Partial Evaluation and Residual Theorems in Computer Algebra

Michael Kucera [1] Jacques Carette [2]

Computing and Software
McMaster University
Hamilton, Ontario, Canada

Abstract

We have implemented a partial evaluator for Maple. One of the applications of this partial evaluator is to find, in Maple, what is the difference between generic or symbolic evaluation, and complete evaluation. More precisely, when asked `degree(a*x^2+3,x)`, Maple replies 2, which is generically true. However, we are interested in the residual formula $\neg(a = 0)$ which, as a guard, makes the answer 2 correct. While special algorithms have been derived in the past for this particular situation, we show how we can derive many of these algorithms as special cases of partially evaluating Maple code.

Key words: Maple, symbolic computation, partial evaluation, specialization problem

1 Introduction

A frequently controversial aspect of computer algebra systems (CAS) is their use of *generic solutions* to different problems. For example, all dynamically typed CASes [3] will return 2 when asked what is the degree in x of the expression $a \cdot x^2 + b \cdot x + c$. If one were to view that expression as a polynomial in the Domain $\mathbb{Z}[a, b, c][x]$, then that is indeed correct. If instead one were to view it as a *parametric* polynomial in $\mathbb{Z}[x]$ with parameters $a, b, c \in \mathbb{Z}$, this becomes a so-called *generic solution*, in other words correct except on a set of co-dimension at least 1. This is frequently termed the *specialization problem*, and is encountered in any parametric problem in which certain side-conditions on the parameters must hold for the generic answer to be correct.

Our goal here is to show something which might be surprising to most CAS system-builders: the *knowledge* necessary to return *complete* solutions instead of

[1] kuceraml@mcmaster.ca
[2] carette@mcmaster.ca
[3] In other words, Maple, Mathematica, Macsyma and MuPAD, the 4 Ms

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

merely generic solutions is already present in a CAS written in an appropriately high-level programming language [4] ! In other words, we can take an algorithm written non-parametrically (but over a suitably general algebraic domain), and reuse it over a parametric domain.

To obtain such a result, one must first be willing to change the *representation* of "answers". As has already been argued elsewhere[3], programs are frequently much better representations than expressions for many tasks. This paper offers yet another example of this. Once this shift has been accomplished, various program transformation techniques become applicable. For our purposes, instead of looking for an algorithm A to solve a problem P over a very general domain, (i.e. $A(P)$ gives the answer we are looking for), we instead look for an algorithm B which simultaneously solves problem P' over explicit domains (like $\mathbb{Z}$) and gives (correct) generic answers to parametric problems. Then we use a program transformation technique called *partial evaluation* to give us what we are looking for. More specifically, given a parametric problem Q, we use $\texttt{PE}(\lceil B \rceil, Q)$. By $\lceil B \rceil$ we mean a *representation* of the algorithm B. A partial evaluator takes as input a program representation (here $\lceil B \rceil$ and the *static* (or unchanging) input to that program. The result is a new program which takes as input the *dynamic* (or varying) inputs to the original program B, and finally produces the desired output. This is essentially the situation we are faced in when looking at parametric problems. Optimally, the results we are looking for are always of the form

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

Instead of using an expression encoding (as in the above), we will use a program, where `if-then-else` will be used to represent the cases. Finally, we will call guarded equations (like $a \neq 0 \rightarrow 2$) *residual theorems*. This perhaps should be simply termed a "guarded result", but in the context of partial evaluation it makes sense to use the qualifier residual instead. In effect, what we call a residual theorem is both a set of residual proof obligations, as well as a complete set of correct answers to the original problem.

In the next section, we will outline our main tool, partial evaluation. In section 3, some details of our implementation of a partial evaluator for Maple can be found, with more details available in [9]. For reasons of space, we cannot give a real introduction to the specialization problem, and instead refer the reader to Ballarin and Kauers' excellent SIGSAM Bulletin article [2] (and the references therein). The next section displays our results; we show how a general degree routine, a general Gaussian Elimination routine, and a mini-integrator can be properly specialized. We finish with some conclusions and outline where we want to take this work.

[4] We define *appropriate* to mean that current partial evaluation techniques are applicable and successful

2 Partial Evaluation

Partial evaluation (PE) is a program transformation technique that fixes a subset of a program's inputs to specific values, then generates a specialized version of the program based on those values. The resulting program is called the *residual program* or *specialized program*. Essentially partial evaluation attempts to execute a program with some of the inputs missing. Program statements and expressions that cannot be fully evaluated due to missing information are reduced as much as possible, and then residualized. The residual program will finish the computation when the rest of the inputs become available. The fixed input and all information known at partial evaluation time is described as *static*, and all program variables that have unknown value are called *dynamic*. These classifications are known as *binding times*.

Partial evaluation performs aggressive optimizations including constant propagation, loop unrolling, function unfolding and so on. It can be very useful when some particular inputs to a program change infrequently. As much computation is performed at partial evaluation time as possible, and thus the specialized program can be considerably optimized. The classical example is that of a powering function `pow` which computes x^n for $n \geq 0$. Of course, in the context of a symbolic computation system like Maple, this is rather pointless, since this can be done *symbolically*. An example which cannot be done purely symbolically would be

```
cond_expr := proc(cond, x, y)
    if cond(x,y) then x + y
    else x - y
    end if
end proc
```

Original routine.

```
`cond_expr_<` := proc(x,y)
    if x < y then x + y
    else x - y
    end if
end proc
```

Specialized code with `cond = '<'`

Attempting to "run" `cond_expr` with `cond` set to `'<'` and other parameters symbolic predictably leads to a run-time error as the boolean condition cannot (yet) be evaluated. However the specialized code is more efficient as a parameter and a function call have been removed.

Notice also that in the first example, all references to the static parameter n have been removed and all of the conditionals, function calls, subtractions, and parameter bindings have been performed at partial evaluation time. The only operations remaining are the multiplications that the partial evaluator could not perform because the input variable x was dynamic.

There are two main flavors of partial evaluation, *online* and *offline*. An offline partial evaluator does not use the concrete values of program variables when making the decision to remove or residualize a program construct [8]. Offline partial evaluators depend on one or more preliminary analysis phases to gather information to be used by the specializer. The most common analysis is binding time analysis (BTA), which is used to gather binding time information. Whether to each program fragment can be removed (i.e. pre-computed) or needs to be residualized is decided during this preprocessing step.

A partial evaluator is said to be online if the concrete values of program variables computed during specialization can affect the choice of action taken. An online PE makes the remove/residualize decision during specialization, and therefore has much more information available about variables, including possibly their types and values. It is believed that online partial evaluation is usually more precise than offline, leading to better specialization [6]. Since all decisions are made during specialization, an online partial evaluator is very reactionary in its computations, as it has no prior knowledge of program points it has yet to encounter. The degree of specialization resulting from an online strategy is directly related to the specializer's ability to infer and maintain as much static information as possible [10].

Online PE can be more precise than offline, however it is frequently less efficient mainly due to the fact that there is more decision making and environment manipulation during specialization. In particular the binding times of variables must be examined often. Hybrid online/offline approaches have been explored in order to exploit the specialization benefits of online and the efficiency benefits of offline. Sumii [13] showed how a type-based representation analysis could improve the performance of an online PE by adding a limited amount of binding time information that optimizes the specialization process by removing unnecessary computation. Unfortunately, as Maple is fundamentally *dynamically typed* such an approach is extremely difficult for Maple. A partial evaluator lies somewhere in between an interpreter and a compiler. When it evaluates code, it is acting as an interpreter, and when it is generating residual code, it is acting as a compiler [11]. If a PE is run on a program with all inputs given, then the PE will completely evaluate the program and produce a static result. This shows that a PE completely subsumes the functionality of an interpreter. On the other hand, if a PE is run on a program without any inputs given it may still be able to perform optimizations. This shows that a PE is complementary to an optimizing compiler. A PE differs from an optimizing compiler in that a PE will generate several specialized versions of a particular program point based upon multiple usages with different static values; this is known as *polyvariance*. Furthermore, a PE will have to deal with strict constructs such as dynamic conditionals, so in general termination of the partial evaluator can be an issue, and is not always guaranteed. Finally, it is up to the programmer to initiate the PE process and to provide information to guide the process in the form of static inputs. Compiler optimization techniques are usually completely automatic, requiring no additional information to be provided by the programmer.

While it is possible to bootstrap partial evaluation through self-application (leading to the Futamura projections [8]), this is usually very difficult. A more direct approach is to treat a program as its own generating extension by extending the semantics of the language with support for dynamic constructs. This is known as the *cogen approach* [14,12] to partial evaluation.

3 Maple Partial Evaluator

We have implemented a partial evaluator called MapleMIX for a subset of the Maple programming language. This section is a very concise overview of the main technical points, many more details are available from [9]. It has the following characteristics:

- **Online.** No pre-analysis of the code is performed. The PE is written 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 reification/reflection functions of Maple (i.e. `FromInert` and `ToInert`) as well as access to the underlying interpreter. This allows us to stay as close to the semantics of Maple as possible. Furthermore 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.
- **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.

MapleMIX has been designed as an interpreter that has the additional functionality of generating residual code for deferred computations that cannot be performed at partial evaluation time [15]. Through Maples reification function, MapleMIX has access to the code of the entire Maple library. Only the bodies of built-in routines are unaccessible as they are implemented directly in the Maple kernel.

MapleMIX maintains a callstack of environments to store all statically known values. If there is no static binding for a variable in the environment then it is dynamic. 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 a function call is specialized as it is encountered. Thus there may be several functions in the process of specialization at the same time.

The partial evaluator must be able to handle dynamic conditionals correctly. This presents two main challenges: firstly, 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 in the correct environment.

For efficiency reasons we do not wish to create two environments by copying (all or part of) the initial environment. Our solution is to implement the online

environment as a stack of variable bindings. We shall call each element of this stack a *setting*. The stack will to 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. 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 restored to the state before the stack was grown by simply popping the stack.

Code coming after an if statement may have to be specialized with respect to two different environments. This is achieved by representing the input program as a DAG where each statement sequence ends with a pointer to the statement sequence which could be executed next.

3.1 M-form

MapleMIX is a syntax-directed partial evaluator, proceeding through the specialization process by following the structure of the abstract syntax. However, as the inert form given by Maple's `ToInert` is not always the most convenient form to manipulate, we introduce a new form, M-form, which better serves our purpose.

Traditionally many existing partial evaluators first transform their input into a simpler language. For example the C partial evaluator C-mix first transforms its input into a base language called Core C [1]; they remove all loops and replace them with conditional jumps and GOTOs. This approach reduces the syntactic forms that the specializer must support. The disadvantage is that invariants inherent with certain syntactic forms are lost. Some languages (e.g. Fortran) have `for` loops that are guaranteed to terminate. M-form is designed to meet the needs of the specializer, and especially to keep the specializer as small and simple as possible. As such, we found that having redundant syntactic forms (i.e. multiple kinds of assignments) did not make things harder, and in fact made things more precise since having to recover invariants from over-simplistic core language was much more difficult. Translation from inert form to M-form is nevertheless straightforward.

- In order to separate the concerns of expression reduction and environment update, M-form stipulates that all expressions must be side-effect free. This is achieved by splitting non-intrinsic function calls out of expressions and generating new assignment statements.
- If statements in Maple may have arbitrarily many elif blocks and an optional else block. In M-form, an if statement must consist of a conditional expression and exactly two branches. This transformation is achieved by replacing elif blocks by nested if-else statements.
- Inert form mixes `for` and `while` loops; M-form syntactically separates these out. This way the specializer can safely unroll a for loop without risking non-termination (assignment to the loop index variable is not allowed).
- There are several other transformations performed for the purpose of conve-

nience in the sense that they reduce the complexity of the specializer.

3.2 Expression Reduction and Partially Static Terms

The expression reducer evaluates expressions as far as possible given the available static information. The reducer supports operations on most Maple datatypes from simple numbers and strings to lists, higher-order functions and tables. The implementation of the reducer is inspired by an online cogen approach to PE as outlined by Sumii [12]. The idea is to replace the underlying operators of the language with smarter ones which correctly handle dynamic arguments.

This scheme has been extended to provide a novel approach to handling partially static data structures. For example a list may have dynamic elements, but the size of the list may be statically known. Thus it may still be possible to iterate over a list even though some of its elements are dynamic. This is achieved by taking the idea of "smart operators" a step further, by giving certain intrinsic functions the additional ability to properly handle dynamic terms. For example a dynamic list $[a, b, 2]$ where a and b are dynamic local variables will be represented in M-form as `MList(MExpSeq(MLocal("a"), MLocal("b"), MStatic(2)))`. Clearly the size of the list is statically known, and we can exploit such static information present within the dynamic representation. For example the built-in Maple `nops` function has been extended (in MapleMIX) to return a static result in such cases. Several of Maple's built-in functions (and also some syntactic constructs) have been extended to add support for partially static lists and polynomials.

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 (if available) from the environment. If the variable is bound to a dynamic expression then it is substituted for that expression. Special care must be taken not to introduce duplicate computations. A special syntactic form is used to track such substitutions, if the dynamic expression that was retrieved from the environment is not consumed then it will not be residualized, instead the variable for which it was substituted is residualized.

Support for partially static terms has been explored mostly within the context of offline PE. One approach is to use a BTA to determine which elements of a partially static data structure are static and which ones are dynamic. Another approach uses abstract interpretation as a shape analysis to gather static shape information as a pre-phase [7]. Our approach is completely online and exploits the full information available during specialization.

In traditional PE, especially when a BTA is used, it is very common for values to go from static to dynamic. Thus a snowball effect may be observed in which more and more constructs become dynamic. With our approach it is possible for reduction involving a dynamic term to still result in a static value. One side effect of this approach is that the PE tends to generate residual code that becomes dead code as dynamic information leads to static computations. Dead code is removed

by a simple post-phase cleanup.

3.3 Parameterization and optimization

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 specific problem, or write one generic program that solves all the problems. The generic program is 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. First the choice of pivot, which effects the complexity properties of the algorithm. Second the choice of comparison function.

Listing 1: In-place QuickSort

```
swap := proc(A, x, y) local temp;
    temp := A[x]; A[x] := A[y]; A[y] := temp;
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:

quicksort := proc(A, m, n, pivot, compare) local p;
    if m < n then
        p := partition(A, m, n, pivot, compare);
        quicksort(A, m, p-1, pivot, compare);
        quicksort(A, p+1, n, pivot, compare);
    end if;
end proc:
```

Listing 2: Sorting ascending with pivot last element

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

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
```

Using MapleMIX produces a highly specialized result. 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 percent performance increase. See also [5,4] for a purely generative approach to solving the same problem.

4 Residual Theorems

We now put such a partial evaluator to work. First, one really needs to start with a straightforward example, just to ensure that simple things work absolutely perfectly. For simplicity, we will again use the degree example. To further simplify, we will assume that our polynomials are expanded polynomials in a monomial basis. In the first listing, we see some Maple code to convert from an expression representation into a list representation, and then compute the degree. It is safe to use Maple's built in function `degree`, as this will always return a conservative estimate of the actual degree (i.e. an upper bound). If we test this on a generic second degree polynomial, the residual code is shown in listing 5, and is as expected.

Listing 4: Simple degree function

```
coefflist := proc(p) local d;
    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:
```

Listing 5: PE result

```
goal := proc(a, b, c) local p;
    p := a*x^2+b*x+c;
    mydegree(p, x)
end proc;

result := proc (a, b, c)
    if   a <> 0 then 2
    elif b <> 0 then 1
    elif c <> 0 then 0
    else             -infinity
    end if
end proc
```

Of course, we are more interested in larger examples. We have one from Linear Algebra (the ever-popular Gaussian Elimination), as well as one taken from Calculus (indefinite integration). Both of these cases share some aspects in common: the input code is quite straightforward[5], and the results show how useful it is to consider a *code* representation for the output.

Listing 6 presents a simple Maple implementation of fraction-free Gaussian Elimination for augmented matrices represented as Maple tables.

Listing 6: Fraction-free Gaussian Elimination

```
GE := proc(AA, n, m) local B,i,j,k,r,d,s,t,rmar,pivot,ii;
    B := table(); # make a copy
    for ii to n do for j to m do B[ii,j] := AA[ii,j] end do end do;
    rmar := min(n,m); s := 1; d := 1; r := 1;
    for k to min(m,rmar) while r <= n do
        # Search for a pivot element.  Choose the first
        pivot := -1;
        for i from r to n do
            if (pivot = -1) then
                if (B[i,k] <> 0) then
                    pivot := i;
                end if;
            end if;
        end do;
```

[5] Although we admit that it has been adapted to the current feature set of our implementation

```
        if pivot>-1 then # interchange row i with row r is necessary
            if pivot <> r then
                s := -s;
                for j from k to m do
                    t := B[pivot,j];
                    B[pivot,j] := B[r,j];
                    B[r,j] := t
                end do;
            end if;
            for i from r+1 to n do
                for j from k+1 to m do
                    B[i,j] := (B[i,j]*B[r,k]- B[r,j]*B[i,k])/ d;
                end do;
                B[i,k] := 0;
            end do;
            d := B[r,k];
            r := r + 1    # go to next row
        end if;
    end do;  # go to next column
    eval(B);
end proc:
```

Listing 7 shows the result we get on the Matrix from [2]. The code has been hand-formatted to display the Matrix entries as clearly as possible. Admittedly, this code could be simplified further through more inlining since for example we know that `B1[2,2] = 2` so that the first condition could read `x + 4 <> 0` or even `x <> -4`. However, as we expect from [2], we clearly see the 3 cases to consider: $x = -4$, $x = 0$ and the generic ("symbolic") case.

It is quite gratifying to see that from a simple implementation of Gaussian Elimination, given a particular Matrix with parametric entries, the cases to consider naturally "drop out" of the computation. Unlike [2], new ideas or algorithms are not necessary to achieve this result, just a powerful-enough partial evaluator. Of course if one is worried about efficiency, our method is very far from competitive, and will likely never be competitive with a specialized algorithm.

Listing 7: Result on parametric Matrix

```
goal := proc(x) local A;
    A := table([ (1,1) = 1, (1,2)=-2, (1,3)=  3, (1,4)=1,
                 (2,1) = 2, (2.2)= x, (2,3)=  6, (2,4)=6,
                 (3,1) =-1, (3.2)= 3, (3,3)=x-3, (3,4)=0]);
    GE(A, 3, 4);
end proc:

result := proc(x) local B1;
    B1[2,2] := x;
    B1[3,3] := x - 3;
    B1[2,2] := B1[2,2] + 4;
    B1[3,3] := B1[3,3] + 3;
    if B1[2,2] <> 0 then
        B1[3,3] := B1[3,3] * B1[2,2];
```

```
        B1[3,4] := B1[2,2] - 4;
        if B1[3,3] <> 0 then
            B1[1,1] := 1; B1[1,2] := -2; B1[1,3] := 3; B1[1,4] := 1;
            B1[2,1] := 0;                B1[2,3] := 0; B1[2,4] := 4;
            B1[3,1] := 0; B1[3,2] := 0;
            eval(B1)
        else
            B1[1,1] := 1; B1[1,2] := -2; B1[1,3] := 3; B1[1,4] := 1;
            B1[2,1] := 0;                B1[2,3] := 0; B1[2,4] := 4;
            B1[3,1] := 0; B1[3,2] := 0;  B1[3,3] := 0;
            eval(B1)
        end if
    else
        B1[2,3] := B1[3,3];
        B1[1,1] := 1; B1[1,2] := -2; B1[1,3] := 3; B1[1,4] := 1;
        B1[2,1] := 0; B1[2,2] := 1;                B1[2,4] := 1;
        B1[3,1] := 0; B1[3,2] := 0;  B1[3,3] := 0; B1[3,4] := 4;
        eval(B1)
    end if
end proc
```

Listing 8 shows a bit of code that we may expect to find somewhere in a symbolic integrator. While actual integration code tends to be more complex, we hope that the code below is representative enough to illustrate our point. This code takes in 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 8: 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 9: 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;
    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.

5 Conclusion

Through a variety of pertinent examples, we have shown that current computer algebra code already contains the information necessary to produce correct results for all cases of parametric problems. In other words, it would appear that the *specialization problem* is not quite as thorny as it was originally thought to be, and could turn out to be tractable. To obtain these results, we have had to switch from an *expression* perspective to a *code transformation* perspective, and more specifically to partial evaluation. This switch has been quite effective, and we feel that this is just the beginning.

It is worthwhile pointing out that certain partial evaluation techniques were crucial in obtaining these results. First, being able to treat *partially static structures* was essential. While in traditional partial evaluation this seems to be uncommon, this is completely natural in a symbolic computation setting, as a "symbolic expression" is really isomorphic to a partially static expression. Second, to properly propagate conditional information through *computations* instead of just through syntactic code, we have employed a state-passing technique (related but different

from Continuation Passing Style) inspired in part by Sumii's work [13]. Third, the use of M-form, a richer language instead of the more usual sub-language approach, allows the partial evaluator to be more precise. And lastly, the use of an (aggressively) online approach has also helped. We hope to report on these techniques in a forthcoming publication.

Obviously, there are some Maple features that MapleMIX does not currently support (most notably arrays) that we would like to implement. We would then like to apply our work to pieces of the Maple library; one obvious area is to enable generic programming (as in part embodied by Maple's `Domains` package) to be efficient. In this direction, our results with Quicksort parametrized via higher-order functions are quite promising.

During this work, it has become quite clear that some level of type inference could really improve the precision of MapleMIX for larger examples. Being able to tell if a function is pure (i.e. has no side-effects), if a variable can never be an expression sequence, and so on would allow MapleMIX to be even more precise. Also, figuring out how some optimizations (like dead code elimination, single-use variable folding, etc) could be integrated directly into the specialization phase would be interesting.

In conclusion, we have shown a novel use of partial evaluation: it can be used to "mine" code for residual algorithms that work properly and correctly on parametric problems. We have used the moniker "Residual Theorem" for the results we obtain from residualizing a "generic" algorithm with respect to a parametric problem, as the results we thus obtain really encodes the complete answer (i.e. a theorem) to the original parametric problem.

References

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

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

[3] Jacques Carette. Understanding expression simplification. In *ISSAC '04: Proceedings of the 2004 international symposium on Symbolic and algebraic computation*, pages 72–79, New York, NY, USA, 2004. ACM Press.

[4] Jacques Carette. Gaussian elimination: a case study in efficient genericity with metaocaml. *Science of Computer Programming*, 2006. accepted.

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

[6] Niels H. Christensen and Robert Gluck. Offline partial evaluation can be as accurate as online partial evaluation. *ACM Trans. Program. Lang. Syst.*, 26(1):191–220, 2004.

[7] Daniel Elphick, Michael Leuschel, and Simon 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] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International(UK) Limited, 1993.

[9] Michael Kucera. Partial evaluation of maple programs. McMaster University, May 2006.

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

[11] Ulrik P. Schultz, Julia L. Lawall, and Charles Consel. Automatic program specialization for java. *ACM Trans. Program. Lang. Syst.*, 25(4):452–499, 2003.

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

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

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

[15] Daniel Weise, Roland Conybeare, Erik Ruf, and Scott Seligman. Automatic online partial evaluation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, June 1991.