences between inert form and M-form.

3.3.1 Assignments

While Maple is an imperative language with global state and side-effecting expressions, statements cannot occur in an expression context, and thus the only expression which might create side-effects is a function call. In order to separate the concerns of expression reduction and environment update, M-form adds the stipulation that all expressions must be side-effect free.

The M-form translator maintains a list of known intrinsic functions. An intrinsic function will never be specialized, instead any call to an intrinsic function will be treated as an atomic operation that may be performed at partial evaluation time. Most built-in functions are considered intrinsic except side-effecting I/O functions. Some library functions are also considered intrinsic in order to simplify residual code. All non-intrinsic function calls are removed from expressions by generating a *new* assignment statement for each call and then replacing the original calls by the names generated[3]. We call this a *splitting* transformation:

Original Code	Transformed Code

`a := f(g(x)) + h(x);`	`m1 :=' g(x);`
	`m2 :=' f(m1);`
	`m3 :=' h(x);`
	`a := m2 + m3;`

A new syntactic form of assignment is generated by this transformation, `:='`, specifically to represent assignment of a function call to a variable. This way the specializer can syntactically decide to do a simple reduction, or perform specialization an entire function body. The splitting transformation has the unfortunate effect of possibly creating many new assignment statements. However, if the specializer decides not to unfold a split function call, or if the function unfolds into a single assignment statement, then we know that the new variable is only assigned to once and only used once. When translating back to Maple, such expressions will always be inlined.

Maple allows expressions to be used in statement context, often used in conjunction with Maple's implicit return mechanism. However we do not want the statement specializer to have to account for every expression form. The solution is to tag standalone expressions and standalone function calls (so that the tag implicitly becomes a new statement form).

3.3.2 If Statements

If statements in Maple may have arbitrarily many `elif` blocks and an optional `else` block. M-form has a simpler *MIfThenElse* construct that always consists solely of a conditional expression and two branches. Any Maple if statement with a list of `elif` blocks is converted into nested *MIfThenElse* statements. Empty else blocks are added as necessary. This transformation works hand-in-hand with the splitting transformation in order to correctly maintain the ordering of function calls in conditional expressions:

Original Code	Transformed Code

if f(x) **then**
  S1
elif g(x) **then**
  S2
end if
	m1 := f(x);
	if m1 **then**
	  S1
	else
	  m2 := g(x);
	  if m2 **then**
	    S2
	  else
	  end if
	end if

3.3.3 Loops

Inert form has two kinds of loop, both variations of for loops. There is the common *for* loop **for** i **from** 1 **to** 10 **do** ... **end do** as well a the **for** e **in** [1,2,3] **do** ... **end do** loop that accesses all elements of a linear data structure such as a list or set, commonly called a *foreach* loop. Both loops have an optional while clause, which is checked on the start of each iteration, causing the loop to exit if the expression is false. Most parts of a loop definition have defaults and can be omited in concrete syntax, but are always present in the AST. For example, a while loop is actually a for-from loop with all default clauses except the while clause.

For loops and while loops contain different static information. Unlike a while loop, a proper for loop where all write access to the loop index variable is controlled by the loop statement itself will not (by itself) be a source of non-termination. This is crucial if the partial evaluator is to reliably unroll loops without risking non-termination. While it is possible to have the specializer check this dynamically, it is simpler to transfer the burden to the M-form translator. Therefore in M-form we support three types of loops instead of two[4]. These are the general while loop, a for-from loop with an optional while condition, and the for-in loop with optional while condition. Note that the for-from loop can be unrolled, but the while condition (if present) must be checked on each iteration; if it evaluates to `false`, unrolling is stopped. Any assignments that are generated by splitting function calls out of a while condition expression are inserted both before the loop and in the body of the loop at the bottom.

Currently MapleMIX does not support the use of `next` or `break` inside a loop. If one is encountered during translation to M-form an exception is thrown. There is however one case where a simple transformation can remove the use of `next`; we illustrate both these ideas here:

Original Code	Transformed Code

while f(x) **do**
  if C **then**
      next
  end if;
  S1;
end do;
	m1 := f(x);
	while m1 **do**
	  if not C **then**
	    S1;
	  end if;
	  m1 := f(x);
	end do;

3.3.4 Other Syntactic Forms

There are many other lesser transformations that are performed when converting from inert form to M-form. For example, the abstract syntax for function parameter lists can become quite convoluted in inert form. In M-form it has been cleaned up significantly for the sole purpose of making it easier to deal with this construct in

[3]this is essentially *let* insertion for an imperative language

[4]Assignment to the loop index variable is currently not supported.

the specializer. This is an example of a simplifying transformation. Other transformations have to do with tables, which are a built-in Maple datatype with language support. For example, Maple allows the creation and initialization of a table at the same time using the built-in `table` function. Dynamic uses of this particular function are transformed into a series of table index assignment statements. This relieves the specializer from having to deal with the `table` function as a special case. Some of the resulting assignments may be static and some may be dynamic at specialization time and will be treated accordingly.

3.4 Further ideas.

The core module of the specializer has the task of deciding when to share already specialized functions (via a simple hashing technique), as well as deciding when a specialized function should be unfolded. It also remembers previously generated residual code for later reuse. The expression reducer, given an M-form of an expression, will reduce it as far as possible using the static information provided by the environment. It may return a static value or a dynamic M-form. Its implementation is inspired by the online cogen approach. There is a module just for the implementation of the function unfolding transformation, which can become complex in certain contexts, for example when target function's body contains many return statements.

4. TECHNIQUES

In this section, we deal with the techniques we used in implementing MapleMIX. If we used a standard technique, this is either mentioned quickly or sometimes that aspect is not covered at all. We concentrate instead on what we perceive to be either novel techniques or interesting variations on older techniques.

4.1 Expression Reduction

The expression reducer serves the role of evaluating expressions as far as possible given the available information stored in the environment. The reducer supports operations on most Maple data types from simple numbers and strings to lists, polynomials, higher-order functions, arrays and tables. The implementation of the reducer is inspired by an online cogen approach to PE as outlined by Sumii and Kobayashi [16]. The idea is to replace the underlying operators of the language with smarter ones that correctly handle dynamic arguments. They first proposed this idea as a solution to the limitations of type-directed partial evaluation, here we use the essence of the idea in a syntax-directed online setting. A reduction function is created for each pure Maple operator which works as follows: if all arguments are static then apply the underlying Maple operator on the arguments, essentially handing control over to the Maple interpreter to perform the actual static operation; otherwise build a dynamic expression and return it. Reduction of static expressions is thus guaranteed to be identical to the already existing semantics of Maple expressions.

4.2 Online Approach to Partially Static Data

In the context of Computer Algebra, it is very common to have partially static data. For a program specializer to produce good results it must use as much static information as possible. In many situations, while the exact value is not known, type information and/or the "shape" of the value might still be statically known. For example a list may have dynamic elements, however its length might be static. Avoiding unnecessary approximations is key to preserving static information [14]. Our approach to supporting partially static data is to take the idea of "smart operators" a step further, by extending certain intrinsic functions with the additional ability to properly handle dynamic terms. Take for example the list `[a, b, 2]` where `a` and `b` are dynamic; clearly the length of this list[5] does not depend on the values (or types) of `a` and `b`. We can determine the length of the list at reduction time by examining the structure of the M-form and counting the number of "holes" for data. In particular, the built-in Maple `nops` function can be used to return the number of elements (operands) in a list. We have extended `nops` with the ability to return a static result in the case where it is given a partially static list as a dynamic input. This approach generalizes, and we thus exploit the static information present within the dynamic representation. Several of Maple's intrinsic functions have been extended in this way to add support for partially static lists and polynomials. Syntactic constructs such as indexing and list concatenation have also been extended in a similar way. This is a tedious task, and more research is needed to better understand what is involved.

In order to propagate dynamic terms through the program they are stored in the environment alongside static values. When the reducer encounters a variable, it retrieves its representation from the environment, which may store a static value, a dynamic representation or not have a binding at all. If the variable is bound to a dynamic representation then it is substituted. Special care must be taken not to introduce duplicate computations in this way. A special syntactic form `MSubst` is introduced by the reducer to track such substitutions, consisting of the variable name and the dynamic representation retrieved from the environment, basically representing a *let* insertion. If the dynamic expression is not consumed during further reduction then the entire `MSubst` will be output by the reducer. Later, when the M-form representation of the residual program is being transformed back to inert form, the dynamic representation part of the `MSubst` will be discarded and the name used instead.

Support for partially static terms has been explored mostly within the context of offline PE. One approach is to use a binding time analysis (BTA) to determine the binding times of individual elements of a partially static data structure [9]. Another approach uses an abstract interpretation as a shape analysis to gather static shape information as a pre-phase [7]. Our approach is completely online and has the potential to exploit the full information available during specialization. However it must be noted that quite a bit of custom support for various dynamic representations must be added to the reducer in order to achieve this.

All function calls within the expression must be to functions that are considered intrinsic. These are pure functions that the specializer will treat as atomic in the sense that it will never try to specialize them. If a call to an intrinsic function has all arguments static then the function will be applied at partial evaluation time. Since any side-effects will go unnoticed it is essential that the function be side-effect free. Most built-in functions are pure except for some I/O functions such as `print` and `read`. These will not be considered intrinsic but will still be detected as built-in and so are treated as a special case by the specializer. I/O functions will be split out of expressions and always be residualized. Many non-built-in functions can also be treated as intrinsic such as `curry`, which performs partial application. Some library functions have non-standard semantics such as `seq` (the function for sequence comprehensions), which are also treated as special cases by the re-

[5]The similarity with parametric polymorphism is not accidental!

ducer[6]. The reducer contains a table of handlers for these "special" functions. For example, all calls to the *eval* family of functions (`eval`, `evalb` and `evalf`) are always residualized.

4.3 Closures

MapleMIX supports the use of static closures in the subject program. The requirement for a closure is that its surrounding lexical environment must contain a static value for any lexical local that is encountered during the evaluation of the function body. Or put more simply, when a lexical local is encountered its value must be static. This means that the function's closure can have dynamic parts as long as they are never accessed. This may occur if a dynamic lexical local is in a branch of a static conditional that is never evaluated. A lexical local may become dynamic at one point and then acquire a static value again later; as long as it is not accessed while dynamic, our restriction is not violated.

Maple is a dynamically typed language and as a result existing Maple code contains a great deal of dynamic type tests (usually as part of error checking code). Therefore, to be semantically correct, it is necessary that the reducer not change the type of any static term. This poses a challenge for handling function closures, as they must be represented as an active Maple function in order to be consistent in our treatment of static values. However the values of the function's lexical locals are stored in the partial evaluator's environment. We solve this problem by performing a simple transformation on the body of the function. Each lexical local is replaced by an application of an inline function. These "thunks" will call back into the partial evaluator to retrieve the variable's value, and will throw an exception if there is no static value available. The M-form of the closure is then converted into active Maple code. This way a static function closure can be applied when needed, a feature essential for supporting higher-order built-in functions such as `map` and `fold`. If applied to some dynamic arguments it will be converted back into M-form and specialized. This converting of code to and from active Maple is inefficient but unavoidable with this scheme.

4.4 Side Effects and Termination

Pure functional languages are characterized by *referential transparency*, meaning that multiple calls to a function with the same arguments will always produce the same result. This property allows a specialization strategy where the partial evaluator does not have to be concerned with the order of specialization of function points [9]. The presence of side-effects and global state puts a restriction on the specialization strategy. The ordering of statement execution must be respected during specialization and be preserved in the residual code [1]. The result is a depth-first specialization strategy where every time a function call is encountered it must be specialized immediately. Because of nesting, there may be several functions in the process of specialization at the same time.

MapleMIX uses a simple function sharing scheme for two purposes: to reuse specialized functions in cases where multiple calls to the same function with the same static arguments are encountered, and to avoid termination problems inherent with recursive procedures. When a function call is encountered its *call signature* is computed. It will consist of values of static arguments and placeholders for dynamic ones. If the call signature has not been encountered before then the function is specialized. The call signature is then saved along with the specialized code. The next time the same call signature is encountered the specialized code is simply retrieved and reused.

This strategy also improves termination properties of the partial evaluator as call signatures are used to help detect static recursion. The depth-first online specialization strategy makes it possible for several functions to be in the process of deferred specialization. If one of those functions is recursive (or mutually recursive) then the problem of infinite specialization arises. The partial evaluator can tell when a call signature refers to a function that is currently in the process of being specialized. When such *static recursion* is detected a call to the recursive function is simply residualized. This strategy relies on detection of identical call signatures, thus if some static value is changing under dynamic control, infinite specialization is still likely [9].

4.5 If Statements and the Online Environment

Partial evaluation of an `if` statement is done by fist reducing the conditional expression. If it statically reduces to a boolean value then the appropriate branch is simply fed to the statement sequence specializer. The much more interesting case is when the conditional reduces to a dynamic expression. The partial evaluator does not know which branch to follow, so it must follow *both*.

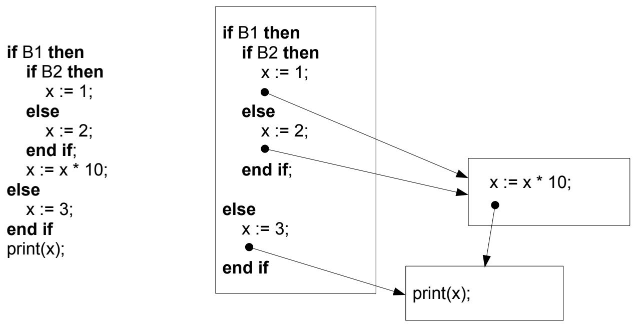
Handling of `if` statements is very different than handling if expressions in partial evaluation of expression oriented languages. There are two main challenges: First, each branch must be able to mutate the environment independently leading to the creation of two likely different environments, and second, code that is below the `if` statement must be handled correctly. The first problem can be handled by copying the environment [1, 7]. However, for efficiency reasons we do not wish to create two environments by copying (all or part of) the initial environment. We also wish to have a solution that scales to handling nested `if` statements in a straightforward manner. Furthermore, code that comes after an `if` statement may have to be specialized with respect to two different environments. We have implemented the online environment specifically with these two challenges in mind.

Our online environment is a stack of variable bindings. We shall call each element of this stack a *setting*. The stack will grow with each branch of a dynamic conditional. Any modifications to the environment are recorded in the topmost setting. An environment lookup initiates a linear search for the binding starting with the topmost setting and working downwards. Thus a binding in a setting will override any bindings of the same name in settings below it. Each setting maintains a dynamic mask to represent static variables that become dynamic. After the first branch of a dynamic conditional has modified the environment it can be trivially restored to its previous state via a simple pop.

Specialization of a dynamic `if` statement requires that all the code that could execute after the `if` statement be specialized with respect to each branch. In order to facilitate this, M-form is further translated before specialization into a DAG (Directed Acyclic Graph) representation. This is especially easy in Maple as the internal representation for expressions is as DAGs [11]. A pointer is added to the bottom of each branch that will point to the code that comes below the `if` statement. The code that comes below is then removed from its original location. This transformation is then performed recursively on each branch. The result is a DAG repre-

[6]Maple is a language that has "evolved" over 25 years, mostly by non-programming-language experts and, unsurprisingly, has many constructs which are "special".

sentation in which all code that can be executed after a branch of an `if` statement can be easily visited by simply following pointers, schematically represented in Figure 3.



```
if B1 then
  if B2 then
    x := 1;
  else
    x := 2;
  end if;
  x := x * 10;
else
  x := 3;
end if
print(x);
```

Figure 3: DAGform

4.5.1 *Specialization Algorithm*

The specialization algorithm for `if` statements, using the online environment, proceeds as follows: Specialization of the first branch begins by pushing a new empty setting onto the environment. All effects of the statements in the first branch are recorded in this new setting. Simply popping the stack restores the environment to the state it was in before specializing the first branch. A new empty setting is then pushed, and the second branch is specialized with respect to the initial environment. If we keep a copy of each of these newly popped settings, we can now easily compare the effects of each branch on the current state. In pseudo-Maple, if the input was **if** B **then** C1 **else** C2 **end if**; S, the algorithm is

Listing 1: Pseudo code for dynamic if

```
Br := reduce(B);
if type(Br, 'dynamic') then
  C1' := grow_and_specialize(C1);
  if bottom_reachable(C1') then
    S1 := grow_and_specialize(S);
    pop();
  end if;
  set1 := top(); pop();
  C2' := grow_and_specialize(C2);
  set2 := top();
  if set1=set2 or not bottom_reachable(C2') then
    pop(); case1
  else
    S1 := specialize(S);
    pop(); case2
  end if;
else
  # Br is static, reduce proper branch
end if;
```

In the above code, `case1` indicates that the residual code
if B' **then** C1' **else** C2' **end if**; S1
is produced, while `case2` corresponds to
if B' **then** C1'; S1 **else** C2'; S2 **end if**;.
The routine `bottom_reachable` ensures that there is no escaping control flow (like a `return` or an `error`), so that `S` is never unnecessarily specialized. Duplication of `S` is avoided in situations where execution of either branch would effect the environment in the same way. This is common with error checking code where the body of the `if` simply has an `error` statement. In the situation where each branch produces a different state we get two specialized versions of S, which results in a high level of polyvariance. The DAG form ensures that no code is specialized in an invalid environment. An example of the results of this algorithm (where `<dynamic>` stands for an arbitrary dynamic boolean expression) can be seen in Figure 4

Original Code	Specialized Code
if <dynamic> **then**	**if** <dynamic> **then**
if <dynamic> **then**	**if** <dynamic> **then**
x := 1;	print(1);
print(x);	print(10);
else	print(100)
x := 2;	**else**
end if;	print(20);
print(x*10);	print(200)
else	**end if**
x := 3;	**else**
end if;	print(300)
print(x*100);	**end if**

Figure 4: Example of `if` statement specialization

4.5.2 *Comparison with other methods*

A natural approach to specializing `if` statements is by merging environments [7]. The initial environment is duplicated by copying it, then each branch of the dynamic conditional is specialized. The two environments are then merged at the end in such a way that only commonalities between the two environments are preserved. For code like **if** <dynamic> **then** x := 1; **else** x := 2; **end if**, the two specialization environments will record different values for x. The merged environment would then store as much static data as possible such as type, shape or a set of values. We could store that x is a positive integer or that it may have a value from the set $\{1, 2\}$. This way certain expressions involving x may still be static such as `type(x, integer)` or `x < 5`, while others will be dynamic such as `x > 1`. This approach discards static data by making approximations, which may lead to an unsatisfactory level of specialization. Furthermore the merging process may be very complex, it requires copies of environments, and the reducer is more complex. Our approach does not make approximations and it never copies environments. However our approach may result in *overspecialization* in that the differences between the code specialized in each branch may be minimal.

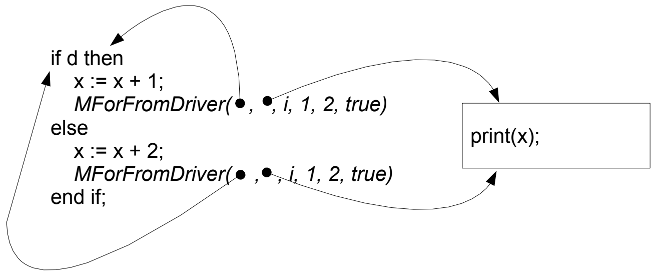
Offline methods perform a Binding Time Analysis, which is essentially a worst case analysis. Since it is safe to approximate everything as dynamic and very difficult to guarantee that a result will be static, any dynamic value tends to propagate through the program creating a snowball effect. The same is mostly true for online methods in a functional setting. However in an online imperative setting it is possible for a variable to change binding time! For example it is possible for a static variable to become dynamic due to assignment to a dynamic expression or assignment within a dynamic context. However with our online partial evaluator, it is possible for a dynamic variable to become static (however unlikely it may be to find code that does this). For example in
x := <dynamic>; ...; x := 5; the last assignment causes x to be bound to the static value 5 in the online environment, regardless of the fact that x was previously dynamic.

4.6 Static Loops

When all the control clauses of a for loop definition are static, the loop may be unrolled, since in Maple for loops are guaranteed to terminate whenever the body of the loop does not contain an assignment to the loop index variable. In that latter case, the entire loop would effectively become dynamic. As MapleMIX is online and we do not want to perform useless computation and then backtrack, this information needs to be detected syntactically, which is why assignments to the loop index is not supported by MapleMIX.

An interesting challenge arises when we consider the case of dynamic conditionals within a static loop. Our `if` statement specialization algorithm relies on the ability of the specializer to have access to the entire execution path that could occur after a dynamic `if` statement, so that this path may be specialized with respect to both branches. When a conditional is inside a loop then the execution path includes all of the subsequent iterations of the loop! A dynamic conditional will essentially cause the path of computations to split. The implementation of the online environment makes it easy and efficient for the specializer to explore every possible computation path.

Our solution to allow the computation path of a loop to split is to use a novel *on-the-fly syntax transformation* technique. When a static for loop is encountered it is removed and replaced with a set of *loop drivers*; in effect, we are replacing our loops by smart gotos! The loop drivers are placed at the end of each DAG path in the body of the loop. The loop index variable is then set to its initial value in the environment and the newly transformed loop body is given to the statement sequence specializer. For example, the Original Code in Figure 5, gets rewritten (in DAG M-form) as:



```
if d then
  x := x + 1;
  MForFromDriver(•, •, i, 1, 2, true)
else
  x := x + 2;
  MForFromDriver(• ,•, i, 1, 2, true)
end if;
```

```
print(x);
```

There are two forms of loop drivers, one for for-from loops and one for for-in loops; as they both work similarly, we will concentrate on the former. The `MForFromDriver` consists of 6 pieces of information, namely: a pointer to the top of the loop body, a pointer to the code that comes after the loop, the name of the loop index variable, the *by* value of the loop, the *to* value of the loop (i.e. the termination value), and the while condition. The operational semantics of this construct are straightforward. Note that the value of the loop variable is retained in the environment after the loop has been fully unrolled, as it is legal to refer to this variable in Maple after the loop has ended. If the loop bounds are such that the loop will never iterate, then the entire loop is eliminated. If a while condition exists, it is checked on each iteration; if it evaluates to false at any point then the unrolling is stopped.

The result is that the context of the loop is propagated into each computation path in the body of the loop. The computation path may continue to split as long as there are dynamic conditionals. The advantages to this approach are a high level of specialization and the lack of any need to merge environments. The main disadvantage is a possible exponential blowup in the size of the residual code. In our experiments we have not found this to be a problem, in fact we have found that this scheme works well in situations where a conditional is dynamic on some iterations and static on others. An example is iterating over a partially static list when the loop contains a conditional that depends on the binding time of the list elements. However, the code in Figure 5 would be of size $O(2^n)$ if the loop were from 1 to a static positive integer n.

4.7 Dynamic Loops

Dynamic loops pose a significant challenge to specialization. Since it is unknown how many times a dynamic loop will iterate, it must

Original Code	Specialized Code
`x := 1;`	**`if`** `<dynamic>` **`then`**
`for` `i` **`from`** `1` **`to`** `2` **`do`**	**`if`** `<dynamic>` **`then`**
`if` `<dynamic>` **`then`**	`print(3)`
`x := x + 1;`	**`else`**
`else`	`print(4)`
`x := x + 2;`	**`end if`**
`end if``;`	**`else`**
`end do``;`	**`if`** `d` **`then`**
`print(x);`	`print(4)`
	`else`
	`print(5)`
	`end if`
	`end if``;`

Figure 5: Example of dynamic conditional in a static loop

always be residualized. It would be unsound to partially evaluate the body of the loop with respect to the current environment. The problem is that the loop may contain a static assignment, however that assignment may be performed an unknown number of times, making the assignment dynamic.

Partial evaluators for other imperative languages take novel and complex approaches to analyzing dynamic loop bodies. For example the MATLAB partial evaluator performs an iterative data-flow analysis involving abstract interpretation [7]. MapleMIX takes a very conservative approach to specialization of dynamic loops. A simple syntactic analysis is done on the body of the loop in order to detect unsupported cases. However our approach is simple to implement and still works for many real-world situations. We will not describe this approach here, we leave it as future work to add sophisticated support for dynamic loops to MapleMIX.

4.8 Static Data and Lifting

Sometimes a static value must be embedded within a dynamic context, this process is known as *lifting*. Traditionally this is done by inserting a textual representation of the value within the residual program. This is easily achieved for simple types such as integers and strings but for more complex types lifting may be difficult or even not possible [7]. Structured types may be difficult to rebuild, and may not have a representation that can occur on one line.

Fortunately it is possible for MapleMIX to sidestep the problem of lifting static data in most situations. MapleMIX does not generate residual code as text, instead it generates an inert form representation that is converted by Maple itself directly into an active (executable) internal representation. Inert form provides a very handy construct `_Inert_VERBATIM` for embedding any Maple value within an inert representation.

All static data is represented in M-form by wrapping it in an *MStatic* constructor. The `FromM` translator will translate *MStatic* directly to `_Inert_VERBATIM`, making the embedding of static data in the residual program an extremely simple operation. Complex types such as static tables are simply embedded directly into the residual program. Allowing a certain flexibility in the output language can often be a convenient way to solve challenging problems in partial evaluation.

5. RESULTS

There are two approaches when attempting to write a program that solves a family of computational problems; write a family of specific subprograms for each problem, or write one generic program that solves all the problems. The generic program is often easier to write, maintain and extend. However it will not be as efficient as the specialized programs. Listing 2 presents an example of a parameterized in-place quicksort algorithm. Two design decisions have been abstracted as functional parameters: the choice of pivot, which effects the complexity properties of the algorithm, and the choice of comparison function.

Listing 2: In-place QuickSort

```
swap := proc(A, x, y) local temp;
  temp := A[x]; A[x] := A[y]; A[y] := temp;
end proc:
quicksort := proc(A, m, n, piv, comp) local p;
  if m < n then
    p := partition(A, m, n, piv, comp);
    quicksort(A, m, p-1, piv, comp);
    quicksort(A, p+1, n, piv, comp);
  end if;
end proc:
partition := proc(A, m, n, pivot, compare)
  local pivotIndex, pivotValue,
        storeIndex, i, temp;
  pivotIndex := pivot(A, m, n);
  pivotValue := A[pivotIndex];
  swap(A, pivotIndex, n);
  storeIndex := m;
  for i from m to n-1 do
    if compare(A[i], pivotValue) then
      swap(A, storeIndex, i);
      storeIndex := storeIndex + 1;
    end if;
  end do;
  swap(A, n, storeIndex);
  return storeIndex;
end proc:
```

Function qs1 which calls the quicksort function with static parameters for the pivot and compare functions. The given pivot function will return the index of the last element of the section of the array that is being sorted. Maple's own built-in <= function is used as the compare function.

```
qs1 := proc(A, m, n) local p, c;
  p := (A, m, n) -> n; c := `<=`;
  quicksort(A, m, n, p, c)
end proc:
```

Running MapleMIX on qs1 produces a highly specialized result as can be seen in Listing 4. All non-recursive function calls have been in-lined and the higher order functional parameters have been integrated into the residual program at their points of use. The optimizations lead to a 500% performance increase.

Listing 3: Specialized QuickSort

```
quicksort_1 := proc(A, m, n)
  local pivotIndex1, pivotValue1, temp1,
        storeIndex1, i1, temp2, temp3, p;
  if m < n then
    pivotIndex1 := n;
    pivotValue1 := A[pivotIndex1];
    temp1 := A[pivotIndex1];
    A[pivotIndex1] := A[n];
    A[n] := temp1;
    storeIndex1 := m;
    for i1 from m to n - 1 do
      if A[i1] <= pivotValue1 then
        temp2 := A[storeIndex1];
        A[storeIndex1] := A[i1];
        A[i1] := temp2;
        storeIndex1 := storeIndex1 + 1
      end if
    end do;
    temp3 := A[n];
    A[n] := A[storeIndex1];
    A[storeIndex1] := temp3;
    p := storeIndex1;
    quicksort_1(A, m, p - 1);
    quicksort_1(A, p + 1, n)
  end if
end proc
```

All CASes use *generic solutions* in their approach to certain problems. For example, when asked for the degree of a polynomial, degree(a*x^2 + b*x + c), Maple will respond with 2 as an answer. However this answer ignores the case when $a = 0$. If that expression is viewed as a polynomial in the domain $\mathbb{Z}[a, b, c][x]$, then Maple's answer is indeed correct. If instead one were to view it as a *parametric* polynomial in $\mathbb{Z}[x]$ with parameters $a, b, c \in \mathbb{C}$, this becomes a so-called *generic solution*, in other words, correct except on a set of co-dimension at least 1. Interestingly enough this is termed the *specialization problem* [3], and is encountered in any parametric problem in which certain side-conditions on the parameters must hold so that the answer to the global problem is correct. In particular we are looking for precise answers of the following form:

$$\texttt{degree}(a \cdot x^2 + b \cdot x + c, x) = \begin{cases} 2 & a \neq 0 \\ 1 & a = 0 \wedge b \neq 0 \\ 0 & \text{otherwise.} \end{cases}$$

In order to use partial evaluation toward this goal, one must first be willing to change the representation of answers. In our case we will use a residual program to represent the answer to a parametric problem, as programs can be a better representation of answers than expressions for many tasks. In our encoding of answers the if-then-else statement will be used to represent the cases. The next listing shows a program that computes the degree of a polynomial. It is safe to use Maple's built-in degree function because it will always return a conservative answer as explained above.

```
coefflist := proc(p) local d, i;
    d := degree(p,x);
    return [seq(coeff(p, x, d-i), i=0..d)];
end proc:
mydegree := proc(p, v) local lst, i, s;
  lst := coefflist(p, v); s := nops(lst);
  for i from 1 to s do
    if lst[i] <> 0 then return s-i end if;
  end do;
  return -infinity;
end proc:
```

In order to use PE to extract the cases we must treat the polynomial coefficients as dynamic variables. Here most of the structure of the polynomial is static so a large amount of specialization is possible. Our treatment of partially static data structures is crucial toward getting a suitable result. In particular the coeff function has been extended in the reducer to be able to return the dynamic coefficients of the partially static polynomial. The function goal:=(a,b,c)->mydegree(a*x^5+b*x+c, x) when called directly (with symbols for a, b, c) will return 5, but residualizes to

```
proc(a, b, c)
  if   a <> 0 then 5
  elif b <> 0 then 1
  elif c <> 0 then 0
  else -infinity end if
end proc
```

Our Calculemus paper [10] shows larger examples involving symbolic integration (of particular interest to people in computer algebra) and another involving Gaussian Elimination, with similar results. The Gaussian elimination example in particular requires all of the techniques we have outlined in this paper to successfully work. It involves 2 static loops over a partially static data-structure, where the inner loop contains a dynamic `if` statement. However, using a completely vanilla Gaussian Elimination routine and our partial evaluator, we were able to reproduce the results of [3] *without* having to invent a specialized algorithm!

Further to [10], as a more classical test of our partial evaluator, we also wrote an interpreter (in Maple) for a small imperative language. Our partial evaluator was able to remove almost all of the interpreter overhead (only environment manipulation remained); in particular, it was able to specialize a recursive (purely functional) binary powering function into an equivalent straight-line imperative program when given a static n. A simple static single-use post-processor could be written to eliminate all remaining overhead.

6. CONCLUSION

MapleMIX is a syntax-directed online partial evaluator which processes a form of abstract syntax we call M-form. This M-form is designed to translate Maple's program representation into one more suited to the needs of a specializer. Contrary to most other approaches, our intermediate form contains *more* primitives than the language itself, which we believe has greatly contributed to the modularity and extensibility of our online partial evaluator.

MapleMIX uses highly online strategies when specializing statements. The online environment has been designed with the depth-first strategy and dynamic conditionals in mind. Transformation to DAG form and a novel approach to treating static loops by performing on-the-fly syntax transformations allows precise specialization without the need to discard static information or merge environments.

We believe that we have achieved our goals of writing an effective online partial evaluator for a large, dynamic language like Maple. It allows us to write more generic yet still efficient code, as well as being able to extract more information out of specific algorithms in the form of what we call "residual theorems".

We plan to continue exploring the applications of partial evaluation to computer algebra. We want to try to apply our PE to larger pieces of Maple's own library — but to do so, we need to take care of some Maple-specific oddities (like the so-called *last-name-eval* rules for tables and procedures, weird parameter passing exceptions, etc), as well as implemented many more "smart" builtins (we need roughly 100 of the 217 builtins to be made smart to get good results on the larger Maple routines). However, it does appear that this should result in some efficiency gains (because of generic code being specialized) as well as "information extraction" from non-parametric routines. All the code and a substantial test suite is available by emailing either authors; we will make the code publicly available in the coming year.

7. REFERENCES

[1] L. O. Andersen. C program specialization. Technical report, DIKU, University of Copenhagen, May 1992.

[2] A. W. Appel. *Modern Compiler Implementation: In ML*. Cambridge University Press, New York, NY, USA, 1998.

[3] C. Ballarin and M. Kauers. Solving parametric linear systems: an experiment with constraint algebraic programming. *SIGSAM Bull.*, 38(2):33–46, 2004.

[4] J. Carette. Gaussian elimination: a case study in efficient genericity with metaocaml. *Science of Computer Programming*, 62(1):3–24, September 2006. Special Issue on the First MetaOCaml Workshop 2004.

[5] J. Carette and S. Forrest. Mining Maple code for contracts. In Ranise and Bigatti [13].

[6] J. Carette and O. Kiselyov. Multi-stage programming with functors and monads: Eliminating abstraction overhead from generic code. In *GPCE*, pages 256–274, 2005.

[7] D. Elphick, M. Leuschel, and S. Cox. Partial evaluation of MATLAB. In *Proceedings of the second international conference on Generative Programming and Component Engineering*, pages 344–363. Springer-Verlag New York, Inc., 2003.

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

[9] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International(UK) Limited, 1993.

[10] M. Kucera and J. Carette. Partial evaluation and residual theorems in computer algebra. In Ranise and Bigatti [13].

[11] M. B. Monagan, K. O. Geddes, K. M. Heal, G. Labahn, S. M. Vorkoetter, J. McCarron, and P. DeMarco. *Maple 10 Advanced Programming Guide*. Waterloo Maple Inc., 2005.

[12] D. R. Musser and A. A. Stepanov. Generic programming. In *ISSAC 1988: Proceedings of the International Symposium on Symbolic and Algebraic Computation*, volume 358 of *Lecture Notes in Computer Science*, pages 13–25. Springer-Verlag, 1989.

[13] S. Ranise and A. Bigatti, editors. *Proceedings of Calculemus 2006*, Electronic Notes in Theoretical Computer Science. Elsevier, 2006.

[14] E. Ruf and D. Weise. Preserving information during online partial evaluation. Technical Report CSL-TR-92-517, Stanford University, April 1992.

[15] D. Stoutemeyer. Crimes and misdemeanors in the computer algebra trade. *Notices of the AMS*, pages 701–785, 1991.

[16] E. Sumii and N. Kobayashi. Online type-directed partial evaluation for dynamically-typed languages. *Computer Software, Iwanami Shoten, Japan*, 17(3):38–62, May 2000.

[17] E. Sumii and N. Kobayashi. A hybrid approach to online and offline partial evaluation. *Higher-Order and Symbolic Computation*, 14(2/3):101–142, 2001.

[18] P. Thiemann. Cogen in six lines. In *Proc. ACM SIGPLAN International Conference on Functional Programming 1996*, pages 180–189, May 1996.

[19] P. Thiemann. *The PGG System - User Manual*, March 2000.

[20] P. Thiemann and D. Dussart. Partial evaluation for higher-order languages with state. available from first author's web page, July 1999.