Partial Evaluation of Maple Programs

By
Michael Kucera, B.Sc.

A Thesis
Submitted to the School of Graduate Studies
in Partial Fulfilment of the Requirements for the Degree of

Master of Science

Department of Computing and Software

McMaster University

MASTER OF SCIENCE (2006) McMaster University
(Computing and Software) Hamilton, Ontario

TITLE: Partial Evaluation of Maple Programs

AUTHOR: Michael Kucera, B.Sc.(Ryerson University)

SUPERVISOR: Dr. Jacques Carette

NUMBER OF PAGES: 1, 104

This work is dedicated to my wife, Kate.
Her love and support helped me through the long task of writing this thesis.

Abstract

Partial Evaluation (PE) is a program transformation technique that generates a specialized version of a program with respect to a subset of its inputs. PE is an automatic approach to program generation and meta-programming. This thesis presents a method of partially evaluating Maple programs using a fully online methodology.

We present an implementation called MapleMIX, and use it towards two goals. Firstly we show how MapleMIX can be used to generate optimized versions of generic programs written in Maple. Secondly we use MapleMIX to mine symbolic computation code for residual theorems, which we present as precise solutions to parametric problems encountered in Computer Algebra Systems.

The implementation of MapleMIX has been modularized using a high-level intermediate language called M-form. Several syntax transformations from Maple to M-form make it an ideal representation for performing program specialization. Many specialization techniques have been explored including a novel online approach to handle partially-static data structures and an on-the-fly syntax transformation technique that propagates loop context into the body of dynamic conditionals.

Acknowledgements

I would like to thank Professor Jacques Carette for providing direction, supervision and guidance. He always provided me with a huge amount of encouragement and inspiration. This thesis would never have been possible without him.

I would also like to thank Professors Emil Sekerinski and Wolfram Kahl for being members of my committee and excellent teachers. My grasp of Computer Science has multiplied exponentially under their tutelage.

Of course I must thank my lab mates: Tim Paterson, Olivier Dragon, Steve Forrest and Greg Herman for asking questions, listening to my lame jokes and letting me talk on and on about my problems.

Thanks to my parents and grandparents for their encouragement and for always believing in me. And finally, thanks to my long time friends for being the best friends a person can have.

Table of Contents

List of Figures

Chapter 1

Introduction

There are two conflicting goals in the practice of computer programming that must live side by side. On one hand programmers wish to use high levels of abstraction and generality in order to create programs that are readable, modular, reusable, portable and maintainable. On the other hand it is necessary to write programs that are efficient. These two goals are in conflict; abstraction usually adds overhead and reduces the chances for optimization. Meanwhile the need for efficiency pulls the programmer away from abstraction and promotes the use of platform and algorithm specific optimizations. Compiler optimization is a starting point for reducing overhead involved in writing programs in a high level language. Compiler optimization techniques are generally concerned with removing redundancy within source code and generating optimal code for a particular architecture. They will not however generate several optimized versions of a particular algorithm based upon different usages of the algorithm, or based upon design decisions that were abstracted from when the algorithm was written. This type of optimization is in the realm of partial evaluation.

Partial evaluation (PE) is a program transformation technique that uses a subset of the inputs to a program to generate a specialized version of the program that will then accept the rest of the inputs. Partial evaluation performs aggressive optimizations by performing as much computation as possible based on the given inputs while generating code that will perform the deferred computations. Several specialized versions of a program section may be generated based upon different usages within the program. An optimized program generated by a partial evaluator may be larger in size than the original program but will perform less computations at run-time.

1.1 Maple

Maple is a Computer Algebra System (CAS) and visualization environment. It uses a high-level language, which is also called Maple, to perform symbolic and numeric computations. Its core library contains a vast amount of mathematical knowledge and problem solving power. Two problems of interest arise. Firstly since Maple is high-level there are many opportunities for improvement in terms of efficiency. Secondly symbolic computation code written in Maple contains a great deal of embedded knowledge and we wish to be able to extract that knowledge. Partial evaluation is used toward both of these goals.

1.2 Thesis Aims and Outline

The main contribution of this work is a partial evaluator for Maple called MapleMIX. It is a program transformation and optimization tool for Maple programs. MapleMIX is a completely online partial evaluator written in Maple itself. It has the ability to produce highly specialized results from input programs and user provided static data. We have used MapleMIX to optimize Maple code and to extract precise answers to parametric problems from Maple programs written to produce generic solutions. Several standard approaches have been used in the design of MapleMIX with the intent of using it for practical situations encountered in Computer Algebra Systems. As such we have concentrated on real-world solutions and so have not considered theoretical results such as self-application or generation of generating extensions. However we have developed many new techniques toward the design of practical online partial evaluators such as: a unique design for the online environment, an online approach to partially static data structures, a novel technique for handling dynamic conditionals within loops, an approach to program transformation that uses higher level constructs than those available in the input language, and a sound way of modularizing the internal structure of MapleMIX.

This thesis is organized as follows. In Chapter 2 we provide an introduction to the Maple programming language and its features of interest to this work. Chapter 3 provides an exposition of partial evaluation, examines current work and theoretical ideas and explains our choices in direction. Chapter 4 describes the high-level architecture of MapleMIX and the various syntax transformations that are performed to facilitate specialization. Chapter 5 explains expression reduction. Chapter 6 goes into detail

about the specialization of statements and the design of the online environment. In Chapter 7 several examples and results are presented. Chapter 8 discusses future work and draws conclusions. The appendix provides a reference for certain technical information.

Chapter 2

Overview of Maple

Maple is a computer algebra problem-solving and visualization environment. It was originally developed by the University of Waterloo's Symbolic Computation Group (SCG) where the Maple project was started in 1980 [1]. Waterloo Maple Inc was created in 1988 for the purpose of refining and commercializing the Maple system as well as to provide support. Today Maple has an estimated user base of over three million people [2].

Maple's support for mathematics is enabled through its large library of symbolic and numerical routines. At the core of the system is Maple's built-in programming language which is also commonly referred to as Maple. Most of the library is written in Maple with only a small set of routines implemented directly in the Maple kernel. Maple is a dynamically typed, interpreted language that supports both imperative and functional features. For the purposes of this thesis we only need to consider a subset of the Maple language. MapleMIX was written for and tested with Maple version 10.

2.1 The Maple System

The Maple system consists of three parts:

[1] http://www.scg.uwaterloo.ca/SCG/history.html
[2] http://www.nserc.ca/news/2004/p041028bio.htm

- At the center of the Maple system is the kernel. It contains the Maple programming language interpreter, the automatic simplifier, built-in arithmetic routines and the memory manager. It is written in the C programming language to be small, efficient and portable.
- Built on top of the kernel is the Maple core library. This is where the majority of Maple's vast mathematical functionality resides. The Maple library is written in the high-level Maple programming language and can be viewed and modified by users.
- The user interface, which comes in both command-line and GUI varieties. The common worksheet mode is essentially an interactive interpreter prompt.

The Maple language has the following main characteristics:

- **Dynamic type system.** All type checking occurs at runtime and variables can be assigned a value of any type. Maple's extensible runtime type assertion mechanism provides an easy and powerful way for the programmer to make explicit type checks.
- **Imperative.** Characterized by global and local state, statements, side-effects and loops.
- **Functional features.** Language support for higher order functions and closures. Library support for partial application, function composition and pattern matching.
- **Symbolic.** All identifiers are called names or symbols and do not have to be declared. Any name can be used as a value without being initialized. Any unassigned name will evaluate to itself.

Maple is a very large and complicated language that has evolved over a long period of time. It is often possible to find multiple ways of doing the same thing. In this chapter we will only be concerned with the language features that are of interest to the topic of this thesis. We do not wish to go into detail about the syntax of Maple and instead informally describe its features and provide examples. For a detailed discussion of the Maple language refer to [20] and [19].

2.2 Statement Forms

Maple has the following statement forms: assignment, conditionals, return, loops, try/catch and error. Additionally any expression may be placed on a line by itself, this will be referred to as a *standalone expression*. The statement separators in Maple are semicolon (`;`) and colon (`:`). The Maple worksheet interface will not display the results of a statement that is terminated by a colon.

2.2.1 Assignment

Most common is the standard form of assignment of an expression to a name. In addition multiple variables can be assigned at once. In fact Maple accepts any non-empty sequence of names as an L-value, including sequences of names that are dynamically built. Assignment to table indices and record fields is also supported. A name can be unassigned by assigning it to itself. (`>` is the Maple worksheet prompt.)

```
> x := 3;
                                  x := 3
> (y, z) := (9, 10);
                               y, z := 9, 10
> a || (1..5) := 2, 4, 8, 16, 32;
              a1, a2, a3, a4, a5 := 2, 4, 8, 16, 32
```

2.2.2 Conditional statements

If statements consist of at least one conditional expression followed by arbitrarily many `elif` branches and an optional `else` branch. Additionally Maple has a functional style if expression similar to the ternary `?:` operator found in languages like C and Java.

```
if a = b then
    x := 1;
elif c >= d then
    x := 2;
elif e < f then
    x := 3;
else
    x := 4;
end if;
```

```
`if`(y = z, expr1, expr2);
```

2.2.3 Return

A `return` statement can be used to explicitly exit the currently executing function at any point and return a value. All Maple functions return a value, void functions can be simulated by returning the special value `NULL`. If a `return` statement does not have an argument then `NULL` will be returned. A `return` statement is not explicitly required to return a value from a function. When a function runs off the end without a `return` then the last evaluated expression will be returned. It is considered good style to make use of this implicit return mechanism when possible.

```
> double := proc(x)
      x * x
  end proc;
```

2.2.4 Loops

Maple provides two kinds of loop, `for-from` loops and `for-in` loops. A `for-from` loop iterates an index variable over a range a values, a `for-in` loop iterates over a linear data structure such as a list or set. Additionally a `while` condition may be attached to a loop, the `while` condition is checked before each iteration and the loop is terminated when it evaluates to false. Most of the clauses of a loop definition may be left blank in which case default values are assumed. A standard `while` loop can be simulated by a `for-from` loop with all clauses left blank except the `while` clause.

```
for i from 2 to 10 by 2 while x < 100 do
    x := x * i;
end do;

for i in [1,2,3,4] do
    print(i);
end do;

while x < 20 do
    ...
end do;
```

Within a loop body the `next` keyword will skip the rest of the current iteration. The `break` keyword exits the loop entirely. Additionally a `return` within a loop will exit the loop.

2.2.5 Try/Catch and Error

The `error` statement explicitly triggers an exception to occur. Exceptions may also be triggered by erroneous computations such as attempting to perform an operation on a type that does not support the operation. An exception is represented as a string, which according to convention usually describes the nature of the exception. When an exception occurs, the execution of the current statement sequence is interrupted. Control is passed to the appropriate matching clause in the enclosing `try/catch` statement. If there is no matching clause then the exception is automatically re-raised at the next enclosing `try/catch`. This process continues until a matching `catch` is found or the exception propagates all the way to the top level. If the exception is unhandled, it becomes an error; the message is displayed and the current execution terminates.

A `try` may have several `catch`es. Matching of `catch` clauses is done based on the exception string. Each `catch` clause may declare a catch string; if the catch string is the prefix of the exception string then a successful match has been found. A `finally` clause may also be declared which is always executed before control leaves the `try` statement.

```
try
    FileTools:-MakeDirectory(dirName);
    lprint("lib directory created");
catch "directory exists" :
    print("directory ", dirName, " already exists");
end try;
```

2.3 Expressions and Data Types

Complex expressions can be built by combining constants, operator-based expressions, data structures and function calls. Maple, being a computer algebra system, has a plethora of numeric and algebraic data types, operators and functions. Many data structures are available; the most common ones are described here.

2.3.1 Names

A name is a sequence of characters that uniquely identifies a variable. A name that has not been bound to a value is called a symbol. Names are first class citizens in Maple; they can be passed to functions and bound in a different scope than where they were first created. Names are usually global unless explicitly defined as local to a particular procedure or module scope. Sometimes when a name is bound it is still desired that the name evaluate to itself instead of evaluating to its bound value. This can be achieved through the use of unevaluation quotes (') around the name.

```
> x;
                                    x
> x := 9;
                                x := 9
> x;
                                    9
> x := 'x';
                                x := x
> x;
                                    x
```

Maple is a symbolic language, meaning that any unassigned name simply evaluates to itself. This is in contrast to many common dynamically typed languages where normally the presence of an unassigned name will trigger an error. A very common mistake in Maple is to misspell an identifier. However when execution reaches the misspelled identifier Maple will not issue an error, instead the identifier evaluates to itself and execution continues. This very common type of programming error can be difficult to track down, as the error may not manifest itself in the same place that it was caused. There is no way to tell Maple to treat unassigned names as errors or to force all variables to be declared.

2.3.2 Tables

The table data structure is a special object for representing a finite map between keys and values. Tables are implemented directly in the kernel and are very efficient. Many other data structures use tables in their implementation, such as matrices and arrays. In Maple it is very common to program with tables and to use them as part of design patterns. In addition to tables Maple provides many other useful data structures including lists and sets.

```
> t := table([1 = "a", 2 = "b"]):
> t[3] := "c":
> t[1], t[2], t[3];
                                    "a", "b", "c"
```

2.3.3 Expression Sequences

A sequence is a group of expressions separated by commas. Sequences are generally used to form sets, lists and in calling functions. When sequences are nested they are automatically flattened. The special name `NULL` represents the empty expression sequence.

```
> s := 2, 3;
                                    s := 2, 3
> 1, s, 4, 5;
                                  1, 2, 3, 4, 5
> numlist := [1, s, 4];
                              numlist := [1, 2, 3, 4]
```

2.3.4 Functions

Maple has higher-order functions, meaning a function is a value that can be treated as any other value. Functions can be passed to other functions as arguments, returned from functions or stored in tables. Every Maple function takes one expression sequence as its argument. Named parameters are matched up to the elements of the sequence. The standard calling mechanism is *call by evaluated name*. Each argument is evaluated then each parameter in the body of the function is substituted by its corresponding argument. (Duplication is avoided since a Maple program is represented internally as a DAG [3]). Parameter passing is quite flexible. Each parameter specification may include a type assertion and possibly a default value. Additionally special optional parameters may be defined that can only be given an explicit value using a special calling scheme. The terms "procedure" and "function" mean the same thing in Maple and are often used interchangeably.

[3] Directed Acyclic Graph

```
> p := proc(x, y::integer, z::integer := 0, {verbose:=false}) local result;
>     # error if y not an integer
>     result := x + y + z;
>     if verbose then
>         print(result);
>     end if;
>     result
> end proc:

> p(1, 2, verbose=true); # using default value for z
                                        3
```

Additionally special expression keywords are available in the body of the function; the `args` keyword refers to the entire expression sequence of arguments used in the function call, `nargs` evaluates to the number of arguments (equivalent to `nops([args])`). Procedures may be defined using the "arrow" syntax. In this case the body of the procedure must be an expression.

```
double := x -> x * x;
```

2.3.5 Special Functions

Many special functions with non-standard calling semantics exist, usually defined as built-ins. For example Maple provides a function called `seq` that is used as an expression sequence comprehension. This function does not evaluate its arguments before the function call, instead the first argument is evaluated multiple times according to the second argument.

```
> seq(i*2, i=1..5);
                                  2, 4, 6, 8, 10
```

The `assigned` function is used to test if a name or a table cell has been assigned to a value. It is used very often with tables to test if a mapping exists before doing a lookup.

```
> assigned(t[1]);
                                     false
> t[1] := "a";
                                  t[1] := "a"
> assigned(t[1]);
                                     true
```

2.3.6 Closures

It is possible to describe Maple as having many of the common features that are associated with functional programming languages, including closures. The need for function closures arises naturally when nested functions and higher-order functions are allowed. A function closure can be described as a pair consisting of the code of the function and an environment that stores values of the variables defined in the function's surrounding lexical scope. Since functions are values they may escape the scope in which they are defined. Closures maintain the state of the lexical environment in which the function was defined.

The following example demonstrates the use of closures to write clean and compact code. The procedure `newCounter` returns an inner procedure that will act as a counter. The inner procedure references the variable `x` which is defined in the scope of the outer procedure.

```
newCounter := proc() local x;
      x := 0;
      return proc()
          x := x + 1;
          return x;
      end proc
end proc:
```

Counter "objects" can then be created and used. There is no way to access the state of the closure from outside as closures represent private state. Closures allow some object oriented patterns and designs to be used in Maple even though Maple is not considered an object oriented language.

```
> count1 := newCounter(): count2 := newCounter():
> count1();
                                    1
> count1();
                                    2
> count1();
                                    3
> count2();
                                    1
> count2();
                                    2
```

In a lexically [4] scoped language it is possible to implement scopes using a stack. Every

[4] Lexical scoping gets its name from the fact that it is possible to determine the scope of a name

time a scope is entered a new environment is pushed onto the stack, and when the scope is exited the environment is popped. A closure then simply contains one or more pointers to the outer lexical environments that it references on the stack. When one of these environments gets popped it is still kept alive by any closures that have references to it. This is a simple scheme in an implementation that supports garbage collection. As soon as there are no more references to an environment it becomes available for garbage collection.

2.3.7 Modules

Modules are similar in functionality to procedures and closures but they differ in two main respects. Firstly modules can contain locals which act similar to variables defined in a function's closure and module locals may be exported. Second, modules act like a record, allowing several procedures and variables to be treated as a unit. The syntax of module definition provides a clean way to declare the interface and the intent of the module.

It is possible for several functions to share the same closure environment and to be used together to access and update it. This is shown in the following example of a counter that can be reset.

```
newResetCounter := proc() local x;
     x := 0;
     count := proc()
          x := x + 1;
          return x;
     end proc;
     reset := proc()
          x := 0;
     end proc;
     return count, reset;
end proc:

> (count, reset) := newResetCounter():
> count();
                                1
> count();
                                2
> reset(): count();
                                1
```

by examining the program text. Lexical scoping is also known as static scoping.

In the above example two procedures are returned by `newResetCounter` which must be independently maintained. Below is an example of a reset counter generator implemented using a module. The module must still be nested within a "constructor" procedure in order to be able to generate many instances of the module. The special module procedure `ModuleApply` allows a module name to be used like a procedure. Module exports are accessed using the module member selection syntax (`m:-e`).

```
newResetCounterModule := proc()
    module()
        description "a counter that increments in steps of one";
        local x; # private
        export ModuleApply, reset; # public
        x := 0;
        ModuleApply := proc()
            x := x + 1;
            return x;
        end proc;
        reset := proc()
            x := 0;
        end proc;
    end module;
end proc;

> c := newResetCounterModule():
> c(); # calls ModuleApply
                                1
> c();
                                2
> c:-reset():
> c();
                                1
```

2.4 Additional Features

2.4.1 Automatic Simplification

Maple automatically simplifies expressions, for example by simplifying $a + a$ to $2a$. Automatic simplification takes place after parsing, so it is common for Maple to display an expression slightly differently from how it was typed in. Automatic simplification is also applied to function bodies and in specific cases can cause code to be removed. For example a while loop where the condition is hard-coded to `false` will

be removed entirely. This feature sometimes misleads people to think that Maple has some kind of automatic partial evaluation system built in, but this is not the case.

2.4.2 Last Name Evaluation

Specific evaluation rules apply to certain types of expressions. Most Maple types are evaluated using standard full recursive evaluation. For example, names that are assigned to integers are subject to full evaluation.

```
> x := 20;
                                  x := 20
> y := x;
                                  y := 20
> y;
                                    20
```

However some special types are not subject to full evaluation. Instead they are evaluated to the last name to which they were assigned. This is known as the Last Name Evaluation (LNE) rule. Procedures, modules and tables are all subject to this special rule. The `eval` function can be used to fully evaluate an expression.

```
> t := table([1="a"]);
                        t := table([1 = "a"])
> t;
                                    t
> t[1];
                                  "a"
> s := t;
                                  s := t
> s;
                                    t
> s[1];
                                  "a"
> eval(s);
                          table([1 = "a"])
```

2.4.3 Type Assertions

The Maple language is dynamically typed, meaning that no attempt is made at statically resolving the types of terms. In order for the programmer to ensure some

level of type safety, Maple does provide an explicit runtime type assertion system. The type of an expression can be checked by using the built-in `type` function or in certain contexts by using the double colon (`::`) operator. A type is either a structured type definition or a predicate that returns true for values of the type. Structured type definitions are built up by combining simpler, more primitive types. A type predicate is given by a boolean valued "slash" function of the name `'type/T'` where `T` is the name of the type. Since a type is just a predicate many interesting types are available including `even`, `prime`, `local`, `global`, `expanded` and many more. There is no well defined subtyping relation.

2.4.4 Reification and Reflection

In order to apply program transformations and to be able to execute programs after they have been transformed we require two important operations; *reification* and *reflection*. Reification is the process by which a part of a program is made accessible to the program itself as a data structure. Reflection is the opposite operation, allowing a data structure representation to be made into a runnable program. Reflection is the process by which a program may alter its own structure and behavior. Interpreted programming languages are ideally suited to support reflection because a representation of the source code is available to the interpreter.

Reification is performed in Maple by using the `ToInert` function and reflection using `FromInert`. `ToInert` takes a maple expression and returns an abstract syntax tree representation, referred to as the *inert form* of the expression. This abstract syntax representation is required for symbolic manipulation.

```
> inrt := ToInert(x+y);
                inrt := _Inert_SUM(_Inert_NAME("x"), _Inert_NAME("y"))
```

`FromInert` takes a valid inert form and converts it into an active Maple object that may be further evaluated.

```
> e := FromInert(_Inert_SUM(_Inert_NAME("x"),_Inert_NAME("y")));
                                        e := x + y
> x := 1; y := 2;
                                          x := 1
                                          y := 2
> e;
                                            3
```

The original purpose of `ToInert` is to convert an active representation into an inert one that can be freely modified without worry of evaluation. The inert form is essentially an external representation of a data structure that is normally internal to the interpreter. Naturally `ToInert` and `FromInert` are implemented as built-ins.

Chapter 3

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. The resulting program is called the *residual program* or *specialized program*. Essentially PE 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 known as *static*. 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 in times 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 should be highly optimized. Consider the following example of a function named `pow` written in Maple that computes x^n for $n \geq 0$.

```
pow := proc(x, n)
    if n = 0 then
        1
    else
        x * pow(x, n-1)
    end if
end proc
```

Running MapleMIX on `pow` with respect to $n = 5$ yields the following residual program. (The multiplication by 1 was removed by the automatic simplifier.)

```
pow_s := proc(x)
    x * x * x * x * x
end proc
```

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. Hence we have automatically derived a function that computes x to the power of fixed value. Several different versions of `pow` computing x to the power of different values could be automatically generated.

Traditionally PE has been done for declarative expression oriented languages such as Scheme [16, 28, 11, 27] , ML [5, 12], Prolog [16] and the lambda calculus [26, 23, 16, 18]. However there does exist partial evaluators for imperative languages such as C [1, 10], Java [30] and Matlab [13]. Most early work was done in the context of Scheme with the emphasis on writing self-applicable partial evaluators.

Most partial evaluators only support a subset of the language they were designed for. Having said that, it is worth noting that the PGG system for Scheme supports the entire language including imperative style code [27]. Scheme as a language is a good test bed for PE research because it has a simple semantics, is dynamically typed and is symbolic. Perhaps if PE were considered during the early stages of programming language design it would become a more widely available tool.

3.1 Mix Equation

A partial evaluator must satisfy the *mix equation* [16].

$$[\![p]\!](s, d) = [\![([\![PE]\!](p, s))]\!](d)$$

The partial evaluator produces a residual program $p^s = [\![PE]\!](p, s)$ from the subject program p and the static inputs. When the residual program p^s is run on the remaining dynamic inputs the results should be equal to running the subject program p on all its required inputs $[\![p^s]\!](d) = [\![p]\!](s, d)$. This means that the partial evaluator PE

must preserve the semantics of p.

3.2 Partial Evaluation Strategies

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. Instead they depend on a preprocess known as binding time analysis (BTA) to gather the necessary binding time information. The remove/residualize decision is made during this preprocess. A BTA precomputes a *division*, a classification of sections of the subject program according to binding time. The division is usually embedded into the subject program via annotations, then the specializer simply obeys the division (and possibly the results of other prephase analyses). Traditionally BTA has come in two forms. One method is abstract interpretation over an abstract value domain consisting of the possible binding times [16]. The other approach is to use a type inference system to build a set of constraints on the binding times of program variables. The constraint set is then solved giving the binding time of each variable [15]. In either case BTA is essentially a worst case analysis.

A common approach to offline PE is to define a two level syntax for the language that adds annotations for static and dynamic. The partial evaluator is then defined as an extended semantics of the two level syntax. Constructs annotated with static use the standard language semantics and constructs annotated with dynamic use the extended semantics. It is believed that offline partial evaluators are easier to write because the partial evaluator is split into two phases yielding good separation of concerns [29]. Offline partial evaluators are currently the only choice for effective self-application [16].

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. The degree of staticness of a variable is directly related to the amount of information known about the variable. If a concrete value is available then the variable is static. If only limited information is available such as the type or shape of the variable, or if the variable represents a compound value for which only certain elements are known, then it is partially static. If there is no information then it is dynamic. It is

believed that online partial evaluation is more precise than offline leading to better specialization [16, 9]. However one must realize that there is no preprocess, therefore the online partial evaluator is very reactive in its nature because while it is computing it has no prior knowledge of program points it is yet to encounter.

For example consider the following program construct annotated with s and d for static and dynamic, assume the variables can only contain integer values.

$x^d := a^d * b^s$

The variable a is dynamic therefore BTA would be forced to annotate x as dynamic. Under the offline methodology this statement would always be residualized. An online PE however would have the value of b available when making this decision. In the case where b is 0 the value of a does not matter, value of x could be computed to be 0 and the statement could be removed. 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 [21].

Online PE can be more precise than offline, however online specializers are usually less efficient mainly due to 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. For example Sumii showed how a type-based representation analysis could improve the performance of an online PE by adding a limited amount of binding time information to optimizes the specialization process by removing unnecessary computations [24].

A specializer is *monovariant* if the divisions it computes can contain at most one classification of binding times for the parameters of a function. The consequence of this approach is that the most dynamic usage of each parameter will become the binding time for all usages of that parameter throughout the subject program. This should be considered a severe restriction, because it is possible for some static information go unexploited. In contrast, *polyvariant* specialization can produce more than one binding time classification for a particular function's parameters, leading to better specialization. A result of this approach is that it is possible for the residual program to be larger than the subject program. This is accepted as long as the amount of computations required for the residual program to arrive at a result is reduced. It is also possible to generalize the concept of *variance* to deal with a wider class of program points instead of just functions.

3.3 Parameterization

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.

Parameterization allows the creation of generic code that can have its behavior controlled by "functional" parameters. For example, the Java language provides a sorting method that can sort any `List` object according to the natural ordering of its elements. However, it is also possible to supply a `Comparator` object as a additional parameter to the sort method. The `Comparator` provides a custom method used to compare elements in the `List`. This pattern is both flexible and extensible. Sorting behavior (i.e. sorting ascending or descending) can be customized. New user defined data types can be sorted by the existing sort function by providing a new `Comparator`. Thus there is only the need for a single sorting function in the Java core library.

Despite obvious advantages to parameterization there is a major disadvantage, reduced efficiency. Often in order to write optimized code it is necessary to manually create specialized versions of an algorithm for specific types of inputs. Gaussian Elimination (GE) is an example of an algorithm that has the potential to be highly parameterized. In practice however writing a generic GE procedure has shown to result in slow and bloated code. Currently in the Maple library there are at least 35 different implementations of GE each exposed through a specialized interface [7].

3.4 Multi-Stage Programming

Multi-stage programming (MSP) is a new paradigm designed specifically for the reliable creation of programs that generate programs. MSP allows for generic programming while minimizing the runtime cost that naturally comes along with it. Multi-stage programming is a special case of meta-programming. The key to MSP is a type system that ensures that both the generic program and the dynamically generated programs are type-safe. MetaOCaml [1] is an MSP extension to the OCaml functional programming language [25]. Three new constructs are added to OCaml.

[1] http://www.metaocaml.org/

- **Brackets** (`.<` *expr* `>.`) When inserted around an expression will delay its execution. Creates an expression of type *code*.
- **Escape** (`.~` *expr*) Allows one code fragment to be included within another. In this way larger code fragments and ultimately the residual program is built.
- **Run** (`.!` *expr*) Dynamically compiles and executes generated code within the MetaOCaml environment.

A new *code* type is added to distinguish delayed computations from other values. MSP allows fine grained control over the code generation process while guaranteeing that generated code is syntactically and type correct. Also MetaOCaml tends to be quite efficient at generating code and dynamically executing it. Let us consider an example of a staged powering function written in MetaOCaml.

```
(* power :: int code -> int -> int code *)
let rec power x n =
    if n = 0
        then .<1>.
        else .< .~x * .~(power x (n-1))>.
;;
```

We can now use this staged version of power to generate a specialized version.
(`#` is the MetaOCaml prompt).

```
# .<fun x -> .~(power .<x>. 3)>. ;;
- : ('a, int -> int) code =
.<fun x_1 -> (x_1 * (x_1 * (x_1 * 1)))>.
```

MSP and PE can both be used to reduce the computational overhead involved in writing generic algorithms. However these two approaches differ greatly in one respect. MSP is a programming paradigm and therefore the burden of generating correct residual programs is on the shoulders of the programmer. In particular the multi-stage programmer must determine which parts of the program are static and dynamic, ensure good binding times, avoid code duplication, explicitly control inlining and may have to write a program in a non-intuitive manner in order to achieve these requirements. MetaOCaml is purely generative, meaning that generated code

cannot be manipulated or post-processed in any way. This requires the programmer to "get it right" in the first place [8]. MSP programs are made more complex by the additional language constructs and typing rules. Furthermore the strange syntax of the new MetaOCaml constructs can render its programs difficult to read.

Conversely PE is an automatic technique ideally requiring minimal intervention from the programmer. Design time and specialization time are separated. At most a programmer may need to keep certain considerations in mind to help the PE such as ensuring good termination conditions and keeping good separation of binding times. A PE may have a post-process to "clean up" by removing or compressing residual code. However there is usually a great deal of computational overhead involved with PE.

PE is less restrictive than MSP in that it allows a function to be specialized several ways with respect to different static permutations of its inputs. MetaOCaml's type-system requires that this decision be made at program design time. For example our Maple `pow` function could be specialized in two ways, with x static and n dynamic or with x dynamic and n static. The MetaOCaml `power` function given above can only generate code specialized with respect to n, this is directly apparent from its type. In fact if we try to write the MetaOCaml `power` function with x static and n dynamic we quickly run into problems.

```
(* powerbad :: int -> int code -> int code *)
let rec powerbad x n =
    .<if .~n = 0 then 1 else x * .~(powerbad x .<(.~n-1)>.) >.
;;

#  .<fun n -> .~(powerbad 3 .<n>.)>.;;
Stack overflow during evaluation (looping recursion?).
```

The problem is that the condition that terminates the recursion is dynamic (along with the entire if expression). We have essentially created a strict if expression where both branches are always evaluated.

Partial evaluators also tend to experience termination problems during specialization for the same reason. There has been some success in overcoming these problems, one simple approach is `function sharing`. When the PE encounters a call to an already specialized function or to a function that is currently being specialized it reuses the

specialized body that has already been generated. We will go into more detail about termination issues of PE in chapter 6.

Carette has shown that MSP allows one to write a general Gaussian Elimination algorithm and automatically generate specialized versions using the code generation facilities of MetaOCaml [7]. We will show that PE can also be used toward a similar goal. Partial evaluation bridges the gap between the conflicting goals of parameterization and efficiency by allowing the automatic generation of optimized versions of highly parameterized algorithms.

3.5 The Futamura Projections

Partial evaluation is a very powerful technique that has some interesting properties known as the Futamura Projections, originally discovered by Yoshihiko Futamura in 1971 [14, 16]. The first Futamura Projection states that compilation can be achieved given just a partial evaluator and an interpreter. First one must realize that the inputs to a program written in an interpreted language are actually inputs to the interpreter.

$$[\![source]\!](d) = [\![interpreter]\!](source, d)$$

Then we can achieve compilation of a program written in an interpreted language by specializing the source code of the interpreter with respect to the program, this is stated in the first Futamura Projection.

$$target = [\![PE]\!](interpreter, source)$$

The resulting program *target* is made up of fragments of the interpreter. Intuitively *target* is the interpreter specialized to run only one source program. The power of partial evaluation can be taken even further when we realize that it is possible to apply a partial evaluator to its own source. This of course comes with the requirement that the partial evaluator be written in its own input language or be translatable into that language. The second Futamura Projection states that a compiler can be generated from self-application and an interpreter.

$$compiler = [\![PE]\!](PE, interpreter)$$

In order to help understand this projection it is useful to realize which dynamic inputs are missing from the projection, and to consider what they are. In this case the dynamic input to *compiler* is the *source* program from the first Futamura Projection. Hence the first Futamura Projection compiles a program while the second shows that a compiler can be generated by using self-application to delay the results of the first projection. The third Futamura Projection takes the idea of self-application to the extreme.

$cogen = [\![PE]\!](PE, PE)$

This states that a compiler generator (called *cogen*) can be generated through double self-application. When *cogen* is applied to an interpreter it yields a compiler. The relationship between the three projections can be realized by showing that each projection simply removes one of the static inputs of the previous projection.

$cogen = [\![PE]\!](PE, PE)$
$compiler = [\![cogen]\!](interpreter)$
$target = [\![compiler]\!](source)$

Hence self-application of a partial evaluator simply delays partial evaluation, and double self-application delays partial evaluation twice.

3.6 The cogen Approach

The Futamura Projections are actually more specific than necessary and their scope can be expanded. Consider the second Futamura projection:

$compiler = [\![PE]\!](PE, interpreter)$

The static input does not have to be an interpreter, in fact it can be any program. When self-application of PE is done with respect to an arbitrary program p the result is p^{gen}, the generating extension of p.

$p^{gen} = [\![PE]\!](PE, p)$

A generating extension p^{gen} is a program that when given some of the inputs to p

will output p^s, the specialized version of p. Thus p^{gen} is a program generator.

$$p^s = [\![p^{gen}]\!](s)$$

The specialized program p^s is normally given using the standard vision of partial evaluation given by the mix equation.

$$p^s = [\![PE]\!](p, s)$$

Thus we can generate p^s by directly applying PE to the program p and its static inputs s, or we can first generate a generating extension p^{gen} and use that with s to generate p^s. This indeed shows that self-application simply delays partial evaluation. It has been shown that using a generating extension to create many specialized versions of a program is more efficient that using a partial evaluator directly [26]. Now consider the third Futamura projection:

$$cogen = [\![PE]\!](PE, PE)$$

A compiler is a program generator, therefore a compiler generator is a program generator generator, and this is exactly what is achieved through double-self-application. Thus a simple renaming is at hand.

$$PGG = [\![PE]\!](PE, PE)$$

Running PGG on program p produces the generating extension of p.

$$p^{gen} = [\![PGG]\!](p)$$

If a program p must be specialized several times with respect to different static inputs then first generating the generating extension p^{gen} and using that to generate the specialized versions of p may yield performance gains [26].

In practice writing a partial evaluator that is self-applicable has shown to be very difficult to achieve [16]. A self-applicable PE must be written in its own input language and many existing partial evaluators only support a subset of the language they were designed for. The more complex the input language becomes the more complex the partial evaluator must be. However, researchers have noticed that it is possible to write PGG directly by hand. This has become known as the *cogen approach* to partial evaluation (referred to as cogen for historical reasons) [26]. This approach sidesteps the need for self-application. Thiemann showed how an offline cogen PE could be

derived from an interpreter [26] and Sumii showed a similar process for deriving an online cogen PE [24].

The cogen approach has the following benefits:

- PGGs generated through double self-application tend to be bloated and inefficient. The cogen approach is more efficient at generating generating extensions.
- Easier to write than a self-applicable PE. Separation of binding times must be taken into consideration when writing a self-applicable PE. Furthermore the PE must be kept as small as possible, every level of self-application has the potential to expand the size of the residual program.
- There are no constraints on the language that PGG is written in, a self-applicable PE must be written in its own input language. A PGG can be written in a different language than its input and output languages.

The cogen approach has been realized in an offline setting by defining the semantics of annotated syntax. Evaluation of the annotated program under the extended semantics yields a specialized program. With this scheme an annotated program is a generating extension and the BTA is the PGG. However, in order to achieve this, the need for environment manipulation must be removed from the extended semantics. This is achieved through the use of higher order abstract syntax (HOAS) which moves the burden of variable binding to the meta level [26, 12, 23].

The cogen approach is also possible in an online setting. The idea is to replace the standard operators of the language with smarter ones. They will behave exactly the same when applied to static arguments but also have the extended functionality of correctly treating dynamic arguments by returning residual code [24, 2]. For example consider an extended plus operator `+'` that would behave as expected for static arguments but also work with dynamic arguments.

```
(5 +' 4) -> 9
(5 +' d) -> Plus(Int(5), Var("d"))
```

In either case the cogen approach is realized by extending the semantics of the source language to correctly treat values that are dynamic. A program becomes its own generating extension under the extended semantics.

3.7 Type-directed PE

A syntax-directed partial evaluator performs specialization by symbolically manipulating an abstract syntax tree. Type-directed partial evaluation (TDPE) is a different approach [23, 22, 12]. Specialization is performed by first expanding a term into a two-level term which mixes the meta language and the abstract language. This expansion is done according to the type of the term and not its syntactical structure. The two-level term is then reduced using standard evaluation of the meta-language. In theory this is more efficient because the underlying evaluator is used directly, removing the need for symbolic manipulation. However expanding a term into a two-level term is non-trivial and may even be impossible for certain types. The situation becomes even worse for dynamically typed languages because it may not be possible to determine the exact type of a term. This approach to TDPE is unsuited to practical programming languages [23].

3.8 PE Functionality

If a partial evaluator is applied to a subject program with all its inputs static then the residual program generated should simply consist of a single return that gives the static result. This shows that a PE subsumes the functionality of an interpreter. In fact one can view a standard PE as an interpreter with the added ability of generating residual programs. Even with the cogen approach this is still true. A generating extension can be thought of as a compiled program. When applied to all the expected input it will have all the information necessary to compute a final result. Furthermore if a PE is applied to a subject program with no static inputs given then the PE may still be able to perform many optimizations such as constant propagation and function inlining. This shows that a PE can be considered complimentary to an optimizing compiler. In reality a partial evaluator merges the functionality of an interpreter and a compiler. Since it evaluates a program it is an interpreter and since it generates code it is a compiler [2].

Many compiler optimization techniques do not require much intervention on behalf of the programmer. More powerful optimizations may be possible if the programmer were to provide additional information not readily available in the source program such as invariants. A partial evaluator gains extra power over an optimizing compiler since the specialization process is explicitly initiated by the programmer. Further-

more the programmer must supply additionally information to be exploited in the optimizing process. In the case of partial evaluation the extra information is in the form of static inputs. Additionally running a partial evaluator does not come with the guarantee of termination which is standard with virtually all compilers [2].

3.9 Partial Evaluation of Maple

If a partial evaluator is written in its own input language and that language is interpreted, then it may hand off static computations to the underlying interpreter. This is the approach used in this thesis. We present a partial evaluator for Maple called MapleMIX. To our knowledge this is the first attempt at creating a partial evaluator for Maple.

MapleMIX 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 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.
- **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 may be used when writing the partial evaluator.
- **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

[2] Some compilers for languages that support dependent typing generally do not guarantee termination.

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.

Chapter 4

Syntax Transformations and Internal Architecture

In this chapter we present the high level architecture of the Maple partial evaluator MapleMIX. We also present several program transformations that are performed before and after the specialization phase. The internal organization of MapleMIX is motivated by the need to perform these transformations. Dividing MapleMIX into several modules that communicate via syntax representations allows for a highly modular design and good separation of concerns. The same idea is often employed in the design of compilers [3].

MapleMIX can be thought of as an interpreter that has the additional functionality of generating residual code for deferred computations that cannot be performed at partial evaluation time. MapleMIX is loaded into the Maple system in the same scope as the core library. This way through Maple's reification function `ToInert` it has access to the code of almost the entire core library. Only the bodies of built-in routines are unaccessible because they are implemented directly in the Maple kernel.

4.1 Input and Output

Traditionally the input to a partial evaluator is the entire source text of a program and this will be the only code that is considered. In contrast MapleMIX has access

to all the code that is loaded into the current Maple session. Therefore the specialization process must 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 goal function is a regular Maple function, it does not need a special name. The parameter list of the goal function will be the dynamic inputs to the specialized program. A user supplied goal function is in contrast to some other partial evaluators that automatically create a goal function or treat a function with a particular name as the goal function. For example, the C partial evaluator C-Mix takes the `main()` function as the goal function and if it doesn't exist then it takes the first function in the program [1]. We believe that a user supplied goal function is an easy way to call MapleMIX. The `pow` example of Chapter 3 was created with the following command.

```
> pow_S := OnPE( proc(x) pow(x, 5) end proc ):
```

MapleMIX may generate several residual functions, these are packaged together with the specialized goal function and returned as a module. The specialized goal function will become the ModuleApply function of the returned module. The name of the returned module is supplied by the user.

4.2 M-form

MapleMIX is a syntax-directed partial evaluator, meaning it proceeds through the specialization process by following the structure of abstract syntax. The Maple reification function `ToInert` will return the abstract syntax tree of a procedure, referred to as the *inert form* [1]. However, MapleMIX does not directly manipulate inert forms. Instead they are first translated into a new form which we shall simply call *M-form*. The M-form of a procedure is designed to be much more convenient for specialization.

```
> 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 language (that may be a subset of the input language). For example the C partial evaluator C-mix first transforms its input into a base language called Core C

[1] The executable form of a Maple procedure is sometimes referred to as the *active form*.

[1]. Many of the constructs of C are removed and replaced with more basic constructs. For example all loops are removed and replaced with conditional jumps and GOTOs. This approach reduces the syntactic forms that the specializer must support. The possible disadvantage of this approach is that invariants inherent with certain syntactic forms are lost. For example some languages have for loops that are guaranteed to terminate.

M-form is not a subset of inert form. M-form simplifies the inert form in some ways and adds to it in others. Since MapleMIX is syntax-directed, syntax transformations and new syntactic forms can naturally be used to direct the specializer. We have found that adding syntactic forms does not add to the length of the specializer but in fact makes it more compact. The removal of syntactic forms can actually add complexity because the specializer may have to infer information that is lost.

We want to keep all the knowledge inherent in the inert form intact while keeping translation between M-form and inert form a straightforward process. M-form is designed to meet the needs of the specializer, and especially to keep the specializer as small and simple as possible. In this chapter we describe where M-form differs from inert form and the motivations for such changes. Translation from inert form to M-form is done by the *ToM* module and translation back is done by the *FromM* module.

4.2.1 Expressions

Maple is an imperative language with global variables, therefore expressions may be side-effecting. Since Maple has no `++` operator and does not allow statements in expression context, the only expression form that may be side-effecting is a function call. In order to separate the concerns of expression reduction and environment update M-form stipulates that all expressions must be side-effect free.

The M-form translator maintains a list of known intrinsic functions. An intrinsic function will never be specialized, instead any call to an intrinsic function will be treated as an atomic operation that may be performed at partial evaluation time. Most built-in functions are considered intrinsic except side-effecting IO functions. Some library functions are also considered intrinsic in order to simplify residual code. All non-intrinsic function calls are removed from expressions by generating a new assignment statement for each call and then replacing the original calls by the names generated. This transformation is known as *splitting expressions*, see figure 4.1.

Original Code	Transformed Code

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

Figure 4.1: Splitting of functions from expressions

A new syntactic form of assignment is added, *MAssignToFunction*, specifically for assignments of a function to a variable that have been generated by the splitting transformation. This way the specializer can syntactically distinguish between situations where simple reduction of an expression is sufficient or specialization of an entire function body must be performed.

The splitting transformation has the unfortunate effect of possibly creating many new assignment statements. However, if the specializer decides not to unfold a split function call, or if the function unfolds into a single assignment statement, then we know that the new variable is only assigned to once and only used once. The form *MSingleUse* is added to track variables that are generated by the splitting transformation. This allows the FromM translator to easily re-inline expressions that were split out by ToM.

Maple allows expressions to be used in statement context. This is often used in conjunction with Maple's implicit return mechanism. However we would not like the statement specializer to have to account for every expression form. The solution is to wrap standalone expressions within *MStandaloneExpr*, and wrap standalone function calls in *MStandaloneFunction*.

4.2.2 Assignment

Inert form has one type of assignment, M-form has several. We have already shown the need for a special *MAssignToFunction* form to tell the specializer that a function must be specialized. However there are other variations in assignment that must be taken into special consideration. The first is assignment to a indexed table element. Tables are treated specially in the environment therefore we separate assignment to a table index into its own M-form, *MAssignTableIndex*. Furthermore Maple allows multiple table indices on the left side of an assignment. We simplify this situation

by only allowing one table index in an assignment, this is done by another splitting transformation.

Original Code	Transformed Code
`t[x][y] := e;`	`m1 := t[x];` `m1[y] := e;`

Figure 4.2: Splitting of table indices

Furthermore we must identify assignment statements that are created in this manner. This is required because Maple allows the implicit creation of tables. In the above example if `t` was unassigned then it would have been implicitly assigned to a newly instantiated table, then `t[x]` would also be instantiated to a table. We must indicate to the specializer this special case when it must create a table. Therefore we have an assignment form for generated assignments where the right side is a simple table lookup, *MAssignToTable*. Multiple assignment is currently not supported.

4.2.3 If Statements

If statements in Maple may have arbitrarily many `elif` blocks and an optional `else` block. M-form has a simpler *MIfThenElse* construct that always consists solely of a conditional expression and two branches. Any Maple if statement with a list of `elif` blocks is converted into nested *MIfThenElse* statements. Empty else blocks are added as necessary, see figure 4.3.

Original Code	Transformed Code

`if C1 then`
`  S1`
`elif C2 then`
`  S2`
`end if`
	`if C1 then`
	`  S1`
	`else`
	`  if C2 then`
	`  S2`
	`  else`
	`  end if`
	`end if`

Figure 4.3: Transformation of if statements

This transformation works hand-in-hand with the splitting transformation in order to

correctly maintain the ordering of function calls in conditional expressions, see Figure 4.4.

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

Figure 4.4: Transformation of if statements with splitting

4.2.4 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** 1 **to** 10 **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 loops have an optional while clause, the boolean expression in the while clause is checked on the start of each iteration of the loop and if the expression is false the loop is exited. Maple allows most clauses of a loop definition to be optional. A while loop is actually a for-from loop with all the clauses unspecified except the while clause. In inert form the missing clauses are given default values. For example:

```
while x < y do
  x := x + 1;
end do;
```

In inert form (pretty printed) we can see that the slots for the loop variable and its value range have been given default values. The loop variable has an empty expression sequence (`NULL`) as its default value.

```
FORFROM
  EXPSEQ
  INTPOS 1
  INTPOS 1
  EXPSEQ
  LESSTHAN
    NAME "x"
    NAME "y"
  STATSEQ
    ASSIGN
      NAME "x"
      SUM
        NAME "x"
        INTPOS 1
```

For loops and while loops are inherently different. A proper for loop where all write access to the loop index variable is controlled by the loop statement itself has a strong invariant that the loop is guaranteed to terminate. This guarantee is an absolute must if the partial evaluator is to reliably unroll loops without risking non-termination. It would be possible for the specializer to tell if a loop is a proper for loop or a while loop by checking for the presence of the known default values. We prefer to lift this concern from the specializer to the M-form translator. Therefore in M-form we specify three types of loop instead of two. Assignment to the loop index variable is currently not supported.

- **MWhile(whileCondition, body)** A regular while loop consisting of just a while condition and the loop body.
- **MWhileForFrom(loopVar, fromVal, toVal, byVal, whileCondition, body)** A for-from loop with a while condition. This type of loop can be unrolled but the while condition must be checked on each iteration. At the point where the condition evaluates to false the unrolling of the loop is stopped. Any assignment statements created by splitting the loop condition are added both before the loop and inside the loop after the original loop body.
- **MWhileForIn(loopVar, loopVar, inVal, body)** A for-in loop with a while condition.

The while condition is executed on each iteration of the loop. Any assignments that are generated by splitting function calls out of the while condition expression are inserted before the loop and in the body of the loop at the bottom. See figure 4.5.

Original Code	Transformed Code

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

Figure 4.5: Splitting of while condition

Currently MapleMIX does not support the use of `next` or `break` inside a loop. If one is encountered during translation to M-form an exception is thrown. However there is one case where a simple transformation can remove the use of `next`, see Figure 4.6.

Original Code	Transformed Code

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

Figure 4.6: Removal of next

4.2.5 Parameter Passing

Parameter passing in Maple can become quite complex. MapleMIX supports all forms of parameter passing and most usages of `args` and `nargs`. Parameter passing is a feature that has evolved slowly over successive versions of Maple, the result being that the inert form of function parameter lists is unnecessarily complex. We simplify and standardize all parameter specification forms. The M-form of a procedure will have two lists of parameter specifications, *MParamSeq* and *MKeywords* for regular and keyword parameters respectively. A *MParamSpec* consists of the parameter's name, an optional default value and an optional type assertion. Figure 4.7 shows a pretty printed view of the the part of a procedure that defines the parameters in both inert and M-forms. Note that *MKeywords* is added at the end of the proc structure as to not renumber the operands that already existed.

```
> p := proc(x::int, y::int := 0, {z::int := 0})
      ...
  end proc:
```

Original Code	Transformed Code

`> inrt := ToInert(eval(p)):`
`> InertForms:-Print(inrt);`
`PROC`
`  PARAMSEQ`
`  DCOLON`
`  NAME "x"`
`  NAME "int"`
`  ASSIGN`
`  DCOLON`
`  NAME "y"`
`  NAME "int"`
`  INTPOS 0`
`  SET`
`  EXPSEQ`
`  ASSIGN`
`  DCOLON`
`  NAME "z"`
`  NAME "int"`
`  INTPOS 0`
`  ...`
	`> m := M:-ToM(inrt):`
	`> M:-Print(m);`
	`MProc`
	`  MParamSeq`
	`  MParamSpec`
	`  "x"`
	`  MType`
	`  int`
	`  MDefault`
	`  MParamSpec`
	`  "y"`
	`  MType`
	`  int`
	`  MDefault`
	`  MInt 0`
	`  ...`
	`  MKeywords`
	`  MParamSpec`
	`  "z"`
	`  MType`
	`  int`
	`  MDefault`
	`  MInt 0`

Figure 4.7: Parameter specifications

4.2.6 Tables

Maple allows the creation and initialization of a table at the same time using the built-in `table` function. 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.

Inert form has a special structure *_Inert_TABLE* to allow the full definition of tables to be defined directly in abstract syntax. We have found that *_Inert_TABLE* can

Original Code	Transformed Code
`A := table([ (1,1) = 1, (1, 2)=x-3, (2,1) = 2, (2, 2)=x, (3,1) =-1, (3, 2)=3]);`	`A[1,1] := 1; A[1,2] := x - 3; A[2,1] := 2; A[2,2] := x; A[3,1] :=-1; A[3,2] := 3; A;`

Figure 4.8: Transforming table creation to assignments

be removed and replaced with a call to Maple's builtin table creation function. The result is the removal of a syntactic form that is unnecessary. When a table is defined directly in abstract syntax it is usually the case that the table is completely static, therefore we do not perform the transformation to assignments described above in this case.

4.3 Architecture

MapleMIX consists of several modules, key modules and their roles are listed below.

- **OnPE.** The core of the specializer. Contains the statement block partial evaluator. Generates specialized functions and decides when to share already specialized functions. Maintains a call stack of environments. Decides when a specialized function should be unfolded. Maintains a table that stores generated residual code.
- **ToM.** Translator from inert form to M-form.
- **FromM.** Translator from M-form to inert form.
- **OnENV.** The online environment. Stores values of static variables and partially static tables. Has a unique implementation that allows for easy treatment of dynamic conditionals. This will be discussed in detail in the next chapter.
- **ReduceExp.** The expression reducer. When given the M-form of an expression will reduce it as far as possible using the static information provided by the environment. May return a static value or a dynamic M-form. Its implementation is inspired by the online cogen approach.
- **Unfold.** Performs the function unfolding program transformation.

- **BuildModule.** Takes the residual code generated by OnPE and packages it as an active Maple module.
- **NameGenerator.** Responsible for generating new unique names for use during M-form translation and during specialization.

4.4 Summary

MapleMIX is a syntax-directed partial evaluator that process a form of abstract syntax called M-form. This M-form is designed to translate inert form into a representation that is more palatable to the needs of a specializer. In most respects M-form is virtually identical to inert form except for certain key areas including: expressions, if statements, loops, assignments, and parameter specifications. We believe that inventing an intermediate language specially designed for the specializer is a good way of modularizing a syntax-directed online partial evaluator.

Chapter 5

Expression Reduction

The expression reducer serves the role of evaluating expressions as far as possible given the available static 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 and tables. The implementation of the reducer is inspired by an online cogen approach to PE as outlined by Sumii [23]. The idea is to replace the underlying operators of the language with smarter ones that correctly handle dynamic arguments. Sumii 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 that works as follows. If all arguments are static then apply the underlying Maple operator on the arguments, this essentially hands control over to the Maple interpreter to perform the actual static operation. Otherwise, in the case that any of the arguments are dynamic, build a dynamic expression and return it. It is easy to see that reduction is implemented as an extension to the already existing semantics of Maple expressions.

For example, reduction of the term $x + y$ when the static values $x = 1$ and $y = 2$ are bound in the environment yields `MStatic(3)`. Reduction of the same term with $x = 1$ static and y dynamic yields `MSum(MStatic(1), MLocal("y"))`. The special M-form tag `MStatic` is used to "wrap" static values returned by the reducer. This has the following advantages:

- It is very easy to test the binding time of a reduced term.

- Static values can be embedded within the dynamic representation in a straightforward way. This allows static values to be easily *lifted*, as will be explained in greater detail in the next chapter.
- It is necessary to wrap static expression sequences within an `MStatic` to avoid expression sequence flattening.
- Wrapping of values that are subject to last-name evaluation rules prevents them from picking up names defined in the specializer.

5.1 Online Approach to Partially Static Data

For a program specializer to produce good results it must utilize as much static information as possible. There are often situations where a concrete value cannot be known, however the type or the shape of the value can still be statically known. For example a list may have dynamic elements, however it may still be possible to know the size of the list. Avoiding unnecessary approximations is key to preserving static information [21]. For example, an online PE may treat a list with a dynamic element by making the approximation that the list is completely dynamic, thereby treating it as an "unknown" value. While this approximation is safe, in the sense that the residual code will be correct, it does not make good use of 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. For example a dynamic list `[a, b, 2]` where `a` and `b` are dynamic will be represented in M-form as:

```
MList(MExpSeq(MLocal("a"), MLocal("b"), MStatic(2)))
```

It is easy to see that the size of the list is statically known in this case by simply examining the structure of the abstract syntax. Our approach is to exploit the static information present within the dynamic representation. For example, the built-in Maple `nops` function has been extended to return a static result in such cases. Several of Maple's intrinsic functions have been extended to add support for partially static lists and polynomials. Syntactic constructs such as indexing and list appending have also been extended in a similar way.

In order to propagate dynamic terms through the program they will be stored in the environment alongside static values. When the reducer encounters a variable it will retrieve its representation from the environment which may store a static value, 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. A special syntactic form `MSubst` is introduced by the reducer to track such substitutions. `MSubst` consists of the variable name and the dynamic representation retrieved from the environment. For example:

```
MSubst(
        Local("x"),
        MList(MExpSeq(MLocal("a"), MLocal("b"), MStatic(2)))
)
```

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 into inert form, the dynamic representation part of the `MSubst` will be discarded and the name used instead.

Support for partially static terms has been explored mostly within the context of offline PE. One approach is to use a BTA to determine the binding times of individual elements of a partially static data structure [16]. Another approach uses an abstract interpretation as a shape analysis to gather static shape information as a pre-phase [13]. Our approach is completely online and has the potential to exploit the full information available during specialization. However it must be noted that quite a bit of custom support for various dynamic representations must be added to the reducer in order to achieve this.

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 when dynamic information leads to static computations. Dead code is removed by a simple post-phase cleanup.

5.2 Input and Output

The reducer has the following preconditions:

- The input term must be in M form.
- All function calls in the term must be to Maple built-in functions and must also be pure, this ensures the expression is side-effect free.
- There must be an online environment provided. Every time a variable is encountered the environment is queried for the binding time of the variable and if static also its value.

The output of the reducer will be in one of three different binding times:

- **Static**: Will return a value wrapped in a *MStatic* constructor.
- **Dynamic**: Will return an M form other that *MStatic* and *MBoth*. This term is reduced as far as possible and may contain static data embedded within it.
- **Both**: This binding time is used exclusively with tables. *MBoth* has a dynamic part and a static part. The dynamic part is the result of reduction with partially static tables treated as dynamic and vice versa. This allows the specializer to identify situations when the environment must both be updated with a static value and residual code must be generated. We will discuss this situation in more detail later in this chapter.

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 IO functions such as `print` and `read`. These will not be considered intrinsic but will still be detected as built-in and so will be treated as a special case by the specializer. IO functions will be split out of expressions and always be residualized. Many non-built-in functions can also be treated as intrinsic such as `curry`, which performs partial application. Some

library functions have non-standard semantics such as `seq` (the function for sequence comprehensions). These are treated as special cases by the reducer. Every time a function call is encountered a table of special functions is first checked to see if a custom handler for the function exists. All calls to *eval* functions such as `eval`, `evalb` and `evalf` are always residualized.

A call to the `assigned` function is treated as a special case. Sometimes it is possible to statically know whether a name is assigned to a value even if the value is unknown (dynamic). In general if an assignment statement is dynamic we cannot assume that the name that is assigned to will actually carry a value because the dynamic expression may be the variable's own unevaluated name (this is how a name becomes unassigned). However only limited usage of unevaluation quotes is supported. The expression between the quotes must be syntactically identifiable as a single name. This means that unassignment of table cells is not supported by MapleMIX. This restriction allows the partial evaluator to know when a table cell will have a value at runtime even though the value is unknown at partial evaluation time. Detection of this case is implemented as a simple extension of the online environment's dynamic masking feature which will be explained in detail in section 6.3.5.

Some expressions are subject to last name evaluation rules. This poses a certain challenge for writing the reducer. Expressions such as procedures, tables and modules always pick up the last name they were assigned to. It is possible for raw LNE rule values to pick up local names defined in the MapleMIX source if special care is not taken. The solution is to never directly assign values in the reducer, instead intermediate results are always wrapped in a list. This keeps the value isolated from the internals of the partial evaluator.

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

Maple is a dynamically typed language and as a result existing Maple code contains a great deal of dynamic type tests (usually as part of error checking code). Therefore, to be semantically correct, it is necessary that the reducer not change the type of

any static term. This poses a challenge for handling function closures, they must be represented as an active Maple function in order to be consistent in our treatment of static values. However the values of the function's lexical locals are stored in the partial evaluator's environment. This problem is solved by performing a simple transformation on the body of the function. Each lexical local is replaced by an application of an inline function. These "thunks" will call back into the partial evaluator to retrieve the variable's value, and will throw an exception if there is no static value available. The M-form of the closure is then converted into active Maple code. This way a static function closure can be simply applied when needed, a feature essential for supporting higher-order built-in functions such as `map` and `fold`. If applied to some dynamic arguments it will be converted back into M-form and specialized. This converting of code to and from active Maple is inefficient but unavoidable with this scheme.

5.3 Dead Code Removal

We use a simple, easy to implement scheme for dead code removal. The algorithm starts at the bottom of a function body and proceeds upwards. For every statement, if it is not dead then add any names referenced to a set of live variables. When an assignment is encountered the name on the left side must be in the set of live variables or the statement is dead code and is removed. For if statements perform this process on each branch, then union the two sets of live variables produced to get a new set. For loops it is necessary to find loop dependent variables, these are variables that are referenced in the loop. The loop dependent variables are then added to the set of live variables before the body of the loop is processed.

This algorithm is sufficient to remove many cases of dead code but it has shortcomings. Some dead code, especially within loops, may not be removed. A better approach would be to do dead code removal as an optimizing compiler would, by using data-flow analysis. However our simple approach is sufficient for the time being.

5.4 Summary

The expression reducer in MapleMIX is inspired by the online cogen approach. It has the ability to reduce a wide variety of Maple expressions, including support for a novel

online approach to partially static data structures. Some tricky situations, such as intrinsic functions with non-standard semantics, are simply handled as special cases. Higher-order functions are supported by transforming the active body of the function by adding callbacks that retrieve static values from the specializer's environment. A special binding time `MBoth` is meant specifically for partially static tables which are handled in a special way in the online environment. In the next chapter we will discuss the specialization of statements and the inner workings of the online environment.

Chapter 6

Statement Specialization and the Online Environment

The specialization process begins with the supplied goal function. The partial evaluator converts the goal function into M-form, creates a new empty environment and begins by specializing its body. Each non-intrinsic function call encountered initiates the following process. If the function has not been encountered before then retrieve its source from the Maple library using `ToInert` and convert it to M-form. Cache the M-form for easy retrieval if a call to the same function is encountered again. A call stack of function environments and a separate global environment are maintained. Specialization of a function begins by creating a new environment, initializing it with the function's static parameters and pushing it onto the call stack. Specialization terminates when the end of the goal function is reached. Finally all residual functions are converted into active Maple code, packaged together as a module and returned to the user.

6.1 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 [16].

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 specialization strategy that is depth-first. Every time a function call is encountered it must be specialized immediately. There may be several functions in the process of specialization at the same time.

6.1.1 Function Sharing and Termination Issues

MapleMIX uses a simple function sharing scheme for two purposes. Firstly to reuse specialized functions in cases where multiple calls to the same function with the same static arguments are encountered. And secondly 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, avoiding unnecessary specialization.

This strategy also improves termination properties of the partial evaluator as call signatures are used to help detect static recursion. The depth-first specialization strategy makes it possible for several functions to be in the process of deferred specialization. If one of those functions is recursive (or mutually recursive) then the problem of infinite specialization arises. The partial evaluator can tell when a call signature refers to a function that is currently in the process of being specialized. When such *static recursion* is detected a call to the recursive function is simply residualized. This strategy relies on detection of identical call signatures, if some static value is changing under dynamic control then infinite specialization is still likely [16].

For example specialization of the `pow` example of chapter 3 with x having the static value 2 and n dynamic results in a function that computes 2^n:

```
pow_1 := proc(n)
    if n = 0 then
        1
    else
        2 * pow_1(n - 1)
    end if
end proc
```

The difficulty with termination is caused by the fact that the termination condition ($n = 0$) is dynamic. When the specializer encounters the recursive call to `pow` it first examines the call signature, in this case ($x = 2, n = DYN$). It realizes that it is currently specializing `pow` with respect to that signature and simply inserts a residual call to `pow_1`.

The function sharing mechanism is currently incompatible with the online environment's support for storing dynamic representations. The abstract syntax dynamic representation may have an effect on how a function is specialized. Therefore any dynamic representations passed to a function must be considered part of its call signature. Doing this in a naive way would hurt the termination properties of the specializer as less cases for reuse would be encountered. We have found in our experiments the need for either function sharing or partially static data structures, but not both at the same time. Therefore, currently in MapleMIX they are implemented as mutually exclusive features. An option can be provided when calling MapleMIX to choose which feature to turn on. We leave it as future work to make these two features work together in a suitable way, possible approaches will be discussed in chapter 8.

6.1.2 Global State

In a language with global state is is usually necessary for functions to be specialized with respect to static parameters and static global variables. Moreover the effect of the function on global state must be remembered if a function is to be shared, and when a function is shared the global state must be effected accordingly [1]. Global state is used often in imperative languages such as C, however it is not an integral part of many Maple programs. Therefore we will not concern ourselves with the problem of coordinating shared functions and global state. Extending MapleMIX with this functionality in the future should be straightforward. Instead we take the following conservative approach; a function may not be shared if it reads from or writes to global state. The specializer maintains a call stack of environments for function calls and also maintains a separate global environment.

6.2 Function Specialization

When a function to be specialized is encountered, the first thing that must be done is matching up of argument expressions to the parameters of the function. Some of the argument expressions may be static and some may be dynamic. Parameters can have type assertions and default values. Also special optional parameters can be defined. It is also common for a function to declare no parameters at all and instead access parameters through `args` and `nargs`.

A new environment for the function is created that will be initialized with bindings for static parameters. Each argument expression is processed one at time, starting with the leftmost one. Two other things must be produced: the residual function call (with static arguments removed and dynamic arguments reduced) and the new parameter list of the specialized function. Each static argument is matched up with its corresponding parameter if one exists. If a type assertion is attached to the parameter, it is checked. If the assertion fails, the default value is taken if it exists; if there is no default value then an error is issued. Maple allows any expression to be given for a default value, however, MapleMIX only supports constant values. A binding is then added to the environment and the parameter is removed from the parameter list of the specialized function. If the argument is dynamic, then it is added to the residual call. Once all the arguments have been resolved, the function body is specialized.

There is a caveat to Maple parameter passing. All functions essentially take a single expression sequence, and expression sequences are automatically flattened. If we have a dynamic argument to a function there is the possibility that at runtime this argument will be an expression sequence. Therefore as soon as a dynamic argument is encountered we can no longer reliably match up arguments to parameters. All remaining arguments must be residualized and unmatched parameters will be dynamic. This is a severe restriction that can lead to a heavy loss of static information. In practice expression sequences are rarely used directly;j they are often used only to build lists and sets. Therefore MapleMIX currently does not consider expression sequences when matching up arguments to parameters.

6.2.1 Unfolding

Unfolding is a well-known program transformation which replaces a function call by the body of the function. It is used to reduce the overhead involved in the function

invocation mechanism, and to enable further transformations. MapleMIX performs unfolding after a function has been specialized. Unfolding cannot be performed when the function has been detected as being recursive, because the recursive function is currently in the process of being specialized. There are several concerns when unfolding a function body, these concerns will be outlined in the following paragraphs.

Local variable names of the unfolded function may clash with variables defined in the calling function. This is solved by consistently renaming the local variables of the unfolded function to new unique names.

The function may use returns, which will not have the same meaning when the body is unfolded. This is solved in most cases by transforming the body of the function in such a way as to remove the need for returns. We shall call the result of this transformation *return normal form*. The transformation works by scanning a code block starting at the top. When a return is encountered all the code in the statement sequence below the return is discarded. When an if-statement is encountered any code below it is removed and copied into the bottom of both branches of the if. Each branch of the if is then processed recursively. In the resulting code all returns may be removed because they are no longer necessary. There are some cases when this transformation is not possible, for example when a return exists within a dynamic loop. If this transformation cannot be performed then the function will not be unfolded.

Dynamic parameters must be replaced by their corresponding argument expressions. It is not sufficient to simply substitute each occurrence of a parameter by its corresponding expression. In the case that a parameter name appears multiple times in the body of the specialized function the expression would be duplicated. Code duplication is not the concern here but rather computation duplication which defeats the purpose of partial evaluation as an optimizing transformation. Instead a technique known as *let-insertion* is used to avoid duplication [16]. An assignment statement for each argument expression is generated which binds the expression to a new variable name. The parameter is then substituted by this name (it would have to be renamed anyway). In the case where an argument expression is just a simple variable then the substitution can be performed directly as there is no risk of computation duplication.

Use of the `args` and `nargs` keywords must be removed, because, like return, they will no longer have the same meaning when the function is unfolded. The solution is to generate a new name for args, bind the entire argument expression sequence to this name and then substitute all occurrences of the keyword with the name. If `nargs` is dynamic then again a new name is generated for the purpose of substitution and this name is bound to the expression `nops([ name-of-args ])` where `name-of-args` is

the new name generated for `args`. These assignments are only generated if the `args` or `nargs` keywords are used in the body of the function. Since unfolding occurs after specialization the use of these keywords may have been specialized away. Consider the following example:

```
f := proc(x, y)
    g(x, y);
end proc;

g := proc()
    args[1] + args[2];
end proc;
```

Partial evaluation of the `f` function produces:

```
proc(x, y) local args1;
    args1 := x, y;
    args1[1] + args1[2]
end proc
```

The function call may be on the right side of an assignment statement. In this case the name on the left side must be bound to the correct value when the function is unfolded. Transforming the function into return normal form facilitates another simple transformation. Implicit returns are simply standalone expressions at the bottom of statement sequences. Each of theses expressions is simply replaced with an assignment statement.

6.2.2 Higher-Order Functions and Closures

All functions in Maple are first-class citizens, they are just values assigned to names. This is apparent from the Maple syntax for procedure declaration which consists of an assignment statement of an expression that evaluates to a procedure. When any non-intrinsic function application is encountered the reducer is first called on the name of the applied function. The reducer will attempt to return the function as a static value which may have been retrieved from either the core library or the online environment. The function is then converted to M-from and specialization takes place as described above.

If a function was retrieved from the online environment then it likely has a closure. As described in the chapter on expression reduction, all lexical locals of the function are replaced with thunks that call into the closure environment to retrieve the

corresponding static value. These thunks contain references to the function's closure environment which keep it from being garbage collected after it has been popped from the call stack. When there are no longer any references to the function it will be available for garbage collection along with its closure environment. This scheme works similar to a common simple approach to implementing closures in an interpreter [3].

6.3 If Statements and the Online Environment

M-form has a very simple if statement that consists of a conditional expression and two statement sequence branches. Partial evaluation 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. In particular 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. This can be done by copying the environment [1, 13]. 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. Second, code that is below the if statement must be handled correctly. This code 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 solution is to implement each online environment 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 in the stack. 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

6.3.1 DAG Form

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 will be further translated before specialization into a DAG (Directed Acyclic Graph) representation. 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. The 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. The DAG form of the following example can be found in figure 6.1.

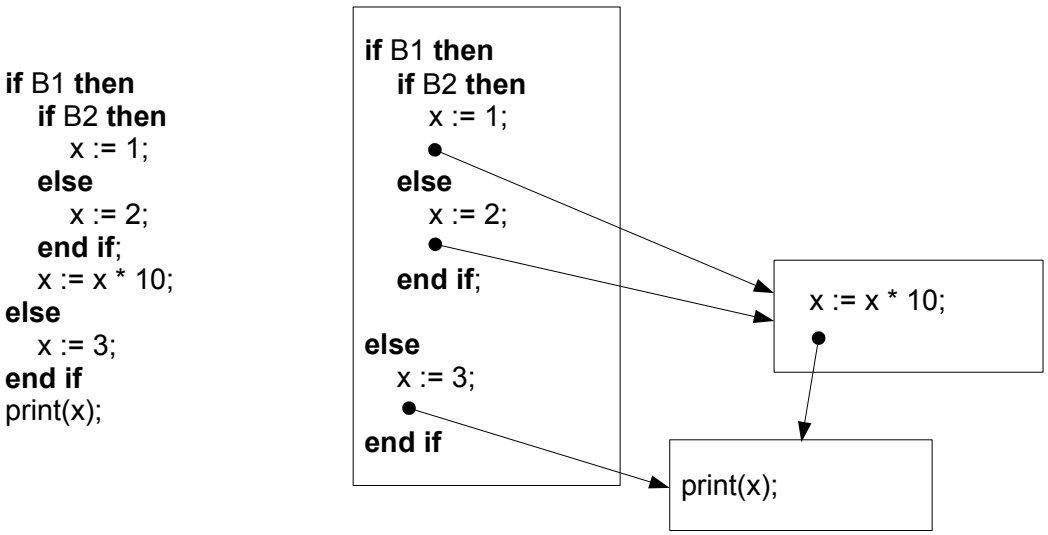


Figure 6.1: DAG form

6.3.2 Dynamic Conditional Specialization Algorithm

Let us begin by describing the high level idea of the specialization algorithm for if statements using the online environment. Each branch of the if statement must be specialized one at a time. Specialization of the first branch begins with growing the stack by pushing a new empty setting. All effects of the statements in the first branch are recorded in this topmost setting. Simply popping the stack restores the

environment to the state it was in before specializing the first branch. The stack is then grown again and the second branch is specialized with respect to the initial environment. Also, we can now easily compare the effects of each branch by comparing the top settings created by each branch. We will now describe the algorithm again, this time in more detail with regard to code of the following form:

```
if B then
  C1
else
  C2
end if;
S
```

First B is reduced; if it is dynamic then the following process is initiated. The environment is grown and C1 is specialized. If C1 does not end with a return or error statement, then the environment is grown once more and S is specialized; this produces a version of S, which we shall call S1, that is specialized with respect to the environment created by C1. The environment is popped twice, once to remove the effects of S1 and second to remove the effects of C1; however, the setting created by C1 is saved. Specialization of C2 can now take place in the initial environment which is grown again with a new empty setting. After C2 is specialized, an important point is reached: the setting produced by C1 is compared to the setting produced by C2. If they are the same, then S1 is simply residualized below the if statement, and specialization of the if statement terminates. The following code will be output:

```
if B' then
   C1'
else
   C2'
end if;
S1
```

If the two settings are different then another version of S, called S2, is specialized in the current environment (there is no need to grow the environment again for this version of S). The two versions of S are residualized by inserting S1 into the first branch below C1 and similarly for S2 and the second branch.

```
if B' then
   C1';
   S1
else
   C2';
   S2
end if;
```

Special care is taken to not unnecessarily specialize S in the case that C1 or C2 ends with a return or error statement. Duplication of S is avoided in situations where execution of either branch would both 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 can be seen in figure 6.2.

Original Code	Specialized Code

`p := proc(...) local x;`
`  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);`
`end proc:`
	`ps := proc (...)`
	`  if <dynamic> then`
	`  if <dynamic> then`
	`  print(1);`
	`  print(10);`
	`  print(100)`
	`  else`
	`  print(20);`
	`  print(200)`
	`  end if`
	`  else`
	`  print(300)`
	`  end if`
	`end proc`

Figure 6.2: Example of if statement specialization

6.3.3 Comparison with Merging Environments Approach

A natural approach to specializing if statements is by merging environments [13]. 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 example consider the following code:

```
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. In our example 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 may 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 complex, environments are copied and the reducer is more complicated. 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.

6.3.4 Comparison with Offline Methods

Binding Time Analysis is essentially a worst case analysis. The dynamic binding time takes precedence over static. As a result the dynamic binding time tends to propagate through the program creating a snowball effect. The same is mostly true for online methods as well. 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. Any expressions involving that variable will also be dynamic causing the snowball effect. However with our online partial evaluator that is written in a similar way to an interpreter it is possible for a dynamic variable to become static (however unlikely it may be to find code that does this).

```
x := <dynamic>;
...
x := 5;
```

In this example the last line causes x to be bound to the static value 5 in the online environment, regardless of the fact that x was dynamic before.

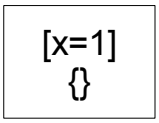
6.3.5 Dynamic Masking

It is possible for a variable to change binding time in a particular setting; however, when the setting is popped, the binding time of the variable in the previous setting must be restored. Special care must be taken when a variable becomes dynamic in

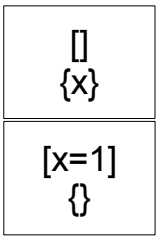
the top setting. This is done by storing a *dynamic mask* in each setting which is simply a set of names. When an environment lookup is requested the search stops when the binding is found or the variable is found in a dynamic mask. Consider partially evaluating the following Maple code fragment:

```
1 p := proc(d) local x;
2     x := 1;
3     if d < 100 then
4         x := x + d;
5     else
6         x := x + 1;
7     end if;
8     print(x);
9 end proc;
```

After line 2 the environment will consist of a single setting with the binding for x (the assignment is removed because it is static). At this point the environment looks like:

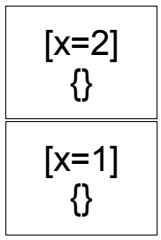


The environment is grown for the first branch of the dynamic conditional. The bottom setting of the environment contains the binding $x = 1$ therefore the expression $x + d$ on line 4 reduces to $1 + d$. The assignment is residualized and x is added to the dynamic mask in the topmost setting. Now the environment has the two settings:



Then the code below the if statement is processed, the argument expression to *print* is dynamic and the print statement is saved for residualization. Then the setting stack is popped back to its original state. The topmost setting that existed after line 3 is saved and the stack is grown again. The else branch is processed, here the expression $x + 1$ is static, the binding is stored in the topmost setting and the assignment is

removed.



Then the topmost setting is compared with the setting that was previously saved. They are different and so the print statement is specialized again, this time the argument is reduced to the static value 2. The resulting code is:

```
p_s := proc(d) local x;
    if d < 100 then
        x := 1 + d;
        print(x);
    else
        print(2);
    end if;
end proc;
```

6.3.6 Tables

MapleMIX fully supports tables as a partially static data-type. Meaning 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 when possible.

Each setting is implemented as a Maple record with the following fields:

- **vals :: name → value**
  Maps (binds) names to regular values.
- **dyn :: name → dynamic**
  Maps names to dynamic representations.
- **mask :: set(name)**
  The dynamic mask for regular values, a set of names.
- **tbls :: name → section**
  Maps a name to a section of a partially static table. Each section is itself represented as a record that consists of the following fields. It is possible for more than one name to be mapped to the same section.
  - **elts :: index → value or section**
    Maps indices to table elements. Dynamic masking of indices is implemented by mapping the index to the special value `DYN`. Nested tables are implemented by mapping an index to a section of the nested table.
  - **link :: section**
    Pointer to the next section of the table below the current section in the stack. Its purpose is to avoid checking every setting on the stack for the sections of a table. This link makes it easier to traverse all the sections that make up the table as only the first section has to be searched for, then all the other sections may be easily accessed in order by following the links. Traversal of the links stops when the link is `NULL`.
  - **dynCount :: integer**
    Maintains a count of the number of dynamic elements in a section of the table. When this value is 0 then the section is completely static. The reducer may query the environment for the binding time of a table, if the overall dynamic count of the table is greater than 0 then the reducer returns the binding time of *Both*. If the count is 0 then the reducer knows to return the binding time of *Static*.

An example of the results of this approach can be found in figure 6.3.

As explained above dynamic masking is implemented for table indices by mapping an index to the special value `DYN`. A table index is dynamically masked when the specializer encounters a dynamic assignment to that table index. Therefore the value of

Original Code	Specialized Code

p := proc(d)
  t[1] := "a";
  if <dynamic> then
  if <dynamic> then
  t[1] := d;
  t[2] := d;
  else
  t[2] := "b";
  end if;
  else
  t[1] := "A";
  t[2] := "B";
  end if;
  t[1], t[2];
end proc:
	ps := proc(d) local t;
	  if <dynamic> then
	  if <dynamic> then
	  t[1] := d;
	  t[2] := d;
	  t[1], t[2]
	  else
	  "a", "b"
	  end if
	  else
	  "A", "B"
	  end if
	end proc

Figure 6.3: Example of tables

the table index is unknown, however it is statically known that the index carries some run-time value. A call to the `assigned` function, which is used to test if mappings exist in a table, can be statically known to be true when the index is dynamically masked.

6.4 Other Statements

For the remaining statement forms we will now discuss how they are handled by the specializer. The addition of syntactic forms in M-form make it easier to identify the required statement specialization strategy using simple case analysis.

6.4.1 Assignments

M-form provides four types of assignment: *MAssign*, *MAssignTableIndex*, *MAssignToTable* and *MAssignToFunction*. We shall discuss each one in turn.

MAssign. Simple assignments are relatively easy to handle. The expression on the right side is reduced and depending on the binding time of the result one of three actions is taken:

- **Static.** The environment is updated with the new binding, No code is residualized.
- **Dynamic.** An assignment to the reduced expression is residualized. If possible the dynamic representation returned by the reducer is added to the environment.
- **Both.** This means the expression reduced to a partially static table. The environment is updated with a new binding. However an assignment statement must still be residualized so that the dynamic parts of the table may be propagated through the residual program. Consider the partial evaluation of the following example:

```
1 p := proc(d) local t, s;
2     t[1] := 100;
3     t[2] := d;
4     s := t;
5     return [s[1], s[2]];
6 end proc;
```

The assignment on line 3 to `t[2]` is residualized because the expression is dynamic. The assignment on line 4 is also residualized because the expression is a partially static table. In the residual program the dynamic part of `t` is assigned to `s` and so the expression `s[2]` will have a value when the residual program is run.

```
p_s := proc(d) local t,s;
    t[2] := d;
    s := t;
    return[100,s[2]]
end proc
```

The left side of the assignment must be a single name, multiple assignment is not supported. The name will be identifiable syntactically except when name concatenation is used. In this case the reducer is called to evaluate the concatenation.

MAssignTableIndex. This is an assignment statement to an indexed name. Treated similarly to *MAssign* except the index expression is also reduced and special action is taken when dynamic. If the index expression is dynamic then the assignment is

residualized regardless of the binding time of the target expression. Since it is not known which table index was affected, the entire table must be treated as dynamic from this point forward, and so the name is added to the dynamic mask. If the index expression is static then the assignment is treated in a similar way to *MAssign*.

MAssignToTable. This syntactic form is the result of the elimination of multiple table indexes on the left side of assignments in conjunction with the need to support Maple's implicit table creation feature. It is known that the expression is a table lookup that itself must evaluate to a table (the result of tables of tables). If the expression is not assigned then that table index is bound to an empty table. The assignment is then treated the same as *MAssign*.

MAssignToFunction. The right side of the assignment is a function call that will be specialized. It is possible for the function to evaluate completely, resulting in the body being a single static return statement. If this is the case then the unfolding transformation will produce an assignment statement to a static expression. This will be detected and the environment will be updated by binding the name to the static value. The specialized function is discarded and no code is residualized.

6.4.2 Statement Sequences

A sequence of statements is partially evaluated by processing each statement in turn while collecting the residual statements into a new residual statement sequence (that may be empty). A statement sequence will not contain any code below an if-statement because of the conversion to DAG form. Any statements that occur beneath a return or error statement are ignored. Once all the residual code for the sequence is collected any useless standalone expressions are removed and the resulting sequence is embedded in the residual program.

6.4.3 Return Statements

Return statements are always residualized. It is up to the unfolding transformation to remove them or replace them with assignment statements. As described above, if unfolding produces a single static assignment statement then it is removed and the environment is updated.

6.4.4 Try/Catch

All of the control flow constructs discussed until this point have involved intraprocedural control flow, meaning control flow is only transferred within the body of the enclosing procedure. The exception handling mechanism is particularly difficult to handle because it involves interprocedural control flow. During normal execution an exception may trigger a jump into the body of another procedure on the call stack at a point other than the function call that was currently being executed.

MapleMIX does support try/catch statements but in a conservative and restricted way. Specialization begins by calling the partial evaluator on the statement sequence inside the try block. However it is not possible to specialize the code within the catch block with respect to the environment generated by the try block. This is because it is unknown at specialization time which of the residual statements within the try block will throw an exception. Therefore it is unknown what the state of the environment will be when the catch block executes (and furthermore it is unknown which catch block will execute). Therefore specialization of catch blocks is allowed but only if the catch blocks do not contain any environment lookups, meaning they cannot contain any variable names. This is a severe restriction although in practice it is sometimes the case that a catch block simply contains a statement that prints a hard-coded message, this will be supported.

If a static exception occurs during specialization of the try block we still cannot reliably specialize the catch blocks. This is because residual code may have been generated before the exception occurred and at runtime it may be that residual code that generates an exception. If a static exception does occur during specialization at any point it is considered an error and specialization is aborted. Furthermore since the execution path of a try/catch cannot be reliably known, any code below the try/catch cannot be reliably specialized.

The specialization algorithm for try/catch proceeds as follows.

- Specialize the try block with respect to the current environment.
- If the try block reduces to an empty statement sequence then simply remove the entire try/catch, the specialization process continues as normal. Thus a completely static try/catch where no exception is thrown in the try is processed as it would be during normal execution.
- If any residual code was generated for the try then check if there is any code

below the try/catch, if there is abort specialization.

- Specialize each catch block and the finally block with respect to a null environment. If any construct is encountered that either reads from or writes to the environment then abort specialization.

6.5 Loops

When all the expression clauses of a for loop definition are static then the loop may be unrolled. Fortunately, in Maple for loops are guaranteed to terminate, but only if the body of the loop does not contain an assignment to the loop index variable. Furthermore, if the body of the loop were to contain a dynamic assignment to the loop index then the entire loop would effectively become dynamic. Since MapleMIX is online it needs to be able to fully determine if a loop is static or dynamic before it decides to unroll the loop. For these two reasons assignment to the loop index is currently not supported by MapleMIX [1].

6.5.1 Static For Loops

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, and to easily "back up" when the end of a path is reached.

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*. 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

[1] Such code is generally considered bad style anyway.

the statement sequence specializer. In order to illustrate our solution the following example will be used.

```
p := proc(d) local x;
     x := 1;
     for i from 1 to 2 do
         if d then
             x := x + 1;
         else
             x := x + 2;
         end if;
     end do;
     print(x);
end proc
```

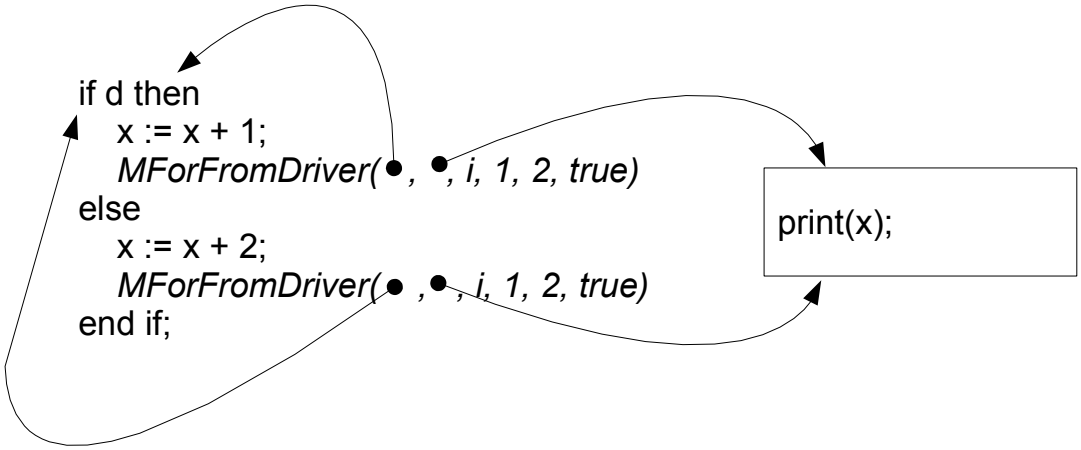


Figure 6.4: Loop Drivers

Loop drivers come in two forms: `MForFromDriver` and `MForInDriver`. Here we will concentrate on `MForFromDriver`; `MForInDriver` works in a similar way. The example will be transformed to look like figure 6.4. `MForFromDriver` consists of the following:

- 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, the termination value.
- The while condition.

A loop driver acts like a conditional GOTO. When it is encountered the specializer updates the loop index variable, then tests its new value against the termination value. If the termination condition fails then the first pointer is taken; if it succeeds the second pointer is taken. The value of the loop variable is retained in the environment after the loop has been fully unrolled. If the loop bounds are such that the loop will never iterate, then the result is that 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 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 several times 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 computation tree. See figure 6.5 for an example of the results of this approach.

Original Code	Specialized Code

`p := proc(d) local x;`
`  x := 1;`
`  for i from 1 to 2 do`
`  if d then`
`  x := x + 1;`
`  else`
`  x := x + 2;`
`  end if;`
`  end do;`
`  print(x);`
`end proc`
	`ps := proc(d)`
	`  if d then`
	`  if d then`
	`  print(3)`
	`  else`
	`  print(4)`
	`  end if`
	`  else`
	`  if d then`
	`  print(4)`
	`  else`
	`  print(5)`
	`  end if`
	`  end if;`
	`end proc`

Figure 6.5: Example of dynamic conditional in static loop

6.5.2 Dynamic Loops

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, however the fact that the assignment may be performed an unknown number of times makes the assignment dynamic.

Partial evaluators for other imperative languages take novel and highly complicated approaches to analyzing dynamic loop bodies. For example the MATLAB partial evaluator performs an iterative data-flow analysis involving abstract interpretation [13]. MapleMIX takes a highly conservative approach to specialization of dynamic loops. A simple syntactic analysis is done on the body of the loop in order to detect many cases that are unsupported. However our approach is simple to implement and still works for many real-world situations. The following situations are considered:

- 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.
- The entire body of the loop must be analyzed. If a function call exists within the loop then the body of the function would also have to be analyzed for statements that affect global state. Furthermore any function calls within its body would have to be further analyzed. 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 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. (Again the first assignment statement is to the expression $x * 1$ but was simplified to x by the automatic simplifier.)

```
iterpow_x := proc(x) local temp1;
    temp1 := x;
    temp1 := temp1 * x;
    temp1 := temp1 * x;
    temp1 := temp1 * x;
    temp1 := temp1 * x;
    temp1
end proc
```

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.

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

While loops are treated in a similar manner to for loops except unrolling will never occur. Therefore all while loops are treated in the same manner as dynamic for loops.

6.6 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 [13]. 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 it.

```
> 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. Thus it may not be possible to tell the static values of the table by looking at the printed representation of the residual program. Dynamic terms may contain static subterms as the result of reduction.

```
> x := M:-FromM(MStatic("Hello"));

                    x := _Inert_VERBATIM("Hello")

> FromInert(x);
                                "Hello"

> tbl := table([1 = "a", 2 = "b"]):
> vbtm := M:-FromM(MStatic(tbl));

                          _Inert_VERBATIM(tbl)

> t := FromInert(vbtm);
                                t := tbl

> t[1];
                                 "a"

>m := MProc(MParamSeq(MParamSpec("x", MType(), MDefault())),
MLocalSeq(), MOptionSeq(), MExpSeq(),
MStatSeq(MStandaloneExpr(MTableref(MStatic(tbl), MParam("x")))),
MDescriptionSeq(), MGlobalSeq(), MLexicalSeq(),
MEop(MExpSeq(MInt(1))), MFlags(MArgsFlag(UNKNOWN),
MNargsFlag(UNKNOWN)), MKeywords()):
```

```
> p := FromInert(M:-FromM(m));

                    p := proc(x) tbl[x] end proc;

> p(1);
                                    "a"
```

Partially static tables must still be lifted in the sense that their static parts will have been removed completely from residual programs. If an entire table need be embedded in the residual program the static part must be rebuilt. This is done by generating a sequence of assignment statements that will rebuild the table at the point where it must be residualized.

6.7 Summary

MapleMIX uses highly online strategies when specializing statements. As Maple is an imperative language, side-effects and global variables must be taken into account. Side-effects are respected by following a depth-first specialization strategy. Function sharing is used to provide better termination properties in the face of recursion. 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. Dynamic loops however are currently not treated in a precise way. In the next chapter we will take a look at the results of larger examples.

Chapter 7

Results

In this chapter the effectiveness of MapleMIX will be evaluated based on several examples. The input programs have been created specifically to demonstrate the capabilities of MapleMIX based on a wide range of criteria. Furthermore we will show how partial evaluation is an effective technique for solving the so called *specialization problem* that is found in all Computer Algebra Systems.

7.1 Quicksort

Parameterization allows the creation of generic code that can have its behavior controlled by functional parameters. Despite obvious advantages to parameterization there is usually a major disadvantage, reduced efficiency. PE allows one to write generic code and have specialized versions generated automatically. Here we present an example of a parameterized in-place quicksort algorithm found in listing 7.1. It has been written with extensibility, reusability and readability in mind. 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. This allows sorting behavior to be customized (for example the code can be used to sort ascending or descending) as well as providing extensibility as new comparison function can be provided for user defined data-types.

Listing 7.1: 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 7.2 shows a function `qs1` which calls the `quicksort` function with the following functional parameters:

- A pivot function that returns the index of the last element of the section of the array that is being sorted.
- Maple's own built-in `` `<=` `` function for use as a compare function.

Listing 7.2: 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 7.3 shows the highly specialized result of running MapleMIX on `qs1`.

Listing 7.3: Specialized QuickSort

```
1  quicksort_1 := proc(A, m, n)
2      local pivotIndex1, pivotValue1, temp1,
3            storeIndex1, i1, temp2, temp3, p;
4      if m < n then
5          pivotIndex1 := n;
6          pivotValue1 := A[pivotIndex1];
7          temp1 := A[pivotIndex1];
8          A[pivotIndex1] := A[n];
9          A[n] := temp1;
10         storeIndex1 := m;
11         for i1 from m to n - 1 do
12             if A[i1] <= pivotValue1 then
13                 temp2 := A[storeIndex1];
14                 A[storeIndex1] := A[i1];
15                 A[i1] := temp2;
16                 storeIndex1 := storeIndex1 + 1
17             end if
18         end do;
19         temp3 := A[n];
20         A[n] := A[storeIndex1];
21         A[storeIndex1] := temp3;
22         p := storeIndex1;
23         quicksort_1(A, m, p - 1);
24         quicksort_1(A, p + 1, n)
25     end if
26 end proc
```

Several things are of note:

- All non-recursive function calls have been inlined, including the functional parameters which have been integrated into the specialized program at their points of use. This is a result of MapleMIX's aggressive approach toward function unfolding.
- The `swap` function has been specialized three times and inlined in each of the three places where it was called in the original program. This is an example of function-point polyvariance.
- The local variables used by the various functions have been renamed to avoid name clash.

- MapleMIX recognized the use of a built-in function `‘<=‘` and residualized it as an operator application (`A[i1] <= pivotValue1`) instead of a function application (`‘<=‘(A[i1], pivotValue1)`).
- MapleMIX terminated and produced a correct result in the face of a recursive algorithm.

Figures 7.1 and 7.2 show the 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	bytes%
partition	1	7044	3.285	69.61	46494668	55.31
swap	1	90928	0.859	18.20	25116568	29.88
quicksort	32	14089	0.575	12.18	12445516	14.81
qs1	1	1	0.000	0.00	716	0.00
total:	35	112062	4.719	100.00	84057468	100.00

Figure 7.1: Timing results for generic quicksort `qs1`

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

Figure 7.2: Timing results for specialized quicksort_1

To further illustrate the ability to produce many specialized versions of one program we will now show the results of specializing the generic quicksort based upon a different set of design decisions. Listing 7.4 presents a function `qs2` that provides as functional parameters:

- A more advanced pivot function. The median of the first, middle and last element of the partition is chosen. This approach improves the average complexity

of the quicksort algorithm [1]. The use of three elements reduces the chance of a bad choice of pivot, especially for partially sorted data. The naive pivot function used in the previous example can result in a worst case complexity of $O(n^2)$ if the input array is already sorted. Since the true median can be computed in $O(n)$ time this approach improves wost case complexity to $O(n\ log\ n)$. However the additional overhead can be quite costly. Also notice that a nested function `middle` is used to choose the median value. The `middle` function will be part of the pivot function's closure.

- Maple's built-in greater-than function `‘>‘` as a compare function, simply for the purpose of showing how easy it is to customize sorting behavior with this approach.

Listing 7.4: Sort descending with mean pivot

```
1  qs2 := proc(A, m, n) local middle, p, c;
2
3      middle := proc(mid, y, z)
4          (mid >= y and mid <= z) or (mid >= z and mid <= y)
5      end proc;
6
7      p := proc(A, m, n) local midindex, x, y, z;
8          midindex := iquo(m+n,2);
9          x := A[m];
10         y := A[n];
11         z := A[midindex];
12         if middle(x, y, z) then
13             m
14         elif middle(y, x, z) then
15             n
16         else
17             midindex;
18         end if;
19     end proc;
20
21     c := `>`;
22     quicksort(A, m, n, p, c)
23 end proc:
```

Again MapleMIX produces a highly specialized result as expected. Note how the `middle` function has been inlined. The Maple automatic simplifier replaces the use of `>` with `<`.

[1] http://en.wikipedia.org/wiki/Quicksort

Listing 7.5: Specialized QuickSort

```
quicksort_1 := proc(A, m, n)
    local midindex1, x1, y3, z3, m4, pivotIndex1, pivotValue1,
          temp1, storeIndex1, i1, temp2, temp3, p;
    if m < n then
        midindex1 := iquo(m + n, 2);
        x1 := A[m];
        y3 := A[n];
        z3 := A[midindex1];
        if y3 <= x1 and x1 <= z3 or z3 <= x1 and x1 <= y3 then
            m4 := m
        else
            if x1 <= y3 and y3 <= z3 or z3 <= y3 and y3 <= x1 then
                m4 := n
            else
                m4 := midindex1
            end if
        end if;
        pivotIndex1 := m4;
        pivotValue1 := A[pivotIndex1];
        temp1 := A[pivotIndex1];
        A[pivotIndex1] := A[n];
        A[n] := temp1;
        storeIndex1 := m;
        for i1 from m to n - 1 do
            if pivotValue1 < A[i1] 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
```

7.2 Complexity of Algorithms

The `pow` example of Chapter 3 is a classical example in partial evaluation. However it is naive to implement powering in this way because the complexity is linear. It is illustrative to demonstrate how the complexity of an algorithm has a direct relation to the number of computations performed by the residual program. For example when the simple `pow` example is specialized with respect to $n = 72$ the result has 71 multiplications. (The 72nd multiplication is a multiplication by 1 and was removed by the Maple automatic simplifier).

Listing 7.6: Specialized `pow`

```
1 proc(x)
2     x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*
3     x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*
4     x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*
5     x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*
6     x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*
7     x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*
8     x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x*x
9 end proc
```

Listing 7.7 shows a powering algorithm with much better complexity known as *binary powering* or *fast powering*. Binary powering tests for the case when the exponent is even and recursively splits the computation in half. For example x^{72} will result in the computations $((((((x^2)^2)^2) * x)^2)^2)^2$. When the algorithm is specialized with respect to $n = 72$, as shown in listing 7.8, the result contains only seven multiplications. (Again, all multiplications by 1 are removed by the Maple automatic simplifier.) It is easy to see that the structure of the residual program matches exactly the expected computations.

Listing 7.7: Binary Powering

```
1 pow2 := proc(x, n) local y;
2     if n=0 then 1
3     elif n=1 then x
4     elif (n mod 2 = 0) then
5         y := pow2(x, n/2);
6         y*y;
7     else x*pow2(x, n-1)
8     end if;
9 end proc:
```

Listing 7.8: Specialized Binary Powering

```
1 proc(x) local y1, y2, y3, y4, y5, y6;
2     y1 := x;
3     y2 := y1 * y1;
4     y3 := y2 * y2;
5     y4 := x * y3 * y3;
6     y5 := y4 * y4;
7     y6 := y5 * y5;
8     y6 * y6
9 end proc
```

7.3 Residual Theorems in Computer Algebra

Here we present three examples of using partial evaluation as an approach to solving parametric problems in Computer Algebra Systems. We use PE to derive *residual theorems* from symbolic computation code written in Maple.

7.3.1 Degree

All Computer Algebra Systems (CAS) use *generic solutions* in their approach to certain problems. For example when asked `degree(a*x^2 + b*x + c)` Maple will respond with 2 as an answer. However this answer ignores the case where $a = 0$. 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 $a, b, c \in \mathbb{Z}$, 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* [4], 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 [6]. In our program encoding of answers the `if-then-else` statement will be used to represent the cases.

We will use the power of partial evaluation to extract so called *residual theorems* from existing code written to provide generic solutions. The first example is of a small 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.

Listing 7.9: Degree

```
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. Listing 7.10 shows a goal function that will be used to derive our results.

Listing 7.10: Degree goal

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

When called directly with symbols provided for the polynomial coefficients the `goal`

function will return 2. However, when it is partially evaluated with no inputs given the result is a residual program representation of the desired result.

Listing 7.11: Residual Theorem

```
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
```

We provide two more examples of MapleMIX returning a residual program answer for different polynomials. The following examples show that statically known cases, i.e. when the application of `coeff` can be statically reduced to 0, are specialized away leaving the desired result.

Listing 7.12: Residual Theorem

```
goal2a := proc(a,b) local p;
    p := a*x^17+b*x^12;
    mydegree(p, x);
end proc;

proc(a, b)
    if   a <> 0 then 17
    elif b <> 0 then 12
    else -infinity
    end if
end proc
```

Listing 7.13: Residual Theorem

```
goal2c := proc(a)
    local p;
    p := (a-5)*x^17+(a^2-1)*x^12+3*x;
    mydegree(p, x);
end proc;

proc(a)
    if   a - 5   <> 0 then 17
    elif a^2 - 1 <> 0 then 12
    else 1
    end if
end proc
```

7.3.2 Gaussian Elimination

Now we move on to a more complicated example from Linear Algebra. Listing 7.14 provides a simple Maple implementation of fraction-free Gaussian Elimination for augmented matrices represented as Maple tables.

Listing 7.14: 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;
        # interchange row i with row r is necessary
        if pivot>-1 then
            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:
```

In order to illustrate the generation of residual theorems from this program we will use the following matrix as a running example.

$$\begin{bmatrix} 1 & -2 & 3 & 1 \\ 2 & x & 6 & 6 \\ -1 & 3 & x-3 & 0 \end{bmatrix}$$

When the GE algorithm is used directly it produces a generic solution, however when the special cases $x = 0$ and $x = -4$ are considered the results should be different.

$$\text{generic solution} \begin{bmatrix} 1 & -2 & 3 & 1 \\ 0 & x+4 & 0 & 4 \\ 0 & 0 & x(x+4) & x \end{bmatrix}$$

$$\text{when } x = -4 \begin{bmatrix} 1 & -2 & 3 & 1 \\ 0 & 4 & 0 & 4 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad \text{when } x = 0 \begin{bmatrix} 1 & -2 & 3 & 1 \\ 0 & 1 & -4 & 1 \\ 0 & 0 & 0 & 4 \end{bmatrix}$$

Listing 7.15 provides a goal function that will call the above gaussian elimination algorithm on the example matrix represented as a standard Maple table.

Listing 7.15: Matrix represented as a table

```
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:
```

In the residual program, shown in listing 7.16, the cases to consider naturally drop out of the computations. By tracing the statements of the residual program it becomes clear the the first if statement is testing for the case where $x = -4$ (which is handled in the else branch). The inner if is testing for $x(x + 4) = 0$, however since the first guard ensures that $x <> -4$ this is really a test for $x = 0$.

Listing 7.16: Gaussian Elimination Result

```
ge_res := 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
```

7.3.3 Symbolic Integrator

Listing 7.17 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 coefficients 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` (and thus the partial evaluator does not look at the code, but in other situations this could result in calls to `ln(1)` being embedded in the residual program). In the second case, there is nothing to do as this branch is never taken, and thus never examined.

Listing 7.17: Symbolic 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:
```

Listing 7.18 provides a goal function that will call the integrator. Here a polynomial is represented as a list of lists. Again the ability to handle partially static data is required to derive highly specialized results.

Listing 7.18: Calling Symbolic Integrator

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

The result shows the cases we expect, with the guard case depending on whether $n = -1$ or not. The call to `ln` has been residualized without the PE attempting to specialize the body of `ln`.

Listing 7.19: Symbolic Integrator 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
```

7.4 First Futamura Projection

The first Futamura projection states that compilation can be achieved given a partial evaluator, an interpreter and a source program written for the interpreter. This projection states nothing about the quality of the resulting "compiled program", intuitively it is essentially the interpreter specialized to run just one program. Therefore the compiled program will contain pieces of the interpreter, however its structure will resemble that of the source program. Since most interpreters tend to contain a great deal of computational overhead the compiled program is likely to be clunky and inefficient compared to a functionally equivalent program written directly in the language that the interpreter is written in. Running the compiled program directly will however be more efficient than running the original source program on the interpreter.

Here we present an example of the first Futamura projection in action. The purpose of this example is twofold, first to demonstrate an interesting property of partial evaluation using a concrete example, and secondly to assert the correctness of MapleMIX using a well established acceptance criteria for partial evaluators. Listing 7.20 shows a simple interpreter written in Maple for a minimal language consisting of if-expressions, binary operators, function definitions, function calls and simple bindings.

Listing 7.20: Interpreter written in Maple

```
1  MiniMapleInterpreter := module() option package;
2      export ModuleApply;  local evalStat, evalExpr, evalBin;
3
4      ModuleApply := proc(prog, input) local defs, d;
5          defs := table();
6          for d in prog do defs[op(1,d)] := d end do;
7          evalStat(op(3,op(1,prog)), input, defs);
8      end proc;
9
10     evalStat := proc(s, env, defs) local h, t, c, var, e1;
11         h := op(0, s);
12         if h = mmIfElse then
13             c := evalExpr(op(1,s), env, defs);
14             if c then evalStat(op(2,s), env, defs);
15             else evalStat(op(3,s), env, defs);
16             end if;
17         elif h = mmExpr then
18             evalExpr(op(1,s), env, defs);
19         else error "unknown statement form: %1", h;
20         end if;
21     end proc;
22
```

```
23      evalExpr := proc(e, env, defs)
24          local h, e1, e2, o, def, ags, newEnv, param, i;
25          h := op(0, e);
26          if h = mmInt or h = mmString or h = mmName then
27              op(1,e);
28          elif h = mmVar then
29              env[op(1,e)]
30          elif h = mmBin then
31              o := op(1,e);
32              e1 := evalExpr(op(2,e), env, defs);
33              e2 := evalExpr(op(3,e), env, defs);
34              evalBin(o, e1, e2);
35          elif h = mmUn then
36              o := op(1,e);
37              e1 := evalExpr(op(2,e), env, defs);
38              evalUn(o, e1);
39          elif h = mmCall then
40              def := defs[op(1,e)];
41              ags := op(2,e); i := 1; newEnv := table();
42              for param in op(2,def) do
43                  newEnv[param] := evalExpr(op(i,ags), env, defs);
44                  i := i + 1;
45              end do;
46              evalStat(op(3,def), newEnv, defs);
47          else error "unknown expression form: %1", h;
48          end if;
49      end proc;
50
51      evalBin := proc(mm, e1, e2)
52          if   mm = mmEq    then evalb(e1 = e2)
53          elif mm = mmPlus  then e1 + e2
54          elif mm = mmTimes then e1 * e2
55          elif mm = mmAnd   then e1 and e2
56          elif mm = mmOr    then e1 or e2
57          else error "unknown binary operator: %1", mm;
58          end if;
59      end proc;
60  end module:
```

Listing 7.21 shows the classic powering function example coded in the language of our simple interpreter. This may seem like a simple example however the termination properties of MapleMIX will be tested as this is a recursive program and all of the program's variables will be dynamic. Even this simple example uses almost all the functionality of the interpreter (which would involve a great deal of overhead when executed on the interpreter).

Listing 7.21: Powering function

```
power := mmProgram(
  mmDef("pow", mmParams("x", "n"),
    mmIfElse(mmBin(mmEq, mmVar("n"), mmInt(0)),
      mmExpr(mmInt(1)),
      mmExpr(
        mmBin(mmTimes,
          mmVar("x"),
          mmCall("pow",
                 mmArgs(mmVar("x"), mmBin(mmPlus, mmVar("n"), mmInt(-1)))
          ))))))
```

Listing 7.22 shows a goal function that calls the interpreter on the example program. Note that the interpreter requires the program's input to be passed as a table.

Listing 7.22: Calling interpreter on powering function

```
goal := proc(x, n) local t;
    t := table(["x" = x, "n" = n]);
    MiniMapleInterpreter(power, t);
end proc;
```

Listing 7.23: Compiled code using interpreter and partial evaluator

```
compiled_pow := module()
    ModuleApply := proc(x, n) local t;
        t["x"] := x;
        t["n"] := n;
        evalStat_1(t)
    end proc;

    evalStat_1 := proc(env) local e12, c, e16, newEnv1, e14, e22;
        e12 := env["n"];
        c := evalb(e12 = 0);
        if c then
            1
        else
            e16 := env["x"];
            newEnv1["x"] := env["x"];
            e14 := env["n"];
            newEnv1["n"] := e14 - 1;
            e22 := evalStat_1(newEnv1);
            e16*e22
        end if
    end proc
end module;
```

Listing 7.23 shows the compiled program that is the result of running the partial evaluator on the interpreter and the example program. As expected pieces of the interpreter are left over, in particular there are several residual statements that are concerned with passing around the environment.

As a further example we provide a goal function, Listing 7.24, which provides a static value 5 for n. The purpose is to show that compilation and specialization of the source program can be achieved at the same time. The resulting program, shown in Listing 7.25, computes x^5. Again the environment passing behavior of the interpreter has been residualized.

Listing 7.24: Calling interpreter on powering function

```
goal := proc(x) local t;
    t := table(["x" = x, "n" = 5]);
    MiniMapleInterpreter(power, t);
end proc;
```

Listing 7.25: Compiled code and specialized code

```
proc(x)
    local t, e110, newEnv5, e18, newEnv4, e16, newEnv3,
    e14, newEnv2, e12, e22, e24, e26, e28, e210;
    t["x"] := x;
    e110 := t["x"];
    newEnv5["x"] := t["x"];
    e18 := newEnv5["x"];
    newEnv4["x"] := newEnv5["x"];
    e16 := newEnv4["x"];
    newEnv3["x"] := newEnv4["x"];
    e14 := newEnv3["x"];
    newEnv2["x"] := newEnv3["x"];
    e12 := newEnv2["x"];
    e22 := 1;
    e24 := e12*e22;
    e26 := e14*e24;
    e28 := e16*e26;
    e210 := e18*e28;
    e110*e210
end proc;
```

7.5 Summary

In this chapter we have shown the effectiveness of MapleMIX on a variety of Maple programs. We have shown how MapleMIX can be used to optimize a program that was written in a generic way, as demonstrated by the quicksort example. Effective use of function unfolding and constant propagation removed a great deal of the runtime cost of the generic algorithm. The termination properties of MapleMIX were also tested by this example, and it was shown that the simple function sharing mechanism was enough to ensure termination in the face of a recursive quicksort procedure.

Use of partial evaluation as a means of solving the so called specialization problem in Computer Algebra Systems has been demonstrated. We believe that PE has a great potential to provide results in this field. Use of residual programs as answers to parametric problems is an effective way to derive the needed results. We have shown three examples; degree, Gaussian Elimination and an integrator that demonstrate the effectiveness of PE as a solution. Our special treatment of partially static data structures such as lists and polynomials was crucial to derive these results.

We have also demonstrated the power of partial evaluation to generate "compiled programs" from interpreters. This asserts the correctness of MapleMIX. In the next chapter we will discuss future direction for further development of MapleMIX.

Chapter 8

Future Work and Conclusions

8.1 Supported Constructs

Currently MapleMIX does not provide support for modules, while it does provide limited support for function closures. These two constructs are very similar in their internal representations. It should be a straightforward extension to provide limited support for modules in a similar way as function closures. However it would be even better to discover a new approach that is not so conservative. Currently dynamic lexical locals are not supported. The main problem is in propagating the dynamic context from one part of the program to another. This problem has been solved in an offline setting using BTA to provide the specializer with prior knowledge that certain lexical locals will be dynamic [28]. In an online setting there is no prior knowledge of this, and so the specializer has no easy way to know that a dynamic variable will have to be passed around the residual program. It would be a great advancement in the area of partial evaluation to discover an elegant solution to this problem in an online setting.

MapleMIX provides very little support for expression sequences. They hinder the specializer in many ways. Firstly, matching up arguments expressions to parameters in a function application becomes highly unreliable when expression sequences are considered. Furthermore our online approach to partially static data structures would also be hindered because we would have to consider the case where dynamic variables within dynamic representations could be expression sequences at runtime. Expression sequence flattening is a highly non-robust aspect of Maple in general, it is rather

disappointing that the language supports this "feature" at all.

Last name evaluation (LNE) is also a problem. Special care must be taken so that raw values that are subject to LNE rules do not pick up a name from within the partial evaluator. This has provided a great source of difficulty while writing MapleMIX. This is also considered to be a rather dubious feature of the Maple language, and like expression sequences, only gets in the way of partial evaluation.

Maple is a very large language. Adding support for additional language features to MapleMIX would naturally extend its scope of application. Currently MapleMIX does not work reliably on certain sections of the Maple core library, mostly due to the use of modules and problems with LNE. If these problems were to be solved the scope of MapleMIX would be dramatically improved.

The try/catch construct currently has very limited support. This is a very difficult to handle construct because it involves interprocedural control flow. It is possible to add better static support for exceptions. Currently MapleMIX maintains a call stack of functions in the process of specialization. If a static exception were to occur under the right circumstances then it may be possible to search the stack for an appropriate handler. Currently MapleMIX treats a static exception by aborting specialization. While we have not found this limiting in our experiments better support for try/catch would be required to tackle larger programs.

8.2 Feature Conflict

As explained in Chapter 5, two of the main features of MapleMIX do not currently work at the same time. Function sharing requires the computation of call signatures and currently there is no support for considering dynamic representations in call signatures. It would be straightforward to add this support, however that would harm the termination properties of the partial evaluator. A more sophisticated approach to computing call signatures is recommended for future work. The idea would be to record what properties of the initial environment are actually used during the specialization of a function. If, for example, the value 5 were passed to a function, but the function only ever tested that the value was odd, then the resulting specialized code would work with any value that was an odd integer. This would also need to be extended to record what aspects of dynamic representations are used. The result would be better termination properties and more specialized code reuse.

8.3 Additional Features

Several additional features are recommended for consideration for future work. While adding support for additional language features is a natural extension it must also be noted that quality of specialization is directly related to the ability to maintain and infer static data.

8.3.1 Post-processing

Post-processing of residual code is performed as a clean-up phase. Dead code is removed and a very small amount of additional inlining is performed. Currently MapleMIX uses very simple algorithms in its post-processing of residual code. Extending these to more sophisticated algorithms would improve the results. For example, data-flow techniques could be used to remove dead code.

8.3.2 Iterative Specialization

In some of our experiments we have found that it is possible to take code that was output from MapleMIX and specialize it again. This has shown to produce small amounts of additional specialization in some cases. We believe that running the specialization core as an iterative fixed point will produce better specialization. Translation to and from M-form need only be performed once while the specialization phase repeats iteratively on the intermediate representation.

If this approach were to be taken it would be possible for the specializer to gather additional information during specialization. For example to track the number of uses of each variable name. On subsequent iterations this information would be available and would be quite useful. For example the specializer could inline assignments in the case that it knows a variable name is used exactly once. An online PE like MapleMIX is very reactionary in its nature. The lack of a pre-process means the specializer proceeds completely unaware of statements it has yet to encounter. An iterative approach would provide useful static information to each pass after the first one. This approach has the potential to eliminate the need for a post-processing phase.

8.3.3 Dynamic Loops

MapleMIX has excellent support for static loops in the sense that an incredibly large amount of specialization is possible. However support for dynamic loops is currently very limited. Development of a sophisticated approach to dynamic loops is recommended for future work.

8.3.4 Types and Shapes

While MapleMIX is prided for being a completely online partial evaluator we are not opposed to adding pre-phase analyses that would benefit specialization. Hybrid approaches to partial evaluation have used shape or representation analyses to gather some static information before specialization. MapleMIX is capable of determining some type and shape information in an online way through its unique support for partially static data structures. However a proper pre-phase shape analysis would still benefit the specialization phase by providing more information. Furthermore it has become apparent that type information would also be a huge benefit to the specializer. A type inference phase would greatly improve results, especially on Maple code that does a great deal of dynamic type checking.

8.3.5 Output Language

Allowing a certain flexibility in the output language is often a good way to solve challenging problems in partial evaluation. The Java partial evaluator JSpec produces aspect-oriented programs in the AspectJ language as output [30]. AOP is an extension to the Object Oriented paradigm that improves separation of concerns by allowing code that implements crosscutting concerns to be grouped into a separate programming unit called an aspect [17].

JSpec is concerned with specializing program slices, then reinserting the specialized slices back into the original program. Part of the specialization process is concerned with safely bypassing encapsulation mechanisms for the purpose of improved efficiency. Specialized methods are not inserted into the class for which the original unspecialized method was defined because the new method would now be available to the rest of the program breaking encapsulation. Instead specialized methods are

grouped into aspects. Access modifiers on the aspect ensure that specialized methods are only called by other specialized methods in the same aspect. The aspect is then weaved into the executable program by the AspectJ compiler. In this case allowing the output language of the partial evaluator to be an extended version of the input language provides a simple solution to an otherwise difficult problem.

The online partial evaluator FUSE generates a residual program by returning a trace of suspended computations [11]. This trace is represented as a graph rather than program text. A separate process can then be used to translate the graph into a particular programming language. This strategy allows a choice of output language as well as facilitating the removal of duplicate computations.

Maple also allows flexibility in how the output program is represented. Any Maple value may be embedded directly into the residual program via *_Inert_VERBATIM*. MapleMIX exploits this to avoid the problem of lifting when it can. It is possible for MapleMIX to generate code that the Maple parser would not generate. This has lead us to believe that taking full advantage of the flexibility of *_Inert_VERBATIM* should be pursued.

8.4 Conclusions

Chapter 7 showed some very promising results. We believe that partial evaluation is a viable approach toward the optimization and specialization of Maple programs. It is necessary to extend our approach to make it applicable to sections of the Maple core library. Unfortunately the core library was not written with partial evaluation in mind and as a result there are a huge amount of language features that would have to be supported in order to use partial evaluation in this setting. Because of the high-level nature of the Maple language, partial evaluation is a very viable optimization technique with the potential to produce impressive results.

8.4.1 Contributions

Our contributions have been:

- An online partial evaluator for a non-trivial subset of the Maple programming

language.

- An approach to modularizing the partial evaluator using a high level intermediate representation called M-form. We have demonstrated that using appropriate syntax transformations can make the specializer easier to implement.
- An online environment that is implemented as a stack of settings. This approach eliminates the need to copy environments and makes the restoration of the environment to an earlier state an extremely simple operation. Our approach to specialization of dynamic if-statements works hand in hand with our online environment.
- Support for partially static data structures such as lists and polynomials implemented in a completely online way.
- A novel on-the-fly syntax transformation approach to static for loops that leads to a high level of specialization and removes the need to merge environments.
- Demonstration that PE can be used to provide precise solutions to parametric problems encountered in Computer Algebra Systems.

Appendix A

M-form Abstract Syntax

NAME ::= LOCAL | GLOBAL

LOCAL ::= *MLocal* STRING
| *MParam* STRING
| *MSingleUse* STRING
| *MGeneratedName* STRING
| *MLexicalLocal* STRING
| *MLexicalParam* STRING
| *MLocalName* STRING
| *MAssignedLocalName* STRING

MNAME ::= *MName* STRING

GLOBAL ::= MNAME
| *MAssignedName* STRING

EXPSEQ ::= *MExpSeq* { EXPR }

UNARYOP ::= *MNot* EXPR – logical not (not)

BINARYOP ::= *MEquation* EXPR EXPR – equation and equals (=)
| *MInequat* EXPR EXPR – inequation and notequal (<>)
| *MPower* EXPR EXPR – power (x^n)
| *MCatenate* EXPR EXPR – name concatenation (||)
| *MLesseq* EXPR EXPR – less than or equal (<=)
| *MLessThan* EXPR EXPR – less than (<)
| *MImplies* EXPR EXPR – logical implies (implies)
| *MAnd* EXPR EXPR – logical and (and)
| *MOr* EXPR EXPR – logical or (or)
| *MXor* EXPR EXPR – logical xor (xor)
| *MRational* EXPR EXPR – rational number (1/2)
| *MComplex* EXPR EXPR – complex number (5 + 4 * I)
| *MRange* EXPR EXPR – range (x..y)

NARYOP ::= *MSum* { EXPR} – sum (x + y + z)
| *MProd* { EXPR} – product (x * y * z)

FUNCTIONCALL ::= *MFunction* NAME EXPSEQ

TABLEREF ::= *MTableref* NAME EXPSEQ

STRINGLIT ::= *MString* STRING

EXPR ::= NAME
| EXPSEQ
| UNARYOP
| BINARYOP
| NARYOP
| FUNCTIONCALL
| TABLEREF
| PROC
| STRINGLIT
| *MInt* INT
| *MFloat* INT INT
| *MList* EXPSEQ
| *MSet* EXPSEQ
| *MArgs*
| *MNargs*
| *MUneval* EXPR – unevaluation quotes ('x')
| *MMember* EXPR NAME – module member access (module:-member)

STMT ::= LOOP
| DRIVER
| *MStatSeq* { STMT }
| *MStandaloneExpr* EXPR
| *MAssign* NAME EXPR
| *MAssignToTable* NAME TABLEREF
| *MAssignTableIndex* TABLEREF EXPR
| *MAssignToFunction* NAME FUNCTIONCALL
| *MStandaloneFunction* FUNCTIONCALL
| *MIfThenElse* EXPR STMT STMT
| *MReturn* EXPR
| *MError* EXPSEQ – throws an exception
| *MTry* STMT { *MCatch* STMT } STMT – try/catch/finally
| *MCommand* STRING – special command to MapleMIX
| *MRef* POINTER – DAG pointer

LOOP ::= *MWhile* EXPR EXPR EXPR EXPR STMT
| *MWhileForFrom* EXPR EXPR EXPR EXPR EXPR STMT
| *MWhileForIn* EXPR EXPR EXPR STMT

DRIVER ::= *MForFromDriver* POINTER POINTER LOCAL EXPR EXPR EXPR
| *MForInDriver* POINTER POINTER LOCAL LOCAL EXPR EXPR

TYPE ::= *MType* MAPLETYPE

DEFAULT ::= *MDefault* EXPR

PARAMSPEC ::= *MParamSpec* STRING TYPE DEFAULT

PARAMSEQ ::= *MParamSeq* { PARAMSPEC }

KEYWORDPARAMSEQ ::= *MParamSeq* { PARAMSPEC }

LOCALSEQ ::= *MLocalSeq* { MNAME }

OPTIONSEQ ::= *MOptionSeq* { MNAME }

DESCRIPTIONSEQ ::= *MDescriptionSeq* { STRINGLIT }

GLOBALSEQ ::= *MGlobalSeq* { MNAME }

LEXICALPAIR ::= *MLexicalPair* MNAME MNAME

LEXICALSEQ ::= *MLexicalSeq* { LEXICALPAIR }

NULL ::= *MExpSeq*

FLAG ::= *true* | *false* | *UNKNOWN*

ARGSFLAG ::= *MArgsFlag* FLAG

NARGSFLAG ::= *MNargsFlag* FLAG

ARGSFLAGS ::= *MFlags* ARGSFLAG NARGSFLAG

PROC ::= *MProc*

PARAMSEQ	– sequence of parameter names
LOCALSEQ	– sequence of names of local variables
OPTIONSEQ	– sequence of options
NULL	– the remember table
STMT	– the procedure body
DESCRIPTIONSEQ	– documentation
GLOBALSEQ	– global variables declared by the procedure
LEXICALSEQ	– lexical locals
EOP	– execution order permutation
ARGSFLAGS	– flags that indicate if the procedure uses args or nargs
KEYWORDPARAMSEQ	– list of keyword parameters

Bibliography

[1] L. O. Andersen, "C program specialization," tech. rep., DIKU, University of Copenhagen, May 1992.

[2] K. R. Anderson, "Freeing the essence of a computation," *ACM SIGPLAN Lisp Pointers*, vol. Volume VIII, no. Issue 2, May 1995.

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

[4] C. Ballarin and M. Kauers, "Solving parametric linear systems: an experiment with constraint algebraic programming," *SIGSAM Bull.*, vol. 38, no. 2, pp. 33–46, 2004.

[5] L. Birkedal and M. Welinder, "Partial evaluation of standard ml," Master's thesis, DIKU, University of Copenhagen, October 1993.

[6] J. Carette, "Understanding expression simplification," in *ISSAC '04: Proceedings of the 2004 international symposium on Symbolic and algebraic computation*, (New York, NY, USA), pp. 72–79, ACM Press, 2004.

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

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

[9] N. H. Christensen and R. Gluck, "Offline partial evaluation can be as accurate as online partial evaluation," *ACM Trans. Program. Lang. Syst.*, vol. 26, no. 1, pp. 191–220, 2004.

[10] C. Consel, J. L. Lawall, and A.-F. L. Meur, "A tour of tempo: a program specializer for the c language," *Science of Computer Programming*, vol. volume 52, 2004.

[11] E. R. Daniel Weise, Roland Conybeare and S. Seligman, "Automatic online partial evaluation," in *Proceedings of the Conference on Functional Programming Languages and Computer Architecture, 1991*, June 1991.

[12] O. Danvy, "Type-directed partial evaluation," in *Partial Evaluation - Practice and Theory, DIKU 1998 International Summer School*, June 1998.

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

[14] Y. Futamura, "Partial evaluation of computation process- an approach to a compiler-compiler," *Higher Order Symbol. Comput.*, vol. 12, no. 4, pp. 381–391, 1999.

[15] F. Henglein, "Efficient type inference for higher-order binding-time analysis," in *Functional Programming Languages and Computer Architecture (FPCA'91)*, August 1991.

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

[17] R. Laddad, *AspectJ in Action*. Greenwich, Conn, USA: Manning Publications, 2003.

[18] T. Mogensen, "Self-applicable partial evaluation for pure lambda calculus," in *Partial Evaluation and Semantics-based Program Manipulation*, 1992.

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

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

[21] E. Ruf and D. Weise, "Preserving information during online partial evaluation," Tech. Rep. CSL-TR-92-517, Stanford University, April 1992.

[22] E. Sumii, "Bridging the gap between tdpe and sdpe." Presentation slides.

[23] E. Sumii and N. Kobayashi, "Online type-directed partial evaluation for dynamically-typed languages," *Computer Software, Iwanami Shoten, Japan*, vol. 17, no. 3, pp. 38–62, May 2000.

[24] E. Sumii and N. Kobayashi, "A hybrid approach to online and offline partial evaluation," *Higher-Order and Symbolic Computation, Kluwer Academic Publishers, the Netherlands*, vol. 14, no. 2/3, pp. 101–142, 2001.

[25] W. Taha, "A gentle introduction to multi-stage programming.," in *Domain-Specific Program Generation*, pp. 30–50, 2003.

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

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

[28] P. Thiemann and D. Dussart, "Partial evaluation for higher-order languages with state." July 1999.

[29] P. Thiemann and R. Gluck, "The generation of a higher-order online partial evaluator," in *Fuji Workshop on Functional and Logic Programming*, pp. 239–253, July 1995.

[30] J. L. L. Ulrik Pagh Schultz and C. Consel, "Automatic program specialization for java," *Transactions on Programming Languages and Systems (TOPLAS)*, no. 25(4), 2003.