Partial Evaluation of Maple

Jacques Carette [1]

McMaster University, 1280 Main St. West, Hamilton, Ontario Canada L8S 4K1

carette@mcmaster.ca

Michael Kucera [2]

IBM Toronto Lab, 8200 Warden Ave, Markham, Ontario Canada L6G 1C7

mkucera@ca.ibm.com

Abstract

Having been convinced of the potential benefits of partial evaluation, we wanted to apply these techniques to code written in Maple, our Computer Algebra System of choice. Maple is a very large language, with a number of 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 conduct our experiments, first on small codes, and now on actual routines taken from Maple's own library. Here, we document the techniques we had to invent or adapt to achieve these results.

Key words: 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 [**?**,**?**]), should be a very powerful tool when combined with a Computer Algebra System (CAS). Certainly code generation from a CAS had already shown

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

[1] Supported in part by NSERC Discovery Grant RPG262084-03.

[2] The content in this document is based on investigations undertaken prior to employment with IBM.

itself [**?**,**?**] to be a successful technique. Eventually, our high hopes were proven correct, as reported previously [**?**,**?**]. But what that paper does not really say is how difficult this turned out to be. The basic theory and practice of partial evaluation as lucidly explained in [**?**], and various papers [**?**,**?**,**?**,**?**,**?**] 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 of choice. We have named our partial evaluator MapleMIX. Note that since doing any kind of static analysis of Maple programs seems outlandishly difficult [**?**,**?**], we felt that the best approach would be to write an *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 [**?**] 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 [**?**] did not become standard practice. We wanted to keep that programming style, but without the efficiency cost. While [**?**] 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* (although have nevertheless obtained some significant speedups), 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 [**?**], it is not necessary to invent parametric algorithms to deal with this, as current algorithms combined with partial evaluation is sufficient.

We assume that the average reader 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-known features in any of Scheme, Java, C, Ocaml or Haskell). One particularity 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 proc;
```

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 explicit program transformation. We are, of course, doing some implicit program transformation using expression manipulation. In other words, the partial evaluation community is quite justified in its belief that `power` is too simple to illustrate much of anything.

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 [**?**] also reports having to work rather hard to get their results. Of course all the work on partial evaluation for (full) Scheme (like [**?**]) 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 (see discussion below) and 2) the demonstration that partial evaluation is an effective tool when applied to a Computer Algebra language. 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. It is worth remembering that we did not seek

to do research in partial evaluation, we were attempting to *use partial evaluation* as a tool to solve problems; thus our implementation is a mixture of well-known, conventional ideas and novel approaches.

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. (For example Maple has one syntactic form for assignment whereas our intermediate 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 use a variety of binding time lattices to deal with structures which are partially static and partially dynamic. Note that since our partial evaluator is online, this binding time information is collected as it is discovered during specialization, and is used as the specialization (and residualization) progresses. This is especially useful for dealing with partly static hash tables, static procedures which refer to (dynamic) lexically scoped variables, and with variables which can revert to being static after having been dynamic for part of the execution.

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, something rather common in CAS codes.

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. We give representative examples of our results in Section 5.

This paper is an extension of [**?**]. We have made the description of the Maple programming language more thorough, to give a better sense of the scale of the task that we face. We have similarly improved the description of the partial evaluator itself; we present a description of the current implementation of MapleMIX which has evolved since we last reported on it. This evolution has been generally driven by getting larger examples of real Maple code (mostly taken from the Maple library) to properly specialize. We needed to completely rethink how we approached closures, had to deal with `procname` recursion, simplified the handling of routines with special evaluation rules, properly residualize dynamic errors, deal with multiple assignments, rtables, functional variants of builtins (like `` `if` `` and `` `+` ``)

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

Fig. 1. Simplified Maple Abstract Grammar

and typed locals, as well as making sure that boolean operators are evaluated using McCarthy semantics, even in mixed static/dynamic cases. We will present new examples in Section 5 which needed these changes.

2 Maple

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. However, at its heart still lies a sophisticated programming language [**?**].

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 although **not** its con-

crete syntax, but influenced by it. This grammar is a slightly simplified version for expository purposes, and uses standard regular expression syntax in the definition of s. 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 auxiliary production which denotes expression sequences of specific terms (identifiers or type-decorated identifiers in this case). An *expression sequence* is the name given to an ordered sequence of either 0 or ≥ 2 expressions, as 1 element expression sequences automatically "flatten" to the expression itself. This is a pervasive structure in Maple, and is used as the "contents" of lists, sets and (unevaluated) function calls. We use the $+(e)$ notation to denote an n-ary operator (in this case $+$, $*$, function application and the expression sequence constructor , are all n-ary). $\mathbb{F}$ denotes the floating point numbers, and `string` denotes string literals. The single biggest difference between Maple and other languages is that some of its fundamental operations (like $+$ and $*$) can return *unevaluated*. 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). Furthermore, it is common to have routines that look like

```
proc(x)
  if type(x, 'numeric') then
    # some numeric computation
  else
    'procname'(args)
  end if
end proc
```

which evaluates to a numeric value given a numeric argument, but otherwise evaluates to *itself* (via its name, given by `procname`, and its expression sequence of arguments, given by `args`). These first-class expressions really are models of *open terms*, which few languages possess, as is the concept of returning an unevaluated version of a function call as a "result".

Figure 2 gives a few of the unusual rules for the operational semantics for Maple. The first three rules detail the evaluation of identifiers, and are some of the strangest in any programming language. The third rule basically says that unassigned identifiers are fine in Maple, they stand for *themselves*. The first two rules use the predicate LNE, which is short for "last name evaluation"; basically, if x is an identifier which has a value in the current store σ, then x will evaluate to its value *unless* it satisfies the predicate LNE. Currently, LNE returns true if the value assigned to x is a table, a procedure or a module; in other words, for tables, procedures and modules assigned to an identifier, the "value" associated to that identifier is again "itself" by the default evaluation mechanism, rather than the underlying value. This is a very large source of complexity in the partial evaluator. Note that the historical motivation for this rather strange rule was printing: in many cases, one is more in-

$$\frac{x \in \sigma \quad \neg LNE\,(\sigma x)}{\sigma x} \qquad \frac{x \in \sigma \quad LNE\,(\sigma x)}{x} \qquad \frac{x \notin \sigma}{x}$$

$$\frac{e_1 \Rightarrow e_3 \quad e_2 \Rightarrow e_4}{e_1{+}e_2 \Rightarrow e_3{+}e_4} \qquad \frac{}{i_1{+}\ i_2 \Rightarrow i_1{+}i_2} i_1, i_2 \in \mathbb{Z}$$

$$\frac{}{z_1{+}\ z_2 \Rightarrow z_1{+}z_2} z_1, z_2 \in \mathbb{F} \qquad \frac{e_1 \Rightarrow e_2}{x{+}e_1 \Rightarrow x{+}e_2} x \in \texttt{identifier}$$

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

Fig. 2. Operational Semantics fragment

terested in the name of an LNE object than its actual value. It is also worthwhile noting that unassigned local identifiers obey the same rules; in particular, it is possible for a named closure to escape its definition context – a mechanism which turns out to be frequently used for name generation.

The next 3 rules are the usual ones for $+$, but the last one on the third line says that adding an identifier to anything will simply return an unevaluated $+$. We then have a rule saying that the unevaluation quotes do just that, they prevent further evaluation. The next rule expresses that $=$ is just a data-constructor for equations, so that when a boolean is needed (for example by `if` as in the next rule), 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, given in Appendix A.3) have unusual semantics 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, `DEBUG` will invoke the debugger, `_jvm` starts a Java Virtual Machine, `evalhf` is an interpreter for an embedded sub-language which corresponds roughly to Fortran but using Maple syntax [and thus creates an environment in which programs have different semantics], `pointto` returns an integer which is actually the *pointer* to a Maple object (!), etc. Furthermore, there are common environment variables (see A.2), also called fluid variables (as available in Common Lisp and now Ruby as well) such as `Digits` to control the number of (decimal!) digits to use for the evaluation of floating point expressions, and `Order` to control the order of expansion of a series. There are also builtin routines like `` `+` ``, `` `*` ``, `assign` and `` `if` `` which are functional equivalents of builtin syntactic operators

and statements.

It is rather unfortunate, but there is no canonical reference for the semantics of Maple (operational or otherwise). While the reference documentation [**?**] and the even more extensive online documentation built into the product do give informal operational descriptions of most of the language, these are incomplete and sometimes ambiguous. However, since expression reduction is done by calling the underlying language, this part of our partial evaluator is always faithful. While we would have liked to give the complete operational semantics that we have built in to our tool, this would have taken over a dozen pages.

3 A Partial Evaluator for Maple

This section presents an overview of the architecture and design of MapleMIX.

We decided early on that our partial evaluator would not only be syntax-directed, but that a number of features would be implemented via transformations on abstract syntax trees. This had major repercussions on the design of various modules. Overall, the system is structured as a specific sequence of program transformations, with a special emphasis on particular transformations occurring before and after the specialization phase. This is inspired in part by the design of some compilers [**?**]. We believe that this approach leads to a highly modular architecture for practical online partial evaluators for complex languages. We view MapleMIX as being essentially an interpreter that has the additional functionality of generating residual code for deferred computations. It contains an expression reducer that was influenced by the *cogen* approach [**?**] to partial evaluation. Furthermore we have implemented a novel online approach to handling partially static data-structures such as polynomials with known monomials but unknown coefficients.

3.1 Characteristics

MapleMIX has the following characteristics:

- **Online.** No pre-analysis of the code is performed. The goal is to exploit as much static information as possible in order to achieve good specialization. Furthermore we have implemented a novel online approach to handling partially static data-structures such as lists and polynomials.
- **Written in Maple.** This allows direct access to the excellent reification/reflection functions of Maple (i.e. `FromInert` and `ToInert`) as well as access to the underlying interpreter. This permits us to stay as close to the semantics of Maple as possible. Additionally, scanning and parsing of Maple programs does not need

to be considered. Maple's automatic simplification feature, instead of hindering us, sometimes helps to slightly clean up residual code [3] .

- **No annotations necessary.** As well as being online, MapleMIX requires no annotations at all. This was really the only possible choice as we wanted to be able to specialize part of Maple's own library, which is much too large and complex to be annotated.
- **Not self-applicable.** We are more concerned with manipulating Maple code than with producing generating extensions. 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 were used used when first writing the partial evaluator. Nevertheless, we may eventually be able to apply the partial evaluator to itself.
- **Standard approach.** Since we are not concerned with producing generating extensions we are free to take the standard approach to PE. This allows the PE to be written in a direct way similar to an interpreter which will make it easier to refine and add new features to the partial evaluator in the future. Having said that, the specializer contains an expression reducer that has been inspired by the online cogen approach.
- **Function-point polyvariant.** Whenever necessary, the partial evaluator will generate several specialized versions of a function.
- **Syntax-directed.** Maple allows easy access to the abstract syntax tree of a term through its `ToInert` function. In this way the entire core library of Maple may be easily retrieved. Furthermore we have used transformations on the abstract syntax to facilitate the specialization process. We believe this approach leads to a highly modular design for practical online partial evaluators.

3.2 Architecture

MapleMIX consists of several modules, whose overall structure is illustrated in Figure 3. The various modules communicate with each other via an intermidate AST representation which we call M-form. This AST representation has been designed to be convenient for specialization.

- **Inert form to M-form Translator.** Takes an AST that has been output by ToInert and transforms it into an M-form AST. The M-form translator is called on every function that is pulled in by the specializer. M-form ASTs are cached so that even if a function needs to be specialized many times, it is only transformed once.
- **M-form to Inert form Translator.** The reverse transformation that is needed to convert M-form back to Inert form.
- **The Specializer** The specializer drives the entire partial evaluation process. It

[3] The computer algebra literature is replete with examples of so-called premature simplification which lead to incorrect results [**?**].

contains specialization routines for each statement form, generates specialized functions and decides when to reuse them, maintains a call stack of environments, decides when a specialized function should be unfolded and maintains a table that stores the generated residual code.

- **The Expression Reducer** When given the M-form of an expression the reducer will evaluate it as far as possible using the static information provided by the environment. It may return a static value or a dynamic M-form. Builtin functions are handled by the reducer as expressions and are called at partial evaluation time if all the arguments are static. The reducer has handlers for certain Maple builtin functions that have non-standard semantics, such as `seq` which acts as sequence comprehension. The implementation of the expression reducer is inspired by the online cogen approach to partial evaluation.
- **The Online Environment** Stores values of static variables and partially static tables. It also stores structure information for variables that are assigned to partially static data such as lists, which is part of our online approach for handling partially static data and will be discussed in detail later. The online environment has a unique implementation that allows for easy treatment of dynamic if statements.
- **The Function Unfolder** In expression oriented languages unfolding a function is a trivial operation. This is not the case in Maple as there are many caveats to function unfolding. Firstly certain Maple keywords such as `return`, `args`, `nargs` and `procname` depend on the context of the function that contains them and must be handled with care. Secondly if the return value of the function is assigned to a variable then any returns in the function must be converted into assignment statements by the unfolding, and this must be done in a controlled way that preserves semantics. We will not describe the function unfolding transformation in detail but it is important to note that it is complicated enough to warrant its own module.
- **DAG Transformer** Our approach for handling dynamic if-statements and static loops requires a transformation that converts an M-form AST into a DAG (Directed Acyclic Graph). This transformation is done on-the-fly inside the specializer and will be described in more detail later.
- **The Unique Name Generator** Responsible for generating new unique names for use during M-form translation and during specialization.
- **The Module Packager** When the specializer is done it will have produced a set of specialized functions in M-form. Before being converted back to active maple code the residual functions are subjected to a dead code removal transformation. Then each function is converted to inert form and then converted to active Maple code that can then be run. The module packager is responsible for building a Maple module that contains all the residual functions. This way MapleMIX can return a single object that encapsulates the results of partial evaluation.

We followed what seemed to be the "most natural design" for each of the above modules. Fairly predictably, this lead to most of the modules having a purely functional interface, with local state used in the implementation for efficiency and to

follow normal Maple style. Somewhat surprising (to us) was that the most natural design for the online environment was a pure object-oriented design. Internally, it implements a stack of frames, corresponding to the current lexical scope being reduced.

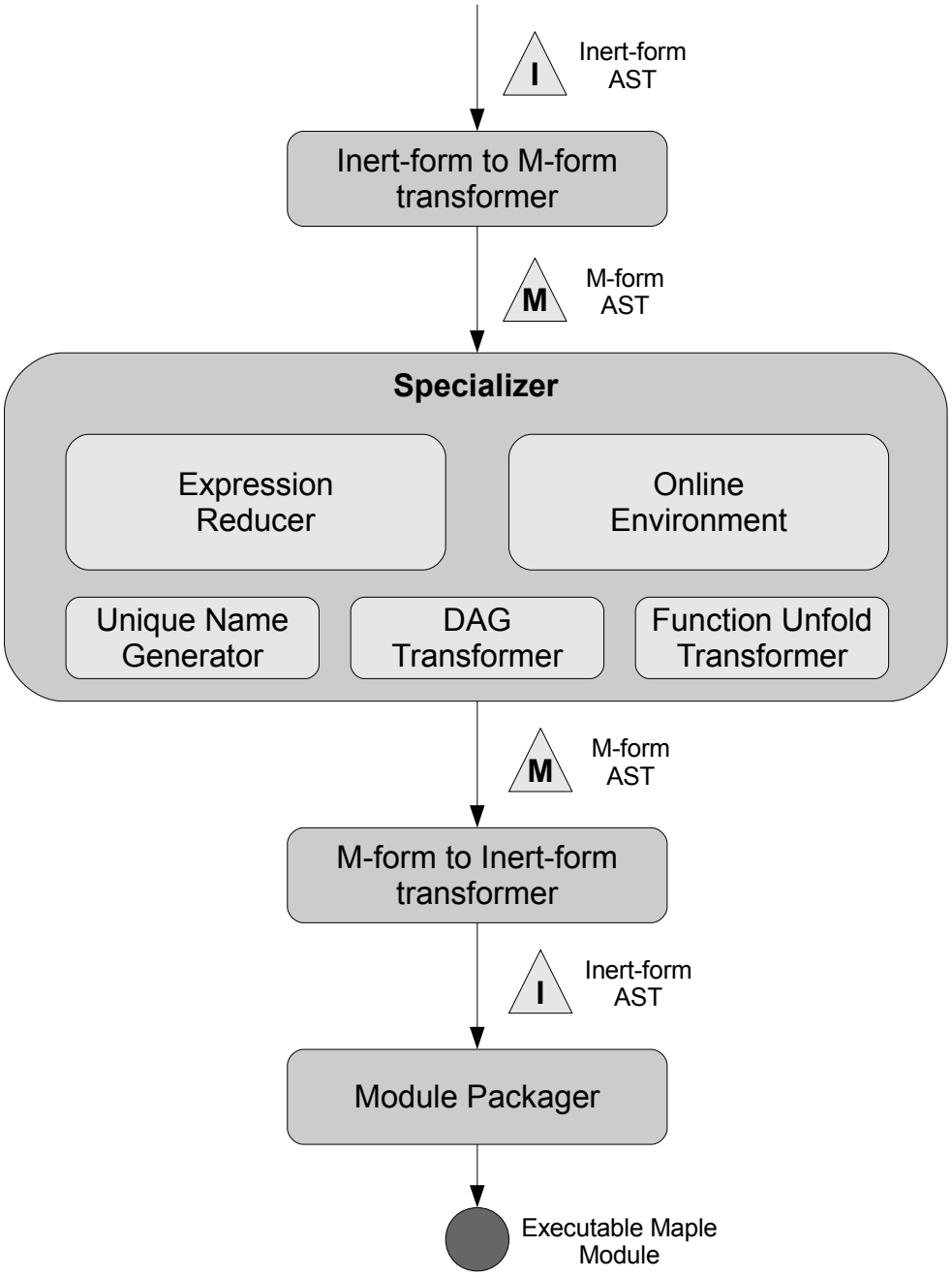


Fig. 3. Architecture of MapleMIX

3.3 Input and Output

Traditionally the input to a partial evaluator is a complete program. But since MapleMIX has direct access to the source of the full Maple library, as well as any other definitions in the current session (not including builtin routines), this gives us scope for considerably more specialization. However, one would not want to specialize everything in the "current session" as that would include the 1,000,000 lines of the Maple library. 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 of the resulting specialized program. MapleMIX may generate several residual functions, which will be 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 as long as they are written in the supported subset of the language. In theory, this now includes the complete language, however the set of fully supported builtin functions and constants is more restricted. As long as the semantics of the builtin functions which are used (but not yet fully supported) is standard, this mostly results in suboptimal residual code rather than in incorrect code.

3.4 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. The AST produced by `ToInert` was not designed with partial evaluation in mind. We have chosen to transform it into another format, which we have named *M-form*, that has been designed to be convenient for specialization. In simple cases, it is essentially isomorphic to the inert form:

```
> inrt := ToInert(x + y);
       inrt := _Inert_SUM(_Inert_NAME("x"), _Inert_NAME("y"))
> m := M[ToM(inrt)];
                m := MSum(MName("x"), MName("y"))
```

Traditionally many existing partial evaluators first transform their input into a simpler core language (for example the C-mix partial evaluator transforms C to CoreC [**?**]). This approach reduces the syntactic forms that the specializer must support, for example by transforming loops into into unstructured code with gotos. However there may be a cost for this apparent simplification: it is possible to lose 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 75.

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 4.7). 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.

3.4.1 Assignments

Maple is an imperative language with global state and side-effecting expressions. Statements cannot occur in an expression context, so 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 for side-effecting I/O functions. Some library functions are also considered intrinsic (like the Vector and Matrix constructors) in order to simplify the 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 [4] . 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, `MAssignToFunction`, is generated by this transformation to specifically represent assignment of a function call to a variable.

[4] This is essentially *let* insertion for an imperative language.

In this way the specializer can decide, based on syntax alone, to do a simple reduction, or perform specialization on a function body. The splitting transformation has the unfortunate effect of possibly creating many new assignment statements. However, we know that the new variable is only assigned to once and only used once. Thus if the specializer decides not to unfold a split function call, or if the function unfolds into a single assignment statement, then this knowledge can be used later. In particular, when translating from M-form 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).

3.4.2 If Statements

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`

An `if` statement 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.

3.4.3 Loops

Inert form has two kinds of loop, both variations of for loops.

- *_Inert_FORFROM* Represents a common for loop of the form:
  `for i from 3 to 11 by 2 do ... end do`
- *_Inert_FORIN* Represents a loop that accesses all elements of a linear data structure such as a list or set, also commonly called a foreach loop.
  `for e in [1,2,3] do ... end do`

Both kinds of loop have an optional `while` clause, which is checked at 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 omitted in concrete syntax, but are always present in the AST. For example, a while loop is actually a for-from loop with all clauses left to default values except the `while` clause.

There is different static information contained in `for` loops and `while` loops. 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 [5] . 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.

Original Code	Transformed Code

`while f(x) do`
`  ...`
`end do`
	`m1 := f(x);`
	`while m1 do`
	`  ...;`
	`  m1 := f(x);`
	`end do`

Fig. 4. Splitting of while condition

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 of a loop. If one is encountered during translation to M-form, an exception is thrown. There is however one case where a simple transformation is performed to remove the use of `next`:

Original Code	Transformed Code

`... do`
`  if C then next end if;`
`  S1;`
`end do;`
	`... do`
	`  if not C then`
	`  S1;`
	`  end if;`
	`end do;`

Fig. 5. Removal of next

3.4.4 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

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

significantly for the sole purpose of making it easier to deal with this construct in 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.

3.4.5 Summary

The following table outlines the differences between Inert form and M-form.

Construct	Inert form	M-form
if statements	`elif` branches and optional `else` branch.	Exactly two branches
loops	`for` and `foreach`, both with optional `while` condition.	`for`, `foreach` and `while` loop.
names	Locals, params, lexicals and pre-assigned	Add two: generated names, and single-use.
expressions	Several expression forms.	In statement context, only `MStandalone` and `MStandaloneFunction`.
assignment	one form	4 cases: from an expression, from a function, from a nested table, and where lvalue is a tableref.
procedures	unique (but version dependent)	version independent; additional fields to encode `args` and `nargs` usage.

Furthermore the following constructs are available only in M-form.

Construct	M-form
commands	Embedded commands (`&onpe("command")`) for debugging
DAG pointers	`MRef`, for sharing
loop drivers	Computed goto inserted by specializer to unroll statis loops.

3.5 Options

As partial evaluation is a rather complex process, there is ample room for variations. MapleMIX supports various options to give the user some control over this. The simplest option allows one to declare that a function is PURE, INTRINSIC or DYNAMIC. One can also control function sharing, how dynamic variables are propagated, and whether assignments should be inlined (when possible).

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 [**?**]. 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 [**?**]. 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 [6] 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 (most notably `op`, `degree`, and `coeff`, as well as `nops`) 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.

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 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 [**?**]. Another approach uses an abstract interpretation as a shape analysis to gather static shape information as a pre-phase [**?**]. Our approach is completely online and has the potential to exploit the full information available during specialization. However, it

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

must be noted that quite a bit of custom support for various dynamic representations had to be programmed into the reducer in order to achieve good results. This is a tedious task, and more research is needed to better understand what is involved. In the context of symbolic computation, our current conjecture is that if the original function is applied to the reified (open) term obtained by replacing the dynamic parts of a partially static by a fresh name, and the result is correct under all substitutions of values for those fresh names, then this is a correct reduction. All of our implementations follow that pattern, but with a caveat: for example, `degree` will not return "the" degree, but rather a guaranteed upper-bound for the degree. In other cases, like `length`, this scheme always gives the correct result.

In traditional PE, especially when a BTA is used, it is very common for values to go from static to dynamic. This can cause a snowball effect in which more and more constructs become dynamic as specialization progresses. With our approach it is possible for reduction involving a dynamic term to result in a static value. One side effect of this approach is that the PE tends to generate residual code which becomes dead code when dynamic data leads to static results. Such dead code is removed by a simple post-phase cleanup.

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 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 reducer [7]. 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 now fully supports closures in the subject programs. Any lexically nested procedure which contains references to its outer environment is treated as being *essentially static*. A new tag, `MEssentiallyStatic`, which introduces a new binding time, is introduced to represent this. An `MEssentiallyStatic`

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

structure always contains the dynamic representation of a procedure. However, we know that the only dynamic parts of such a procedure are the lexical references, which are few – thus the "essentially static". What is done is that the procedure definition, instead of being stored as a static element in the environment, is stored in its dynamic form. Then, whenever an actual call is made to that procedure, the lexical bindings are looked up in the current environment; if these are static, then the procedure can be reified as fully static, otherwise it needs to be residualized. This allows the partial evaluator to support higher-order functions, especially in the forms that frequently occur in Maple code. The only drawback is that this can involve a fair bit of reification, which is inefficient, but unavoidable.

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 [**?**]. 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 [**?**]. 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 multiple functions are 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 [**?**].

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 [**?**,**?**]. 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 implemented as 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 [**?**]. 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 representation 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 6.

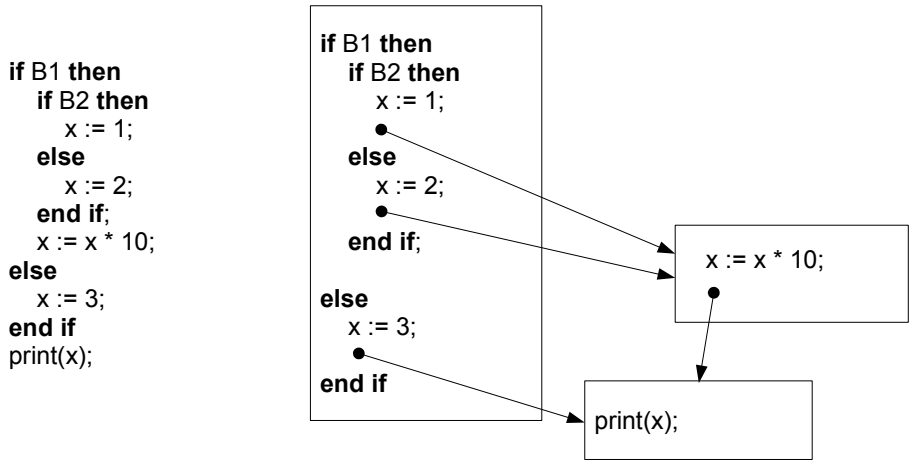


Fig. 6. DAGform

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

```
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;
```

Original Code	Specialized Code

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

Fig. 7. Example of `if` statement specialization

In the above pseudocode, `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. In other words, it checks that there is at least one edge in the control flow graph which goes from `C1` to `S`. 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 7.

4.5.2 Comparison with other methods

A natural approach to specializing `if` statements is by merging environments [**?**]. 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 Tables and rtables

MapleMIX fully supports both tables (i.e. hash tables) and rtables (i.e. various shaped n-dimensional arrays) as a partially static data-type. Below, we will call both of these table. Partially static here means that some elements of a table may be static while others are dynamic. Also tables require special treatment because a variable of type table is actually a reference to a table. Therefore it is possible for a function to be side-effecting through its input parameters. Support for tables is implemented as an extension to the design of the online environment. The environment will provide as part of its interface methods for manipulating regular variables and separate methods for manipulating table elements.

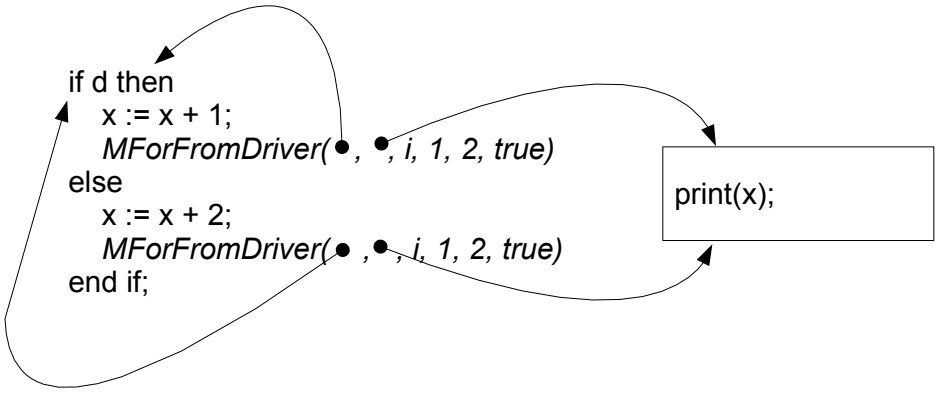
Each environment consists of a stack of settings. Bindings for tables will be kept separate from bindings for regular values. The idea for handling tables is to be able to represent only part of a table in one particular setting. The complete table may be rebuilt by starting at the top of the stack and working downwards. The environment provides methods for directly querying and retrieving the elements of specific table indices. The point is to avoid traversal or rebuilding of the entire table whenever possible.

One distinction must be made between tables and rtables: tables have last name evaluation rules, while rtables have normal evaluation rules.

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. This is the only restriction on the body of loops, all other side-effects are supported.

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 8, gets rewritten (in DAG M-form) as:



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. When the specializer encounters a loop driver it will simply evaluate the loop condition and follow the appropriate pointer. 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 while condition must always be statically reducible to a boolean value. If at any point it is dynamic [8] an error is issued and specialization is aborted. MapleMIX currently does not support partial unrolling of a loop.

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 8 would be of size $O(2^n)$ if the loop were from 1 to a static positive integer n.

Original Code	Specialized Code

`x := 1;`
` for i`
` from 1`
` to 2`
` while true`
` do`
`  if <dynamic> then`
`  x := x + 1;`
`  else`
`  x := x + 2;`
`  end if;`
`end do;`
`print(x);`
	`if <dynamic> then`
	`  if <dynamic> then`
	`  print(3)`
	`  else`
	`  print(4)`
	`  end if`
	`else`
	`  if <dynamic> then`
	`  print(4)`
	`  else`
	`  print(5)`
	`  end if`
	`end if;`

Fig. 8. Example of dynamic conditional in a static loop

[8] It is possible to construct code in which this happens, but we have not encountered such code used in practice.

Dynamic loops pose a significant challenge to specialization. Since it is unknown how many times a dynamic loop will iterate, it must 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 which might 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 [**?**]. 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.

The following two cases are considered: First, any assignment statement would effectively cause the target variable of the assignment to be dynamic. If a target variable is already dynamic then there is no problem. If a target variable is currently static then its value must be made available to the residual program before the loop executes. Its value is then removed from the environment and it becomes dynamic. Thus, only *statically invariant* values are maintained in the environment in that case. The second situation requires an analysis of the entire body of the loop must be analyzed as well as any of the bodies of the functions that are called (and so on), to see if there are statements which affect the global state. If the global state is changed, the loop is not specialized. Besides being highly inefficient this approach would lead to termination problems when a dynamic loop contained a call to a recursive procedure. For these practical reasons non-intrinsic function calls are currently not allowed within dynamic loops.

Consider an iterative, imperative version of the power function.

```
iterpow := proc(x, n) local temp, i;
    temp := 1;
    for i from 1 to n do
        temp := temp * x;
    end do;
    return temp;
end proc:
```

When specialized with respect to $n = 5$ the loop is static and is unrolled a fixed number of times, as explained in the previous section, with the resulting code being the obvious 6 sequential assignments. When specialized with respect to $x = 5$ the results are quite different. The loop is now dynamic because the value of n is unknown. The first assignment to $temp$ is initially removed by the specializer. Then

when the loop body is analyzed it becomes known that the static value of the $temp$ variable is needed and so a new assignment statement is generated and inserted before the loop. While loops are treated in a similar manner to for loops except unrolling will never occur.

```
iterpow_n := proc(n) local temp1, i1;
    temp1 := 1;
    for i1 to n do
        temp1 := 5 * temp1
    end do;
    temp1
end proc
```

4.9 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 [**?**]. 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.

```
> FromInert(
  _Inert_SUM(_Inert_POWER(_Inert_NAME("x"), _Inert_INTPOS(2)),
             _Inert_INTNEG(5)));

                              x^2 - 5

> FromInert(_Inert_SUM(_Inert_VERBATIM(x^2),
                       _Inert_VERBATIM(-5)));

                              x^2 - 5
```

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

We show some MapleMIX results. We show examples which we felt representative of our successes. Additional examples can be found in [**?**], and in the second author's Master's Thesis [**?**]. We cannot yet report on a truly broad evaluation of our techniques as "typical" Maple *library* code as we only recently were able to support essentially the full language, although we still have work to do to support the semantics of more builtin functions, constants and predefined environment variables. However, MapleMIX seems to have now reached the point where implementing these features looks straightforward if somewhat tedious, rather than daunting.

Listing 1. 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:
```

5.1 Quicksort

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 1 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. 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
```

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 2. 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.

Figures 9 and 10 show an example of results of timing tests of both the original and the specialized versions of quicksort (as given by Maple's `profile` function). Each algorithm was tested on an array of 10000 elements where each element is a random integer in the range 1..5000. The specialized quicksort shows a huge performance gain of almost 500 percent, most likely due to the elimination of the overhead involved in the function calling mechanism.

function	depth	calls	time	time %	bytes
partition	1	7044	3.285	69.61	46494668
swap	1	90928	0.859	18.20	25116568
quicksort	32	14089	0.575	12.18	12445516
qs1	1	1	0.000	0.00	716
total:	35	112062	4.719	100.00	84057468

Fig. 9. Timing results for generic quicksort `qs1`

function	depth	calls	time	time %	bytes
quicksort_1	32	14185	0.953	100.00	23724852
total:	32	14185	0.953	100.00	23724852

Fig. 10. Timing results for specialized quicksort_1

5.2 Residual Theorems

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* [**?**], 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
```

5.3 Integration

Listing 3 shows a bit of code that we may expect to find somewhere in a symbolic integrator. While actual integration code tends to be much more complex, the code below is representative enough to illustrate our point. This code takes as input a polynomial represented as a list of monomials, each of which are represented as a coefficient and a pure power. We then use a sub-function to integrate pure powers of a variable. Note that this sub-function contains calls to two large pieces of Maple code: `ln` and `int` itself. In the first case, we have to tell the partial evaluator to

always residualize code for `ln` (i.e. the partial evaluator does not look at the code, although this could result in calls to `ln(1)` being residualized). In the second case, there is nothing to do as this branch is never taken, and thus never examined.

Listing 3. Mock integrator

```
int_pow := proc(i,var)
    if op(1,i)=var then
        if op(2,i)=-1 then
            ln(var)
        else
            var^(op(2,i)+1)/(op(2,i)+1)
        end if
    else
        int(i,var)
    end if;
end proc:

int_sum := proc(l, var) local res, x, i;
    res := 0;
    for i from 1 to nops(l) do
        x := op(i, l);
        res := res + x[1]*int_pow(x[2],var);
    end do;
    res;
end proc:
```

And, as expected, the result shows the cases we expect, depending on whether $n = -1$ or not. As far as mathematical correctness goes, this result is frankly better than the output of any CASes we know; Derive's "correct in the limit" answer of $x^{n+1}/(n+1) - 1/(n+1)$ is nice, but difficult to deal with since it is an intensional rather than extensional result (i.e. one cannot just "plug in" values of n, the expressions always have to be interpreted via limits).

Listing 4. Integrator result

```
goal := proc(n) local x;
    int_sum([[5,x^2], [-7,x^n], [2,x^(-1)]], x)
end proc:

result := proc(n) local m1, res1, x;
    if n = -1 then
        m1 := ln(x)
    else
        m1 := x^(n + 1)/(n + 1)
    end if;
    res1 := 5 * x^3/3 - 7 * m1;
    res1 + 2 * ln(x);
end proc
```

It is also worthwhile noting that automatic expression arithmetic will take care of the cases where $n = 2$ or $n = -1$, and the resulting expression will have the correct terms, so that no additional cases need to be treated. In other words, if we are willing to shift our representation to code instead of expressions, then our straightforward algorithms already contain all the necessary information for all the cases, which a partial evaluator can "dig out".

5.4 Generic Linear Algebra

After experimenting with very generic programming [**?**], Monagan et al. [**?**] implemented a simpler mechanism for generic linear algebra in Maple, and made the resulting code publicly available [**?**]. From this, we extracted the code for matrix multiplication, as well as the implementation of Berkowitz's Algorithm.

In Appendix B, we show the full code for generic matrix multiplication, the commands necessary to specialize it to the integers, and the resulting code. It should be clear that all the overhead has been eliminated. For example, on 200x200 matrices, the specialized version was 2.1 times faster, and used 5 times less memory; this gets asymptotically better as the sizes of the matrices is increased.

The results for the much more complex Berkowitz Algorithm are similar, so we omit the details.

5.5 Further examples

We have many more examples of increasingly large pieces of code which can be successfully handled by MapleMIX. We are in the process of building a web site [**?**] to make these examples easily available.

There we also show variations on one of the most interesting examples from [**?**], namely Gaussian Elimination on a matrix with some generic elements, yielding an answer (as code) which contains exactly the right conditionals for all of the subcases. 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 [**?**] *without* having to invent a specialized algorithm!

We also show a new example, that of the reduction of a code generator for the 4 Weierstrass functions (which are doubly-elliptic functions, quite important in certain areas of mathematics). These 4 routines were so similar that, in the Maple library, the first author used lexically scoped higher-order functions to "generate" them. While elegant, the drawback with this approach is having to dereference and

execute static code at run-time. We show how MapleMIX, when applied to the generator, eliminates this overhead and automatically residualizes these 4 routines to what had previously been hand-written code with considerable duplication.

As a more classical test of our partial evaluator, we also wrote an interpreter (in Maple) for a small imperative language with recursive calls. 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. Furthermore, in the embedded language, we implemented a self-interpreter; in fact, this 2^{nd} level interpreter actually implements a slightly richer language. We were able to reduce simple expressions in this doubly-nested situation as well, and remove the double overhead. This example can also be found at [**?**].

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 are well on our way to apply our partial evaluator to substantial pieces of Maple library code. We are aware of a few Maple oddities whose semantics we have not thoroughly tested (like all the corner cases of last-name-evaluation rules), but by and large we seem to have implemented the semantics of the language. To get good results, we now need to attack those builtin routines, constants and environment variables which have special semantics. We estimate that once we have roughly 100 of the 217 builtins made "smart", as well as the constants listed in Figure A.1

and the environment variables in Figure A.2, we should get very good results Furthermore, 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 at [**?**].

A Builtins

Figure A.1 lists some of the more common constants, in other words reserved names which are not associated to a value but rather have nominal semantics. Figure A.2 lists the more common environment variables; note that any variable whose name starts with the charaters `_Env` are automatically environment variables as well, and about 100 of them are in common use in Maple. Finally, Figure A.3 gives a list of all the functions which are built directly into Maple's kernel, and whose operational semantics [in the partially static case] must be directly implemented in the partial evaluator.

```
ModuleApply, ModuleLoad, ModuleID, ModulePrint, lasterror,
undefined, remember, infinity, pos_infinity, neg_infinity,
real_infinity, cx_infinity, Pi, gamma, Catalan, @, @@, in, I,
O, true, false, FAIL, And, Not, Or, Float, SFloat, Fraction
```

Fig. A.1. Selected constants

```
Digits, Order, Normalizer, NumericEventHandlers, Rounding,
Testzero, UseHardwareFloats, _ans, printlevel
```

Fig. A.2. environment variables

`$`, `*`, `**`, `+`, `..`, `<`, `<=`, `<>`, `=`, `>`,
`>=`, `?()`, `?[]`, ASSERT, Array, ArrayOptions, CopySign,
DEBUG, Default0, DefaultOverflow, DefaultUnderflow,
ERROR, EqualEntries, EqualStructure, FromInert, Im,
MPFloat, MorrBrilCull, NextAfter, Normalizer, NumericClass,
NumericEvent, NumericEventHandler, NumericStatus, OrderedNE,
RETURN, Re, SDMPolynom, SFloatExponent, SFloatMantissa,
Scale10, Scale2, SearchText, TRACE, ToInert, Unordered,
UpdateSource, `[]`, `^`, _jvm, _maplet, _treeMatch, _unify,
_xml, abs, add, addressof, alias, anames, `and`, andmap,
appendto, array, assemble, assigned, attributes, bind,
call_external, callback, cat, coeff, coeffs, conjugate,
convert, crinterp, debugopts, define_external, degree,
denom, diff, disassemble, divide, dlclose, `done`, entries,
eval, evalb, evalf, `evalf/hypergeom/kernel`, evalgf1,
evalhf, evaln, expand, exports, factorial, frem, frontend,
gc, genpoly, gmp_isprime, goto, has, hastype, hfarray,
icontent, `if`, igcd, ilog10, ilog2, `implies`, indets,
indices, inner, `int/series`, `intersect`, iolib, iquo,
irem, is_gmp, isqrt, `kernel/transpose`, kernelopts, lcoeff,
ldegree, length, lexorder, lhs, lprint, macro, map, map2,
max, maxnorm, member, min, `minus`, modp, modp, modp1,
modp2, mods, mul, mvMultiply, negate, nops, normal, `not`,
numboccur, numer, op, `or`, order, ormap, overload, parse,
piecewise, pointto, print, `quit`, readlib, reduce_opr,
remove, rhs, rtable, rtableInfo, rtable_convolution,
rtable_eval, rtable_histogram, rtable_indfns, rtable_is_zero,
rtable_normalize_index, rtable_num_dims, rtable_num_elems,
rtable_options, rtable_redim, rtable_scale, rtable_scanblock,
rtable_sort_indices, rtable_zip, savelib, searchtext,
select, selectremove, seq, series, setattribute, sign,
sort, ssystem, `stop`, streamcall, subs, `subset`, subsop,
substring, system, table, taylor, tcoeff, time, timelimit,
traperror, trunc, type, typematch, unames, unbind, `union`,
userinfo, writeto, `xor`, `{}`, `||`, Digits, Order,
Normalizer, NumericEventHandlers, Rounding, Testzero

Fig. A.3. Built-in routines

B Generic Linear Algebra

We present an implementation (verbatim from [**?**]) of a generic matrix-matrix multiplication algorithm.

```
MatrixMatrixMultiplyOperations := [`0`,`+`::procedure,`*`::procedure]:

HasOperation := proc(D,f)
  if type(D,table) then assigned(D[f]) else member(f,[exports(D)]) fi;
end:

# Type check
GenericCheck := proc(P,T) local D,f,n,t;
  if not type(P,indexed) or nops(P)<>1 then
    error "\%1 is not indexed by a domain",P fi;
  D := op(1,P);
  if not type(D,{table,`module`}) then
    error "domain must be a table or module" fi;
  for f in T do
    if type(f,`::`) then n := op(1,f); t := op(2,f);
    elif type(f,symbol) then n := f; t := false;
    else error "invalid operation name: \%1", f;
    fi;
    if not HasOperation(D,n) then error "missing operation: \%1",n; fi;
    if t <> false and not type(D[n],t) then
      error "operation has wrong type: \%1", f fi;
  od;
  D
end:

MatrixMatrixMultiply := proc(A::Matrix,B::Matrix)
  local D,n,p,m,C,i,j,k;
  D := GenericCheck( procname, MatrixMatrixMultiplyOperations );
  if op(1,A)[2]<>op(1,B)[1] then error
     "first matrix column dimension (\%1)
     <> second matrix row dimension (\%2)",
         op(1,A)[2], op(1,B)[1]; fi;
  n,p := op(1,A);
  m := op(1,B)[2];
  C := Matrix(n,m);
  for i to n do
     for j to m do
        C[i,j] := D[`+`](seq(D[`*`](A[i,k],B[k,j]),k=1..p))
     od
  od;
  C
end:
```

If we wish to specialize this to the integers, then via

```
(Z[`0`],Z[`1`],Z[`+`],Z[`-`],Z[`*`],Z[`=`]) :=
    (0,1,`+`,`-`,`*`,`=`);

goal := proc(x,y)
  MatrixMatrixMultiply[Z](x,y);
end proc;
```

We get the expected

```
proc (x, y) local n1, p1, m6, C1, i1, j1;
  if op(1,x)[2] <> op(1,y)[1] then
    error "first matrix column dimension (\%1)
           <> second matrix row dimension (\%2)",
               op(1,x)[2], op(1,y)[1]
  end if;
  n1, p1 := op(1,x);
  m6 := op(1,y)[2];
  C1 := Matrix(n1,m6);
  for i1 to n1 do
    for j1 to m6 do
      C1[i1,j1] := `+`(seq(x[i1,k]*y[k,j1],k = 1..p1))
    end do
  end do;
  C1
end proc;
```