UNIVERSITY OF SOUTHAMPTON

Partially Evaluating MATLAB

by

Daniel R. Elphick

A thesis submitted in partial fulfillment for the
degree of Doctor of Philosophy

in the
Faculty of Engineering, Science and Mathematics
School of Electronics and Computer Science

June 2005

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING, SCIENCE AND MATHEMATICS
SCHOOL OF ELECTRONICS AND COMPUTER SCIENCE

Doctor of Philosophy

by Daniel R. Elphick

More and more scientific code is being written in higher level languages than the traditional choice of Fortran. Such languages are more accessible to less computer science minded people and bring with them large libraries, geared towards solving mathematical problems. MATLAB is such a language and allows the rapid development of high performance codes. In this thesis, we describe the problems associated with the creation of high performance code for mathematical computations. We discuss the advantages and disadvantages of using a high level language like MATLAB and then propose partial evaluation as a way of lessening the disadvantages at little cost. Partial Evaluation is a program transformation technique, which propagates constants and performs static computations, resulting in faster residual programs. We present our approach to partially evaluate MATLAB programs using the online methodology, whereby decisions about static or dynamic values are made at specialisation time rather than as a pre-process. We also describe an implementation called MPE (MATLAB Partial Evaluator). As well as propagating static values to produce fast residual programs, MPE uses extensive shape and type inferencing to extract static information from otherwise dynamic contexts. Partially evaluated programs frequently contained redundant code, necessitating a post-processing phase to prevent irrelevent computations at run-time and also to reduce the amount of code produced. The post-processor is able to remove much of this redundancy by a combination of dead code elimination using ud-chains and duplicate function detection, using structural equivalency. To ascertain the efficacy of our approach, we obtained empirical results for a variety of real MATLAB programs. These results showed performance increases in various programs, from the relatively simple Chebyshev approximation and Lagrange Interpolation codes (with speed-ups of up to 100%) to the several different Ordinary Differential Equations solvers (speed-ups range from 57% to 568%) to a Computational Fluid Dynamics solver, which was sped up by 92%. Since these results were produced with a minimal of manual interference, we suggest that partial evaluation is a viable automated technique for use by those actually coding scientific codes.

Contents

Listings

Nomenclature

$\mathbb{N}$	The set of positive integers including 0.
$\mathbb{Z}^+$	The set of positive integers excluding 0.
$M^{d_1 \times d_2 \times \cdots \times d_n}$	The set of matrices with n dimensions and dimension lengths $d_1 \ldots d_n$ composed of elements from M.

Acknowledgements

I am very grateful to my two supervisors Professor Michael Leuschel and Professor Simon Cox for allowing me the opportunity to do this research and for their help and direction in reaching this final milestone.

I would also like to thank Dan Underwood, who provided the Mars Lander experiment code, which proved to be a very useful benchmark and testing platform, and Professor Mike Giles at Oxford University, who provided the Arcadia CFD code.

Finally, I could never have gotten this far without the help of my parents, who have supported me both financially and emotionally throughout.

Chapter 1

Introduction

Scientific computing can be described as the use of computers to solve scientific problems. As computers have become cheaper and faster, it has become possible to model ever more complicated phenomena with ever more accuracy. Rarely is any large engineering effort undertaken today, without some model first being built, which can be tested to find flaws before anything real has been constructed.

1.1 The Problems of Scientific Computing

Problems range in complexity from calculating the amount of thrust required to put a satellite into a desired orbit to calculating the effects of "greenhouse gases" on the temperature of the earth over a hundred years. Even medicine is benefiting from advances in computing, as biologists can now model how proteins "fold" using large scale distributed computing efforts and then use the information to find out what causes diseases such as Alzheimer's and BSE [22].

At the core of much of this work is the production of new algorithms to solve previously insoluble problems. Doubling the clock speed of a processor can at best double the speed of a model. Models are however rarely linear and so doubling the size of a model will often have a far larger effect on the time taken to compute it. Replacing an algorithm with cubic time complexity by one with quadratic time complexity can greatly increase the size of problems that are manageable.

To this end, the barriers to the construction of algorithms need to be broken down so that scientists do not struggle with problems that need not concern them. If algorithms exist to perform linear algebra operations and which fully take advantage of the hardware available, then those algorithms should be made readily available in the form of environments which make them integral.

Unforgiving environments, designed for ease of implementation by compiler writers in mind, can rarely offer users the flexibility and the ease of use desired by users without computer science backgrounds since their focus is necessarily different. Learning an arcane syntax which does not easily map to their other scientific experiences should be avoided.

Scientific computing is moving away from hand-coded, hand-optimised codes, written in low to medium level languages like FORTRAN, towards general codes written in high-level domain specific programming languages or using problem solving environments, e.g. MATLAB [75], Maple [46] MATHEMATICA [47]. This trend is also apparent in general purpose computing, where languages like Python are now being adopted for projects traditionally written in C/C++, since the large standard library includes string processing and network communications in a way that saves users knowing the specific architectural features of the computer or operating system [27].

With the current diversity of computer architectures (due to differences in cache configurations, speeds and memory bandwidths as well as processor types), it is becoming increasingly difficult to produce near optimal code without expending excessive time. Even then the code ties the developer to one architecture, which is no longer ideal in an era of heterogeneous systems. With high-level languages and integrated problem solving environments, these details are both solved and yet hidden from the user, greatly easing their work.

1.2 MATLAB as a Solution

The use of high level languages allows rapid prototyping to ease development, while hiding the complex implementation details from the user, who is only interested in getting fast (both in execution and implementation) and accurate results. Users will happily sacrifice some execution speed in the initial development period, if it means they can quickly produce a working solution. This applies not just to the choice of platform but the development methodology, where general solutions are sought which can be applied to a wide variety of problems albeit slowly due to their generality. The approach results in programs which can contain redundant and irrelevant computations, thus causing significant reductions in performance. Later, once the prototype has been verified, serious optimisation efforts can be made, often by rewriting the program in a language like FORTRAN, which offers low level control.

MATLAB easily fills this hole in the development model. It closely maps to mathematical notation and treats matrices and multi-dimensional arrays as "first-class citizens", so that they can be manipulated with the ease of scalars in more traditional languages. It relies on highly optimised libraries such as ATLAS [80] for linear algebra routines and

FFTW [23] for Fast Fourier Transforms, so that users never have to worry about the hand-coding of such fundamental operations.

Programs in MATLAB tend to be very general and multi-purpose, using input parameters to determine and configure which particular method is to be used in a particular invocation. A familiar example of this might be a general purpose numerical library, where selection and configuration of an eigenvalue solver is done at run time by setting some parameter values. However, it also happens in full-scale engineering calculations – a particular example arises in the process of design search and optimisation, in which the performance of a device, system, or component is systematically improved by a series of computational simulations. At the heart of this process is a device and the computational experiment needed to analyse its performance, which is described by a large number of variable parameters, which can each be modified to change some aspect of either the device itself (e.g. its size/material/topology) or the analysis process (resolution of simulation, number of iterations, etc.). The design process consists of a number of studies, in each of which the code is executed a number of times with a (different) large subset of the parameters fixed while the remaining ones are varied at run-time. In each of these cases, we believe that partial evaluation could offer performance benefits to users.

1.3 Partial Evaluation

Partial evaluation is the process of transforming programs by the early specification of some of the program parameters. The resulting programs contain the precomputed results of calculations that depended only on the pre-specified parameters; since the new program does not need to perform these calculations itself, it is usually faster than the original program. Optimisations such as loop unrolling, previously impossible due to lack of information, become possible.

With component-based software engineering, programmers are encouraged to create general code which can be reused, thus saving later development time. The trade-off is that performance can suffer. With access to an effective partial evaluator, software engineers can produce the general code that will ultimately lower their production costs and yet often achieve the performance of a bespoke system.

High level code optimisation and partial evaluation techniques are quite different from the low level optimisation done, for instance, on the assembly language produced by C compilers. High level languages often provide special functionality not provided in low or medium level languages which needs to be exploited to get maximum performance: e.g. vectorisation in MATLAB and also FORTRAN 90. However unlike in low level languages, there are few opportunities to influence the exact details of basic operations like arithmetic operations. Indeed in a typical well written MATLAB program, most of the computation is done inside the run-time libraries, which we cannot influence. Since these libraries are very general, they must cater to many eventualities including invalid or incompatible parameters, in which case checks must be performed first regardless of whether we can infer that they are redundant. Recent efforts involving *Just-In-Time* compilation seem to offer a solution to this problem, but there are often static aspects of programs which, if found before execution, can be dealt with effectively using partial evaluation.

More commonly partial evaluation has been applied to the more academic side of computer science. The focus of most research has been on functional and declarative languages, like Scheme [37] and Prolog [45], although partial evaluators do exist for C [5] and Java [64]. While there has been research on numerical applications for Fortran [41], this work has not been continued recently, and certainly has not been updated for languages like MATLAB and Maple.

1.4 Thesis Aims and Outline

The aim of this project is to address these issues by taking general programs written (or produced by other programs) in high level languages such as MATLAB and producing code that executes more quickly using partial evaluation and other high level code optimisations.

The main contribution of this work is an online partial evaluator for MATLAB called MPE (MATLAB Partial Evaluator). This is a program transformation tool, which takes a MATLAB program and produces a new one specialised according to some specified static data. This transformation is guided by our extensive abstract interpretation system, which allows information about the dynamic types of MATLAB to be captured. The tool is firmly grounded in practicality and so eschews some of the more theoretical decisions taken to achieve results such as self-application as this is superfluous to our goal of speeding up mathematical codes.

In combination with our partial evaluation techniques, we also demonstrate an effective post-processing phase, which combines dead code removal using ud-chains and function duplication elimination via structural equivalency detection.

This work demonstrates that program specialisation can effectively improve the performance of MATLAB programs without placing too much burden on the end-user. Typical results for non-trivial systems such as the solution of ordinary differential equations using the generic solver provided as part of MATLAB show that partial evaluation can give 100% increases in speed. When the MATLAB compiler is used, speed-ups are retained mostly intact and in some cases compilation in combination with partial evaluation gives much larger performance increases.

In Chapter 2, we examine the MATLAB programming language, looking at its structure and performance issues as well as examining previous work on increasing execution speed using techniques like parallelisation and compilation. In Chapter 3, we examine existing work in the field of partial evaluation. Chapter 4 presents a formal analysis of the type system used in MPE. The loading and parsing of a MATLAB program including all of the supporting functions is described in Chapter 5. The core partial evaluation details are given in Chapter 6. Chapter 7 describes how we remove dead code and perform other post-processing optimisations. In Chapter 8, we demonstrate the effectiveness of automatic partial evaluation by applying our tool to several test programs and comparing timings. Future work that could enable further improvements is described in Chapter 9 and our final conclusions are presented in Chapter 10. Some of the early work of Chapters 4, 6, 7 and 8 appeared in [20].

Chapter 2

Overview of MATLAB

MATLAB is a problem solving environment sold by The Mathworks [75], whose users world-wide have grown in number from 400,000 in 2000 [72], to around 1,000,000 today [71].

The Mathworks sells MATLAB to a variety of industries, from the aerospace and defence industry, to communication equipment manufacturers, to semi-conductor suppliers [73]. In the Aerospace and Defence industries, MATLAB and its related tools are used to "provide a flexible software environment for designing multidomain systems simulating high fidelity behavioral dynamics, testing and generating safety-critical flight computer code". Semi-conductor engineers use it in their design process from "algorithm development and the creation of specifications, to system-level simulation, testing, and verification". As communications equipment manufacturers strive to improve "the transmission of voice, music, data, and video", they have found that the MATLAB family of tools "accelerate[s] tasks such as data analysis, algorithm development, large-scale system simulation, performance analysis, hardware and software verification, and automatic code generation for prototyping and deployment".

The MATLAB environment enables these applications through its large mathematical libraries and the ready mapping from traditional notation to implementation by its controlling language, also known as MATLAB. From now on when we refer to MATLAB, we are referring to the language unless otherwise stated. The main characteristics are that it is a dynamically typed imperative language which is normally interpreted. Variables do not need to be declared and can change type. Matrices and arrays are not of fixed size but are reshaped when assignments are made to subscripts outside the current bounds. Function calls are always *call-by-value*, (although a copy-on-write approach is taken internally) and it is not possible for data structures to share values. As such this means that there are none of the aliasing problems that exist in languages like C [52].

For the purposes of this project we intend only to look at a subset of MATLAB. This subset includes most MATLAB features and is large enough to handle non-trivial programs without modification. This subset has grown significantly since [20] as wider testing revealed further desirable features. In this chapter the structure of MATLAB will be discussed in order to give a better understanding of the later chapters. Sections 2.1 to 2.3 describe the features we include in the subset, while Section 2.4 outlines the main features that we have chosen to exclude. A mostly complete EBNF[1] will be given to show this structure. This description is not intended to be complete; in particular, the use of whitespace to delimit matrix columns and rows is not discussed. A fuller if slightly outdated discussion on parsing MATLAB can be found in [36].

2.1 M-files

MATLAB code is always stored in a file with a `.m` extension, called an m-file. An m-file is either a script or a function, depending on whether the file starts with the `function` keyword. Scripts are executed within the current scope whereas functions execute within their own scope. For simplicity we deal only with functions as their scoping rules are simpler.

Functions are declared using the `function` keyword. This must come at the start of the file (excluding whitespace and comments), although additional local function definitions can appear later in the same file.

Functions can return zero or more values and take zero or more parameters. Whatever values are stored in the output variables at the end of the execution of the function are returned. E.g.

```
function [a,b] = f(x,y)
```

It is also possible to declare functions that take a variable number of parameters by adding a final parameter called `varargin`. Functions can return any number of outputs by making the last output `varargout`. The EBNF of functions is given in Figure 2.1.

MATLAB statements are either separated by new lines, semi-colons or commas. Using a comma to delimit a statement is equivalent to using a new line and so from now on only new lines will be mentioned. All the legal MATLAB statements are shown in Figure 2.1 using EBNF.

[1]Extended Backus Naur Form

```
m_file = script_file
       | function_file ;

script_file = { statement } ;

function_file = function { function } ;

function = func_pream { global } block ;

func_pream = "function" [ ret_vars "=" ] identifier [ params ] ;

ret_vars = identifier
         | "[" { identifier "," } identifier "]" ;

params = ( "(" { identifier "," } identifier ")" ) ;

global = "global" identifier { identifier } ;

block = { statement ( "\n" | ";" | "," ) }
```

FIGURE 2.1: EBNF for MATLAB functions

2.2 Statements

In the following pages we will now describe the effect of each of the MATLAB statements given in Figure 2.1.

Expressions. MATLAB is in many ways like a calculator and can be used just to evaluate expressions. If an expression is terminated by a new line, the result is printed to the screen. If it is terminated by a semi-colon, the result is computed but not displayed. The result would normally be stored in a special variable called ans but we will require explicit assignments, as the use of ans is mostly employed only in the interactive environment, where MATLAB is being used as a calculator. The make-up of expressions will be discussed in more detail later in this section.

As some functions do not return anything, it is possible for an expression to have no value. Such a function cannot be used anywhere where a value is required, which means that it can only appear as an expression statement (but not part of a binary expression). When these functions are executed, no output value is displayed even if the semi-colon is omitted. Such functions would merit their own case distinct from expressions, except that determining that a function returns nothing is not trivial since the presence of return variables in a function declaration only means that it can return values if these variables are defined when the function exits. E.g.

```
1 + a * [2 3]'
disp('hello')
```

```
statement = expression
          | special_func
          | assignment
          | for_loop
          | while_loop
          | if_statement
          | switch
          | try_catch
          | control_flow_change ;

assignment = ( target "=" expression )
           | ( subscript "=" "[]" )
           | ( "[" { target "," } target "]" = func_call ) ;

special-func = identifier { string } ;

for_loop = ( "for" variable "=" expression block "end" ) ;

while_loop = ( "while" expression block "end" ) ;

if_statement = ( "if" expression block else_block ) ;

else_block = "end"
           | "else" block "end"
           | "elseif" expression block else_block ;

switch = "switch" expression
         { "case" choice block }
         [ otherwise block ]
         "end" ;

choice = expression
       | ( "{" { expression "," } expression "}" ) ;

try_catch = "try" block "catch" block "end" ;

control_flow_change = "return"
                    | "break"
                    | "continue" ;
```

FIGURE 2.2: EBNF for MATLAB statements

Assignments. There are three kinds of assignment, all indicated by the = operator. The most common is an assignment from an expression to a variable or to *within* a variable using indices. In addition multiple variables can be assigned if the right hand side is a function call. The final kind of assignment is the *delete assignment*, indicated by an assignment from the empty matrix [] to a subscript of a variable. This can be used to delete elements from arrays. If no semi-colon is present, then the values of all assigned variables are displayed. E.g.

```
a = 1
a(2) = 1
[a,b] = size(c)
a(5) = []
```

`if` statements. `if` statements check the value of an expression before executing a list of statements. There is also an optional `else` clause with `elseif` being "syntactic sugar" for a nested `if` statement inside an `else` block. The final list of commands is terminated by an `end` on its own. Usually the expression will be a scalar, in which case if it is non-zero the first set of commands will be executed. In the case of arrays, every value must be non-zero. E.g.

```
if a == 1      % every element of a must be 1
    n = 20
elseif a       % every element of a must be non-zero
    n = 30
else
    n = 0
end
```

Logical operators are evaluated differently inside `if` statements. Normally neither | nor & shortcuts (skips evaluation of the second operand when the evaluation of the first predetermines the final result), but it does in `if` statements. When no shortcut occurs, the semantics revert to the normal behaviour of the logical operators. Unfortunately this is implemented in a manner likely to cause confusion, with the possibility of incorrect results.

```
if [1 0] | [0 1]
    disp(1)
else
    disp(2)
end

if [1 0] | ([0 1] & [0 1])
    disp(1)
else
    disp(2)
end
```

The above examples were run in MATLAB 6.1 and 6.5, and both printed 1 and then 2 on the screen, when most likely the expected result would be 1 and 1 or 2 and 2. In the first example `[1 0]` is evaluated and found to be *false*, no shortcut occurs and so it is logically *OR*ed with `[0 1]` which produces the result `[1 1]`, which is *true*.

In the second example, `[1 0]` is evaluated and found to be *false*, then `[0 1]` is evaluated and also found to be *false* and so the result is logically *OR*ed with `[0 1]`, which gives `[0 1]`. Instead of then logically *OR*ing `[1 0]` with `[0 1]` and getting *true*, it actually logically *OR*s 0 with `[0 1]` and so evaluates to *false* causing 2 to be printed. This behaviour is ultimately the result of the ambiguous semantics obtained by combining shortcuts with an element-wise logical operator.

MATLAB 6.5 introduces `&&` and `||` as shortcut operators but does not remove the shortcut semantics from `&` and `|`. The new shortcut operators can only be used with scalar operands, and so it is not possible to write `a || b == c` when either `a`, `b` or `c` are non-scalar. A better implementation would implicitly convert operands to logical scalar values. We do not fully support MATLAB 6.5 (including these new operators), although we plan to shift to a newer version in future.

`for` loops. These execute a series of statements multiple times, while varying a loop variable on each iteration. The loop variable takes its values from a list, which in most cases is an arithmetic progression, although any list is allowed. The end of the loop is marked by the `end` keyword. In the example below, `1:10` expands to `[1 2 3 4 5 6 7 8 9 10]`, although the interpreter and compiler can frequently optimise loops like this. E.g.

```
for a = 1:10
    b = b + c(a)
end
```

`while` loops. These statements repeatedly evaluate the value of an expression, checking its value for zero (in the same way as with `if` statements) before executing a series of statements. The end of the loop is marked by the `end` keyword. E.g.

```
while a > 0
    b = b + c(a)
    a = a - 1
end
```

As with `if` statements, logical operators in `while` loops have shortcut semantics.

`switch` statements. These evaluate an expression which must be either a scalar or a string and check the value against a series of `case` clauses, executing the appropriate code if it matches before control flows to the end of the switch block. There can also be an `otherwise` clause which is executed if no other clauses match. The final list of commands is terminated by the `end` keyword. E.g.

```
switch a
    case 1
        n = n + 1
    case {2, 3, 4}
        n = 0
    otherwise
        n = 1
end
```

`return`, `break` and `continue` statements. Normally functions finish executing when control passes off the textual end of the function. However `return` can be used to explicitly end the function at any point, with the current values of the return variables immediately returned.

`break` causes the execution of the innermost loop to terminate and control to flow to the end of it. If a break appears outside of a loop, it terminates the function in the same way as `return`. `continue` causes the execution of the current iteration of the innermost loop to terminate and control to be passed back to the beginning of the loop; it can only appear inside loops.

2.3 Expressions

MATLAB allows the creation of complex expressions in a very intuitive way using the syntax described in Figure 2.3. The basic construct is the matrix, of which vectors and scalars are special cases. Matrices are created by enclosing the elements in square brackets (`[...]`). Apart from standard matrices which are two dimensional, MATLAB also has n-dimensional arrays. No syntax exists to create n-dimensional arrays directly, but they can be created using built-in functions and through indexed assignments. Arrays are laid out in memory in rows first, then columns and then any further dimensions. Arrays have a class associated with them as well as some type traits, which are somewhat independant of class. Type traits indicate whether arrays are real, complex or logical. The combination of traits and class makes up the *intrinsic type* of an array. Logical arrays are returned from boolean operators and built-in functions like `isreal` and `isinf`. While generally they will have the values 0 or 1, they can have any real value. Strings are arrays of class 'char', which contain only integer values (which use 2 bytes of storage). Arrays always have the same number of elements in each row, column or further dimension. The intrinsic type of an array applies to every element within it.

```
>> a = [1 2 3; 4 5 6; 7 8 9]
a =
        1     2     3
        4     5     6
        7     8     9

>> b = a > 4
```

```
expression = ( expression bin_op expression )
           | ( un_prefix_op expression )
           | ( expression un_postfix_op )
           | ( "(" expression ")" )
           | ( expression ":" expression )
           | ( expression ":" expression ":" expression )
           | ( "[" { row ";" } row "]" )
           | ( "'" string "'" )
           | ( "@" identifier )
           | target
           | number
           | complex_number
           | matrix
           | func_call
           | ":"
           | "end" ;

target     = indexable
           | ( indexable "(" { expression "," } expression ")" ) ;

indexable  = variable
           | ( indexable "{" { expression "," } expression "}" )
           | ( indexable "." identifier ) ;

variable   = identifier ;

func_call  = identifier
           | ( identifier "(" { expression "," } expression ")" ) ;

bin_op     = "+" | "-" | "*" | "/" | "\" | "^" | "|" | "&" |
             "<" | ">" | ".*" | "./" | ".\" | ".^" | "==" |
             "~=" | ">=" | "<=" ;

un_prefix_op = "~" | "-" | "+" ;

un_postfix_op = "'" | ".'" ;

general_number = number [ "e" [ "+" | "-" ] digit* ] ;

complex_number = general_number "i" ;

number = ( [ digits ] "." digits ) | ( digits [ "." ] ) ;

digits = digit { digit } ;

digit = "0..9" ;

row = expression | ( { expression "," } expression ) ;
```

FIGURE 2.3: EBNF for MATLAB expressions

```
b =
     0     0     0
     0     1     1
     1     1     1

>> c = 'a string'
c =
a string

>> d = a + b * j
d =
   1.0000              2.0000              3.0000
   4.0000              5.0000 + 1.0000i    6.0000 + 1.0000i
   7.0000 + 1.0000i    8.0000 + 1.0000i    9.0000 + 1.0000i
```

Nearly all binary operators in MATLAB are *array* operators. This means they are *element-wise* and require that both operands have the same *shape*. The shape of an array is described by the sizes of its dimensions. Array operators perform a computation for each pair of elements in the two operands with equivalent positions and return an array with an identical shape. For instance, using the + operator on the two vectors `[1 3]` and `[4.5 0]` gives us `[5.5 3]`. If one of the operands is a scalar (and therefore has 2 dimensions, both of size 1), then the result will have the same shape as the other operand and the result will be the result of applying the operator to the scalar and each of the elements in the non-scalar. If both are scalars, the result is also a scalar.

The exceptions to this are matrix operators like *, \, / and ^. Given one or more scalar operands, these behave exactly like array operators. To achieve the same for non-scalar operands, the array forms must be used: .*, .\, ./ and .^. Each of the matrix forms has different operand requirements although all require operands be 2-dimensional:

- * requires the number of columns in the first operand be the same as the number of rows in the second.
- \ requires the second operand to have as many rows as the first operand. The result will have as many rows as the first has columns and as many columns as the second has columns.
- / requires the second operand to have as many columns as the first operand. The result will have as many rows as the first has rows and as many columns as the second has rows.
- ^ requires that one of the operands be a scalar and the other a square matrix. The result will have the same shape as the matrix operand.

One consequence of allowing * to be used both as an array and a matrix operator in different circumstances is that it is no longer truly associative. Array multiplication and *true* matrix multiplication are both associative but the combination that MATLAB presents in the * operator is not. For instance:

```
>> a = [1 2; 4 5]; b = [1 2 3]; c = [3; 2; 1];
>> a * (b * c)
ans =
    10    20
    40    50
>> a * b * c
??? Error using ==> *
Inner matrix dimensions must agree.
```

This has consequences for bracketing when printing MATLAB code as we cannot remove what would appear to be redundant brackets without first checking whether it is safe to do so. In practice, we do not perform this check and so never rebracket, which also means that our partial evaluator will not change the order of execution, which in some cases can lead to a change in complexity (see Section 9.3).

Ordinary arrays and matrices can only store values of the same type and each of these values must be a scalar. While *cell arrays* must be regularly shaped, each element can have any type including cell arrays themselves. Cell arrays are created by enclosing the elements in braces ({...}).

```
>> a = {1 2 3; 4 5 6; 7 8 9}
a =
    [1]     [2]     [3]
    [4]     [5]     [6]
    [7]     [8]     [9]

>> b = {a, 'hi'; 5 + j, 10}
b =
           {3x3 cell}     'hi'
    [5.0000+ 1.0000i]     [10]
```

Structures consist of a number of fields each containing a value. The fields are strings, while the value can be any MATLAB array including cell arrays and other structures. Fields can be added to a structure at any time and there is no concept of named structures as in C. Structures can either be created incrementally by assigning to each of the fields in turn (using the '.' operator) or by using the `struct` built-in. Fields can be accessed using the '.' operator or the `getfield` function and can be deleted using the function `rmfield`.

```
>> a = [];
>> a.field1 = 5
a =
    field1: 5
>> a.anotherfield = {'hello', [1, 2]}
a =
          field1: 5
    anotherfield: {'hello'  [1 2]}
>> b = struct('field1', 5, 'anotherfield', {{'hello' [1, 2]}})
b =
          field1: 5
    anotherfield: {'hello'  [1 2]}
```

Indexing into matrices is done using round brackets, as in a(3). MATLAB allows more than just scalars as indices. An appropriately sized matrix is also a valid index. In particular, ranges can be used to extract parts of matrices. Indices start at 1 and end can be used to get the last element. Finally, if an index is logical it predicates which part of the matrix should be extracted. E.g.

```
>> a = [1, 2, 3; 4, 5, 6; 7, 8, 9];
ans =
     1     2     3
        4     5     6
        7     8     9

>> a(1,:)
ans =
     1     2     3

>> a(2,2:end)
ans =
     5     6

>> a(a > 4)
ans =
     5     6     7     8     9
```

It is possible to index into a matrix using more dimensions than the matrix has, as long as the extra indices are all equal to 1. If fewer dimensions are used then the dimensions that are not explicitly specified are flattened, so that the final index can be used to access all of them. For example, matrices can be indexed linearly (remembering of course that the values are stored in row-column order).

```
>> a = [1, 2; 3, 4];
>> a(2, 2, 1, 1, 1)
ans =
     4

>> a(:)
ans =
     1
     3
     2
     4
```

Indexing into cell arrays using round brackets produces a cell array as the result and so it is not suitable for extracting the elements themselves. Instead braces need to be used as in a{1}. As this extracts the elements themselves and not a subset of the cell array, the behaviour is very different when the indices are not scalars. In this case such an expression would evaluate to more than one value and so it only makes sense in a context that allows comma separated expressions (like a function call invocation or in the construction of a matrix). This is of special importance for functions which have the varargin parameter. This parameter will be a cell array containing all the extra parameters passed into the function. In order to pass all of these extra parameters onto a second function, they must be expanded in the function invocation.

```
function y = f(varargin)
y = g(varargin{:}) + 10;
```

Cell array expansion using braces, can be used in any place where comma-separated lists are accepted. This includes function call parameter lists, index lists and matrices.

Another important type is the *function handle.* These are like pointers to functions in C. They are created by prefixing a function name with `@` and also by using certain functions. The built-in `feval` is used to invoke the function represented by the function handle. These are used by many general numerical solvers like `quad` which evaluates integrals using adaptive Simpson quadrature and `ode45` which solves non-stiff differential equations using a medium order method.

The description of matrices given in Figure 2.3 is incomplete as the use of white space to denote column and row separators is not given, as it is just "syntactic sugar" for the form given. It is also an ambiguous grammar as there is nothing to distinguish a function call from an array access. This is because it is impossible to make this distinction using a context-free grammar and the method for performing this *disambiguation* will be described in Section 5.4.

2.4 Unhandled Constructs

In order to simplify, there are several MATLAB features that we do not consider:

`try-catch` statements. This construct is used to handle exceptions and is mostly of interest when dealing with I/O. While we do not attempt to do any partial evaluation of this construct, our parser does recognise it. This allows partial evaluation of functions to go ahead when functions use the construct but the statement appears on an execution path not taken by the partial evaluator. Partial evaluation of this construct is especially difficult as control could theoretically flow from any point in the `try` block to the `catch` block making data flow analysis difficult.

Struct arrays. By creating a cell array of structs, it is not possible to constrain the fields that each struct must have. Indeed it is not even possible to assure that each element is even a struct. An alternative is the struct array, which is a generalisation of the struct as described earlier. In addition it can stores more than one element. We ignore it as it is rarely used and complicates parsing. As a result we require that the `size` function always returns `[1 1]` for a struct.

Classes. Classes allow the definition of user defined objects which can have methods executed on them polymorphically. While we support some features of classes we do not support the creation of user defined classes or any class which is not directly built into MATLAB.

When MATLAB is to execute a function call, the function that MATLAB invokes is dependent on the class of the first parameter in the invocation. If the parameters is of class A, MATLAB searches for directories called `@A` containing a matching function. If no such function is found then the default function is called instead.

It would not be difficult to handle polymorphic function calls when the class can be statically determined, but when the class is unknown any number of functions could be invoked, thus complicating any analysis. One could abstractly interpret or partially evaluate each possible function, but this could be expensive. Currently we completely ignore the existence of functions contained in class directories, which could lead to incorrect code generation even for built-in classes.

Global and persistent variables. Global variables are declared on lines prefixed with `global`. Globals can be declared at any point of a program, including after a variable has been used as a local variable. To simplify matters, we required that global variable declarations come immediately after the function declaration and that they cannot appear anywhere else. This explicitly makes illegal declaring a variable as a global after it has already been used. We also require that global variables do not shadow function parameters, which means a function cannot declare that it takes a parameter and also use a global variable with the same name.

Persistent variables are like static local variables in C. They retain their values between function invocations, but are not available outside of the function in which they were declared. They are declared just like global variables but using the `persistent` keyword. For now these are treated as if they were ordinary global variables.

It is necessary for a partial evaluator to parse global declarations since they can occur in the main MATLAB libraries, but beyond this we largely ignore them. We do not store any information about them since they could be overwritten by other functions. It is rare that global variables are used in MATLAB programs in ways that partial evaluation could help. Any code that does can often be written to avoid using globals.

Recursive functions. Recursive functions with dynamic control cannot be handled as cycles are not detected by the polyvariant specialisation. This means that recursive functions which would normally terminate can be infinitely specialised leading to non-termination of the partial evaluator. This could be handled by marking the function call signature as currently being specialised and immediately return control to the caller.

This would not help with recursive functions which statically altered their parameters, but use of the widen annotation described in Section 6.1.7 could help.

If the recursion is completely static, as with a recursive power function where the exponent is static, then the partial evaluator will terminate but will produce a new function for each recursive call, which may or may not be desired. In combination with inlining it could reduce the simple recursive power function into a single line function.

Special functions. Certain functions can have unpredictable effects on the execution of MATLAB programs. These include functions like clear and assignin which manipulate variables. The clear function is mostly used interactively and it is uncommon to see it used in functions.[2] The function deletes a variable and makes it undefined. If it is used with static input, it might be possible to correctly handle it. However if the variable to be cleared cannot be determined statically at partial evaluation time, to ensure correctness we would have to throw away everything we know about all variables from that point on as any one of them might have been affected.

Multiple outputs through cell array subscripts. As previously stated, functions can return a variable number of outputs. Through the use of cell array subscripts it is possible to make a program, in which the number of outputs to a function cannot be statically determined.

```
[a{b}] = f(x)
```

If the size of b is not known, then the value of nargout will be unknown in f. Also if the size of b is known then the only way to split the assignments if the result of f(x) is static, is as follows (where b has 3 elements):

```
a{b(1)} = ...
a{b(2)} = ...
a{b(3)} = ...
```

Ideally, this transformation should occur before the function call is partially evaluated as follows:

```
[a{b(1)} a{b(2)} a{b(3)}] = f(x)
```

Unfortunately this transformation produces a reduction in performance, and even if some of the outputs are static, it is unlikely that speed-ups will be possible as assignments will have to be inserted which are unlikely to be removed by post-processing as the dynamic outputs will mark the whole of a as dynamic.

[2] clear is often seen in scripts which operate in the caller scope and need to clear variables to avoid conflicts.

Ultimately our partial evaluator assumes that each entry in the output list requires one and only one output and so in the original example, `b` will be assumed to be scalar and `nargout` for `f` will be 1.

In addition, we also require that the outputs do not contain any conflicts. I.e. two outputs cannot be to the same variable and a variable cannot be written to if it is used as an index in another output.

By limiting the set of MATLAB features that we can handle we limit the number of MATLAB programs with which we can initially work. However we have to make a pragmatic decision to ignore certain features that are not critical to testing our hypothesis that partial evaluation is a viable technique for the optimisation of MATLAB programs. In the future, it is hoped that these features could also be added to our tool.

2.5 Optimising MATLAB — Current Technologies Review

The Mathworks provides a compiler called MCC [74], which translates MATLAB into C. This C code is then compiled by the native compiler to produce an executable which can be executed without a full MATLAB installation. The C code produced is portable but, since it requires the MATLAB runtime libraries, is tied to the platforms that MATLAB supports. The code produced consists mostly of function calls and very little attempt is made to use native C types. This is due to the dynamic typing which means that a variable could contain anything from a matrix to a function handle. Because the MATLAB base libraries call optimised routines in libraries such as ATLAS [80], LAPACK [44] and FFTW [23], these function calls are executed very quickly but have to inspect the parameters to determine the type of function to call.

Another compiler is FALCON, which produces Fortran 90 code [57]. This uses extensive type inferencing at compile time to produce code which does very little type checking. Using user defined types (encompassed by existing MATLAB types), [26] describes how to make FALCON take advantage of structural information such as diagonal or upper triangular matrices, to improve results. Initial results showed FALCON outperforming MCC, but there have been improvements to the Mathworks compiler recently and so it is not clear which would perform better now as the FALCON source code was not made available and no recent comparisons have been reported by its authors.

Following on from FALCON, Almási has developed MaJIC, a MATLAB *Just-In-Time* compiler described as "an interactive front-end that looks like MATLAB and compiles/optimises code behind the scenes in real time, employing a combination of just-in-time and speculative ahead-of-time compilation". [3] Because MaJIC compiles code in an interpreted environment, it has information about the parameters used to call functions and attempts to produce more appropriate code. When compiling *Just-In-Time* it eschews most optimisations in favour of fast compilation times and so cannot easily perform the types of aggressive optimisation seen in offline compilers and partial evaluators. MATLAB 6.5 also introduced *Just-In-Time* compilation as part of its normal operation and this often gives better results and is reported to be much improved in the recently released MATLAB 7.

Other approaches to speeding up MATLAB execution have involved parallelisation. Mostly this involves adding parallel extensions to the language, like MultiMATLAB [50] or the DP toolbox [53]. However Otter is an attempt to translate MATLAB scripts into C programs targeting parallel computers supporting ScaLAPACK [54]. This approach produces varying results depending on the sizes of matrices and the complexity of the operations performed on them.

In [48] and [49], the authors discuss source-level optimisations that would be appropriate for MATLAB. These include vectorisation of loop operations to take advantage of the more efficient MATLAB libraries, preallocation of arrays to prevent repeated resizing and expression optimisation through reordering. Many of these issues would already be apparent to MATLAB programmers, who would seek to avoid these pitfalls. However other automatic source level transformations, including partial evaluation, may yield opportunities to perform these optimisations.

In [35], the authors use a static shape analysis to try and optimise the memory used by MATLAB programs. This results in quite significant speed increases as cache utilisation is improved. The shape analysis is described in more detail in [32]. This shape analysis is capable of determining that the shapes of two arrays must be the same, even when the shape itself is unknown, leading to fewer run-time checks.

Type inferencing [1, 30] has been examined for other dynamically typed languages, like Scheme, since it is enables early detection of errors due to type mismatches as well as enabling many optimisations, such as replacing general arithmetic operators with integer specific operators.

2.6 Summary

In this chapter we have seen the structure of the MATLAB language, including some of its idiosyncrasies. The language was clearly designed to try and minimise the distance from the mathematical description of a problem to its solution. In doing so, many of the methods used to implement languages in the past have been ignored, since they serve only to distract from the immediate problem. In ignoring these methods, the developers of MATLAB have created a different set of problems, which must be solved to achieve satisfactory performance.

MATLAB was developed outside academic computer science and is a niche-language since it is not readily applicable for general computation. It is however an important niche growing in size as computers become fast enough to allow realistic models to be built before any concrete engineering is even contemplated. To allow it to grow even further, more methods from the academic world need to be introduced. Approaches like compilation have served to give better performance, and indeed such work is ongoing.

The common theme to MATLAB usage, is the flexibility and the easing of traditionally complex tasks. Engineers can focus on solving the problem at hand rather than the reinvention of existing technology for each new application. This culture of reuse and componentisation can gain significantly from processes which seek to solve its associated problems.

We will, in the next chapter, introduce one such process which could improve the productivity achievable with MATLAB, both through execution time reductions and reduced development effort. Partial evaluation has more often been applied to traditionally academic languages, like Scheme and Prolog, and very mainstream languages like C and Java. In this work, we seek to expand its domain to MATLAB.

Chapter 3

Partial Evaluation

Partial evaluation is a technique to partially execute a program, when only some of its input data are available. [51]

The above statement means to take a program, for which some of the inputs are known prior to full execution, and execute as much of the program as possible. In cases where programs are executed many times with only a few parameters changing, dramatic savings can be made as many calculations can be performed during the partial evaluation and thus only once. Partial evaluators can also perform aggressive optimisations like loop unrolling and inlining which, while also possible in traditional compilers, are less easy to control or see the effects of when the transformation is not source to source. A program generated by partial evaluation is called a *residual* program.

3.1 Review

The most complete description of partial evaluation can be found in [37]. A similar technique known as *supercompilation* is described in [77].

Traditionally partial evaluation has been mostly applied to declarative languages, like Scheme [37] or Prolog [45]. But there are also partial evaluators for C [5], Java [64] and Fortran [41]. Frequently the work on partial evaluation of languages like Scheme and Prolog has focussed on efficient self-application [15, 38]. While it is possible to compile programs, by specialising interpreters with respect to a static source program, by specialising the partial evaluator with respect to the interpreter, it is possible to produce a compiler [25]. This *compiler* can be much faster than specialising the interpreter and so self-applicable partial evaluators are often desirable.

There are two main forms of partial evaluation, online and offline. In *offline* partial evaluation, a *Binding-Time Analysis* (BTA) is performed first which given a source program and a division for the initial input, determines which parts of the program are static and which parts are dynamic. This data is then embedded in the source file in the form of annotations which are used by the partial evaluator to produce the final result. This final process is sometimes called *reduction.*

In *online* partial evaluation, there is no binding-time analysis step but instead decisions about static vs. dynamic expressions are made as late as possible, and it is thus, in principle, more precise. In general, offline partial evaluators can be made more efficient and predictable. On the other hand, online partial evaluators are typically slower but can detect and thus evaluate more static expressions.

```
%# x size [STATIC DYNAMIC]
%# c STATIC
if size(x, c) == 1
  ....
else
  ....
end
```

In the example above, x is a matrix for which the number of rows is declared static but the number of columns is dynamic. This code checks to see if the size of the dimension indicated by c is equal to 1. Even though c is static, offline partial evaluation will not easily allow us to remove the if-statement if c turns out to be 1 (or greater than 2). With online partial evaluation, the expression is only examined when we know the value of c and so the if-statement can be removed.

The advantage of offline partial evaluators is that they are simpler to build and as the binding-time analysis is separate from the specialisation phase, it can be done just once while the specialiser is then called multiple times for different parameters. The binding-time analysis can also be manually adjusted in cases where the automatic analysis is imperfect. Offline partial evaluators are currently the only way to effectively handle self-application.

One approach to offline partial evaluation is not to generate a full program directly from the annotated source (after binding-time analysis), but to generate a program which takes as input the static inputs. This program then outputs the final program. This program is called a *generating extension* and the approach is known as the *cogen* approach.

Most *offline* partial evaluators use a *monovariant* BTA. This means that each expression at each program point has at most one binding type. The BTA will normally determine this to be the binding that works given the binding division of the input variables. This means that a function called with dynamic parameters at one program point and static parameters at another, will only be residualised based on the most conservative set of bindings, i.e. the most dynamic ones. A *polyvariant* BTA could produce two different sets of bindings for the function and so could produce more suited residual code. *Polyvariance* can be achieved by function cloning, where every time a different binding is required, the called function is cloned and the BTA is performed on the new function [76]. Polyvariant binding time analysis need not work at the function level, but instead program points can have multiple binding types associated with them, where the most appropriate one is selected at specialisation time. This allows the code fragment given earlier to be specialised by an offline partial evaluator in the same way as an online one would.

Monovariant specialisation produces, for every function in the source program, at most one function in the residual program. *Polyvariant* specialisation produces more than one function if required. A maximally *polyvariant* specialiser would produce a new function for every residualised function call, although more generally duplicate function signatures are combined so that functions can be shared. *Polyvariant* specialisers are more prone to termination issues as they can potentially produce infinitely many functions in cases where a static parameter grows under dynamic control in a recursive function. The process of detecting and eliminating this problem is called *generalisation.*

There has been some work on hybrid *online/offline* partial evaluators, including [69], which is essentially an *online* specialiser which produces generating extensions, where some offline decisions have already been made in order to speed up the final specialisation. Sperber [66] produced an "online" specialiser which could be realistically self-applied, but this was effectively an offline partial evaluator which could defer some binding-time decisions to specialisation time. Christensen and Glück [12] have also demonstrated that *offline* partial evaluation can be as accurate as *online* using a maximally polyvariant BTA, although as effective generalisation is not possible. In [28], Glück showed that offline partial evaluators in conjunction with binding-time improvements can always achieve as efficient residual programs as online partial evaluators.

For the online partial evaluator FUSE, the authors use fix-point analysis to find the types of return values from functions [61]. This is required because the value of a recursive function is itself dependent on its own type, making repeated iteration necessary. This approach can be generalised to the non-recursive structured loops present in imperative languages. They also introduced the idea of producing a graph of the suspended computation rather than directly producing code [78]. This makes detecting and removing code duplication, due to expanding variables to expressions, much simpler. While most specialisers will recognise when a function is invoked with a signature that has already been specialised, and reuse the earlier specialisation, it is more difficult to recognise when signatures are only slightly different and will lead to the same specialisation. FUSE tackles this [59] by storing information about what properties of the parameters led the function to be specialised in the way that it was. This allows a looser signature to be constructed which would result in the same specialisation. If a signature is then encountered which falls inclusively between the two signatures, the specialisation can be reused safely with no chance of missing optimisation opportunities.

Instead of writing partial evaluators directly, it is possible to produce partial evaluators using an interpreter and an existing self-applicable partial evaluator written in the same language [67]. This requires a specially written interpreter, which divides inputs to the source program into dynamic and static parts. The partial evaluator is then partially evaluated with respect to the interpreter. The result will be a program, which given a source program and the static input, will produce a residual program written in the source language of the interpreter. This program will be a partial evaluator, although its output will not be the same language as its input. This does, however, demonstrate an automatic way of producing a partial evaluator.

Alternatively, source code can be compiled to a language, for which a partial evaluator already exists. For instance, the main MATLAB compiler produces C code, which could then be partially evaluated using cmix [5]. Brief experimentation showed that cmix is not capable of parsing the raw output of the MATLAB compiler, although this could no doubt be worked around. Even then, such partial evaluation would be unlikely to ever take advantage of type information that could be statically inferred for dynamic arrays as the array structure is opaque to everything except the run-time libraries.

Tempo [13] is a partial evaluator for C, which can generate residual programs which perform run-time specialisation, using optimised binary code templates, as well as performing standard partial evaluation. This system has been deployed in various fields, with notable work on operating systems software including Sun RPC and the BSD packet filter.

Psyco [56] is an implementation of the Python language, which performs just-in-time specialisation to remove the overheads introduced by the highly dynamic nature of the Python type system.

A particular problem in languages with complex types, is the concept of *lifting*. This occurs when a static value is required in a dynamic context. For simple types such as integers this is easily achieved by inserting a textual representation into the residual program. For more complex types, this may not be possible. Pointers, for instance, cannot be lifted as their value at the time of partial evaluation is meaningless when the residual program runs. Structured types can often incur a cost to build, or may not even be possible to construct within a single line.

In many cases, the requirement to lift a variable with a complex type, leads to the variable being made dynamic in all cases, thus reducing the number of possible static computations. Hornof et al [31] use an analysis of the uses of a variable to avoid making variables completely dynamic in the Tempo specialiser for C. This problem is also addressed by Asai [6] for functional languages.

Aside from program specialisation, another technique first introduced by [8], which uses early computation to reduce final execution time, is *Data Specialisation*. This technique precomputes static computations and stores them in a cache. A residual program is then created which loads the static data from the cache and uses it to perform the work of the original program. The advantage of this method over program specialisation, is that the residual program does not grow with the size of the input data. The disadvantage is that it is only effective if the cached computations are sufficiently expensive to justify the time and space for storing and accessing them. Knoblock and Ruf [42], demonstrate an automatic system to perform data specialisation for C, especially with regard to improving shader performance. The route taken by Chirokoff and Consel [11] seems promising, in that they combined program and data specialisation in the Tempo partial evaluator. By using both approaches code size does not increase as quickly as with normal program specialisation, and yet performance does not differ by much. This allows much larger data sets to specialised than with partial evaluation alone, where the residual program can become so large that performance is affected adversely.

Reps and Turnidge [55] use a technique know as *program slicing* to specialise programs. This technique extracts a *slice* of the program that produces a specified result. For instance a program that produces two outputs can be specialised to produce only one, in which case computations which do not contribute to this single output can be removed from the residual program. This technique holds an advantage over partial evaluation, in that the slice is not determined by the program parameters and so can be based on a parameterisation not originally envisioned by the program author.

Berlin and Surati [9, 70] used partial evaluation to expose parallelism in large scale numerical applications. This is possible because partial evaluation can remove many conditional checks based on structure sizes, resulting in a program which contains largely only numerical calculations.

Specialisation of Fortran [7] has yielded impressive results for numerical applications, showing that the Fast Fourier Transformation can see a between 3 and 4-fold speed-up by specialising with respect to the number of data points to be returned. Cubic spline interpolation specialised with respect to range conditions, the number of values and the distance between consecutive values is 4 times faster in general and 6 times faster for periodical range conditions.

Continuing in the vein of partial evaluation for high performance computing, there have also been attempts at improving partial evaluation using the high performance computing technique of distributed computing. Sperber [68] describes a client-server system for processing function specialisation requests and gains speed-ups of up to 3 times using up to 6 processors.

3.2 Summary

This chapter has given an overview of the previous work on partial evaluation and related techniques. We have seen that offline partial evaluation can offer efficient specialisation, but requires conservative approximations which can sacrifice some performance in the residual program. On the other hand, online specialisation can produce faster residual programs, but is less predictable and often slower. Offline partial evaluation can be carefully guided to produce the desired result, but this requires expert knowledge from the user. Online partial evaluation, on the contrary, is largely an automatic process possibly making it accessible to more users.

Several functions exist in MATLAB which discover information about the characteristics of MATLAB values. While the values themselves will often change rapidly, the intrinsic type and shape of arrays will often be constant. Successful type inferences can often allow us to replace calls to type-query functions with constants, frequently enabling the unrolling of loops dependent on array shapes and the removal of conditionals that verify parameter types.

To this end, a formalisation of the MATLAB type system is given in the next chapter, which captures information about both the intrinsic type of arrays, which is comprised of its class and certain traits which are common to all classes, and the shape of arrays, made up from a number of dimension sizes.

Chapter 4

Abstract Domains

In this chapter, we describe lattices and then use them to formalise the abstract domains our partial evaluator uses to capture information about arrays, for when concrete values are not available. We also give equations for deriving types of arrays resulting from using operators, calling functions and indexing into arrays. The notation and methodology is mostly based on [3] and [14].

4.1 Partially Ordered Sets, Lattices and Fixpoints

This chapter uses partially ordered sets (*posets*) and lattices extensively as the basis for our symbolic execution system.

Definition 4.1. A *poset*, $(S, \sqsubseteq)$, is a set for which a partial order ($\sqsubseteq$) is defined. This ordering is an antisymmetric relation, such that $\forall x, y \in S : x \sqsubseteq y \wedge y \sqsubseteq x \Rightarrow x = y$. If an ordering exists between all elements ($\forall x, y \in S : x \sqsubseteq y \vee y \sqsubseteq x$) then it is a *total* ordering.

Definition 4.2. An *upper bound* of $X \subseteq S$ is an element $u \in S$, such that $\forall x \in X, x \sqsubseteq u$. The *least upper bound* is an upper bound, which is ordered lower than all of the other upper bounds. Put formally, the least upper bound of $X \subseteq S$ is $l \in S$, where $\forall x \in X, x \sqsubseteq l$ and $\forall u \in S$, such that $\forall x \in X, x \sqsubseteq u \implies l \sqsubseteq u$. If the least upper bound exists, it is unique and is written $\sqcup X$. *Lower bounds* and *greatest lower bounds* are defined in the same way using the reverse ordering ($\sqsupseteq$). The greatest lower bound is written $\sqcap X$.

Definition 4.3. A *lattice* $(L, \sqsubseteq, \bot, \top, \sqcup, \sqcap)$ is a poset $(L, \sqsubseteq)$, for which $\forall X \subseteq L, \exists \sqcap X \wedge \exists \sqcup X$. The *supremum* of L is $\top = \sqcup L$ and the *infimum* is $\bot = \sqcap L$. The least upper bound of the set, $\{x, y | \forall x, y \in L\}$ is $x \sqcup y$ and the greatest lower bound is $x \sqcap y$.

In terms of the approximations we use in our symbolic execution, if $\sqsubseteq$ is an ordering based on accuracy, i.e. $x \sqsubseteq y$ means that x is more accurate than y, then $x \sqcup y$ is an approximation that best *fits* both x and y. When we say the approximation best *fits* x and y, we mean that it is the least conservative approximation possible, which is valid for both x and y. If x and y are sufficiently different then the result will be $\top$, which encompasses all possible values. In addition $x \sqcap y$ is the least precise approximation which is included in both x and y. If no approximation fits at all then the result will be $\perp$. Since we will use more than one lattice, each with their own infimums, supremums and partial orders, we will add subscripts to these to distinguish them.

Definition 4.4. A function $f : X \to X$ is monotone (with respect to the ordering), if $x \sqsubseteq y \implies f(x) \sqsubseteq f(y)$. This means that application of f maintains the ordering.

Definition 4.5. A fixpoint of a function $f : X \to X$ is a $x \in X$ such that $f(x) = x$. It follows that if X is finite and f is monotone, repeated application of f will always result in a fixpoint and therefore iteration will never be infinite. We will later use this to show under what circumstances our analysis terminates. For a more detailed explanation of lattices and their use, see [14].

4.2 Abstract Type System

MATLAB has a complicated type system which has evolved over time from just representing two dimensional matrices to N-dimensional arrays, cell arrays, structures, strings and function handles. In essence expressions evaluate to *arrays*. Each array has a *class*, which can be retrieved using the `class` built-in function. In addition, each array has two properties which are returned by the boolean functions, `isreal` and `islogical`. These functions are overloaded for all classes and for instance can be used for both full matrices and sparse matrices. A more thorough description is given in the previous chapter.

The main aim in producing a type system is to capture information about dynamic expressions. As in [1], we require that types be comparable. This means that we can detect the equivalence of two types and also whether one type is a subtype of another. As such, one can imagine a type system as defining the sets of values that can have each type. If two sets are equivalent, they describe the same type. The operators $\sqcup$ and $\sqcap$ would then be analogous to the set operators $\cup$ and $\cap$, while the relation $\sqsubseteq$ is analogous to $\subseteq$. The *supremum*, $\top$, is like the full set of values, while the *infimum* is like the empty set, $\varnothing$.

4.2.1 Class Information

Definition 4.6. MATLAB classes (shown in Table 4.1), are modelled using the lattice $(\mathbf{K}, \sqsubseteq_k, \perp_k, \top_k, \sqcup_k, \sqcap_k)$, where $\mathbf{K}$ is defined in (4.1) and $\forall k_1, k_2 \in \mathbf{K} - \{\top_k, \perp_k\}$, $k_1 \sqsubseteq_k$

Class	Description
double	Real and complex numbers (scalars, vectors, matrices and N-dimensional arrays
char	Strings (single line, multi-line and N-dimensional)
cell	Cell arrays that can store any other type of array
struct	Stores elements in named fields
sparse	Sparse version of double
single	Like double but single precision
int8, int16, int32	Signed integers of various sizes
uint8, uint16, uint32	Unsigned integers of various sizes
function handle	Similar to pointers to functions in C

TABLE 4.1: MATLAB classes

$k_2 \iff k_1 = k_2$ and $\forall k \in \mathbf{K},\ k \sqsubseteq_k \bot_k \wedge k \sqsupseteq_k \top_k$.

$$\begin{aligned} \mathbf{K} \;&=\; \{\bot_k, \mathit{double}, \mathit{char}, \mathit{cell}, \mathit{struct}, \mathit{sparse}, \mathit{single}, \mathit{int8}, \\ &\quad \mathit{int16}, \mathit{int32}, \mathit{uint8}, \mathit{uint16}, \mathit{uint32}, \mathit{function}, \top_k\} \end{aligned} \tag{4.1}$$

The concrete elements of $\mathbf{K}$, (i.e. $\mathbf{K} - \{\top_k, \bot_k\}$) are equivalent to the values returned by the MATLAB function `class`.

4.2.2 Type Trait Information

In addition to classes, there are *type traits*. The two traits we deal with are *real* and *logical*. An array can be either real, complex or logical; although logical arrays are also real. In MATLAB, the built-in functions, `isreal` and `islogical`, can determine whether an array has a specific trait. An array of type real is made up from double precision floats. Complex arrays use twice the memory of real ones, in order to store the real and imaginary components. Logical arrays are identical to real arrays except for a flag indicating that they are logical. The only functional difference occurs when logical arrays are used as indices. Used as an index, a logical array acts like a filter, where non-zero elements indicate that a value should be retained and zero elements indicate it should be skipped.

Many mathematical operators and functions return complex numbers in MATLAB. This happens automatically when the parameters dictate the result is not real, although it is also possible to create a complex array even when the imaginary component is 0 using the `complex` built-in function. In many cases, there is no way of predicting whether the result of a computation will be complex just based on the presence or absence of the complex trait as for instance the addition and multiplication of two complex conjugates produces a real result. Examples of arrays with the various traits can be seen in Figure 4.1.

```
>> [isreal([-1 2 pi]), islogical([-1 2 pi])]
ans =
     1     0

>> [isreal(3 == 2), islogical(3 == 2)]
ans =
     1     1

>> [isreal(23 + 3i), islogical(23 + 3i)]
ans =
     0     0
```

FIGURE 4.1: Examining MATLAB traits (Non-zero values indicate true)

Definition 4.7. $\mathbf{B} = \{\top_b, \mathit{true}, \mathit{false}, \bot_b\}$ is the extended boolean type. The lattice $(\mathbf{B}, \sqsubseteq_b, \top_b, \bot_b, \sqcup_b, \sqcap_b)$, is such that $\bot_b \sqsubseteq_b \mathit{false} \sqsubseteq_b \top_b$ and $\bot_b \sqsubseteq_b \mathit{true} \sqsubseteq_b \top_b$. This will be used to describe true or false values, where $\top_b$ indicates that the value is unknown and $\bot_b$ indicates an invalid value.

Type traits can be modelled using the extended boolean type given above, with one for the *real* flag and one for the *logical* flag. While in theory this would allow 4^2 possible combinations, in practice the value of one flag often dictates the value of the other flag, as for instance a logical array cannot be complex. There are in fact only 7 valid combinations including *invalid*, which are listed in Table 4.2.

real	logical
$\top_b$	$\top_b$
$\top_b$	false
false	false
true	$\top_b$
true	false
true	true
$\bot_b$	$\bot_b$

TABLE 4.2: Valid type trait combinations

Definition 4.8. To describe type traits, we define the lattice $(\mathbf{T}, \sqsubseteq_t, \top_t, \bot_t, \sqcup_t, \sqcap_t)$, where $T \subset \mathbf{B} \times \mathbf{B}$ is the set of all valid tuples and is given in (4.2). The partial order $\sqsubseteq_t$ associated with $\mathbf{T}$ is given by $\langle x, y\rangle \sqsubseteq_t \langle x', y'\rangle \iff x \sqsubseteq_b x' \wedge y \sqsubseteq_b y'$ and is depicted in Figure 4.2.

$$\begin{aligned}\mathbf{T} \;=\; & \{\langle\top_b,\top_b\rangle, \langle\top_b,\mathit{false}\rangle, \langle\mathit{false},\mathit{false}\rangle, \langle\mathit{true},\top_b\rangle, \\ & \langle\mathit{true},\mathit{false}\rangle, \langle\mathit{true},\mathit{true}\rangle, \langle\bot_b,\bot_b\rangle\} \qquad (4.2)\end{aligned}$$

Using Figure 4.1 as an example, `[-1 2 pi]` has traits $t_1 = \langle\mathit{true},\mathit{false}\rangle$, `3 == 2` has traits $t_2 = \langle\mathit{true},\mathit{true}\rangle$ and `23 + 3i` has traits $t_3 = \langle\mathit{false},\mathit{false}\rangle$. The approximation which best fits t_1 and t_2 is $t_1 \sqcup_t t_2 = \langle\mathit{true},\top_b\rangle$, while the intersection of the two trait

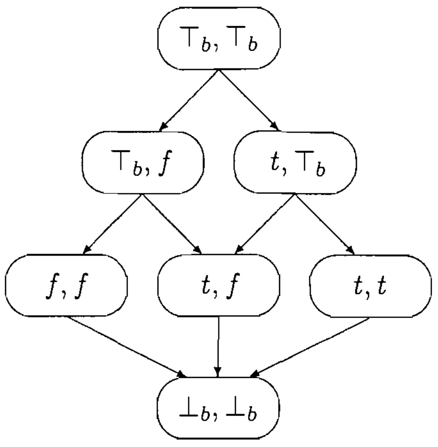


FIGURE 4.2: Visualisation of the type traits lattice, where each line indicates a covering from top to bottom

tuples, t_2 and t_3 is $t_2 \sqcap_t t_3 = \langle \bot_b, \bot_b \rangle$. The significance of these results will be explained later.

4.2.3 Combining Classes and Traits

The class and type traits of an array are not entirely independent, as not all combinations are legal. Specifically character arrays can only be real and cannot be logical, while cell arrays, structures and function handles all return false to both `isreal` and `islogical`. Additionally it makes little sense to have invalid traits with a valid class or valid traits with an invalid class, so if either is invalid, the other must be as well.

Definition 4.9. The type of an array can now be described using the lattice, $(\mathbb{T}, \sqsubseteq_T, \top_T, \bot_T, \sqcup_T, \sqcap_T)$, where $\mathbb{T}$ is defined in (4.3), $\top_T = \langle \top_k, \top_t \rangle$ and $\bot_T = \langle \bot_k, \bot_t \rangle$. If $T_1 = \langle k_1, t_1 \rangle$ and $T_2 = \langle k_2, t_2 \rangle$, then $T_1 \sqsubseteq T_2 \iff k_1 \sqsubseteq k_2 \vee t_1 \sqsubseteq t_2$ and $T_1 \sqcup T_2 = \langle k_1 \sqcup k_2, t_1 \sqcup t_2 \rangle$.

$$\begin{aligned}
\mathbf{T}' &= \mathbf{T} - \{\bot_t\} \\
\mathbf{K}' &= \mathbf{K} - \{\mathit{cell}, \mathit{struct}, \mathit{function}, \mathit{char}, \bot_k\} \\
\mathbb{T} &= \big\{\langle k, t\rangle \mid k \in \mathbf{K}', t \in \mathbf{T}'\big\} \cup \\
&\quad \big\{\big\langle k, \langle \mathit{false}, \mathit{false}\rangle\big\rangle \mid k \in \{\mathit{cell}, \mathit{struct}, \mathit{function}\}\big\} \cup \\
&\quad \big\{\big\langle \mathit{char}, \langle \mathit{true}, \mathit{false}\rangle\big\rangle, \langle \bot_k, \bot_t\rangle\big\}
\end{aligned} \tag{4.3}$$

The following arrays have $T_1 = \big\langle \mathit{cell}, \langle \mathit{false}, \mathit{false}\rangle\big\rangle$ and $T_2 = \big\langle \mathit{double}, \langle \mathit{false}, \mathit{false}\rangle\big\rangle$.

```
{10, [3 4j], 'abc'}
5 + j
```

The best approximation which fits both T_1 and T_2 is $T_1 \sqcup_T T_2 = \langle \top_k, \langle \mathit{false}, \mathit{false} \rangle \rangle$, while no array could have a type approximated by both T_1 and T_2 and so $T_1 \sqcap_T T_2 = \bot_T$. The second result indicates that if the two types are both fully *specified*, (i.e. there are no $\top$ terms), the meet of two types will always be $\bot_T$ unless the two types are identical.

Ours appears to be the first approach to MATLAB, that considers the intrinsic type (made up from the class and its type traits) and attempts to preserve it exactly as it appears in MATLAB. All the compilers mentioned [58, 54, 2] have used a simplified type system whereby types form a single increasing chain from $\bot$ through boolean through integer through real up to complex. This is also the case for the recent work by Joisha [34, 32] which, while it goes further than previously and recognises multi-dimensional arrays, ignores classes, instead continuing with the simplistic approach of others. All of these works use lattices to find the least complex representation for variables which means that memory utilisation can be minimised (since complex numbers require twice as much memory as real numbers) and so that simpler instructions can be used (for instance replacing floating point instructions with integer instructions). We have no control over the actual types used but just wish to pre-evaluate calls that are directly dependant on the types.

These compilers can take this approach because they are not producing MATLAB code as output. Since MATLAB as produced by The Mathworks is the authoritative implementation, we choose to mimic their behaviour, although it could be argued that slavish adherence to this standard sacrifices many optimisation opportunities. This does allow us to use features of MATLAB like function handles and sparse arrays, which are not available to these other compilers.

4.2.4 Dimension Information

The above abstract domain captures the types of arrays. As matrix manipulations are the backbone of MATLAB, to enable many optimisations, we need to capture abstract information about the *shape* of matrices. The shape of an array describes the size of its dimensions. As loops are frequently controlled by the size of a matrix dimension, knowledge of array shapes can enable loop unrolling.

Arrays have a number of dimensions, which is always greater than or equal to 2 and is returned by the `ndims` function. Each dimension then has a size greater than or equal to 0. The `size` function is used to get the dimension sizes. Requesting a dimension beyond the number of dimensions always returns 1. The shape of an array is defined initially in Definition 4.12 and Definition 4.13. Figure 4.3 is an example showing two arrays, `a` and `b`, along with some of the functions used to ascertain information about their shapes. The indexed assignment to `b` creates a 1-by-2-by-3 element array consisting of all zeros apart from a 1 at the indexed element.

```
>> a = [1, 2, 3];
a =
     1     2     3
>> b(1,2,3) = 1
b(:,:,1) =
     0     0
b(:,:,2) =
     0     0
b(:,:,3) =
     0     1

>> ndims(a)
ans =
     2
>> size(a)
ans =
     1     3
>> size(a, 3)
ans =
     1

>> ndims(b)
ans =
     3
>> size(b)
ans =
     1     2     3
>> size(b, 3)
ans =
     3
```

FIGURE 4.3: Examples of creating and examining arrays with different shapes

Definition 4.10. We define the extended set of non-negative integers $\mathbb{N}^\omega = \mathbb{N} \cup \{\omega\}$. We extend the ordering $<$ on $\mathbb{N}^\omega$ by stating $\forall n \in \mathbb{N}^\omega, n \leq \omega$ and also $\omega \leq n \Rightarrow n = \omega$.

Definition 4.11. A *range* is a tuple $\langle l, u\rangle$, where $l \in \mathbb{N}, u \in \mathbb{N}^\omega$ and $l \leq u$. Ranges represent all the numbers between two inclusive bounds, which is to say $\langle l, u\rangle$ is a valid approximation of x, if $l \leq x \leq u$. The most constrained approximation to x would be $\langle x, x\rangle$. We define the set $\mathbf{R}$ to contain all possible ranges with the addition of the $\bot_r$ element to indicate an invalid range ($\mathbf{R} = \{\langle l, u\rangle | l \in \mathbb{N}, u \in \mathbb{N}^\omega, l \leq u\} \cup \{\bot_r\}$). The supremum is the least constrained range possible, i.e. $\top_r = \langle 0, \omega\rangle$. We define two functions: $low(\langle l, u\rangle) = l$ and $up(\langle l, u\rangle) = u$. We also define join ($\sqcup_r$) and ($\sqcap_r$) operations on this set.

$$\begin{aligned}
x \in \langle l, u\rangle &\iff x \geq l \wedge x \leq u, \text{ where } x \in \mathbb{N}^\omega \\
\langle l_1, u_1\rangle \sqsubseteq_r \langle l_2, u_2\rangle &\iff l_1 \geq l_2 \wedge u_1 \leq u_2 \\
\langle l_1, u_1\rangle \sqcup_r \langle l_2, u_2\rangle &= \big\langle \min(l_1, l_2), \max(u_1, u_2)\big\rangle \\
\langle l_1, u_1\rangle \sqcap_r \langle l_2, u_2\rangle &= \begin{cases} \big\langle \max(l_1, l_2), \min(u_1, u_2)\big\rangle & \text{if } l_1 \leq u_2 \wedge l_2 \leq u_1 \\ \bot_r & \text{otherwise} \end{cases}
\end{aligned}$$

Throughout this chapter we use ranges to represent the possible values, shape characteristics can have. Note that while the lower bound must be finite, the upper bound can be ω. This makes the range unbounded, which is useful in the case where little or no information is available.

Definition 4.12. The number of dimensions of an array is a range as defined in Definition 4.11, except that the lower bound is always at least 2, as MATLAB arrays always have at least 2 dimensions. This is given by the set, $\mathbf{N} = \mathbf{R} - \{\langle i, j\rangle \mid i \in \{0,1\}, j \in \mathbb{N}^{\omega}\}$. The top element in $\mathbf{N}$ is $\top_n = \langle 2, \omega\rangle$. The partial ordering $\sqsubseteq_n$ and operations $\sqcap_n$ and $\sqcup_n$ are equivalent to $\sqsubseteq_r$, $\sqcap_r$ and $\sqcup_r$ respectively.

Definition 4.13. To represent the list of dimension sizes for an array, we introduce the set, $\mathbf{D} = \mathbf{R}^* \cup \bot_d$. This includes all sequences of ranges, $\langle r_1, r_2, \ldots, r_n\rangle \in \mathbf{R}^*$ and the invalid list $\bot_d$. We also define two functions $low(i, d) = low(r_i)$ and $up(i, d) = up(r_i)$, which return lower and upper bounds from a list of dimensions, d. The length of a list, d, is given by $|d|$. Below the operators are defined using $d = \langle r_1, r_2, \ldots r_n\rangle$ and $d' = \langle r'_1, r'_2, \ldots r'_m\rangle$, although $\sqcup_d$, $\sqcap_d$ and $\sqsubseteq_d$ are only defined for $n = m$.

$$\begin{array}{rcl} d \sqsubseteq_d d' & \Longleftrightarrow & \forall i \in \mathbb{Z}^+, i \leq |d|,\ r_i \sqsubseteq_r r'_i \\ d \sqcup_d d' & = & \langle r_1 \sqcup_r r'_1, \ldots, r_n \sqcup_r r'_n\rangle \\ d \sqcap_d d' & = & \langle r_1 \sqcap_r r'_1, \ldots, r_n \sqcap r'_n\rangle \\ d \cdot d' & = & \langle r_1, r_2, \ldots, r_n, r'_1, r'_2, \ldots, r'_m\rangle \end{array}$$

Definition 4.14. The function in (4.4), given $i \in \mathbb{Z}^+$, $n \in \mathbf{N}$ and $d \in \mathbf{D}$, gives the range representing the size of the ith dimension. This function combines the information present in the elements of $\mathbf{N}$ and $\mathbf{D}$ and is the basis for our later creation of a canonical form for the shape. The function is comprised as follows: if the desired dimension is in the list of dimensions then the value is the ith range in the list of dimensions; else if the dimension number is less than the number of dimensions, the dimension size is unknown so the value is $\top_r$; otherwise the dimension is beyond the number of dimensions and therefore its size is 1. Not all values of n and d give meaningful values for Definition 4.4 and we will describe constraints in Section 4.2.6.

$$dim(i, n, d) = \begin{cases} \bot_r & \text{if } n = \bot_n \vee d = \bot_d \\ \langle low(i, d), up(i, d)\rangle & \text{if } i \leq |d| \\ \langle 0, \omega\rangle & \text{if } |d| < i \leq up(n) \\ \langle 1, 1\rangle & \text{otherwise} \end{cases} \tag{4.4}$$

4.2.5 Definedness

We also need to consider the possibility that a variable may not be defined. When a function is called, it does not need to be passed as many parameters as there are in the function signature, resulting in some of the function parameters being undefined. Variables can also be undefined if they are only set on one branch of a conditional statement or in the body of a loop that may never be executed. In this case, it would be unknown whether the variable was defined, since we do not what branch will be taken. This is not a problem for a MATLAB interpreter, since the decision about whether the variable is defined is taken only after the branch has been chosen. Although it is quite possible to write code like this and have it run without error (assuming the right branches are always taken), it can be difficult to determine this statically and it is almost always an indication of programmer error. The *defined* flag is $\delta \in \mathbf{B}$. It is an error to use an undefined variable, but the built-in function `exist` can take a variable name as a string and returns whether a variable with that name is defined.

Storing information about *definedness* allows us to remove calls to `exist` at partial evaluation time, which can often lead to the removal of conditional statements. In addition some uses of undefined variables can be caught earlier.

4.2.6 Putting it all together

We have now developed all the components of our type system, which allows us to describe many different attributes of MATLAB arrays when information about their values is not available. The full type of a MATLAB array could now be described by $\mathbb{T} \times \mathbf{N} \times \mathbf{D} \times \mathbf{B}$, but this is insufficient as the individual components of the type are not entirely independent. It is possible to produce many $n \in \mathbf{N}$, $d \in \mathbf{D}$ which give the same values of $dim(i, n, d)$ for all $i \in \mathbb{Z}^+$, for instance:

$$\begin{aligned}
n_1 &= \langle 2,2\rangle, & d_1 &= \langle\langle 1,1\rangle,\langle 1,1\rangle\rangle\\
n_2 &= \langle 3,3\rangle, & d_2 &= \langle\langle 1,1\rangle,\langle 1,1\rangle,\langle 1,1\rangle\rangle\\
n_3 &= \langle 2,3\rangle, & d_3 &= \langle\langle 5,10\rangle,\langle 2,2\rangle\rangle\\
n_4 &= \langle 2,3\rangle, & d_4 &= \langle\langle 5,10\rangle,\langle 2,2\rangle,\langle 0,\omega\rangle\rangle\\
n_5 &= \langle 2,3\rangle, & d_5 &= \langle\langle 5,10\rangle,\langle 2,2\rangle,\langle 0,\omega\rangle,\langle 1,1\rangle,\langle 1,1\rangle\rangle
\end{aligned}$$

In the above examples, $\langle n_1, d_1\rangle$ and $\langle n_2, d_2\rangle$ represent the same shape. However MATLAB would return 2 as the value of `ndims` and so the value of n_2 is wrong. The same values of *dim* would also be given for $\langle n_3, d_3\rangle$, $\langle n_4, d_4\rangle$ and $\langle n_5, d_5\rangle$ so clearly redundant information is present in d_4 and d_5. Finally if $n = \langle 2,2\rangle$ and $d = \langle\langle 1,1\rangle,\langle 1,1\rangle,\langle 2,2\rangle\rangle$, then clearly n and d contradict each other, as there are 3 dimensions. In order to do comparisons of types, there needs to be a concrete description of each type with no ambiguity. In addition if a variable is undefined, it is meaningless for it to have shape or intrinsic type. If its *definedness* is unknown, it can still have shape and type as might be the case when a variable is defined in only one branch of a conditional statement. The constraints on the type are thus given below:

1. If an array, with full type $\langle t, n, d, \delta\rangle$, is undefined, e.g. $\delta = \mathit{false}$, then the values of t, n and d can only be $\bot_T$, $\bot_n$ and $\bot_d$ respectively.
2. The shape must be in the canonical form described in Definition 4.15.

Definition 4.15. The *canonical* form $\langle n', d' \rangle$ of $\langle n, d \rangle$, where $n, n' \in \mathbf{N}$ and $d, d' \in \mathbf{D}$, is calculated as follows with $l = low(n)$, $u = up(n)$, and d_i as the ith range in d. We also define $\underline{1} = \langle 1, 1 \rangle$ and $\underline{\infty} = \langle 0, \omega \rangle$.

1. If $n = \bot_n$ or $d = \bot_d$, then $n' = \bot_n$ and $d' = \bot_d$, otherwise go to step 2
2. If $\forall j \in \mathbb{Z}^+, dim(j, n, d) = \underline{1}$, then choose $l' = u' = 2$ and go to step 5, otherwise go to step 3.
3. Choose u' such that $u' = \max(2, x)$, where $dim(x, n, d) \neq \underline{1}$ and $\forall i > x$, $dim(i, n, d) = \underline{1}$.
4. Choose l' such that $l' = \max(2, y)$, where $dim(y, \langle l, u' \rangle, d) \sqcap_r \underline{1} = \bot_r$ and $\forall j > y, dim(j, \langle l, u' \rangle, d) \sqsupseteq_r \underline{1}$.
5. Choose $n' = \langle l', u' \rangle$, and d' such $d' = trunc(d, i)$,[1] where $dim(i, n', d) \neq \underline{\infty}$ and $\forall k, i < k \leq u', dim(k, n', d) = \underline{\infty}$.

The canonical form produces the correct values for the MATLAB function `ndims` and is the most compact form possible. Without a canonical form, two identical shapes could have different values of $\langle n, d \rangle$, whereas with the canonical form, these values must be identical. This reduces the problem of comparing shapes to checking for equality. The function $canon(n, d)$ gives $\langle n', d' \rangle$, where n' and d' are the canonical forms of n and d. Using the examples from the previous page, we see that $canon(n_1, d_1) = canon(n_2, d_2) = \langle n_1, d_1 \rangle$ and that $canon(n_3, d_3) = canon(n_4, d_4) = canon(n_5, d_5) = \langle n_3, d_3 \rangle$.

Theorem 4.16. *The canonical form $\langle n, d \rangle$ of $\langle n', d' \rangle$ is the most compact form, for which $dim(i, n, d) = dim(i, n', d')$, $\forall i \in \mathbb{Z}^+$.*

Proof. To see that the canonical form is the most compact form possible, it is necessary to try to remove elements from the list of dimension sizes. If it is possible to do this without loss of information then it is not the most compact.

If the canonical form $\langle n, d \rangle$ is not the most compact, then $\langle n, trunc(d, |d| - 1) \rangle$ must be an equally valid form. Validity is determined by whether the same values are calculated by the function dim. If it is valid then:

$$\begin{aligned} dim(|d|, n, d)) &= dim(|d|, n, trunc(d, |d| - 1)) \\ &= \underline{\infty} \end{aligned}$$

But by step 5 in Definition 4.15, d was chosen so that $dim(|d|, n, d) \neq \underline{\infty}$. Thus removing one element must entail a loss of information and so the canonical form is the most compact. □

[1] $trunc(\langle a_1, a_2, \ldots, a_m, \ldots, a_n \rangle, m) = \langle a_1, a_2, \ldots, a_m \rangle$, where $m < n$.

Joisha [33, 32] also describes canonical shapes and coins the term *selective rank demotion* to describe how excess $\underline{1}$ elements can be removed from the end of dimension lists with no loss of information. Since their work is for a compiler, it has a different focus as with the intrinsic types. One of the key aims is to determine if two arrays have the same shape even when this shape information is unavailable. This allows run-time checks for shape equality to be removed. This information is less likely to be useful at partial evaluation time.

Definition 4.17. While the canonical form is the most compact, it is difficult to work with as the two components must be kept synchronised. It is often easier to convert the list of dimensions to an infinite list, manipulate the list and then convert it back, as the infinite list is conceptually simpler. To this end, we give the definition of *canon* : $\mathbf{R}^\infty \rightarrow \mathbf{N} \times \mathbf{D}$.

1. If $\exists i, d_i = \bot_r$, then $s = \langle \bot_r, \bot_d \rangle$ otherwise go to step 2.
2. If $\forall j \in \mathbb{Z}^+, d_j = \underline{1}$, then choose $l' = u' = 2$ and go to step 5, otherwise go to step 3.
3. Choose $u' = \max(2, x)$, where $d_x \neq \underline{1}$ and $\forall i > x, d_i = \underline{1}$.
4. Choose $l' = \max(2, y)$, where $d_y \sqcap_r \underline{1} = \bot_r$ and $\forall j > y, d_j \sqsupseteq_r \underline{1}$.
5. Choose $d' = trunc(d, \max(2, i))$, where $d_i \neq \underline{\infty}$ and $\forall k, i < k \leq u', d_k = \underline{\infty}$, and $n' = \langle l', u' \rangle$. Choose $s = \langle n', d' \rangle$.

From this definition, it can be seen that there are some $d \in \mathbf{R}^\infty$, for which no canonical form exists, since the canonical form must be finite. An infinite list consisting solely of $\langle 1, \omega \rangle$ cannot be canonicalised, which means there is no way to represent the shapes of all non-empty arrays.

Definition 4.18. We define a function $pad(d, u, m)$, where $d \in \mathbf{D}, u \in \mathbb{N}^\omega, m \in \mathbb{N}$, which gives a list of dimensions of size m, padding out any missing elements with $\underline{\infty}$ and $\underline{1}$ as appropriate. This function is necessary as the operators defined in Definition 4.13 can only be used with lists of equal length. Note $x = \min(u, m) - |d|$ and $y = m - x$.

$$pad(d, u, m) = d \cdot \underbrace{\langle \underline{\infty} \ldots \underline{\infty} \rangle}_{x} \cdot \underbrace{\langle \underline{1} \ldots \underline{1} \rangle}_{y}$$

Definition 4.19. We now define the shape to be $\mathbf{S} \subset \mathbf{N} \times \mathbf{D}$ with the restriction that $\mathbf{S}$ must always be in the canonical form described in Definition 4.15. So in fact $\mathbf{S} = canon^*$. We now define the lattice $(\mathbf{S}, \sqsubseteq_s, \top_s, \bot_s, \sqcup_s, \sqcap_s)$, with $\top_s = \langle \langle 2, \omega \rangle, \langle \rangle \rangle$ and $\bot_s = \langle \bot_n, \bot_d \rangle$. The partial order and join and meet operations are now defined with

$s_1, s_2 \in \mathbf{S}, s_1 = \langle n_1, d_1 \rangle, s_2 = \langle n_2, d_2 \rangle$

$$\begin{array}{rcl} d_1' & = & pad(d_1, up(n_1), \max(|d_1|, |d_2|)) \\ d_2' & = & pad(d_2, up(n_2), \max(|d_1|, |d_2|)) \\ s_1 \sqcup_s s_2 & = & canon(n_1 \sqcup_n n_2, d_1' \sqcup_d d_2') \\ s_1 \sqcap_s s_2 & = & canon(n_1 \sqcap_n n_2, d_1' \sqcap_d d_2') \\ s_1 \sqsubseteq_s s_2 & \Longleftrightarrow & n_1 \sqsubseteq_n n_2 \wedge d_1 \sqsubseteq_d d_2 \end{array}$$

We can describe many shapes using our canonical form, but in one area it is deficient: It cannot be used to specify a minimum value that would be returned by `ndims` without also specifying the size of some dimensions. Since an array has 3 dimensions if the third dimension has size 0 or greater than 1, there is no way to specify this without also including 1, which gives the possibility that the array might only have 2 dimensions.

Definition 4.20. The full abstract type can now be represented by $\mathbf{C} \subset \mathbb{T} \times \mathbf{S} \times \mathbf{B}$. The full type lattice is thus $\mathfrak{L}_c = (\mathbf{C}, \sqsubseteq_c, \top_c, \bot_c, \sqcup_c, \sqcap_c)$ with $\top_c = \langle \top_T, \top_s, \top_b \rangle$ and $\bot_c = \langle \bot_T, \bot_s, \bot_b \rangle$. Given $c_1, c_2 \in \mathbf{C}$, where $c_1 = \langle t_1, s_1, \delta_1 \rangle$ and $c_2 = \langle t_2, s_2, \delta_2 \rangle$, with $t_1, t_2 \in \mathbb{T}$, $n_1, n_2 \in \mathbf{N}$, $d_1, d_2 \in \mathbf{D}$ and $\delta_1, \delta_2 \in \mathbf{B}$.

$$\begin{array}{rcl} c_1 \sqsubseteq_c c_2 & \Longleftrightarrow & t_1 \sqsubseteq_T t_2 \wedge s_1 \sqsubseteq_s s_2 \wedge \delta_1 \sqsubseteq_b \delta_2 \\ c_1 \sqcup_c c_2 & = & \langle t_1 \sqcup_T t_2, s_1 \sqcup_s s_2, \delta_1 \sqcup_b \delta_2 \rangle \\ c_1 \sqcap_c c_2 & = & \langle t_1 \sqcap_T t_2, s_1 \sqcap_s s_2, \delta_1 \sqcap_b \delta_2 \rangle \end{array}$$

In later chapters we will use $\mathfrak{L}_c$ extensively in our data flow analysis, to show how type information is propagated as a program is executed. The join operation for the full type ($\sqcup_c$) is of great importance as it can be used to merge the types of variables emerging from two different paths of a conditional statement. The meet operation for shapes ($\sqcap_s$) is of use with operators where the operands must have the same shape. In which case the meet of the two shapes will be $\bot_s$ if the shapes are incompatible. The functions, *shape* : $\mathbf{C} \rightarrow \mathbf{S}$ and *type* : $\mathbf{C} \rightarrow \mathbb{T}$ can be used to extract the shape and type components from a full abstract type tuple.

4.3 Full Type System

In the previous section, we defined the abstract type system. This is used to capture information about dynamic data in programs when the actual value is unknown. On the other hand, full static information is frequently available and is vital to the effectiveness of any partial evaluator.

When the value of a variable or expression is known statically, the value is stored in the MATLAB library type, `mxarray`. This is the structure which is passed to all MATLAB library function calls and is capable of expressing all the MATLAB types available to the libraries.

Definition 4.21. We define the set, $\mathbf{M}$, to contain all possible `mxarray` structures.

Definition 4.22. Combining $\mathbf{C}$, from Definition 4.20, with $\mathbf{M}$, from Definition 4.21, gives us all the expressiveness required for the symbolic execution method used by MPE. Below we define the set of tagged values, $\mathbf{A}$, used by our execution engine.

$$\mathbf{A}^{\sharp} = \{\langle dynamic, c\rangle | c \in \mathbf{C}\} \tag{4.5}$$

$$\mathbf{A}^{\flat} = \{\langle static, m\rangle | m \in \mathbf{M}\} \tag{4.6}$$

$$\mathbf{A} = \mathbf{A}^{\sharp} \cup \mathbf{A}^{\flat} \tag{4.7}$$

Definition 4.23. Often static arrays need to be made dynamic, as in the case of operators where one operand is static and the other is dynamic. For this we define the abstraction function, $\alpha : \mathbf{A} \rightarrow \mathbf{C}$. We also define a concretisation function, $\gamma : \mathbf{A} \rightarrow \mathbb{P}\mathbf{M}$, which gives us the set of all arrays that fit the abstraction. For $a \in \mathbf{A}^{\flat}$, $\gamma(a)$ is a set with only one value. The function $\gamma' : \mathbf{A} \rightarrow \mathbf{M}$, is only defined when $|\gamma(a)| = 1$ (when there is only one concrete value associated with an abstraction) and is equal to that one static value. It is however still defined for some $a \in \mathbf{A}^{\sharp}$, since a fully specified class and traits, along with a completely specified but empty shape, has only one valid static array which could represent it.

The following code demonstrates the information that is extracted from the static array to get the dynamic array representation.

```
n = ndims(a);
dims = size(a);
defined = 1;
isreal = isreal(a);
islogical = islogical(a);
k = class(a);
```

Each of these values is then converted into the form used by the abstract representation. For instance `ndims` returns a positive integer, which is converted into a range using the following equation.

$$convert\_to\_range(n) = \langle n, n\rangle \tag{4.8}$$

The list of dimensions, returned by `size`, is a list of positive integers, which is converted into a equivalently sized list of ranges.

$$convert\_to\_ranges(\langle d_1, \ldots d_n\rangle) = \langle\langle d_1, d_1\rangle, \ldots \langle d_n, d_n\rangle\rangle \tag{4.9}$$

The type properties returned by `isreal` and `islogical` are integers which need to be converted to concrete boolean values.

$$convert\_to\_exbool(b) = \begin{cases} true & \text{if } b = 1 \\ false & \text{otherwise} \end{cases} \tag{4.10}$$

The `class` function returns a string representation of the class which maps directly to a member of $\mathbf{K}$ as in (4.1).

We now define a partial order over $\mathbf{A}$ $(a_1, a_2 \in \mathbf{A})$, with supremum, $\top_a = \langle dynamic, \top_c \rangle$:

$$a_1 \sqsubseteq_a a_2 \iff (a_1 \in \mathbf{A}^\sharp \wedge \alpha(a_1) \sqsubseteq_{a^\sharp} \alpha(a_2)) \vee (a_1 \in \mathbf{A}^\flat \wedge a_1 = a_2) \tag{4.11}$$

With this definition, static values are never looser than dynamic values. The most accurate abstraction of the value of a variable is the value itself. The join operator is defined as follows:

$$a_1 \sqcup_a a_2 = \begin{cases} a_1 & \text{if } a_1 \in \mathbf{A}^\flat \wedge a_1 = a_2 \\ \langle dynamic, \alpha(a_1) \sqcup_c \alpha(a_2) \rangle & \text{otherwise} \end{cases} \tag{4.12}$$

This operator is fundamental to merging the states from the branches of conditional statements. If a variable is assigned the same static value on both branches, the state will contain this same value after the conditional. On the other hand, if each branch sets it to a static but different value, the value will be *abstracted* and an approximation to the type will be recorded in the environment instead.

In the following subsections, we will derive equations for calculating various pieces of type information required for the symbolic execution. In Sections 4.3.1 and 4.3.2, we give equations for calculating the shape resulting from each of the binary and unary operators. In Section 4.3.4, we give the equations for determining the shape resulting from the various forms of array indexing. Section 4.3.5 gives equations for finding the class resulting from using the various operators, while Section 4.3.6 does the same for traits. Section 4.3.7 then brings together the previous sections with regard to calculating full type information for operators.

4.3.1 Shape Equations For Binary Operators

Most binary operators in MATLAB are called *element-wise* binary operators, including `+`, `.*` and `==`. These either operate on two arrays with equal dimensions, one non-scalar and a scalar or two scalars. They always result in an array of the same shape as the non-scalar operand (or a scalar in the case of two scalars). In this section, we will derive equations for calculating the shape resulting from a binary operator given the shapes of its operands. Joisha [33], also derives equations for the shape resulting from binary operators, using his type formalisation.

Assuming a binary expression $a \oplus b$, where a and b have shapes $s_a, s_b \in \mathbf{S}$ and s_s is the shape of a scalar, then the two scalars case is represented by:

$$\begin{aligned} s_1 &= (s_a \sqcap_s s_s) \sqcap_s (s_b \sqcap_s s_s) \\ &= s_a \sqcap_s s_b \sqcap s_s \end{aligned} \tag{4.13}$$

The case where both are non-scalars is given by:

$$s_2 = s_a \sqcap_s s_b \tag{4.14}$$

When the first operand is a scalar and the second an array, we have:

$$s_3 = inc(s_a, s_s) \sqcap_s s_b \tag{4.15}$$

Similarly, when the second is a scalar we have:

$$s_4 = inc(s_b, s_s) \sqcap_s s_a \tag{4.16}$$

The function, $inc : \mathbf{S} \times \mathbf{S} \rightarrow \mathbf{S}$, is defined to be $\top_s$, when the first shape also includes the second and $\perp_s$ otherwise:

$$inc(a,b) = \begin{cases} \top_s & \text{if } a \sqsupseteq_s b \\ \perp_s & \text{otherwise} \end{cases} \tag{4.17}$$

The resulting shape, $binop_s(s_a, s_b)$, is given by the combination of (4.13), (4.14), (4.15) and (4.16):

$$\begin{aligned} binop_s(s_a, s_b) &= s_1 \sqcup_s s_2 \sqcup_s s_3 \sqcup_s s_4 \\ &= (s_a \sqcap_s s_b \sqcap s_s) \sqcup_s (s_a \sqcap_s s_b) \sqcup_s (inc(s_a, s_s) \sqcap_s s_b) \sqcup_s (inc(s_b, s_s) \sqcap_s s_a) \\ &= (s_a \sqcap_s s_b) \sqcup_s (inc(s_a, s_s) \sqcap_s s_b) \sqcup_s (inc(s_b, s_s) \sqcap_s s_a) \end{aligned} \tag{4.18}$$

If $binop_s(s_a, s_b)$ is $\perp_s$, it implies that applying an array binary operator will result in an error as the shapes are incompatible.

The matrix multiply operator, $*$, is the same as the array multiply operator, $.*$, except when neither operand is a scalar. In this case both operands must be matrices with matching inner dimensions, the equation for which is given below, with $s_{ij} = dim(s_i, j)$:

$$s_5 = \langle \langle 2, 2 \rangle, \langle s_{a1}, s_{b2} \rangle \rangle \sqcap_s eq(s_{a2}, s_{b1}) \sqcap_s matrix(s_a) \sqcap_s matrix(s_b) \tag{4.19}$$

The functions, $eq : \mathbf{R} \times \mathbf{R} \rightarrow \mathbf{S}$, $neq : \mathbf{R} \times \mathbf{R} \rightarrow \mathbf{S}$ and $matrix : \mathbf{S} \rightarrow \mathbf{S}$, are defined as follows (neq is not used here, but is required later in this chapter):

$$eq(d_1, d_2) = \begin{cases} \bot_s & \text{if } d_1 \sqcap d_2 = \bot_r \\ \top_s & \text{otherwise} \end{cases} \tag{4.20}$$

$$neq(d_1, d_2) = \begin{cases} \top_s & \text{if } d_1 \neq d_2 \\ \bot_s & \text{otherwise} \end{cases} \tag{4.21}$$

$$matrix(s) = \begin{cases} \top_s & \text{if } ndims(s) \sqsupseteq_r \langle 2, 2 \rangle \\ \bot_s & \text{otherwise} \end{cases} \tag{4.22}$$

The shape resulting from matrix multiplication, $multiply_s(s_a, s_b)$, is then given by the join of (4.13), (4.15), (4.16) and (4.19):

$$\begin{aligned} multiply(s_a, s_b) &= s_1 \sqcup_s s_3 \sqcup_s s_4 \sqcup_s s_5 \\ &= (s_a \sqcap_s s_b \sqcap s_s) \sqcup_s (inc(s_a, s_s) \sqcap_s s_b) \sqcup_s \\ &\quad (inc(s_b, s_s) \sqcap_s s_a) \sqcup_s \big(\langle \langle 2, 2 \rangle, \langle s_{a1}, s_{b2} \rangle \rangle \sqcap_s \\ &\quad eq(s_{a2}, s_{b1}) \sqcap_s matrix(s_a) \sqcap_s matrix(s_b)\big) \end{aligned} \tag{4.23}$$

Left matrix division (`a \ b`), works like array division when the first operand is a scalar, but otherwise requires two matrices, each with the same number of rows. The result will have as many rows as the first had columns and as many columns as the second matrix, since it computes $a^{-1}b$.

$$\begin{aligned} mldivide_s(s_a, s_b) &= (inc(s_a, s_s) \sqcap_s s_b) \sqcup_s \big(matrix(s_a) \sqcap_s matrix(s_b) \sqcap_s \\ &\quad eq(s_{a1}, s_{b1}) \sqcap_s \langle \langle 2, 2 \rangle, \langle s_{a2}, s_{b2} \rangle \rangle\big) \end{aligned} \tag{4.24}$$

Right matrix division (`a / b`), works like array division when the second operand is a scalar, but otherwise requires two matrices, each with the same number of columns. The result will have as many columns as the first had rows and as many rows as the second matrix.

$$\begin{aligned} mrdivide_s(s_a, s_b) &= (inc(s_b, s_s) \sqcap_s s_a) \sqcup_s \big(matrix(s_a) \sqcap_s matrix(s_b) \sqcap_s \\ &\quad eq(s_{a2}, s_{b2}) \sqcap_s \langle \langle 2, 2 \rangle, \langle s_{a1}, s_{b1} \rangle \rangle\big) \end{aligned} \tag{4.25}$$

The final binary operator is matrix power (`a ^ b`), which requires one operand to be a scalar, while the other must be a square matrix. The result has the same shape as the matrix.

$$\begin{aligned} mpower_s(s_a, s_b) &= \big(inc(s_a, s_s) \sqcap_s \langle \langle 2, 2 \rangle, \langle s_{b1} \sqcap_s s_{b2}, s_{b1} \sqcap_s s_{b2} \rangle \rangle\big) \sqcup_s \\ &\quad \big(inc(s_b, s_s) \sqcap_s \langle \langle 2, 2 \rangle, \langle s_{a1} \sqcap_s s_{a2}, s_{a1} \sqcap_s s_{a2} \rangle \rangle\big) \end{aligned} \tag{4.26}$$

4.3.2 Shape Equations For Unary Operators

There are only 4 unary operators in MATLAB: unary plus and minus as well as two types of transpose. The transpose operators, `'` and `.'`, both require a matrix operand and result in a matrix with the number of rows and columns swapped.

$$transp_s(s) \quad = \quad canon(\langle ndims(s) \sqcap_n \langle 2,2\rangle, \langle dim(s,2), dim(s,1)\rangle\rangle) \qquad (4.27)$$

If the operand is not a matrix (i.e. it has more than two dimensions), $ndims(s) \sqcap_n \langle 2,2\rangle$ will be $\bot_r$, indicating an error.

The other two unary operators, `-` and `+`, produce a result with an identical shape to the operand.

4.3.3 Shape Equations For Built-in Functions

In this section, we will give shape equations for the results of executing built-in functions. We only demonstrate a few of these, since there are at least 300 built-in functions in MATLAB 6.1. This seems a lot, but many of these do not return values or return values with trivial shapes such as scalars. In addition many functions share the same shape characteristics. Our interest here is only to demonstrate functions, which return values whose shapes can be determined by examining the shapes of the parameters. The functions examined in this section are `size`, `horzcat`, `vertcat` and `cat`. We also look at `numel` which returns a scalar and is useful in other equations.

The `size` function, when invoked with one parameter and one output, returns a row vector containing the size of each of its dimensions. This vector always has at least two elements as arrays always have at least two dimensions. Even when an array is dynamic, this function can be fully evaluated, when the dimension information itself is static. If it is not static, we can infer the shape of the size vector itself.

$$size(s) \quad = \quad \langle\langle 2,2\rangle, \langle \underline{1}, ndims(s)\rangle\rangle \qquad (4.28)$$

The function `numel` gives the total number of elements in an array. This is given by the product of the sizes of all the dimensions. The number of elements in an array is given by the function, $numel : \mathbf{S} \rightarrow \mathbf{R}$:

$$numel(s) = \prod_{i \in \mathbb{Z}^+} dim(s,i) \qquad (4.29)$$

Multiplication over $\mathbf{R}$ is defined as follows:

$$\langle l_1, u_1\rangle.\langle l_2, u_2\rangle = \begin{cases} \langle 0,0\rangle & \text{if } u_1 = 0 \lor u_2 = 0 \\ \langle l_1.l_2, \omega\rangle & \text{if } u_1 = \omega \lor u_2 = \omega \\ \langle l_1.l_2, u_1.u_2\rangle & \text{otherwise} \end{cases} \tag{4.30}$$

Clearly if $numel(s) = \langle x, x\rangle$, the result is static and the function call can be removed and replaced with a constant value. While (4.29) requires iteration over an infinite set, it is not difficult to rework this for our canonical form:

$$numel(\langle\langle l, u\rangle, d\rangle) = \begin{cases} \prod_{1\le i\le |d|} d_i & \text{if } |d| = u \\ \left\langle 0, \left(\prod_{1\le i\le |d|} up(d_i)\right).\omega\right\rangle & \text{otherwise} \end{cases} \tag{4.31}$$

Another important function is `horzcat`, which concatenates two arrays horizontally. This function is used implicitly in the construction of matrices as `[2 2]` is equivalent to `horzcat(2,2)`. Vertical concatenation is done using `vertcat`, which is used implicitly in expressions like `[1; 2]`. Concatenation along any dimension is possible using `cat` with the dimension number as the first parameter.

If concatenation is along dimension i, then all dimensions except for i must be the same size or an error will result. The resulting shape will have all of these dimensions the same size, while dimension i will be the sum of the sizes of dimension i in the parameter arrays. The one exception is if one of the arrays is the empty matrix, `[]`, which has shape $\langle\langle 2,2\rangle, \langle\langle 0,0\rangle, \langle 0,0\rangle\rangle\rangle$. Concatenating `[]` with x or x with `[]`, results in x. Other *empty* arrays, (e.g. `zeros(10, 0)`), do not have this property. We define two auxiliary functions *empty* and *notempty*, which give $\top_s$ if the shape could possess the requisite property.

$$empty(s) = \begin{cases} \top_s & \text{if } s \sqsupseteq_s \langle\langle 2,2\rangle, \langle\langle 0,0\rangle, \langle 0,0\rangle\rangle\rangle \\ \bot_s & \text{otherwise} \end{cases} \tag{4.32}$$

$$notempty(s) = \begin{cases} \top_s & \text{if } s \neq \langle\langle 2,2\rangle, \langle\langle 0,0\rangle, \langle 0,0\rangle\rangle\rangle \\ \bot_s & \text{otherwise} \end{cases} \tag{4.33}$$

We now derive an equation giving the shape resulting from the function call `cat(n, a, b)`, where `a` and `b` have shapes, s_a and s_b, and $n \in \mathbb{Z}$ is the value of `n`. Below are the two cases where either of the two arrays to be concatenated could be empty.

$$s_1 = empty(s_a) \sqcap_s s_b \tag{4.34}$$

$$s_2 = empty(s_b) \sqcap_s s_a \tag{4.35}$$

To do concatenation of non-empty arrays, we define a function $setdim : \mathbf{S} \times \mathbb{Z} \times \mathbf{D} \to \mathbf{S}$, which, given a shape, is equal to it, but for a specified dimension set to a new size. We also define a function $plus : \mathbf{R} \times \mathbf{R} \to \mathbf{R}$, which gives the sum of two ranges.

$$\begin{aligned} replacedim(\langle d_1, \dots, d_n \rangle, i, d) &= \langle d_1, \dots, d_{i-1}, d, d_{i+1}, \dots, d_n \rangle && (4.36) \\ setdim(\langle \langle l, u \rangle, ds \rangle, i, d) &= canon(replacedim(pad(ds, u, \infty), i, d)) && (4.37) \end{aligned}$$

$$plus(r_1, r_2) = \begin{cases} \bot_r & \text{if } r_1 = \bot_r \vee r_2 = \bot_r \\ \langle low(r_1) + low(r_2), up(r_1) + up(r_2) \rangle & \text{otherwise} \end{cases} \quad (4.38)$$

The shape produced by concatenating two non-empty arrays along a dimension, i, is given be s_3:

$$\begin{aligned} s_3 &= notempty(s_a) \sqcap_s notempty(s_b) \sqcap_s \\ &\quad setdim(setdim(s_a, i, \underline{\infty}) \sqcap_s setdim(s_b, i, \underline{\infty}), i, plus(s_{ai}, s_{bi})) \quad (4.39) \end{aligned}$$

The function $concat : \mathbf{S} \times \mathbf{S} \times \mathbb{N} \to \mathbf{S}$ gives the shape produced by concatenation along a specific dimension and is the combination of (4.34), (4.35) and (4.39).

$$\begin{aligned} concat(s_a, s_b, i) &= s_1 \sqcup_s s_2 \sqcup_s s_3 \\ &= (empty(s_a) \sqcap_s s_b) \sqcup_s (empty(s_b) \sqcap_s s_a) \sqcup_s \\ &\quad (notempty(s_a) \sqcap_s notempty(s_b) \sqcap_s \\ &\quad setdim(setdim(s_a, i, \underline{\infty}) \sqcap_s setdim(s_b, i, \underline{\infty}), i, plus(s_{ai}, s_{bi}))) \\ &\qquad (4.40) \end{aligned}$$

4.3.4 Shape Equations For Array Indexing

The shape resulting from array indexing is dependent on both the shape of the array being indexed and the index. In addition logical indexes have a different behaviour to non-logical indexes. Indexing using normal parentheses (`a(1:3)`) is also different to using braces (`a{1:3}`).

Indexing using braces is used for extracting elements from cell arrays and produces shape results which cannot be inferred from the shape of the cell array. As no information is stored about the contents of cell arrays, we cannot say anything except to recognise when an index is out of bounds. As such, this is an example where we do not want to immediately abstract all of the values involved if one of them is dynamic. For instance, `a(1==0)`, is always `[]`, regardless of the contents of a, so long as it is defined. This is true whenever the index is logical and contains only 0s.

In the following subsections we will derive shape equations for the result of an ordinary (using normal parentheses) array index. The four types of array indexing described are: indexing with a single non-logical value, indexing with multiple non-logical values, indexing with a single logical value and finally indexing with multiple indices where the final one is logical.

4.3.4.1 Single index non-logical subscripts

We will first consider subscripts with only one index. If the index is not logical, then the result will normally have the same shape as the index. Unfortunately it is not quite that simple, in that MATLAB considers vectors to be special cases and gives different behaviours for them. In the case of indices, a vector is an array with 2 dimensions and either only one row or column of non-unit length. The array being indexed is considered a vector if all but one of its dimensions has unit length.

When both the array and the index are vectors by the definitions above, the result will be a vector, along the non-unit-length dimension of the array, with the same size as the index. Consider the array access `p(q)`, where `p` has size `[a b]` and `q` has size `[c d]`, then `p(q)` will have size `[e f]`. Normally `e` = `c` and `f` = `d`, but Table 4.3 below gives the exceptions caused by the vector behaviour. For array accesses on multi-dimensional vectors, the result will also be a multi-dimensional vector, although this is not shown in the table. Here we define the function *array_vector* : $\mathbf{S} \rightarrow \mathbb{PZ}^+$, which gives the

a	b	c	d	e	f
0	1	1	0	0	1
1	0	0	1	1	0
1	y	0	1	1	0
1	y	m	1	1	m
x	1	1	0	0	1
x	1	1	n	n	1

TABLE 4.3: Non-regular matrix accesses using a single index

dimensions which could be of non-unit length when all the others could be unit length. In addition *index_vector* : $\mathbf{S} \rightarrow \mathbf{R}$, gives a *range* approximating the length of the index if it is a vector.

$$array\_vector(s) = \left\{ i | i \in \mathbb{N} \wedge \left(neq(s_i, \underline{1}) \sqcap_r \bigsqcap_{j \in \mathbb{N} - \{i\}} (eq(s_j, \underline{1})) \right) = \top_r \right\} \quad (4.41)$$

$$index\_vector(s) = \bigsqcup_{i \in \{1,2\}} \left(s_i \sqcap_r neq(s_i, \underline{1}) \sqcap_r \bigsqcap_{j \in \mathbb{N} - \{i\}} (eq(s_j, \underline{1})) \right) \quad (4.42)$$

Using these two functions, we can develop a flawed equation for the list of dimensions produced by indexing into an array.

$$d = \bigsqcup_{i \in array\_vector(s_a)} \underbrace{\langle \underline{1} \ldots \underline{1} \rangle}_{i-1} . \langle index\_vector(s_b) \rangle . \underbrace{\langle \underline{1} \ldots \underline{1} \rangle}_{\infty} \quad (4.43)$$

This equation is fine for all cases where the upper limit of the number of dimensions in s_a is finite. Unfortunately if it is ω and the value of $index\_vector(s_b)$ is not $\underline{1}$ or $\underline{\infty}$ then there will be no canonical form of the shape, as the size of ithe dimensions list will be infinite.

Our solution is to sacrifice some accuracy when the list of dimensions is not complete (because it has been padded out with $\underline{\infty}$). This is not a great problem, as lack of information about the number of dimensions usually indicates that not all that much was known in the first place. To do this, we define a function, $vector : \mathbf{S} \times \mathbf{R} \rightarrow \mathbf{S}$, which given the shape of the indexed array and a range which will be the length of any new vector, returns the shape describing all the possible vectors.

$$vector(s, r) = s' \sqcup_s canon(d') \tag{4.44}$$

$$indices = array\_vector(s) \sqcap \{i \in \mathbb{N} | i \leq |dims(s)|\} \tag{4.45}$$

$$s' = \bigsqcup_{i \in indices} canon(\underbrace{\langle \underline{1} \ldots \underline{1} \rangle}_{i-1} . \langle r \rangle . \underbrace{\langle \underline{1} \ldots \underline{1} \rangle}_{\infty}) \tag{4.46}$$

$$d' = \begin{cases} \underbrace{\langle \bot_r \ldots \bot_r \rangle}_{\infty} & \text{if } up(ndims(s)) = |dims(s)| \\ \underbrace{\langle \underline{1} \ldots \underline{1} \rangle}_{|dims(s)|} . \underbrace{\langle \top_r \ldots \top_r \rangle}_{\infty} & \text{if } up(ndims(s)) = \omega \\ \underbrace{\langle \underline{1} \ldots \underline{1} \rangle}_{|dims(s)|} . \langle \underbrace{\top_r \ldots \top_r}_{up(ndims(s)) - |dims(s)|} \rangle . \underbrace{\langle \underline{1} \ldots \underline{1} \rangle}_{\infty} & \text{otherwise} \end{cases} \tag{4.47}$$

No accuracy is lost if $up(ndims(s)) = |dims(s)|$. Using *canon* from Definition 4.17, the final result for a single index subscript involving vectors is:

$$vecsubscript(s_a, s_b) = vector(s_a, index\_vector(s_b)) \tag{4.48}$$

Now it is necessary to bring in the standard case where either the array or index is not a vector ($notvec : \mathbf{S} \times \mathbf{S} \rightarrow \mathbf{S}$).

$$notvec(s_a, s_b) = \begin{cases} \top_s & \text{if } array\_vector(s_a) = \varnothing \vee index\_vector(s_b) = \bot_r \\ \bot_s & \text{otherwise} \end{cases} \tag{4.49}$$

The final equation is therefore ($subscript : \mathbf{S} \times \mathbf{S} \rightarrow \mathbf{S}$):

$$subscript(s_a, s_b) = (notvec(s_a, s_b) \sqcap_s s_b) \sqcup vecsubscript(s_a, s_b) \tag{4.50}$$

4.3.4.2 Multiple index non-logical subscripts

Indexing into an array with multiple indices, none of which are logical, is very simple to understand. Each of the indices is flattened so that all its elements form a vector.

Its length is the product of the sizes of all of its dimensions. This is the value that is returned by the `numel` built-in function, for which an equation was given in (4.31). The resulting array will have as many rows as the first index has elements and as many columns as the second index has elements and so on for further indices. If the indexed array has more dimensions than there are indices then the extra dimensions are flattened into the final one.

The equation for array indexing with multiple indices is therefore ($subscript : \mathbf{S} \times \mathbf{S}^* \to \mathbf{S}$):

$$subscript(s_a, \langle s_1 \ldots s_n \rangle) = canon\big(\langle numel(s_1), \ldots numel(s_n)\rangle . \langle \underbrace{\underline{1} \ldots \underline{1}}_{\infty} \rangle\big) \tag{4.51}$$

4.3.4.3 Single logical subscripts

A single logical index acts like a filter, where non-zero elements select elements in the array to extract. The extracted elements are placed in a vector. The vector can have as many elements as the index has, but could also be empty. This number is given as a range by $lnumel : \mathbf{S} \to \mathbf{R}$.

$$lnumel(s) = \langle 0, up(numel(s)) \rangle \tag{4.52}$$

As with non-logical single index subscripts, when vectors are involved, the analysis becomes complicated. When the indexed array is a vector (along any dimension), the result will also be a vector along that dimension. When the indexed array is not a vector then the type of the vector produced depends on whether the index has more than one row. If it does the result will be a row vector, otherwise it will be a column vector.

The index can be larger than the array into which it is indexing, but an error will occur if a non-zero element occurs in a place which would retrieve a non-existent element. For the purposes of this analysis we will ignore this error case and assume that only valid elements are accessed.

The vector case is covered by:

$$s' = vector(s_a, lnumel(s_b)) \tag{4.53}$$

The non-vector case is covered by (neq is given in (4.21)):

$$s'' = \big\langle \langle 2, 2 \rangle, (neq(s_{b1}, \underline{1}) \sqcap_d \langle lnumel(s_b), \underline{1} \rangle) \sqcup_s (eq(s_{b1}, \underline{1}) \sqcap_d \langle \underline{1}, lnumel(s_b) \rangle) \big\rangle \tag{4.54}$$

class(a)	class(b)	class(c)
double	double	double
double	sparse	double
double	char	double
sparse	sparse	sparse
sparse	char	double
char	char	double

TABLE 4.4: Class table for addition like binary operators. (Only valid classes are shown – all other classes would produce an error).

The final shape equation for single logical subscripts is:

$$\begin{aligned} subscript_l(s_a, s_b) &= s' \sqcup_s s'' \\ &= vector(s_a, lnumel(s_b)) \sqcup_s \\ &\quad \big(nonvec(s_a) \sqcap_s \langle\langle 2,2\rangle, (neq(s_{b1}, \underline{1}) \sqcap_d \langle lnumel(s_b), \underline{1}\rangle) \sqcup_s \\ &\qquad (eq(s_{b1}, \underline{1}) \sqcap_d \langle \underline{1}, lnumel(s_b)\rangle)\rangle\rangle\big) \end{aligned} \tag{4.55}$$

4.3.4.4 Multiple indices with a final logical index

With multiple indices, only the final index can be logical or an error will occur. The final logical index filters over the remaining dimensions, which are all flattened into one. Like the non-logical case, the size of each index gives the size of the corresponding dimension in the result. The dimension corresponding to the logical index has a size between 0 and the maximum number of elements of the logical index (as would be given by the function `numel`). The function *subscript* : $\mathbf{S} \times \mathbf{S}^* \times \mathbf{S} \to \mathbf{S}$, gives the resulting shape.

$$subscript(s_a, \langle s_1 \dots s_n\rangle, s_b) = canon\big(\langle numel(s_1), \dots numel(s_n), lnumel(s_b), \underbrace{\underline{1} \dots \underline{1}}_{\infty}\rangle\big) \tag{4.56}$$

4.3.5 Class Equations For Operators

The class of an array produced by operators is often predictable. For instance adding two real double arrays will produce a real double array. Adding two real sparse arrays will produce a real sparse array, but adding a real double array to a real sparse array will produce a real double array. On the other hand array multiplication of a sparse array by a double array produces a sparse array.

Table 4.4 and Table 4.5 show the resulting class for addition and array multiplication. Since these operators are symmetric, redundant entries have been removed. Table 4.6 shows the class table for the non-symmetric operator, array right division. Left division will be equivalent to swapping the operands. Since only double, sparse or char classes are

class(a)	class(b)	class(c)
double	double	double
double	sparse	sparse
double	char	double
sparse	sparse	sparse
sparse	char	sparse
char	char	double

TABLE 4.5: Class table for array multiplication like binary operators. (Only valid classes are shown – all other classes would produce an error).

class(a)	class(b)	class(c)
double	double	double
double	sparse	double
sparse	double	sparse
double	char	double
char	double	double
sparse	sparse	sparse
sparse	char	sparse
char	sparse	double
char	char	double

TABLE 4.6: Class table for right array division. (Only valid classes are shown – all other classes would produce an error).

class(a)	class(b)	class(c)
double	double	double
double	sparse	sparse
double	char	double
sparse	sparse	sparse
sparse	char	sparse
char	char	double
single	*	uint8
int8	*	uint8
int16	*	uint8
int32	*	uint8
uint8	*	uint8
uint16	*	uint8
uint32	*	uint8

TABLE 4.7: Class table for logical binary operators. (* indicates includes any class in the table).

defined for arithmetic operations, we define the function, $arith_k : \mathbf{K} \rightarrow \mathbf{K}$, to determine whether an operand is valid:

$$arith_k(c) = \begin{cases} \top_k & \text{if } c \in \{double, sparse, char, \top_k\} \\ \bot_k & \text{otherwise} \end{cases} \tag{4.57}$$

Using the previous tables, we can easily derive the following function:, $plus_k : \mathbf{K} \times \mathbf{K} \rightarrow \mathbf{K}$, $multiply_k : \mathbf{K} \times \mathbf{K} \rightarrow \mathbf{K}$, $rdivide_k : \mathbf{K} \times \mathbf{K} \rightarrow \mathbf{K}$ and $ldivide_k : \mathbf{K} \times \mathbf{K} \rightarrow \mathbf{K}$. These functions all use the operand classes to determine the class of the result.

$$\begin{aligned} plus_k(c_1, c_2) = {} & (c_1 \sqcap_k double \sqcap_k arith_k(c_2)) \sqcup_k (c_2 \sqcap_k double \sqcap_k arith_k(c_1)) \sqcup_k \\ & (eq(c_1, char) \sqcap_k double \sqcap_k arith_k(c_2)) \sqcup_k \\ & (eq(c_2, char) \sqcap_k double \sqcap_k arith_k(c_1)) \sqcup_k (c_1 \sqcap_k c_2 \sqcap_k sparse) \end{aligned} \tag{4.58}$$

$$\begin{aligned} multiply_k(c_1, c_2) = {} & (c_1 \sqcap_k sparse \sqcap_k arith_k(c_2)) \sqcup_k (c_2 \sqcap_k sparse \sqcap_k arith_k(c_1)) \sqcup_k \\ & \big((eq(c_1, double) \sqcup_k eq(c_1, char)) \sqcap_k \\ & (eq(c_2, double) \sqcup_k eq(c_2, char)) \sqcap_k double\big) \end{aligned} \tag{4.59}$$

$$\begin{aligned} rdivide_k(c_1, c_2) &= ((c_1 \sqcap_k double) \sqcup_k (c_1 \sqcap_k sparse) \sqcup_k \\ &\quad (eq(c_1, char) \sqcap_k double)) \sqcap_k arith_k(c_2) && (4.60) \\ ldivide_k(c_1, c_2) &= ((c_2 \sqcap_k double) \sqcup_k (c_2 \sqcap_k sparse) \sqcup_k \\ &\quad (eq(c_2, char) \sqcap_k double)) \sqcap_k arith_k(c_1) && (4.61) \end{aligned}$$

The types of the matrix operators, *, / and \, are more complicated than the array binary operators. If the operator is used as a pure matrix operator then the result is different to when one of the operands is a scalar and an array operation takes place. Since this is dependent on knowing the dimensions of the operands as well as their types, the full type equation is deferred to later. Assuming the true matrix operation semantics are being used, then matrix multiplication and left and right division have an equivalent class equation to addition.

Table 4.7 shows the classes produced by logical operators such as equals, greater than and logical and. These operators are all symmetric and produce a valid output for all but the cell array, struct and function handle classes. This is modelled using the function,

$lbinop_k : \mathbf{K} \times \mathbf{K} \rightarrow \mathbf{K}$.

$$\mathbf{K}_u = \{single, int8, int16, int32, uint8, uint16, uint32\} \tag{4.62}$$

$$\mathbf{K}_{inv} = \{cell, struct, function, \bot_k\} \tag{4.63}$$

$$lbinop_k(c_1, c_2) = \begin{cases} \bot_k & \text{if } c_1 \in \mathbf{K}_{inv} \vee c_2 \in \mathbf{K}_{inv} \\ uint8 & \text{if } c_1 \in \mathbf{K}_u \vee c_2 \in \mathbf{K}_u \\ \top_k & \text{if } c_1 = \top_k \vee c_2 = \top_k \\ sparse & \text{if } c_1 = sparse \vee c_2 = sparse \\ double & \text{otherwise} \end{cases} \tag{4.64}$$

4.3.6 Trait Equations For Operators

All the operators in MATLAB can accept both logical or complex operands. Logical arrays only differ in behaviour when used as indices and so are equivalent to non-logical arrays in this context. The main problem with arithmetic operators comes from complex arrays. It cannot be determined statically whether the result of adding two complex arrays will be complex as well. In addition, complex arrays with no imaginary component can be created, meaning that a complex array added to a real array is not necessarily complex. The only certainty exists when both operands are real, in which case the result will also be real. The function, $binop_t : \mathbf{T} \times \mathbf{T} \rightarrow \mathbf{T}$ models this:

$$binop_t(\langle r_1, l_1 \rangle, \langle r_2, l_2 \rangle) = \begin{cases} \langle true, false \rangle & \text{if } r_1 = true \wedge r_2 = true \\ \langle \top_b, false \rangle & \text{otherwise} \end{cases} \tag{4.65}$$

For unary minus, we have $umin_t : \mathbf{T} \rightarrow \mathbf{T}$:

$$umin_t(\langle r, l \rangle) = \begin{cases} \langle true, false \rangle & \text{if } r = true \\ \langle \top_b, false \rangle & \text{otherwise} \end{cases} \tag{4.66}$$

The transpose operators retain the logical trait from the operand and so we have, $transp_t : \mathbf{T} \rightarrow \mathbf{T}$:

$$transp_t(\langle r, l \rangle) = \begin{cases} \langle true, l \rangle & \text{if } r = true \\ \langle \top_b, l \rangle & \text{otherwise} \end{cases} \tag{4.67}$$

4.3.7 Full Type Equations For Operators

In this section, we bring together the equations from the previous sections, to give full type equations encompassing both binding, shape, class and type trait information.

The full equation for unary minus, $umin : \mathbf{A} \rightarrow \mathbf{A}$, is a simple case. If the operand is static, then apply the operator to the concrete representation, otherwise an abstract representation is found:

$$
\begin{aligned}
umin(\langle b, v\rangle) &= \begin{cases} \langle static, interp\_u\_op(\texttt{umin}, v)\rangle & \text{if } b = static \\ \langle dynamic, umin_\alpha(\alpha(v))\rangle & \text{otherwise} \end{cases} && (4.68)\\
umin_\alpha(\langle\langle k, t\rangle, s, \delta\rangle) &= \langle\langle umin_k(k), umin_t(t)\rangle, s, true\rangle && (4.69)
\end{aligned}
$$

The full equation for transpose, *transp* : $\mathbf{A} \rightarrow \mathbf{A}$, is as follows:

$$
\begin{aligned}
transp(\langle b, v\rangle) &= \begin{cases} \langle static, interp\_u\_op(\texttt{transpose}, v)\rangle & \text{if } b = static \\ \langle dynamic, transp_\alpha(\alpha(v))\rangle & \text{otherwise} \end{cases} && (4.70)\\
transp_\alpha(\langle\langle k, t\rangle, s, \delta\rangle) &= \langle\langle transp_k(k), transp_t(t)\rangle, transp_s(s), true\rangle && (4.71)
\end{aligned}
$$

Addition has the function, *plus* : $\mathbf{A} \times \mathbf{A} \rightarrow \mathbf{A}$.

$$
plus(\langle b_1, v_1\rangle, \langle b_2, v_2\rangle)) = \begin{cases} \langle static, interp\_b\_op(\texttt{plus}, v_1, v_2)\rangle & \text{if } b_1 = static \wedge b_2 = static \\ \langle dynamic, plus_\alpha(\alpha(v_1), \alpha(v_2))\rangle & \text{otherwise} \end{cases} \quad (4.72)
$$

$$
plus_\alpha(\langle k_1, t_1\rangle, s_1, \delta_1\rangle, \langle k_1, t_1\rangle, s_1, \delta_1\rangle) = \langle\langle binop_k(k_1, k_2), binop_t(t_1, t_2)\rangle, binop_s(s_1, s_2), true\rangle \quad (4.73)
$$

All arithmetic array binary operators, like subtract, multiply and power, have very similar equations, differing only in which concrete function is called. Logical *and* is an example of a logical binary operator like equals and greater than.

$$
and(\langle b_1, v_1\rangle, \langle b_2, v_2\rangle)) = \begin{cases} \langle static, interp\_b\_op(\text{and}, v_1, v_2)\rangle & \text{if } b_1 = static \wedge b_2 = static \\ \langle dynamic, and_\alpha(\alpha(v_1), \alpha(v_2))\rangle & \text{otherwise} \end{cases} \quad (4.74)
$$

$$
and_\alpha(\langle\langle k_1, t_1\rangle, s_1, \delta_1\rangle, \langle\langle k_2, t_2\rangle, s_2, \delta_2\rangle) = \langle\langle lbinop_k(k_1, k_2), \langle true, true\rangle\rangle, binop_s(s_1, s_2), true\rangle \quad (4.75)
$$

The full equations for matrix multiply, divide and power are very similar to the equations already given here but with the functions changes appropriately.

4.4 Concrete and Symbolic Execution

The main work of MPE is performed by two interpreters: a concrete interpreter, which takes fully static data and produces a fully static result and a symbolic interpreter, which takes possibly static or dynamic data and produces a static or dynamic result.

The symbolic interpreter examines expressions and determines from their sub-terms, whether the result will be static or dynamic, while it symbolically interprets statements. If an expression contains only fully static terms, then the result will generally be fully static. In fact the symbolic interpreter invokes, for static expressions, the same library calls as the concrete interpreter. If an expression is not sufficiently static, the result will be dynamic, but an attempt is made to infer some information about the result even if the value cannot be found. For each operator, an equation like the ones given in the previous section is used to find the abstract interpretation of the result, given its operands.

The concrete interpreter is reasonably fast (hence why it is used instead of the symbolic interpreter where possible), but it is not as fast as MATLAB. While it uses MATLAB library calls to calculate its results, it uses a method now deprecated by Mathworks. This method puts the onus on us to keep track of deleting temporary arrays once they have been used, which is both slower and more complicated to code, unlike the method favoured by Mathworks which automatically deletes temporary arrays used as parameters to library functions. But unfortunately the new technique causes crashes in some circumstances and although this bug is noted on the MATLAB website, it has not been fixed. Using the newer method of managing arrays greatly speeds up interpretation, but this will have to wait until the bugs are eliminated. An alternative method would be to replace the concrete interpreter with one which directly invokes the MATLAB interpreter allowing any advances in the MATLAB engine to speed up our partial evaluator. This may be problematic given the amount of data that would have to be passed between the MATLAB interpreter and the partial evaluator.

In MPE, symbolic execution is always a forwards iterative process. Information inferred about function parameters or operator operands is not used to update the parameters or operands themselves. E.g.

```
%# b size [2 UNKNOWN]
c = b * b;
```

In the above example b is declared to have 2 rows but an unknown number of columns. In the following statement b is multiplied by itself. From (4.23), the shape of c is determined to be $\left\langle \langle 2,2\rangle, \left\langle \langle 2,2\rangle, \underline{\infty}\right\rangle\right\rangle$. Because information was only passed forwards, it was not inferred that since the inner dimensions must match, that b also must have 2 columns for the expression to be valid, in which case c would also have 2 columns. To do backward shape inferencing would require a far more complicated infrastructure than is present in MPE.

In Section 6, we will describe how this symbolic execution framework is actually used. Section 6.1 describes how each statement is partially evaluated and it refers heavily to the abstract domains defined in this chapter. Section 6.1.5 contains a description of the fixed point algorithm used for computing the least upper bound of the entry point to loops. This is later revised in Section 6.3 to work with loops containing `break` and `continue` statements.

4.5 Precision

The abstract domains contained in this chapter will later be used to approximate the types of concrete values. This approximation clearly loses all information about actual values when a concrete value is abstracted, but this is not unexpected. Possible shortcomings however can arise when combining type information following conditional blocks.

If the class of a variable on exit from each branch of a conditional is fixed but different, all class information will be discarded. This insures that the class lattice has only $n + 2$ elements, where n is the possible number of classes, but is perhaps unnecessarily restrictive. Knowing that an array could be either of class *double* or *char* would allow us to infer that arithmetic operations on it were allowed. If the array was either a cell array or struct, then we could immediately infer that any arithmetic operation would always fail. This could be modelled by a bit-vector of size n, which flagged every class that might be possible, leading to a lattice with 2^n elements. Clearly such an approach would become expensive for a system with an unbounded number of classes.

However the main way of extracting further static type information from the class of a dynamic array is through the `class` built-in function, which returns the class as a string. Clearly if this string was compared with certain other static values then it might be possible to remove the comparison. However since no information about values is currently retained,

4.6 Summary

This chapter has seen the creation of a full type system for MATLAB, capturing information about arrays that will later be used to make specialisation decisions. Since MATLAB is a dynamically typed language, to provide input validation, functions must check their parameters types at run-time. While in C, passing an integer (instead of a string) as the first argument to the `printf` function will be spotted at compile time, this will not happen in MATLAB. If we can maintain accurate information about the types of dynamic variables, we should be able to remove these checks without producing an unreliable program.

This chapter contains a formalisation of the MATLAB type system, which is we believe more complete than any other work on MATLAB, particularly since it supports classes.

Details are given for deriving types for all MATLAB operators, including matrix multiplication and array addition. We have described how to infer the shapes for several key built-in functions as well as the many forms of array indexing.

In the following chapter, we will lay the foundations for a partial evaluator by describing details related to loading and parsing MATLAB programs, particularly with regard to differentiating between function calls and variables.

Chapter 5

Handling full MATLAB programs

We have produced an online partial evaluator because many of the problems that offline partial evaluation helps alleviate are less likely to occur in MATLAB. With online partial evaluation, we can easily handle function handles. In addition we wish to perform optimisations based on the characteristics of arrays such as shape and type. For example, while the exact value of a matrix may be unknown, its shape and whether it is real or complex could well be known. In this case built-in functions which try to determine these properties can be replaced by the actual values, which might lead to speed-ups due to loop unrolling and the removal of conditionals. This is less likely to be possible using offline partial evaluation since the static/dynamic divide must be declared before specialisation and must be conservative in cases where it could go either way.

While offline partial evaluators are generally more efficient and predictable, we have different motives in our use of MATLAB. Theoretical results such as self-application are mostly irrelevant to us, as MATLAB would be an unlikely choice for writing interpreters. The language does not use recursion extensively since it is imperative, meaning termination issues that can cause problems in online partial evaluators of heavily recursive languages are unlikely to cause problems.

The complexity of the information that we wish to store about expressions means that using MATLAB to create the partial evaluator itself would be cumbersome. One reason is that MATLAB has no tools for parsing programs, such as *lex* and *yacc*. While this step could be performed elsewhere, MATLAB has many other deficiencies which makes string handling difficult and complex data structures complicated. The lack of *call-by-reference* semantics and sharing of data would also likely make the partial evaluator very inefficient.

Since MATLAB was not an option itself for the implementation language, we were restricted to either C, C++ or Fortran, since these are the languages for which Mathworks provided libraries. Our MATLAB partial evaluator (MPE) was written for GNU/Linux systems in C++, and should be portable to any system with a good C++ compiler and which has MATLAB support.

In this section we describe the steps taken to load in full MATLAB programs and parse them as MATLAB itself would do. While the parser is not perfect, we believe it to be current up to MATLAB 6.1[1] and able to handle the majority of MATLAB codes.

A *program*, as recognised by our system, consists of one or more m-files situated in one or more directories. Files are not explicitly included in a program but are implicitly included by calling the functions present within them.

MATLAB provides many functions in m-files, which can be called by user programs. While some of these are self-contained, others can call many other functions, themselves contained in m-files. This can greatly increase the time to load and parse a MATLAB program as well as the memory usage, especially since the presence of a function call in an m-file does not mean that it will definitely be called. Since partial evaluation will often determine that a function will not be called, we can use this property to avoid parsing unnecessary files. To this end, MPE only loads functions when they are called.

5.1 Overview

Below we give an overview of the stages of the partial evaluator.

1. Parse source file to be partially evaluated (Section 5.2) producing an Abstract Syntax Tree (AST).
2. Insert static values at beginning of AST (Section 5.3).
3. Convert placeholders in the AST (Section 5.4).
4. Partially evaluate the function obtaining a new AST (Chapter 6). This stage requires the partial evaluation of further functions. In each case the process is started from stage 1.
5. Post-process the new tree, removing dead code (Chapter 7).
6. When the original m-file has been completed, traverse new main function AST, marking any functions that are called.
7. Write out all called functions as MATLAB source code.

Each of these stages will be described in more detail in the following sections.

[1]We do not handle the explicit short-cut logical operators of MATLAB 6.5, `&&` and `||`.

MPE is invoked from the command line and produces a single m-file as output. This file contains all the specialised functions. Since all the functions are placed in a single file, there is a chance that two identically named functions could be produced and then one of them would be inaccessible. This could only happen if an additional function in an m-file (accessible only in that m-file) shared the same name with another function. This could be corrected by renaming conflicting functions, but has not proven to be a problem in any of our tests.

5.2 Lexical Analysis and Parsing

MATLAB was designed more to allow mathematicians to read it than for simple parsing. This leads to ambiguous constructs that are fairly simple for a human to understand as they can more easily make contextual judgements, but a lot harder for a lexical analyser. Problems occur because in matrices, spaces can be column delimiters or white space. Outside matrices new lines are treated as an end of command indicator, but inside they are treated as row separators. Fortunately `flex` [21] can be made stateful thus avoiding the need for a hand-written lexical analyser. Joisha et al describe how to parse MATLAB in [36], although since that publication many of the problems, to which they refer, have disappeared as MATLAB was made more "compiler-friendly". This section will not attempt a full description of parsing, but will highlight some areas where our implementation differs from [36].

Due to difficulties in disambiguating variables and function calls, identifiers that could represent either are stored initially as placeholders. In the following stage, these will be replaced with either variables or function calls in the case of identifiers not followed by parentheses or by subscripts or function calls for identifiers followed by parentheses and a list of indices or parameters.

Other problems arise because the MATLAB grammar is not fully documented. There are some (admittedly contrived and probably due to bugs) circumstances in which MATLAB interprets the input in seemingly unpredictable ways. This has improved in recent MATLAB versions as The Mathworks have themselves introduced compiling technology for MATLAB and MATLAB 6.5 has largely eliminated these problems.

There is one quirk of MATLAB functions, that we choose to ignore because dealing with it would require a more complicated lexical analyser for little benefit. The problem is due to MATLAB allowing the calling of functions without parentheses. E.g.

```
shading gouraud
clear a b c
```

These examples are equivalent to `shading('gouraud')` and `clear('a', 'b', 'c')`. Unfortunately when MATLAB sees a function not followed by parentheses it assumes the function will be of this form even if the function takes no parameters. So if `f` is a function then `f * 10` will be interpreted as `f('*', '10')`. Trying to use `lex` to handle this would be difficult, as it would need context to realise that `f` was a function and that everything following it should be interpreted as strings. Function calls are only parsed like this when they begin a line and not as part of an assignment or part of another statement. We call such statements directives and only partially consider them, noting that they traditionally cause side-effects and need to be retained in the residual program. MPE will not recognise `f * 10` correctly if `f` is a function but will instead create a binary expression, but we do not see this causing problems as such code would be unlikely except due to programmer error. When Joisha [36] parses these function calls, he requires that the function arguments be enclosed in quotes. We do not take this approach since this would cause many library functions to fail to parse (although this form is also acceptable to our parser).

The only notable problem that actually arose in the MATLAB libraries is the use of keywords after a function, e.g. `dbstop if error`. Since `if` is a keyword, this will cause a parse error and abort MPE. This can be avoided by wrapping parameters with inverted commas, e.g. `dbstop 'if' error`.

This stage is performed on m-files and produces an m-file object containing a representation of each of the functions found in that m-file. Functions are represented as a list of input parameters, output variables and an AST. Assignment and expression statements have a flag, which indicates whether they are terminated with semi-colons and therefore should be *silent*. Statements can be either expressions, assignments (normal assignments, multi-output assignments and delete assignments), `for` loops, `while` loops, `if` statements, `switch` statements, control-flow-change statements (`return`, `continue` and `break`), directives or annotations.

5.3 Handling command line definitions

To specify the values of parameters to the main function, the user will give a command line flag (`-d`) along with an assignment from some variable to a static value. These assignments are inserted at the start of the AST which will contain the residual program. The expression is parsed just like any other MATLAB code and so can be a matrix or any other valid MATLAB expression. Additionally a file, containing a list of assignments and annotations, can be specified using the `-a` flag.

The function declaration is then modified to remove any parameters that were specified at the command line. E.g.

```
function z = f(x,y)
...
```

After partial evaluation by `mpe -d x 1 f.m`, this becomes:

```
function z = f(x)
...
```

This approach was chosen purely due to its simplicity. By adding assignments no special code needs to be generated to ensure that the values are propagated through the program as this occurs in the normal course of partial evaluation. It also avoids the problems of indexed assignments which assume the variable is already in existence.

5.4 Converting placeholders

This stage is necessary because the distinction between variables and functions is not immediately determinable unlike in C. This stage also appears in most of the literature on MATLAB compilers including [58] and [2], where it is called *disambiguation*. Variables do not have to be declared but are created as required by assignments. Variables can also shadow the names of both built-in and ordinary functions. This means that an identifier could be used to indicate a function call at one point in a function and then later be used to access a variable if there is assignment to the variable in between. Variables can also be deleted using functions such as `clear`, which, if they shadow a function name, means that the function can again be called using that identifier. We choose to ignore the `clear` function as its results are quite unpredictable especially when used with dynamic input. As a result of this ambiguity when the parser encounters a symbol, it inserts a placeholder into the AST for later conversion.

Our criteria for determining whether a placeholder represents a function call or a variable access are similar to those of De Rose, although ours is more extensive than that described in [57], as MATLAB has evolved since it was written. An identifier that has been previously used as a variable will always be a variable. Otherwise a function with that name is sought in the following places with descending precedence[2]:

1. Built-in Functions
2. Functions in current file
3. M-files in current directory called `function-name.m`
4. M-files in a subdirectory, `private`, of the directory of the current m-file, called `function-name.m`
5. M-files in the path called `function-name.m`

[2]this order ignores `@class` directories which MPE does not support.

If no function is found, then the identifier is assumed to represent a variable allowing code like the (extremely contrived) following code to work:

```
for n = 1:10
  if n > 1
    a = [a n];
  else
    a = 1;
  end
end
```

A particularly confusing and yet possibly valid statement is `x(x) = x(x)`, where `x` is a function and has not been used previously as a variable. This problem is discussed by De Rose [57]; in particular he discusses how conditional statements can affect whether an identifier is seen as a variable or function. We take the same approach and assume identifiers which have no visibly defined function are variables as we cannot defer the decision unlike an interpreter.

If an appropriately named file is found for a previously unseen identifier, the placeholder is converted to a function call. The file is not however parsed until the call is partially evaluated (or abstractly interpreted). With many programs this prevents memory being wasted on functions which will never be called and can sometimes allow partial evaluation of programs which contain features we do not support.

The end result of this stage is one parse tree where all placeholders have been replaced with either variable, subscript or function call identifiers, for each function in all m-files in the system.

5.5 Summary

This chapter first described our motivation in using online partial evaluation. It then listed the stages that make up our partial evaluator and then described the early stages, by giving a brief description of the problems associated with parsing large MATLAB programs and them disambiguating variable and function symbols. This is not presented as unique research, since it is documented elsewhere, but is presented for completeness.

Having parsed an m-file and produced the Abstract Syntax Tree for it, it is now possible to partially evaluate it. In the next chapter, we will describe the core of a partial evaluator. This includes a description of how to handle every kind of statement as well as whole functions. It is this work which accounts for the bulk of the partial evaluation time.

Chapter 6

Partial Evaluator Core Functionality

In this chapter we describe the core of the partial evaluator and how the abstract domains and equations from Chapter 4 are used.

The initial input is the first function in the m-file specified on the command line along with a list of parameter values. Each value is from the set $\mathbf{A}$, defined in Definition 4.22, and so can be completely static, dynamic with some shape information, or perhaps undefined. Since the number of parameters and outputs can themselves be dynamic, the list of parameters is also supplemented by two additional parameters, $nargin, nargout \in \mathbf{R}$, whose values are ranges from Definition 4.11.

For non-*varargin* functions, which can only take a certain number of parameters, *nargin* is capped to this maximum ($nargin' = nargin \sqcap_r \langle 0, n \rangle$). The same is done to *nargout* for non-*varargout* functions. For *varargin* functions, the excess parameters are directly mapped into a cell array. If any of these parameters is dynamic, then so will the `varargin` variable. The size of this parameter list is now definitely finite even if *nargin* is unbounded.

The environment is a hash-table mapping symbols to values ($sym \rightarrow \mathbf{A}$). This environment has several extra fields: *nargin*, *nargout* and *current-mfile*. The *current-mfile* field is used when determining where to look for functions. This is not necessary for functions found by the converting placeholders stage from Chapter 5, but for calls to `feval` as will be described later.

We use hierarchical environments to reduce copying at points where execution branches. Each table has a parent field. When partial evaluation starts for a function, this field is `NULL`. On partially evaluating a conditional with two branches, two empty environments will be created, each with the current environment as their parent. If searching for a symbol fails in the current environment, the parent is searched recursively until there are no more parents or the symbol is found. Updates to the environment only affect the most recent environment so as not to affect the state of alternate branches. On completing both branches, the two environments will only contain symbols which were altered and so only these need to be examined.

The initial environment is then populated with the parameter symbols and their values and partial evaluation can now commence. The function is partially evaluated statement by statement, updating the environment as it goes.

6.1 Partially evaluating statements

For each of the statements discussed in Chapter 2, we will now describe how it is partially evaluated.

6.1.1 Expressions

In MATLAB, when an expression appears on a line on its own, it is evaluated and the answer displayed (assuming a semi-colon is omitted), as in a calculator, but more frequently expressions are found in other statements such as assignments. The rules for evaluating expressions as described here apply in all cases, although other statements use the expression values in different ways.

Binary expressions are stored in a binary tree. Our implementation performs a depth first traversal of the tree evaluating wherever possible. The nature of the tree has an effect on the amount of evaluation that is possible. For instance `2 + x + 4` and `x + 2 + 4` will not be reduced to `6 + x` because the tree stores them as $(2 + x) + 4$ and $(x + 2) + 4$. Either a strategy would have to be developed to propagate the addition across the inner binary expression or N-ary expressions with reordering could be used.

Any strategy that used expression reordering based on associativity or commutativity would have to be very careful about causing an adverse change in execution time. Menon and Pingali [49] discuss how we can use the semantic properties of operations in conjunction with type information to realise previously infeasible operations. For instance in evaluating $A * B * x$, if A and B are matrices and x is a vector, the time complexity is $O(n^2)$ evaluating from right to left, but $O(n^3)$ if we evaluate from left to right. It is also possible to introduce rounding errors by reordering expressions, as `1 + 1e-16 + 1e-16 - 1` is evaluated as exactly 0 by MATLAB, where as `1 + (1e-16 + 1e-16) - 1` produces $\sim 2 \times 10^{-16}$.

Other problems can arise from the dynamic typing of MATLAB. For instance it is not possible to assume that multiplication is commutative as matrix multiplication is not.

(Commutativity can be assumed if `.*` is used). $A * 0$ is not necessarily 0 but could be a matrix of 0s or a sparse empty matrix.

Each node in an expression has two pieces of information associated with it. Its *Value* ($v \in \mathbf{A}$) and the code which produces it. These two pieces of information are returned by the expression partial evaluator. In the case of static values ($v \in \mathbf{A}^\flat$), the code to reproduce it not actually created until the function is written out. This means that little time is wasted on the static values of sub-expressions which are immediately consumed by another static expression. Instead the code generated for static values, is just an instruction to *lift* the value when code is actually required. For dynamic values ($v \in \mathbf{A}^\sharp$), the code is immediately created. We assume the availability of functions which call the concrete interpreter on static parameters, like $apply\_op_\gamma : \mathrm{Op} \times \mathbf{M} \times \mathbf{M} \to \mathbf{M}$, which applies an operator concretely to two operands.

We will now describe how the various types of expressions are handled, starting with the binary expression $\mathrm{binop}(op, e_1, e_2)$, where $peval(e_1, E) = \langle e_1', v_1 \rangle$ and $peval(e_2, E) = \langle e_2', v_2 \rangle$, with environment E:

$$peval(\mathrm{binop}(op, e_1, e_2), E) = \begin{cases} \langle \langle static, apply\_op_\gamma(op, \gamma'(v_1), \gamma'(v_2)) \rangle, \mathrm{lift} \rangle \\ \quad \text{if } v_1, v_2 \in \mathbf{A}^\flat \\ \langle \langle dynamic, apply\_op_\alpha(op, \alpha(v_1), \alpha(v_2)) \rangle, \mathrm{binop}(op, e_1', e_2') \rangle \\ \quad \text{otherwise} \end{cases} \tag{6.1}$$

As described here, only fully static binary expressions produce a static result, even in the case of expressions like `x * 0`. Such transformations are ignored, as they are not often applicable and can also produce changes in the semantics of a program. Where the expression component is 'lift', this is a short hand for the expression obtained by lifting the value component. We assume the existence of a function $apply\_op_\alpha : \mathrm{Op} \times \mathbf{A} \times \mathbf{A} \to \mathbf{A}$, which selects the correct type determination function based on the operator.

For the unary expression, represented by $\mathrm{unop}(op, e)$, where $peval(e, E) = \langle e', v \rangle$, partial evaluation proceeds as follows:

$$peval(\mathrm{unop}(op, e), E) = \begin{cases} \langle \langle static, apply\_op_\gamma(op, \gamma'(v)) \rangle, \mathrm{lift} \rangle & \text{if } v \in \mathbf{A}^\flat \\ \langle \langle dynamic, apply\_op_\alpha(op, \alpha(v)) \rangle, \mathrm{unop}(op, e') \rangle & \text{otherwise} \end{cases} \tag{6.2}$$

A constant, represented by $\mathrm{const}(c)$ in the AST, with value, $c \in \mathbf{M}$ (Definition 4.21) is handled as follows:

$$peval(\mathrm{const}(c)) = \langle \langle static, c \rangle, \mathrm{lift} \rangle \tag{6.3}$$

Variables for which there is a static value in the environment are replaced with the static value:

$$peval(\text{var}(sym), E) = \begin{cases} \langle E(sym), \text{lift}\rangle & \text{if } E(sym) \in \mathbf{A}^{\flat} \\ \langle E(sym), \text{var}(sym)\rangle & \text{otherwise} \end{cases} \tag{6.4}$$

An expression Subscript$(sym, format, \langle e_1 \ldots e_n\rangle)$ is an indexed variable access, like `a{3}` or `v.field(2)`. The *format* describes where the indices are placed and takes the form `{?}` and `.field(?)` for the examples respectively. Subscripts are handled by first partially evaluating the indices. Since the indices are in the form of a comma-separated list, there is a chance that more or even fewer than n elements will be produced by partially evaluating the list. This is because one of the indices might be a cell array subscript itself. Such a subscript can expand to any number of values. This can happen anywhere that comma-separated lists are expected, e.g. function parameters.

The function, $peval\_list : \text{Exp}^* \times Env \times \text{Exp}^* \times \mathbf{A}^* \times \mathbf{R} \to \text{Exp}^* \times \mathbf{A}^* \times \mathbf{R}$, is used to partially evaluate a comma-separated list of expressions. It produces a new list of expressions, a list of *values* as well as a range for the size of the list. Where applicable, $\langle e''_s, v'_s, \langle l', u'\rangle\rangle = peval\_in\_list(e_1, E)$ and $n' = \langle l', u'\rangle$.

$$\begin{aligned} peval\_list(\langle e_1 \ldots e_n\rangle, E, e'_s, v_s, n) &= peval\_list(\langle e_2 \ldots e_n\rangle, E, e'_s.e''_s, v_s.v'_s, n + \langle l', l'\rangle) \\ peval\_list(\langle e_1 \ldots e_n\rangle, E, e'_s, v_s, n) &= peval\_list2(\langle e_2 \ldots e_n\rangle, E, e'_s.e''_s, v_s.v'_s, n + n') \\ peval\_list(\langle\rangle, E, e'_s, v_s, n) &= \langle e'_s, v_s, n\rangle \end{aligned} \tag{6.5}$$

The second part of *peval_list* above, matches the case where n' is not a constant, but a range of values. In this case there is no point continuing to build the list of values as it is not known where they will fit. Instead, the rest of the expression list is built up by *peval_list2*, which just passes through v_s. Anything which uses v_s, will also have n and will therefore know that this list is incomplete.

$$\begin{aligned} peval\_list2(\langle e_1 \ldots e_n\rangle, E, e'_s, v_s, n) &= peval\_list2(\langle e_2 \ldots e_n\rangle, E, e'_s.e''_s, v_s, n + n') \\ peval\_list2(\langle\rangle, E, e'_s, v_s, n) &= \langle e'_s, v_s, n\rangle \end{aligned} \tag{6.6}$$

Similar to *peval_list* is *peval_index_list*, which has an additional parameter, *format*, which it returns updated to take account of any expressions that were expanded. For instance in the case of `b(a{[1 2]})`, *format* = '`(?)`' for the indices of the subscript to `b`, but if `a` is static, it will be expanded and so *format'* = '`(?,?)`'. If `a` was not static, it would not have been expanded and *format'* would be the same as *format*.

Each of the individual expressions that make up the list are partially evaluated using the function $peval\_in\_list : \text{Exp} \times Env \to \text{Exp}^* \times \mathbf{A}^* \times \mathbf{R}$. If c is a cell subscript, such that $c = \text{subscript}(sym, format, \langle e_1 \ldots e_t\rangle)$, where $format \in$ CellSubscript (the set of all format strings, which represent cell accesses) then:

$$peval\_in\_list(c, E) = \langle\langle v'_1, \ldots v'_p\rangle, \langle e'_1, \ldots e'_q\rangle, n'\rangle \tag{6.7}$$

where the components are defined below, given that:

$$peval\_index\_list(\langle e_1, \ldots e_t\rangle, format, E) = \langle\langle e''_1, e''_r, \rangle, \langle v''_1, \ldots v''_s\rangle, n, format'\rangle$$

1. If $\forall i, v''_i \in \mathbf{A}^\flat \wedge low(n) = up(n) \wedge E(sym) \in \mathbf{A}^\flat$, the result is static, $p = q$ and using a function $cell\_subscript_\gamma : \mathbf{A}^\flat \times \text{Format} \times \mathbf{A}^{\flat *} \times \mathbb{N} \to \mathbf{A}^{\flat *}$:

$$\begin{aligned} \langle v'_1, \ldots v'_p\rangle &= cell\_subscript_\gamma(\gamma'(E(sym)), format', \langle\gamma'(v''_1), \ldots \gamma'(v''_s)\rangle, low(n)) \\ \langle e_1, \ldots e_p\rangle &= \langle\underbrace{\text{lift} \ldots \text{lift}}_{p}\rangle \\ n &= \langle p, p\rangle \end{aligned}$$

2. Otherwise the result will be dynamic and the subscript needs to be constructed.

$$n = \begin{cases} \prod_{1\ldots s} numel(shape(v''_i)) & \text{if } low(n) = up(n) \\ \underline{\infty} & \text{otherwise} \end{cases}$$

Since no information is known about the contents of a dynamic cell array, the list of values just consists of unspecified but defined values. The subscript expands to the product of the number of elements within all the indices. This is calculated using a combination of *numel* from (4.31) and the fact that if the number of elements, to which the indices expand is unknown, then we have no information about how many elements the outer subscript will expand to. Consequently we only store the maximum number that it definitely expands to and so $p = low(n)$:

$$\forall i, 1 \le i \le p, v'_i = \langle\top_T, \top_s, true\rangle$$

The subscript has not been expanded so $q = 1$ and:

$$e_1 = \text{Subscript}(sym, \langle e''_1, \ldots e''_r\rangle, format')$$

Returning to partially evaluating the elements in a list. Any expression that is not a cell subscript will evaluate to a list of size 1 and is handled by the following equation. (This equation uses . to concatenate arbitrary tuples).

$$peval\_in\_list(e, E) = \langle peval(e, E).\langle\underline{1}\rangle\rangle \tag{6.8}$$

Subscripts in non-list positions like operands to binary expressions, must evaluate to a single value or an error will occur. Given the definition of s as above:

$$peval(s, E) = \langle e_1, v_1 \rangle \tag{6.9}$$

where $peval\_in\_list(s, E) = \big\langle \langle e_1 \rangle, \langle v_1 \rangle, \underline{1} \big\rangle$.

In the following example, a is a 4-by-4-by-2 static real double array, b is 5, c is a dynamic 2-by-1 non-logical array and d is a dynamic cell array with unknown shape information.

```
a(b, b + c, d{c})
```

To partially evaluate this expression, first the indices to the subscript to a would be partially evaluated. The first index is b, so the E(b) is checked (6.4) and found to be static with a value of 5, so the result is $\big\langle \langle static, 5 \rangle, \text{lift} \big\rangle$. The second index is a binary expression, b + c. As before b is static, but c is not and so from (6.1), we get $\big\langle \langle dynamic, a_2 \rangle, \text{binop}(+, \text{const}(5), \text{var}(\text{c})) \big\rangle$, where (with $\underline{2} = \langle 2, 2 \rangle$)

$$a_2 = \Big\langle \langle \langle \top_b, false \rangle, \top_k \rangle, \langle \underline{2}, \langle \underline{2}, \underline{1} \rangle \rangle, true \Big\rangle$$

The final index is a cell array and so its index need to be partially evaluated. Doing so gives $\big\langle \langle dynamic, a_2 \rangle, \text{var}(\text{c}) \big\rangle$. Since $numel(shape(v_2'')) = numel(\langle \underline{2}, \langle \underline{2}, \underline{1} \rangle \rangle) = \underline{2}$, the subscript to d will expand to two elements, but as d is dynamic, we can infer no information about their types or values. Consequently we will get as final code: `a(5, 5 + c, d{c})`, when the inferred shape is the result of indexing with a scalar, a 2-by-1 non-logical array and two unknown elements. From this we can infer (using the shape equations for indexing from Chapter 4) that the shape of the final result has between 2 and 4 dimensions, the first two of which have sizes 1 and 2 respectively. The other 2 dimensions have unknown size. The type will be the same as a, which means it will also be a real double array.

Built-in functions that have static parameters can usually be executed directly via the MATLAB runtime libraries. There are however some built-in functions which cannot be executed directly as they require context or produce side-effects. Examples include `exist`, which can be used to determine the existence of variables as well as files and functions; I/O functions like `disp`, `fopen`, `fprintf` and `fclose`; graphing functions like `plot` and `plot3`; and timing functions like `cputime` and `clock`.

Some built-in functions like `exist` can be evaluated indirectly by examining the environment. It does, however, make little sense to evaluate timing functions while partially evaluating, as it is more likely it was intended that the timing actually take place when the residual program is executed. This is an area where offline partial evaluation gives the user explicit control of which functions to execute early and which to leave to the residual program.

No I/O operations are performed while specialising, but are always inserted into the residual program. This does limit the amount of static data that MPE can see as static data files are never read. It would not be too onerous to remove this limitation through explicit user annotations, if it was later deemed useful.

Built-in functions that cannot be evaluated directly, due to lack of static parameters or side-effects, are handled by the partial evaluator internally. For each built-in function we extract as much information as possible about the return values based on the input passed to it. In the case of functions like `size` and `ndims` which examine the shapes of matrices, we can sometimes fully evaluate them provided we have that information. If insufficient information is available to fully evaluate the function, we instead return an entry describing the shape and type of the returned value in so far as we can determine it.

When a non-built-in function call appears in an expression, there are two main ways of handling it. If all the parameters are static, the call can be fully evaluated and then the result is either embedded in the residual program or is used as part of some other static computation. Alternatively if the parameters are not entirely static, the function call needs to be specialised. Function specialisation is described in Section 6.4, but for now we will assume that the result is a new function name (constructed from the old name and a variant number) and a list of return values, $v_i \in \mathbf{A}$.

If it is a built-in function, a list of *special* functions is checked. If the function is on that list, it will never be concretely interpreted even if the parameters are completely static. Functions which cannot be directly executed either because they are not completely static or because they are on the *special* function list, are instead handled internally by the partial evaluator. In either case the result will be a list of return values, $v_i \in \mathbf{A}$. (For side-effecting built-in functions, which are always *special*, v_i will never contain static elements, i.e. $\forall i, v_i \in \mathbf{A}^\sharp$).

The equations for built-in functions are not given here, but essentially most require fully static parameters to produce a static result, otherwise they will produce a dynamic result. The exceptions to this are the shape accessor functions like `size`, `length`, `isempty` and `numel`; and type accessor functions `isreal`, `islogical` and `class`. These functions can examine the components of a dynamic type and extract a static result if it exists, which means that the function call is replaced by a constant in the residual program.

Given a non-built-in function call, $f = \text{Function}(sym, \langle e_1 \dots e_n \rangle)$, where

$$peval\_list(\langle e_1, \dots e_n \rangle, E) = \langle \langle v_1', \dots v_p \rangle, \langle e_1', \dots e_q' \rangle, n \rangle$$

then the equation for partially evaluating the function is:

$$peval(f, E) = \begin{cases} \langle \text{Function}(sym', \langle e_1' \dots e_q' \rangle), v_1'' \rangle & \text{if } v_1'' \in \mathbf{A}^\sharp \\ \langle \text{lift}, v_1'' \rangle & \text{otherwise} \end{cases} \tag{6.10}$$

where v_1'' is the result of evaluating that function with the parameters $\langle v_1', \ldots v_p' \rangle$ and *nargin*, n. This evaluation procedure will not return $v_1'' \in \mathbf{A}^{\flat}$, unless the function *sym* has no possible side-effect for that call. If the call does not return a static result then a new function *sym'* will be produced. The function call will be replaced by one that calls the new specialised function.

Currently if a call to `feval` is passed a dynamic function handle or string, no attempt is made to infer information about the return value of the function or even to specialise it. This means opportunities are lost, but since we do not attempt to find all the possible function handles that could reach a `feval` call, there is no way to infer anything. Ruf and Weise [60] use control flow analysis in their partial evaluator, FUSE, to find lambda expressions that can reach a dynamic application. A generalised argument list is found which fits each of these lambda expressions by examining every place where the expression could be applied. This is all done in a pass after the initial specialisation is complete and requires a specialisation method, which allows for code to be iteratively improved. All of this is far more important for FUSE, because it partially evaluates Scheme, where recursion and higher-order functions are much more common.

If the call to `feval` uses a static function, then the call to `feval` is replaced with a direct call to the function. This transformed function call is then partially evaluated as normal.

Arrays containing only constants are constructed at parse time to avoid the cost of re-construction, every time an expression is encountered while partially evaluating. Arrays that cannot be fully constructed at parse time, such as ones including expressions, are built by applying the appropriate *horzcat* and *vertcat* functions repeatedly. For fully static values, this will produce a fully static array. If any of the elements that make up an array are dynamic, the whole array becomes dynamic and so the shape determining functions are used. This allows expressions such as `[2, x; 4, y]` to be inferred as a 2-by-2 matrix when `x` or `y` is known to have only one column.

6.1.2 Assignments

Simple assignments discard the old value of the target variable and are relatively easy to handle. In a single value assignment, we partially evaluate the right hand side as described above, and create a new assignment with this new expression. The value, ($v \in \mathbf{A}$) of the expression is then stored in the environment for non-indexed assignments ($E = E(sym \rightarrow v)$). For indexed assignments, the value in the environment is updated if it is static and the indices are static with a new static value. If the indices or the right hand side are not static then the environment is updated with a value inferred from shape analysis.

With multiple output assignments, the right hand side can only be a function. In which case it is evaluated as for function call expressions and the multiple outputs are distributed to assigned variables. Ideally the outputs will be completely static and so the statement can be replaced with one new assignment with a static value for each output.

```
[a,b] = size(c);
```

If the shape of c is known then assignments to a and b are substituted. E.g. If c is a matrix with 4 rows and 2 columns, this will become:

```
a = 4;
b = 2;
```

But if only the number of columns is known then we would have to use a function call. E.g.

```
a = size(c,1);
b = 2;
```

This is more expensive than the original as there are two assignments and a function call, but post-processing could completely eliminate the assignment to b. It is easy to transform the `size` function into multiple assignments as above, but most built-in function calls are not so simple and so we do not perform this kind of transformation when there is a mix of dynamic and static return values.

For functions provided in m-files, moving the static assignment from the called function to the calling function provides more opportunities for dead code removal by dead code removal. E.g.

```
function y = f(x)
[a, b] = g(x, 6);
y = a * b;

function [a, b] = g(x, y)
a = x * 2;
b = y * y;
```

If static outputs are not removed from the function call then the following code will be produced:

```
function y = f__1(x)
[a, b] = g__1(x);
y = a * 36;

function [a, b] = g__1(x)
a = x * 2;
b = 36;
```

The value of b is known and so it can be propagated later in the function meaning that b is never required in the residual code, and any dead code elimination system which works locally on a function will not be able to remove any code from this example. If the static output is removed then the following code is produced.

```
function y = f__1(x)
a = g__1(x);
b = 36;
y = a * 36;

function a = g__1(x)
a = x * 2;
b = 36;
```

While this code is not really any shorter, the redundant assignments to b in f__1 and g__1 can easily be removed in post-processing producing a shorter faster program (and in fact inlining will later be shown capable of completely removing the function g__1).

6.1.3 if statements

If the value of the condition expression is static ($v \in \mathbf{A}^\flat$), the conditional statement is removed and replaced with an appropriate set of commands. To do this the condition expression is partially evaluated and if found to be static and non-zero, the first set of commands are partially evaluated and inserted into the residual code directly. If the condition expression is static and evaluates to zero and there is an else clause, those commands are partially evaluated; otherwise the if statement is removed and no code is inserted in the residual function.

If the condition expression is dynamic, then both sets of commands need to be partially evaluated with the same initial conditions and the resulting environments merged. This means that if both sets of commands set a variable to the same value, it will have that value after the loop. If the value is different, but the type or size of the array are the same, then this information is retained instead. For instance:

```
a = 5;
if a > x
    a = a + 1;
else
    a = 0
end
```

In the above code, if x is static and has the value 1, then the condition expression is static as a is static as well and so the conditional is replaced with the following code:

```
a = 5;
a = 6;
```

If x were unknown then the following code would be produced instead:

```
a = 5;
if 5 > x
    a = 6;
```

```
else
    a = 0
end
```

The environment would then be updated with the a equal to a dynamic real non-logical double scalar.

6.1.4 switch statements

As with `if` statements, if it is possible to predetermine the switch value along with the case values, the `switch` statement can be removed completely and replaced with the appropriate statements. Because the case values do not need to be constants and the cases are checked in order, it is important not to remove a `switch` statement when a case matches if there are one or more dynamic cases before it. In this case, the preceding cases need to be left in place while the matching case is replaced by an `otherwise` clause, thus removing the need for a comparison. Cases that can never match can be removed thus eliminating redundant comparisons. If as the result of eliminating comparisons only one comparison would ever be carried out, then it may be better to substitute an `if` statement as they are faster to execute.

One proviso here is that an error would be thrown if the switch value was neither a scalar or a string. By replacing the `switch` statement with an `if` statement, the program semantics change slightly, unless the switch value can be inferred to be a scalar or string. In addition, the equality operator == requires that both of its operands have the same shape for this optimisation to produce equivalent code. The function, `isequal`, would be a valid substitute but it brings with it the overhead of a function call.

As with `if` statements, if the exact control flow cannot be determined, each branch of the `switch` statement needs to be partially evaluated in parallel and the resulting environments then merged. Clearly branches that will never be reached can be skipped.

In the following example, there are several possible specialisation opportunities:

```
switch a
    case 'foo'
        disp('Monday')
    case 'foobar'
        disp('Friday')
    otherwise
        disp('Another time')
end
```

If a is completely static and has the value `'foobar'` then the code produced is:

```
disp('Friday')
```

On the other hand if a is dynamic, but it is known to have a length of less than 4, then the following would be produced:

```
switch a
    case 'foo'
        disp('Monday')
    otherwise
        disp('Another time')
end
```

If the length was exactly 3, then this would be optimised to the following:

```
if a == 'foo'
    disp('Monday')
else
    disp('Another time')
end
```

6.1.5 `for` loops

The number of iterations for a `for` loop is fixed from the moment that loop execution commences (unlike in languages like C where the loop constraints are re-evaluated after every iteration) and so determining whether to unroll requires no data flow analysis on the loop body. If the loop constraints are static then the loop will be automatically unrolled unless there is an annotation prohibiting it (as described later).

To unroll a loop, the body of the loop is partially evaluated once for each iteration. For each iteration, the loop variable is set to the appropriate static value, although generally an assignment will not be generated for this. The exception to this, is if the loop contains an assignment to the loop variable. In this case, omitting the assignment at the start of the iteration could produce invalid code if the assignment inside the loop is under dynamic control.

At the end of the loop, an assignment will be required to ensure the final value of the loop variable is available to the rest of the program following the loop. This assignment is only required if no assignments to the loop variable were inserted at the start of each iteration. Inserting assignments for the loop variable will often produce redundant code, but in these cases post-processing will remove it at a later stage.

When the loop constraints would mean that the body of the loop is never executed, (e.g. `a:b`, where $b < a$), then this is a special case of unrolling with no iterations in which case the entire loop is skipped. MATLAB does reset the loop variable to `[]` and so an assignment needs to be inserted for this, which will in most cases be removed by post-processing.

When the loop constraints are dynamic, the loop is retained. More complicated loop unrolling could be performed, including partial unrolling but for simplicity this is ignored here. Care now needs to be taken with variables inside the loop. Naively partially evaluating the loop, using the state before the loop was reached, would result in a loop that might only execute correctly for the first iteration. Instead information about the state after each iteration needs to be combined with the initial state.

This is achieved using iterative data-flow analysis. The body of the loop is iterated over using our abstract interpretation method which computes, for each variable, either an abstract value or a concrete value ($x \in \mathbf{A}$) storing the result in the environment. Algorithm 1 requires an initial environment, the list of commands comprising the loop body, a flag indicating whether we know if the loop always iterates at least once as well the identifier for the loop variable along with the abstract description of its value, ($a \in \mathbf{A}^\sharp$). It gives us two environments, one with which to partially evaluate the loop and another giving the state after the loop has been executed. The following algorithm assumes the existence of a function f_{run} which takes an environment and a list of commands and returns the environment that would be obtained by running those commands.

Algorithm 1 Calculate the fixpoint state for the loop

```
Require: $e_{\text{orig}} \in (\text{Ident} \rightarrow \text{Value})$, $comms \in \text{Block}$, $always\_iterates \in \text{Boolean}$,
    $loop\_var\_id \in \text{Ident}$, $loop\_var\_val \in \text{Value}$
Ensure: $e_{\text{input}}, e_{\text{final}} \in (\text{Ident} \rightarrow \text{Value})$
  $e \leftarrow e_{\text{orig}}$
  repeat
    $e_{\text{input}} \leftarrow e[loop\_var\_id \rightarrow loop\_var\_val]$
    $e_{\text{output}} \leftarrow f_{\text{run}}(comms, e_{\text{input}})$
    $e \leftarrow e_{\text{output}}[loop\_var\_id \rightarrow loop\_var\_val] \sqcup e_{\text{input}}$
  until $e = e_{\text{input}}$
  if $always\_iterates$ then
    $e_{\text{final}} \leftarrow e_{\text{output}}$
  else
    $e_{\text{final}} \leftarrow e_{\text{output}} \sqcup e_{\text{orig}}[loop\_var\_id \rightarrow [\,]]$
  end if
```

If it cannot be determined that the loop will ever execute even once, the final state environment of the loop is merged with the environment that would be obtained if the loop never executed. In some cases this could result in a lot of information being lost, and so it is important to examine the loop constraints first in case this merge can be skipped. For example:

```
a = 0;
for n = 1:5
    a = a + n ^ b;
end
```

This loop can be fully unrolled, resulting in a sequence of assignments. (Note that `1 ^ b` is not identified as equivalent to 1, as this does not hold true if `b` is not a scalar).

```
a = 0;
a = 0 + 1 ^ b;
a = a + 2 ^ b;
a = a + 3 ^ b;
a = a + 4 ^ b;
```

```
a = a + 5 ^ b;
n = 5;
```

In the following example, the loop cannot be unrolled when x is unknown.

```
a = 0;
b = 2;
for n = 1:x
    a = a + n ^ b;
end
```

In fact the only optimisation possible is the propagation of the value of b which can be determined not to change, when the least upper bound is found.

```
a = 0;
b = 2;
for n = 1:x
    a = a + n ^ 2;
end
```

The following is an example of code where the loop assignments must be generated to insure equivalent behaviour.

```
a = 0;
for n=1:4
    n(2) = x;
    a = a + n;
end
```

The assignments to n must be written out as there is an indexed assignment to n inside the loop.

```
a = 0;
n = 1;
n(2) = x;
a = 0 + n;
n = 2;
n(2) = x;
a = a + n;
n = 3;
n(2) = x;
a = a + n;
n = 4;
n(2) = x;
a = a + n;
```

While we have not implemented such a transformation, it may be possible to then convert the two assignments to n for each iteration above to a single one of the form: `n = [1 x]`.

There is never a risk that unrolling `for` loops will causes the partial evaluator to loop infinitely as the maximum number of iterations is always bounded. These loops can terminate earlier if `break` or `return` statements are present.

6.1.6 `while` loops

Unlike `for` loops, `while` loops cannot be so easily determined as unrollable. There are two approaches to unrolling such loops: one method is optimistic and unrolls the loop until the loop condition indicates that it has terminated or the loop condition becomes dynamic. The second method uses a binding time analysis (BTA) to determine if the loop condition will be static on every iteration.

Using the optimistic approach, there are no problems if the loop is found to terminate without the loop conditions becoming dynamic. However if the loop conditions become dynamic then unrolling cannot continue and the loop must be inserted into the residual code. The body of the loop and the loop condition are partially evaluated with respect to the least upper bound of the state after any unrolling that was successful.

Using this method, the following loop has the initial iteration unrolled, but then the loop must be inserted as y becomes dynamic.

```
%# x size [1 10]
y = 0;
a = 0;
while y < length(x)
    a = a + x(y);
    y = y + a;
end
```

Using an optimistic approach to unrolling, this becomes:

```
%# x size [1 10]
y = 1;
a = 0;
a = 0 + x(1);
y = 1 + a;
while y < 10
    a = a + x(y);
    y = y + a;
end
```

One way of performing the BTA, would be to abstractly interpret the loop to see if it terminated or if the loop condition became dynamic. If it terminates during abstract interpretation, then the loop is static and can be unrolled. The problem with this approach is that involves doing most of the work of unrolling the loop without generating any code. Once it has been determined that the loop can be unrolled, much of this work must be repeated. If the loop cannot be unrolled then a fixpoint iteration to find the least upper bound would also still be required in the same way as after the optimistic unrolling.

Another approach to determining whether to unroll is to find a conservative approximation of the *staticness* of the loop condition using a simple abstract interpretation model for which variables have three states, *dynamic*, *static* and *undef*. Our more complicated environment maps directly to this without difficulty. We then perform a fixpoint iteration on the loop maintaining a table of bindings for each variable. The reason for the *undef* binding is that the binding table is nested in the same way as the main symbol table. If a variable is *undef* in one table, its parents are checked for a non-*undef* binding. If no *static/dynamic* value can be found in any ancestor table then *undef* is returned by the lookup function. This fixpoint iteration will be considerably faster than the one used to find the least upper bound of the state as no computation is performed.

For each expression we calculate the binding as follows:

- Constants are static.
- Variables are static if they are not dynamic in the environment.
- Binary expressions are static if both operands are static.
- Unary expressions are static if both the operand is static.
- Function calls are static if all the parameters are static, the function exists and does not have side-effects.
- Array accesses are static if the variable is not dynamic in the environment and the indices are all static.

For assignments, the binding is stored in the symbol table. In the case of assignments to subscripts, a static binding is only recorded if the existing binding is static or undef and if the indices are also static.

For conditional statements, if the condition is static, each branch is determined in parallel and the results merged using nested tables. If the condition is dynamic then any assignments appearing inside it mark their variables as dynamic. Static loops are determined by performing a nested fixpoint iteration. Dynamic loops require that all assignments inside the loop mark their variables as dynamic.

This approach can do nothing with the previous example, as the simple approximation would make `x` dynamic and therefore `length(x)` would be dynamic. The code could be rewritten to avoid this by calculating the value of `length(x)` before the loop and storing it in a variable which would then be static for the duration of the loop. This would be both a *binding-time improvement* and an optimisation to the original code since the length of `x` does not change. The advantage of performing a simple BTA step first, is that it almost always guarantees that the loop can be fully unrolled and it does this fairly quickly.

Before attempting the BTA, the loop condition is abstractly interpreted once. If it is static and *false*, then the loop can be immediately deleted. Otherwise the BTA is performed and if it is determined that the loop condition is static it is unrolled by partially evaluating the body of the loop for each iteration. At the end of each iteration the loop condition is checked and if still *true*, the unrolling is continued. If, at the end of an iteration, the loop condition has become dynamic then the partial evaluator aborts (or alternatively the loop could be inserted at this point). Reasons why this might occur are discussed in 6.1.7. Another way of dealing with the loop condition becoming dynamic during unrolling is to insert the original loop at that point as with the other method of unrolling. In practice, very few codes produced different results, presumably because loops which depend on the shapes of parameters are `for` loops which can already be unrolled.

If the loop cannot be unrolled, a `while` loop is then inserted with the loop condition and body partially evaluated with respect to the least upper bound for the loop starting state.

Algorithm 1 can be reused for finding the fixpoint for `while` loops by removing all the references to *loop_var_id* and *loop_var_val*.

Unrolling `while` loops can cause infinite looping. If the loop would never terminate when the program was run normally, then the loop will be unrolled infinitely. One problem is that if the loop would not normally terminate but its execution only occurs when a dynamic condition is met, the partial evaluator will execute both paths. Hence the partial evaluator will always loop infinitely where the original program would only loop infinitely in some cases. Cases like this are not common but are not always contrived:

```
if x < y + w
    y = 10;
else
    while x ~= y
        x = x - 1;
        ...
    end
end
```

If `x` and `y` are static and `w` is dynamic then both branches of the `if` statement will be partially evaluated leading to the second loop being unrolled. If there is an assumption that `w` is always positive then normally the loop would never be executed if `x` was less than `y` and (assuming `x` and `y` differ by an integral amount) the code would always terminate. Since the partial evaluator always unrolls the loop, it will loop infinitely if `x` is less than `y`. In this case the code would need to be rewritten to avoid this problem, possibly by replacing ~= with >.

Listing 6.1 shows an implementation of a *sprintf*-like function in MATLAB using a `while` loop. This relies on two functions, `int2str` and `num2str`, which are provided as part of MATLAB in m-files.

```
function output = strprintf(format_string, varargin)
```

```
output = '';
count = 1;
pos = 1;
length = length(format_string);
while pos <= length
    if format_string(pos) == '%'
        code = format_string(pos + 1);
        switch code
            case 'd'
                output = [output int2str(varargin{count})];
                count = count + 1;
            case 'f'
                output = [output num2str(varargin{count})];
                count = count + 1;
            case 's'
                output = [output varargin{count}];
                count = count + 1;
            case '%'
                output = [output '%'];
        end
        pos = pos + 2;
    else
        output = [output format_string(pos)];
        pos = pos + 1;
    end
end
```

Listing 6.1: A simple implementation of sprintf

If we partially evaluate this function with the following format string:

```
'My name is %s, and I am %d years old'
```

We get the output shown in Listing 6.2. (In fact this is the result after the post-process phase of the next chapter is applied, but the original was considerably longer and could have obscured our result). This new program has no control flow as the `while` loop has been unrolled and the conditional statements have all been determined as static.

```
function output = strprintf__1(varargin)
output = ['My name is ' varargin{1}];
output = [output ','];
output = [output ' '];
output = [output 'a'];
output = [output 'n'];
output = [output 'd'];
output = [output ' '];
output = [output 'I'];
output = [output ' '];
output = [output 'a'];
output = [output 'm'];
output = [output ' '];
output = [output int2str__1(varargin{2})];
output = [output ' '];
output = [output 'y'];
output = [output 'e'];
output = [output 'a'];
output = [output 'r'];
output = [output 's'];
output = [output ' '];
output = [output 'o'];
output = [output 'l'];
output = [output 'd'];
```

LISTING 6.2: A test function for our `strprintf` function

6.1.7 Annotations

In addition to standard MATLAB language constructs, our tool recognises annotations which guide the partial evaluation. These always begin with `%#` and are ignored by MATLAB as comments. Annotations are just copied directly into the final code, but they do modify the symbol table.

There are two types of annotations: variable annotations and general annotations. Variable annotations describe variables, specifying the type, shape and definedness. This is useful for dealing with function parameters and also for preventing a variable identifier from being treated as if it was a function.

```
% Declare that x is a scalar
%# x size [1 1]
% Declare that x is complex
%# x complex
% Declare that y is undefined
%# y undefined
```

For convenience we define two extra annotations: `realdouble` indicates that the variable is of class *double* and is real and not logical. In addition, `realscalar` is the same as `realdouble` but also indicates that the variable is a scalar.

Function annotations describe how the function has been called. They specify the values returned by built-in functions like `nargin` and `nargout`, which return the number of parameters used to call the function and the number of return values requested respectively.

```
% Declare that the function is called with 2 parameters
%# nargin 2
% Declare that the function is called with 1 output
%# nargout 1
```

Currently MPE can loop infinitely on input containing loops which steadily widen shape values. Consider the following code:

```
function y = f(x)
y = 1;
for n=2:x
    y = [y n];
end
```

This function returns a vector of values from 1 to x with increment 1. The current implementation of MPE will iterate over the loop trying to find the shape of y. It can easily determine that it is a two dimensional matrix with 1 row. However each successive iteration will increase the number of columns by 1. Since iteration only ceases when stabilisation of shape information is achieved, the iteration will be infinite. To prevent this problem, the number of columns needs to be *widened*. If the number of columns is set to $\langle 1, \omega \rangle$, no further iteration would be required.

To this end, ideally we would develop heuristics to determine when there is a possibility of infinite iteration. For now the `widen` annotation can be used:

```
function y = f(x)
y = 1;
for n=2:x
    %# y widen 2
    y = [y n];
end
```

In this case, the second dimension of y has been widened, which means that iteration will not be infinite. Currently MPE limits the number of iterations for finding the least upper bound. When the limit is exceeded, it aborts, printing the variables that have not stabilised along with the information for the last two iterations allowing the user to determine which variables need to be widened.

```
Printing differences for function f
y ---------------
e1 : (ndims : 2) (size : 1 1-20) (type : real) (value : unknown)
e2 : (ndims : 2) (size : 1 1-21) (type : real) (value : unknown)
```

By default MPE will always unroll a `for` loop when the loop range is static. In addition if the loop condition of a `while` loop can be inferred to be static then it too will be unrolled. Sometimes this will cause an excessive increase in code size; this can be stopped by inserting a `nounroll` annotation in front of the loop. In addition the simple BTA may sometimes determine a loop to be unrollable when it is in fact not. This occurs when functions are called which are not static even when called with fully static parameters. Examples are `nargin` and `nargout` called with no parameters, which may or may not be dynamic. These could be treated as special cases by storing whether `nargin` and `nargout` are static or dynamic in the binding table. However other functions cannot be dealt with so easily without making the analysis much more complicated. In cases where this mistake is made, a `nounroll` annotation is required to prevent the partial evaluator aborting. Alternatively the optimistic loop unrolling approach can be used, which just inserts a loop after unrolling becomes no longer possible.

Loops which contain `break` or `continue` statements will never be unrolled by default. If the user knows that the loop conditions are static and that the control flow around these statements is also static, then an `unroll` annotation can be added before the loop to force unrolling. For `while` loops, the unrolling test is very conservative and will classify loops as not unrollable when the loop condition depends on dynamic data for which size or type information exists, when using the simple BTA method. If during unrolling, it is found that the conditions were not static then partial evaluation will abort.

Whenever MPE is generating code and it comes across a non-built-in function which cannot be executed directly, it will attempt to generate a specialised version of the function regardless of whether specialisation will actually be useful. It will also specialise MATLAB library calls for which MATLAB code can be found. This is sometimes undesirable and so the `preserve` annotation can be inserted to declare that a function should never be specialised. If the function has fully static data, it can still be directly executed and the call removed from the residual program.

Sometimes functions are called with fully static data, for which it is undesirable to execute directly at partial evaluation time. This might be because the function calls other functions which have side-effects, which would then not be present in the residual code. One approach would be to detect any functions which might have side-effects and bar them from being directly interpreted, but this might bar functions which only have side-effects with certain parameters which are not used in the code to be specialised. To allow specialisation here, we introduce the `nointerpret` annotation which forces the function to be abstractly interpreted even if it has fully static parameters. If a function is directly interpreted and during this interpretation calls a side-effecting built-in then partial evaluation will abort citing the function which called the built-in. In addition directly interpreting a function which calls a function which cannot be directly interpreted because of annotations will cause a similar error.

6.2 Control Flow Change statements

Control Flow Change statements cause the current control flow to jump to another point, such as the end of a function. We also include statements which cause the program to end such as errors and exit statements. For all the possible control flow statements described in the following subsections, abstract interpretation is not described explicitly. Since abstract interpretation has the same effect on the environment as when generating code, the only difference is that code is not generated.

6.2.1 `return` statements

Until now it has been assumed that the only way that a function can exit, is if control reaches the end of the function. However if control reaches a `return` statement then the function immediately terminates and returns the current values of the return variables. To handle this an extra environment is added, which stores the values to be returned from the function.

Its value is calculated by:

$$t_{result} = t_{end} \sqcup \bigsqcup_{i=1...n} t_i \tag{6.11}$$

Where t_{result} is the final environment for the function, t_{end} is the environment when control reaches the end of function (its value is $\perp$ if control never reaches the textual end of the function) and t_i are the environment values at each of the n return statements in the function (which are equal to $\perp$ if control never reaches them).

This is calculated in the partial evaluator by updating t_{result}, whenever the partial evaluator generates a `return` statement or when it reaches the end of the function. Since these statements prevent control flowing further in the function, they immediately cause the partial evaluator to stop working on the current block. If the current block is the top level block, then the partial evaluator would have finished with the whole function. If the statement is contained in a dynamic conditional then the other branches still need to be partially evaluated.

To handle this, the block partial evaluator returns a value, denoting how the block ended. If the block ended normally, it returns `CFC_NONE` and if it was ended by a `return`, it returns `CFC_RETURN`. The full set of values is given in Table 6.1. This set of values allows

CFC_NONE	1
CFC_RETURN	2
CFC_BREAK	4
CFC_CONTINUE	8
CFC_EXIT	16

TABLE 6.1: Exit values returned by the block partial evaluator

blocks to terminate in several other ways, which will be described later. Each of the values corresponds to a single bit being set, which means that we combine and then test the values using the binary operators, OR and AND. The set CFC contains all the possible combinations, where at least one flag is present, e.g. $\text{CFC} = \{x \in \mathbb{N} | 1 \leq x \leq 31\}$.

If one branch of an `if` statement ends with a `return`, and one ended normally, the exit value would be `CFC_NONE | CFC_RETURN`. The partial evaluator would then discard the environment generated by the `return` block when calculating the environment after the `if` statement. Consider the following code:

```
if a == 1
    b = 2;
    return
else
    b = 4;
end
```

In this example, both branches set the value of b. However, any code after this conditional will see a static value of 4 for b since the other branches contributes a value only to the final value of the function. If both branches had ended with `return`, the block containing the `if` statement would itself terminate.

6.2.2 `break` statements

In normal execution, if one of these statements appears outside of a loop, it behaves exactly like a `return` statement. If it appears inside a loop, it causes the loop to terminate and for control to continue from after the loop.

To handle this, we create a new environment when partially evaluating a loop which contains the state when the loop terminates. On reaching a `break` statement, this environment is updated in the same way as the environment for `return` statements. In addition the block partial evaluator returns CFC_BREAK, which is propagated back in the same manner as CFC_RETURN.

6.2.3 `continue` statements

These statements are only allowed to appear inside loop, or an error will be generated. If it appears inside a loop, it ends the current iteration of the loop and passed control to the start of the loop causing the loop condition to be re-evaluated (as well as updating the loop variant if necessary).

To handle this, we create a new environment when partially evaluating a loop which contains the state when the current loop iteration terminates. On reaching a `continue` statement, this environment is updated in the same way as the environment for `break` and `return` statements. In addition the block partial evaluator returns CFC_CONTINUE, which is propagated back in the same manner as CFC_RETURN.

6.2.4 Errors

When an error occurs during partial evaluation, the partial evaluator does not terminate, but continues generating code. This is because much of the control in a program will be dynamic, and so there will be dynamic conditionals which produce errors on one branch and not on the other.

```
function y = f(x, y)
if nargin < 2
    error('This function requires 2 inputs').
end
...
```

Since the partial evaluator does not know which branch will be taken it evaluates both, encountering the error on one branch. When this occurs, the partial evaluator recognises that one branch results in an error and so reproduces that error in the residual code. It then ignores that branch when generating the rest of the function. When the block partial evaluator encounters an error it returns `CFC_EXIT`.

In the example given here, after the `if` statement, execution can only continue if `nargin` is at least 2. If this is the case then y must be defined. The environment can then be updated with information. This is an example of positive context propagation, which will be discussed in Section 9.6.

Errors are also caught in expressions when operators are used with incompatible operands. In this case, the statement containing the error will be removed and replaced with a statement that reports the error that would be generated by executing the code as normal. Doing it this way, ensures that the residual code clearly shows that the input as given to the partial evaluator is responsible for the error (or that the program would produce the error given any input). For expressions with at least one dynamic operand, errors raised by trying to infer the shape of the resulting expression are also detected and flagged as above.

For errors that occur in functions being executed by the concrete interpreter, an exception is raised, which is passed through all the calling functions until it reaches one generating code or abstractly interpreting. In the former case, the exception in the form of an error statement is then inserted into the code instead of the statement which called the original function. This statement also includes the stack trace to help the programmer trace the error. If the called function came from code being abstractly interpreted, then an error is raised at that point, as if it occurred in the current block and the abstract interpretation continues as normal for this case.

6.3 Block Partial Evaluator

We now describe the block partial evaluator in more detail. The block partial evaluator takes a block of statements and partially evaluates them with respect to an environment, in the process producing an updated environment and a new block. A block is a group of consecutive statements, which appear at the same indentation level in a well formatted program. Compound statements like `if` statements and `for` loops also comprise the statements which appear within them.

In Listing 6.3, there are 4 blocks. Block 1 consists of the all the top level statements. Block 2 consists of the `error` statement. Block 3 contains the statements inside the `while` loop and block 4 is the assignment inside the second `if` statement. There are nominally 2 extra branches which represent the empty `else` branches of the `if` statements.

```
function z = pow(x, n)
```

```
if n < 0
    error('pow can only handle values of n >= 0');
end

z = 1;
while n > 0
    if mod(n, 2)
        z = z * x;
    end
    n = floor(n / 2);
    x = x * x;
end
```

LISTING 6.3: Power function

Each of these blocks is partially evaluated at one point. When block 1 is partially evaluated, it partially evaluates both branches of the first `if` statement. The first branch (block 2) contains the `error` function call which causes the block partial evaluator to recreate the command in the residual program and return CFC_EXIT, while the second branch is empty and so causes it to return CFC_NONE. At this point, the two exit values are combined as CFC_EXIT | CFC_NONE, but the CFC_EXIT component is discarded as it contributes no information if combined with a value which allows execution to continue. All other blocks complete with an exit value of CFC_NONE.

```
function y = product(x)

y = 1;
for n=1:length(x)
    if x(n) == 0
        y = 0;
        break
    else
        y = y * x(n);
    end
end
```

LISTING 6.4: Product function

In Listing 6.4, there is a simple function for computing the product of a vector. If any element in the vector is 0, then the function loop stops and 0 is returned.[1]

When partially evaluating the `if` statement, one branch terminates with a static value of 0 for y and an exit value of CFC_BREAK. The other branch ends with a dynamic scalar value for y and an exit value of CFC_NONE. The `break` branch updates the *loop end* environment, while the environment from the other branch is used to continue the partial evaluation.

The least upper bound is computed only using the second branch as taking the other branch would cease iteration. Once the least upper bound is found, the state after the

[1]This function would work just as well if `return` was used instead of `break`.

loop is the join of the normal loop end state and the break end state. Thus, the state after the loop has y as a dynamic scalar.

```
function y = nzproduct(x)

y = 1;
for n=1:length(x)
    if x(n) == 0
        continue
    end
    y = y * x(n);
end
```

Listing 6.5: Non-zero Product function

In Listing 6.5, there is a function which computes the product of a vector while skipping 0s.[2] This time the two branches of the `if` statement end with `CFC_CONTINUE` and `CFC_NONE`. The loop iteration end environment is updated with the environment after the `continue`, and when the iteration is completed, this environment is merged with the main environment in order to compute the least upper bound.

Algorithm 2 gives an updated version of Algorithm 1 for computing the fixpoint state for a `for` loop. This new algorithm takes both the current state, e_{orig}, and the current *return* state, e_{return}. It then produces new states, e_{output}, which is the state after the loop has executed; e_{input}, which is the state that should be used to partially evaluate the loop body (which will in turn reproduce e_{output}); and e_{return}, which is the accumulated state from all `return` statements so far. If while iterating, it is determined that no iteration of the loop will ever complete normally or due to a `continue` statement, the iteration is immediately stopped as the loop can never iterate more than once. In reality, if this determination is to be made, it will only happen on completing the first iteration. The *exit* value is also returned, with the `CFC_BREAK` or `CFC_CONTINUE` flags reset and converted to `CFC_NONE` if necessary.

6.4 Function Specialisation

A conservative approach to function specialisation would be *monovariant* and would create only one residual version of any function appearing in the source program. To do this, the least upper bound of all function signatures (including any information about parameters, the number of parameters and the number of outputs) would have to be found. This is likely to discard static data in the case of functions which are called many times in a program.

Polyvariant specialisation can more effectively specialise programs, as it can produce multiple versions of functions. Naively one could produce a new version of each function

[2]This function could easily be rewritten without the `continue`, but a non-trivial example would be much longer.

Algorithm 2 Calculate the fixpoint state for the loop while handling control flow change statements

```
Require: $e_{\text{orig}}, e_{\text{return}} \in \text{Ident} \to \text{Value}$, $comms \in \text{Block}$, $always\_iterates \in \text{Boolean}$,
    $loop\_var\_id \in \text{Ident}$, $loop\_var\_val \in \text{Value}$
Ensure: $e_{\text{input}}, e_{\text{output}}, e_{\text{return}} \in \text{Ident} \to \text{Value}$, $exit \in \text{CFC}$
  $e \leftarrow e_{\text{orig}}$
  repeat
    $e_{\text{input}} \leftarrow e[loop\_var\_id \to loop\_var\_val]$
    $\langle e_{\text{output}}, e_{\text{break}}, e_{\text{continue}}, e'_{\text{return}}, exit\rangle \leftarrow f_{\text{run}}(comms, e_{\text{input}})$
    $e_{\text{output}} \leftarrow e_{\text{output}} \sqcup e_{\text{continue}}$
    $e \leftarrow e_{\text{output}}[loop\_var\_id \to loop\_var\_val] \sqcup e_{\text{input}}$
  until $e = e_{\text{input}} \vee (test\_flag(exit, \texttt{CFC\_CONTINUE}) \wedge test\_flag(exit, \texttt{CFC\_NONE}))$
  if $always\_iterates$ then
    $e_{\text{output}} \leftarrow e_{\text{output}} \sqcup e_{\text{break}}$
  else
    $e_{\text{output}} \leftarrow e_{\text{output}} \sqcup e_{\text{break}} \sqcup e_{\text{orig}}[loop\_var\_id \to \texttt{[]}]$
  end if
  $e_{\text{return}} = e_{\text{return}} \sqcup e'_{\text{return}}$
  if $test\_flag(exit, \texttt{CFC\_CONTINUE}) \vee test\_flag(exit, \texttt{CFC\_BREAK})$ then
    $exit \leftarrow set\_flag(exit, \texttt{CFC\_NONE})$
    $exit \leftarrow clear\_flag(exit, \texttt{CFC\_CONTINUE})$
    $exit \leftarrow clear\_flag(exit, \texttt{CFC\_BREAK})$
  end if
```

whenever it is called in the residual code. Each function would be specialised for its parameters hopefully reducing the execution time of the function.

To reduce the number of functions produced, we *memoize* each residual function along with its call signature. When a function is to be specialised, the *memo* table is first checked for a matching signature and if one is found the memoized function is returned instead.

Often a loop might be unrolled where the loop variable is used as a parameter to a function. This function would then be specialised for every iteration of the loop, possibly causing massive code explosion and increase in partial evaluation time. We do not currently check for this although annotations can be used to homogenise parameters. Unfortunately these annotations only act on the current environment and so the changes would affect any later use of the variables and also they can only be used on dynamic variables. These limitations are fairly arbitrary and could be lifted without much difficulty.

Once a function is specialised, the function is added to the list of specialised functions for the original function hashed using the call signature (consisting of the number of parameters, their *values* and the number of outputs). The outputs of the function (both static and dynamic) are also saved so that re-evaluation is not required for later invocations. As this is all saved after the function has been specialised it cannot handle recursion (where the termination conditions cannot be determined statically) and this is likely to lead to infinite specialisation.

In Listing 6.3, there is an example of a power function (using integer bases) which exploits the fact that $x^{2n} = x^n x^n$, with an example function which invokes `power` using two different bases in Listing 6.6.

```
function y = testpower(x)

y = x + pow(x, 3) + pow(x, 4);
```

Listing 6.6: Power test function

If we partially evaluate `testpower` without fixing anything, then the code in Listing 6.7 is produced. Note that no post-processing is performed yet and so there are many redundant assignments.

```
function y = testpower__1(x)
y = x + pow__1(x) + pow__2(x);

function z = pow__1(x)
n = 3;
z = 1;
z = 1 * x;
n = 1;
x = x * x;
z = z * x;
n = 0;
x = x * x;

function z = pow__2(x)
n = 4;
z = 1;
n = 2;
x = x * x;
n = 1;
x = x * x;
z = 1 * x;
n = 0;
x = x * x;
```

Listing 6.7: Specialised version of `testpower`

Once functions are specialised, the call signature is checked and any completely static parameters are removed from the list of parameters to the residual call, in a process known as *reparameterisation* [63]. Since static parameters are initially inserted into the function by assignments, there is no danger that a parameter may later be accessed when it is undefined, if it would not normally be. Static parameters include undefined parameters, which occur when functions are called with fewer than the maximum number of parameters.

Varargin functions are those which have a final (or only) parameter called `varargin`. Such functions can be called with any number of parameters, with the excess values put in varargin in the form of a row-vector cell array. The problem with this approach, is that if any of the excess parameters are dynamic, then varargin becomes dynamic and all of the other static parameters are lost. One way of dealing with this, would be to split out the static excess parameters into conventional parameters as in *arity raising* [29].

```
function y = f(varargin)
y = varargin{1} + varargin{2};
```

For instance this code could become:

```
function y = f(a, b)
y = a + b;
```

This would likely lead to a speed-up, but unfortunately it is unlikely that such code would exist. It is more common to see varargin accessed inside a loop using a loop counter as an index. If the loop is dynamic, then there would never be any hope of performing such a transformation. If it was static, it would have to be deferred until after the loop was unrolled, in which case it would require re-evaluation of the code to insert previously unknown static values and propagate them through the function.

Rather than performing this operation, it would probably be simpler (although require a large development effort) to introduce partially static cell arrays (and perhaps structures). For now, no attempt is made to preserve static excess parameters, but we have found in many cases, that either all of the excess parameters are static or they are all dynamic.

The call signature of a *varargin* function has all the excess parameters stored in a cell array, which if dynamic retains only information about its size. This means that if x is dynamic, f(x, 1) and f(x, 2) have the same call signature, although f(x, 1, 2) would have a different signature.

Static outputs (including unrequested outputs) are also removed from the function. For single output functions, this means that unless the function has side-effects, the call can be replaced with the returned value. For multiple output function calls, the dynamic outputs will be retained in the function call, while the static outputs will be inserted via assignments in the calling residual code. This means that our later dead code elimination will be more effective. For multiple output function calls for which all the outputs become static, the function call will be completely removed unless it has side-effects.

Function calls which produce fully static outputs cannot always be removed, as the call may have side effects. In this case the function call needs to be retained, but its outputs can be removed as above, allowing more effective post-processing. This presents problems in the form of side-effect reordering. To prevent this we require that any function with side-effects called with nargout equal to 1, must retain its single output.

This means that function calls that are embedded in expressions will not be moved and so the side effects will remain in order. This requirement is probably unnecessary as users who rely on the order of function call execution are generally relying on undefined behaviour.

6.5 Inlining

In most partial evaluators, simple functions are *unfolded* inside the calling function. This *unfolding* performs a similar role to our loop unrolling for recursive functions. Since recursion is not so important, we do not unfold functions, but always specialise them. Once the function has been specialised, we then make a decision about whether to *inline* the specialised function in the calling function. Our main strategy for inlining is very simple.

- The function to be inlined must be a single line function containing one assignment. The assignment must be to a simple variable and not a subscript.
- Each of the parameters must be used only once.
- The function must not have a `varargin` parameter.

This strategy means no additional variables need be created and that the overhead of calling functions is reduced, without any expressions being executed more than once. For instance:

```
...
a = b * f(10 * c, d + e * g(x));
...
function z = f(x, y)
z = 25 + x * h(y);
```

is transformed into:

```
...
a = b * (25 + 10 * c * h(d + e * g(x)));
...
```

The second rule is rather restrictive as it always bars the inlining in cases like this:

```
...
a = f(a, g(b));
a = f(g(a), b);
...
function z = f(x, y)
z = x * x + y;
```

If the first invocation was inlined as above it would result in `a = a * a + g(b)`, which would cause no problems. The reason for the second rule is the second invocation, which would be transformed into `a = g(a) * g(a) + b`, which leads to g being invoked twice when it was only invoked once in the original program.

The general solution to this would be to create a variable to store `g(a)` and use that. The first reason for not doing this is simplicity: we are creating a partial evaluator not an optimising compiler. The second reason is that MATLAB has no intra-function scoping, so once the variable was created it would not be destroyed until the end of the function, assuming we created unique variables for each inlined function. This problem could be reduced by reusing the variables we use for inlining. It would not be a problem if used with better compilers and interpreters, which detected that a variable was never used again or which provided common sub-expression elimination as then no temporary variables would be required.

An alternative solution maintaining our goal of simplicity, is to allow inlining when the parameter is a simple variable access. This would inline the first example above but not the second. One convenient exception is that subscripts of parameters count as multiple accesses of a parameter.

```
...
a = rand(3);
b = rand(3);
c = f(a([1 1 1],:), b);
c = f(a, b([1 1 1],:));
...
function z = f(x, y)
z = x([1 2 3], [3 2 1]) + y;
```

This exception means that the first invocation of `f` would not be inlined but the second one would be. This prevents the creation of the following invalid MATLAB code :
`a([1 1 1],:)([1 2 3], [3 2 1])`.

The only reason for excluding `varargin` parameters is that it more complicated to map cell array accesses to the calling parameters. This functionality could certainly be added later, although a stepping stone would be the conversion of `varargin` functions to ordinary functions, by arity raising, in the case where the number of parameters is known in advance. Once this transformation is done, our inlining strategy could be used if applicable.

Given the restrictions on how inlining is applied in our partial evaluator, it is unlikely that functions produced as described in the previous section would ever be inlined due to redundant assignments. The next chapter describes how the these functions are post-processed to remove most of this redundant code.

6.6 Least Upper Bound Caching

To partially evaluate a loop which cannot be unrolled requires the least upper bound of the initial loop state to be computed. If the loop itself contains a loop which also cannot be unrolled, its least upper bound must be found in order to find the least upper bound of the outer loop. In fact the inner least upper bound must be computed for each iteration of the outer loop computation. Nesting loops (including loops appearing in functions called inside loops) leads to an exponential increase in specialisation time.

In addition, while the least upper bound is being computed, no code is being generated. Once it is found, one final traversal of the loop is required which generates the code. In our original partial evaluator, on encountering an inner loop, its least upper bound would be recomputed again. Since this final traversal of the outer loop will use all of the same values that were used on the final iteration of the least upper bound calculation, the extra least upper bound calculation on encountering the inner loop while generating code will produce exactly the same result.

As a result, significant time can be saved by caching least upper bounds once they are calculated. The approach we take is to speculatively store a least upper bound for each loop once it is computed. If a least upper bound calculation needs to be redone because of a change in the outer least upper bound calculation, then the cached result is deleted and replaced with the new result.

The least upper bound cache is actually just a list containing the least upper bound states as they are encountered during the computation. When a loop is encountered, the current size of the cache is increased by 1 and the final position is reserved. Once the calculation is completed, its result will be stored in this position. This makes the cache like a queue where instead of adding directly to the back, a place is reserved which can be filled later.

At the start of each iteration of each least upper bound calculation, the cache after the current position is cleared as the following states are to be recalculated. For a program consisting of only unnested loops, the cache will always be empty. The first interesting program would be like the following:

```
for n = 1:x
    for m = 1:y
        ...
    end
end
```

Here we have the n-loop and the m-loop, with the m-loop nested within the n-loop. The least upper bound of the n-loop is calculated before its body can be partially evaluated. During the first iteration, the m-loop is encountered and position 0 is reserved in the cache which now has size 1. Since this loop is not nested, the cache will not have been altered when the calculation is completed. The computed state will then be stored at position 0 and then the first iteration of the outer loop calculation will end.

At this point, the environment will be examined to see if a fixpoint has been reached and if not iteration will continue. This causes the cache from position 0 (i.e. the whole cache) to be cleared. The inner least upper bound will be recomputed and again stored at position 0. Once the outer least upper bound calculation is complete, the result will be a cache containing one element. The body of the outer loop will then be partially evaluated, at which point the inner loop will be encountered. Since the cache is not empty, its least upper bound will not be computed, but the front of the cache will be *popped* and used to partially evaluate the body of the inner loop.

```
a = 1;
for n = 1:x
    for m = 1:y
        ...
        if a ~= 1
            a = (n,m)
        end
        ...
        a = f(m,n)
        ...
    end
end

function y = f(x,y)
for o = 1:x
    ...
end
```

The above example is more complicated since it contains a nested loop at the top level which contains two calls to the function `f` which itself contains a loop that cannot be unrolled. In addition, during the first iteration the first function call will not be examined as `a` is static and equal to 1. On subsequent iterations, assuming `f` cannot be statically determined to return 1, the function will be called twice.

At the end of the first iteration of the outer least upper bound calculation, the cache will contain 2 states and it will be filled as follows:

$$[\,] \rightarrow \begin{array}{|c|}\hline 0 \\ \hline - \\ \hline\end{array} \rightarrow \begin{array}{|c|c|}\hline 0 & 1 \\ \hline - & - \\ \hline\end{array} \rightarrow \begin{array}{|c|c|}\hline 0 & 1 \\ \hline - & s_o^1 \\ \hline\end{array} \rightarrow \begin{array}{|c|c|}\hline 0 & 1 \\ \hline s_m^1 & s_o^1 \\ \hline\end{array}$$

On the second iteration the cache will have size 3 and it will be filled as follows:

$$[\,] \rightarrow \begin{array}{|c|}\hline 0 \\ \hline - \\ \hline\end{array} \rightarrow \begin{array}{|c|c|}\hline 0 & 1 \\ \hline - & - \\ \hline\end{array} \rightarrow \begin{array}{|c|c|}\hline 0 & 1 \\ \hline - & s_o^1 \\ \hline\end{array} \rightarrow \begin{array}{|c|c|c|}\hline 0 & 1 & 2 \\ \hline - & s_o^1 & - \\ \hline\end{array} \rightarrow \begin{array}{|c|c|c|}\hline 0 & 1 & 2 \\ \hline - & s_o^1 & s_o^2 \\ \hline\end{array} \rightarrow \begin{array}{|c|c|c|}\hline 0 & 1 & 2 \\ \hline s_m^1 & s_o^1 & s_o^2 \\ \hline\end{array}$$

Note that none of the cached values on the second iteration are related to the cached values from the first iteration even though we have reused the symbols. If any further iterations are required then, the cache will be filled out in the same manner as the second iteration.

Once the least upper bounds are found, their order in the table will be identical to the order in which the loops will be encountered while partially evaluating, in which case the states can be popped from the cache as required. In doing so 5 repeated calculations (s_m^1 once and s_o^1 and s_o^2 twice each), are avoided. The downside is that the current state needs to be copied for each least upper bound and only the final cached value is actually used. The memory usage of the partial evaluator is thus raised as is the total number of allocated MATLAB arrays. Since the speed, with which MATLAB runtime library operations run, is slowed as more are created, this creates scalability problems which are beyond our control.

One problem that can occur using this approach is that a function containing loops that cannot be unrolled will be called twice with the same signature inside a loop. If the function has not been specialised before, no return information is available for the function and so it must be abstractly interpreted and the least upper bound of the internal loop calculated and stored in the cache. When the second function call is encountered, exactly the same will happen, resulting in another table being added to the table. However, later when the code is being generated, the first function call will be specialised, and the cache will be popped once. The return values will then be stored in the function call signature table, so that when the second call is reached, no specialisation is required and the specialised function can be reused. Unfortunately the cache is now out of sync. If the function is not specialised, the cache will contain an entry that is not used. If there are further loops to be specialised, they will attempt to use this cache entry even though it is not applicable, resulting in incorrect code generation.

```
for n = 1:x
    ...
    f(n);
    ...
    f(n + 1);
    ...
    f(n, 2);
    ...
end

function f(x, y)
for n = 1:x
    ...
end
```

To solve this, before specialising a function, we note the number of loops left intact so far. This is subtracted from the number of loops left intact after specialising the function, giving n, the number of loops left intact in the function and the functions it calls. On encountering a function specification for which a specialisation already exists, while inside an intact loop, the first n entries are removed from the cache.

```
%# x realscalar
f(x, 1);

for n = 1:x
    ...
    f(n, 1);
    ...
    f(n + 1, 1);
    ...
    f(n, 2);
    ...
end

function f(x, y)
for n = 1:x
    ...
end
```

Above, outside the loop there is previous specialisation of `f` which due to the annotation, will have the same signature as the two calls inside the loop. When computing the least upper bound of the loop, there is no need to re-evaluate `f` for these two calls as the return values are already know. However, the work-around for the previous example does not recognise whether functions were actually specialised during the current loop, and so it will try and remove one entry from the cache for each function call. Since there will be only one entry as created by the third call to `f`, the partial evaluator will fail. This is solved by inserting as many dummy entries into the cache as would be removed later.

6.7 Summary

Within this chapter, we have given a full description of majority of the partial evaluator inner workings. We have demonstrated how MATLAB statements are evaluated and how the decision is made to reduce or to residualise. Using the type system from Chapter 4, we have demonstrated what part abstract interpretation plays both while generating code and also when computing the least upper bound of the loop state using a fixpoint iteration.

Having introduced control flow change statements such as `break` and `continue`, we have refined our fixpoint iteration to correctly handle these constructs. Since computing the least upper bound can account for a large part of the partial evaluation time, we demonstrate a strategy for caching least upper bounds in nested loops, even when they are nested within other functions.

The polyvariant specialisation strategy is outlined including how we reuse functions which share call signatures, made up from the parameters values and types. We also describe a simple but safe inlining strategy which is applied after function specialisation and so can take advantage of the caching strategy just mentioned.

In the next chapter, we will describe a post-processing phase designed to tackle the specific problems that this partial evaluation strategy introduces. Assignments are generated even when the right-hand side is static, without checking if it will be required. This means that much code is generated that is entirely redundant, increasing both the residual program size and the time to execute.

Additionally, while specialised functions are cached in a table indexed by the function call signature, many functions can be specialised identically for different signatures. These identical functions increase program size and can hurt cache performance and so a post-processing phase that eliminates them is essential.

Chapter 7

Post-processing

The purpose of this phase is to reduce the size of the code produced by the partial evaluation. It does this by removing *dead* statements. *Dead* statements are ones which do not have any effect on the result of the current function, do not modify global variables and do not produce any side-effects.

We initially used the following simple approach: assume all variables in a function are dead except for the variables returned by the function. Any assignment to a dead variable is redundant and so can be removed. For any statement which is not dead, mark all the variables referenced in it as live. For `if` statements and `switch` statements, perform the above algorithm for each branch in parallel and then merge the list of live variables, so that any variable that is live on at least one branch, is also live before the conditional. For loops, it is necessary first to find the loop *dependencies*. These are variables that are referenced in the loop without first being assigned in the loop. These loop dependencies are then to be added to the list of live variables before applying the original algorithm. This needs to repeated as removal of dead statements may change the loop dependencies, exposing more dead statements.

This algorithm is sufficient for the majority of codes, but it has several problems. It cannot remove assignments inside loops which create values, only consumed on other iterations of the loop. This is because the assigned variables become loop dependencies. E.g.

```
function y = factorial(x)
a = 1;
y = 1;
for n = 1:x
    y = y * n;
    a = a + 1;
end
```

In this example, if the loop cannot be unrolled, the previous algorithm will not be able to remove any statements, even though `a` contributes nothing to the final result.

In addition this initial algorithm was developed when our subset of MATLAB excluded `return`, `break` and `continue` statements. These statements complicate the control flow and extending the algorithm to deal with them would not be easy.

As a result we replaced the dead code removal algorithm with one using ud-chains (Use-Definition). Ud-chains link a use of a variable to all the places where it could be defined. The dual of ud-chains are du-chains (Definition-Use), which link a definition of a variable to all the places where it might be used. These can be created by solving the reaching definitions problem. Using ud-chains means that irregular control flow using `break`, `continue` and `return` is possible and it is also more effective at removing dead code. While we do not require the du-chains for our algorithm, we can use these for further optimisations along with the ud-chains.

To create the ud- and du-chains, we need to solve the *Reaching Definitions* problem. This "finds which definitions of a variable may reach each use of the variable". [52] It is a forward data flow algorithm that uses a lattice of bit vectors, where each bit represents a variable definition. In general, it is used to find the definitions which reach each basic block of a procedure rather than individual statements.

7.1 Dividing into Basic Blocks

This data flow problem is normally performed on low-level or intermediate-level code and not on the high level parse tree that our partial evaluator produces. This creates some small problems, as high-level representations are not so easy to partition into basic blocks. One approach would be to convert the parse tree form into a low level form and then perform the reaching definitions analysis and dead code removal. This could be problematic as the conversion back to the high level form is not likely to be simple.

In order to use the high level form, a method of partitioning into basic blocks must be devised. For most cases we define a basic block using an existing block[1] and two integers, which give the start and end of the basic block. We also have two special blocks which do not contain any commands: the entry point and exit point blocks. All exits from a function, which allow execution to continue, flow through the exit point.

The function shown in Listing 7.1 would be represented by the parse tree given in Figure 7.1. This function is made up of 6 basic blocks including the start and exit blocks. Figure 7.2 gives a directed graph of the basic blocks showing how control can flow from the start of the function to the end. Basic blocks are subdivided as follows:

- Simple statements which do not affect the control flow, like expressions and all types of assignment are just appended to the end of the current basic block.

[1]A block could be the list of top level commands in a function or the list of commands in the body of a loop or conditional statement

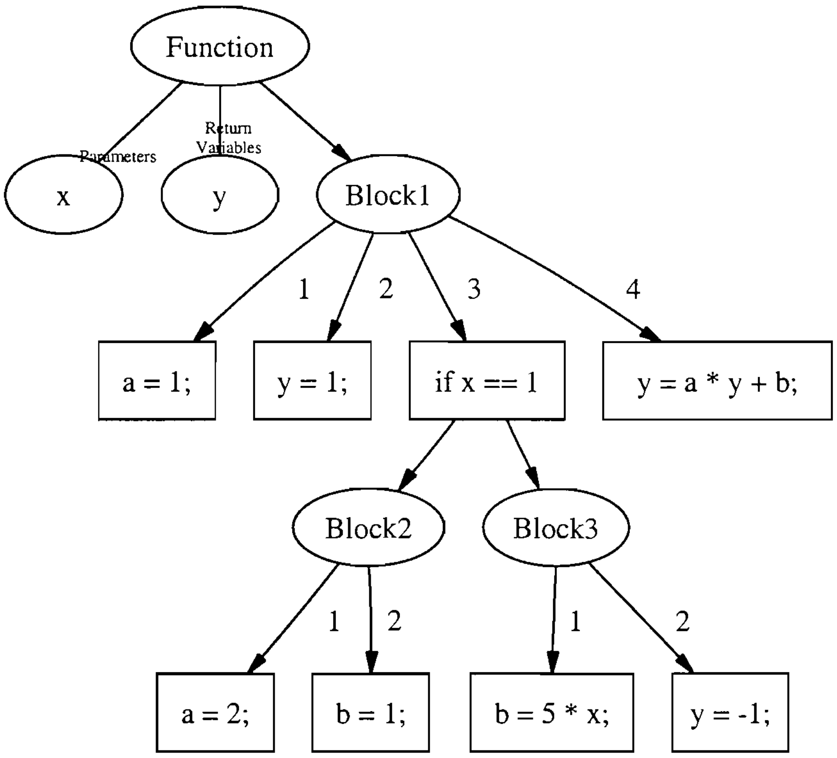


FIGURE 7.1: Parse tree for 7.1

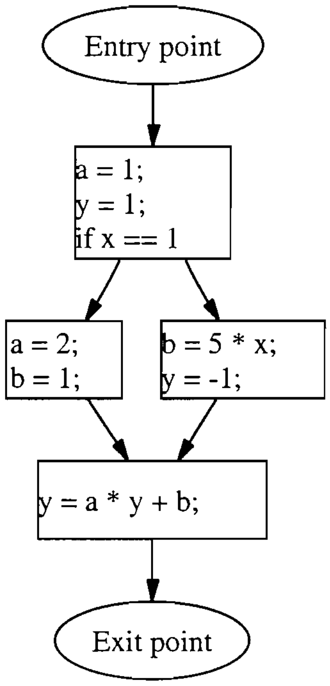


FIGURE 7.2: Graph of basic blocks for 7.1

```
function y = f(x)
a = 1;
y = 1;
if x == 1
    a = 2;
    b = 1;
else
    b = 5 * x;
    y = -1;
end
y = a * y + b;
```

LISTING 7.1: Example Code

- The conditional part of an `if` statement is appended to the current basic block. The first set of statements starts a new basic block and so does the `else` set. A new basic block is created to which two branches point. See Figure 7.3(a).
- The `while` statement starts and ends its own basic block. This block points to a basic block created from the body of the loop. At the end of the loop body, the final basic block points back to the `while` basic block. This then points to an empty basic block to indicate the loop exit. See Figure 7.3(b).
- `switch` statements require a new type of basic block to encapsulate the `case` test. This is a basic block which just contains the `case` and its expression. This is required because `case` is not a valid statement as it can only happen inside a `switch`. The `switch` statement itself is appended to the last basic block, as with `if` statements. It then points to the first case basic block. This will then point to a basic block with the associated commands and also to the next case which will also point to two basic blocks. This is done for all the cases until the last one which points to a basic block for the `otherwise` statements. At the end of the statements for each case and the `otherwise` clause is a pointer to an empty basic block indicating the end of the `switch`. See Figure 7.3(c).
- `for` loops are more complicated than `while` loops as the loop range is only evaluated once while the loop variant is updated on every iteration. As a result, we have two new kinds of basic block: one contains a `for` loop and indicates that it is the *preamble* and only evaluates the loop range. The other contains a `for` loop and indicates that it updates the *variant*. The loop then is comprised of a preamble basic block which points directly to a variant basic block. The variant then points to the body of the loop as well as an empty block at the end (used for loops, where the body never executes). At the end of loop body basic blocks, control flows both to the variant block and the exit block. See Figure 7.3(d).
- `break` causes the current basic block to flow to the exit block of the current loop. See Figure 7.4(a).
- `continue` causes the current basic block to flow to the header block of the current loop. See Figure 7.4(b).

- `return` causes the current basic block to flow to the exit block of the function. See Figure 7.4(c).
- In addition the built-in function `error` causes a change in the control flow as it causes execution of the program to stop. As a result control does not flow at all from a basic block ending with `error`. See Figure 7.4(d).

In the Figure 7.3 and Figure 7.4, solid lines indicate that flow goes directly from one block to the one pointed to by the arrow. If the line is dotted, then there may be other blocks in between, for instance if a `for` loop is nested inside a `while` loop.

For each basic block, i, there exists a set $Pred(i)$ which contains all the basic blocks from which control can flow to i.

7.2 Calculating ud- and du-chains

Once the function has been divided up into basic blocks, the ud- and du-chains need to be calculated. A ud-chain is made up from a *Use* and a list of *Definitions*, whereas a du-chain is made up from a *Definition* and a list of *Uses*. A *Use* comprises a variable and the location of the statement where it is used. A *Definition* is made up from a variable and the location of the statement that writes to it.

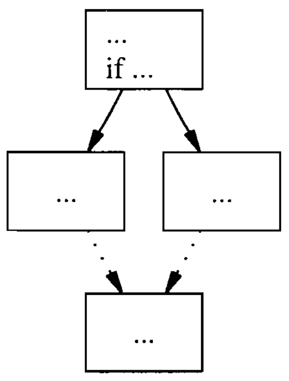
We first calculate the reaching definitions using the method outlined in [52]. This involves creating $kill(i)$ and $gen(i)$ vectors for each basic block, i, indicating which definitions are killed and which are created respectively. To do this we traverse the whole function creating a list of definitions; the position of a definition in this list indicates the position of the relevant bit in the bit vectors.

Each basic block is then traversed and for each assignment, the relevant definition in the $gen(i)$ bit vector is marked set if it is the last assignment to that variable in the block. All other definitions for that variable are then marked in the $kill(i)$ bit vector. In addition the *gen* bit vector for the entry node has the definitions for the function parameters marked.

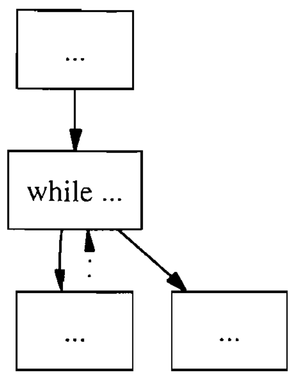
Two additional bit vectors are then created for each basic block: $RCHin(i)$ gives the definitions that can reach the start of a block, i, and $RCHout(i)$ gives the definitions that can reach the end of that block. These can be calculated from:

$$RCHout(i) = gen(i) \vee (RCHin(i) \wedge not(kill(i)))\forall i \quad (7.1)$$

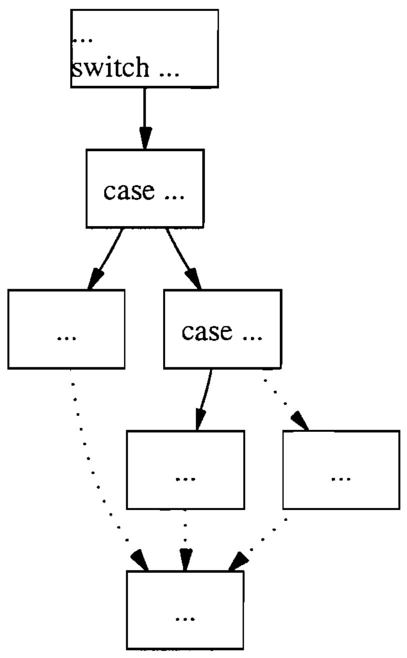
$$RCHin(i) = \bigvee_{j \in Pred(i)} RCHout(j)\forall i \quad (7.2)$$



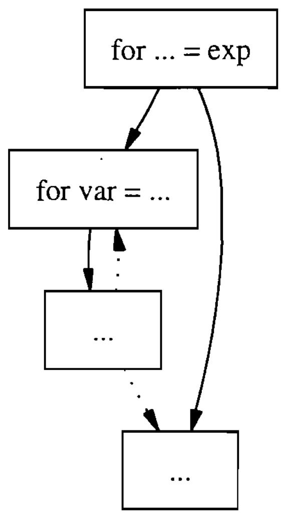
(a) if



(b) while



(c) switch



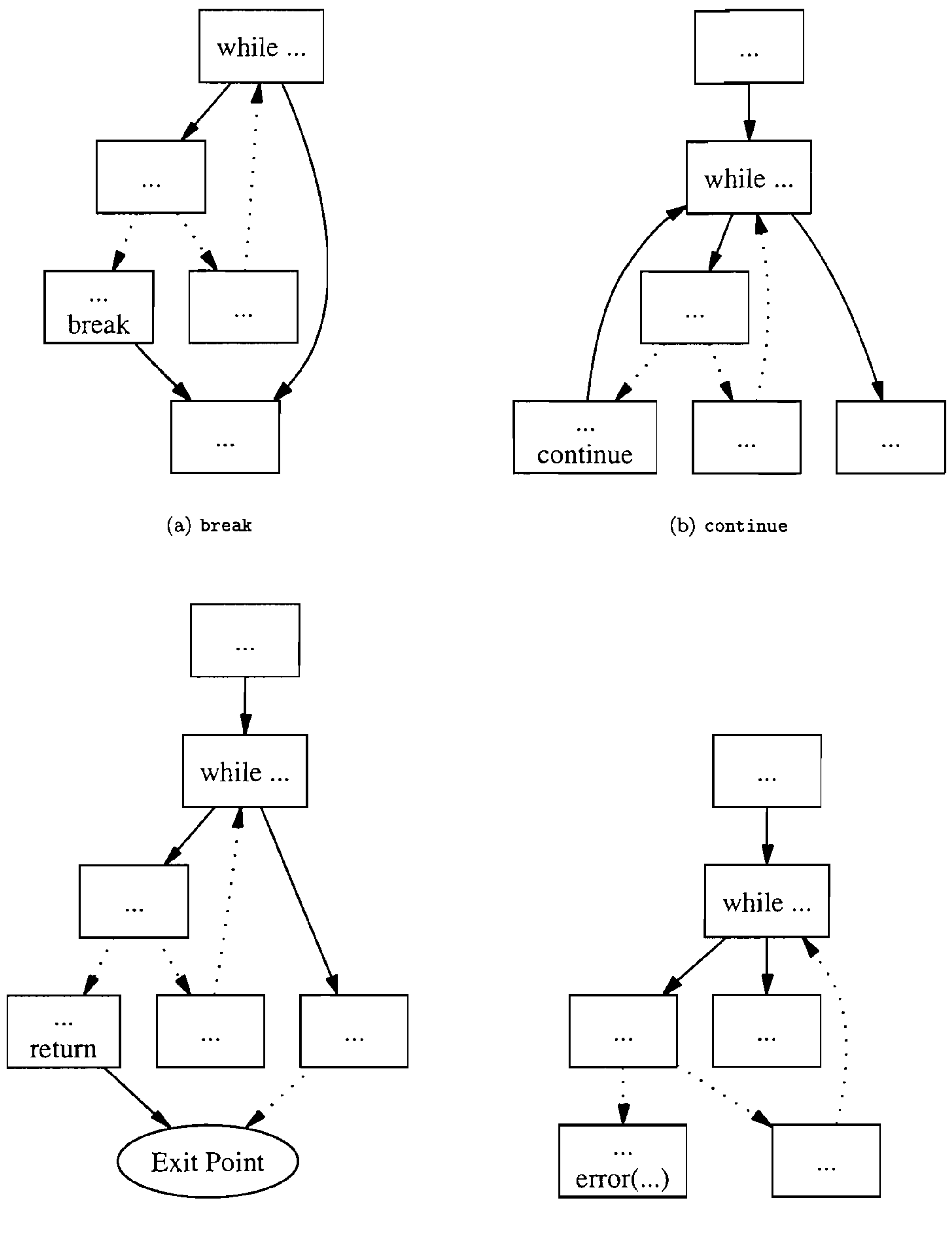
(d) for

FIGURE 7.3: Basic blocks for conditionals and loops.

We now perform a fixed point iteration using equations 7.1 and 7.2. This gives us *RCHin*(i) and *RCHout*(i) for each basic block, i, which can be used to create the ud- and du-chains.

Each basic block is examined, one statement at a time to find uses. Each use is recorded along with the current live definitions. The current live definitions are taken from *RCHin* initially, but assignments in the current block cause the current definition to be replaced. The exit block of the function is considered to *use* the output variables of the function.

Indexed assignments and deletion assignments are both *uses* and *definitions* as they



(a) `break` (b) `continue` (c) `return` (d) `error`

FIGURE 7.4: Basic blocks for the control flow change statements (inside a `while` loop)

both use the old value and assign a new value. Technically any uses of a variable after such an assignment also use any old definitions, but we can ignore this as it just makes the chains longer and has no effect on the result of our algorithm.

7.3 Dead Code Elimination

The Dead Code Elimination occurs in two parts: the first part marks as live the statements directly required to compute the result of the function, while the second part marks all the control flow statements that are required.

Initially all statements are unmarked. We then mark live those statements which produce a side-effect. This includes all statements including calls to functions (both m-files and built-in functions inferred to produce a side-effect) and expressions and assignments which are not terminated by a semi-colon and so would print the result. In addition we also mark live all the control flow change statements (`return`, `continue` and `break`) as working out whether they are redundant is beyond the scope of this work.

As with [52], we use a work-list which we initialise with the set of essential statements as well as the exit point. One by one, we remove statements from the list and examine the uses. For each use, we add to the work-list every currently unmarked statement that creates a definition by examining the appropriate ud-chain as well as marking the statement as live. This process is continued until the work-list is empty at which point, every statement that directly contributes to the result of the function will be marked live.

However conditionals or loops, containing live statements, may not be marked live, so the second stage scans the function for marked statements and marks all the control flow statements that contain them. For instance, if a marked assignment is inside a `for` loop then that `for` loop will be marked live and added to the work-list for the previous stage.

Once the second stage is completed, the first stage is repeated. If no changes were made in the second stage then the work-list will be empty and so it will immediately cease. If the first stage makes no changes then the iteration ceases and the dead code elimination is completed.

If at any point, a statement with side-effects is marked live then the function itself is marked as having side-effects, thus enabling its callers to treat their calls accordingly.

7.4 Duplicate Function Elimination

While using a type system which provides information about dynamic parameters enables more specialisation opportunities, it also has the side-effect that two functions could be produced which are identical even though they have different call signatures.

```
function z = f(x, y)
if x > 5
    z = y + 10;
else
    z = y + 5;
end
```

For instance, if the function above is specialised with static y = 4 and dynamic x, we get:

```
function z = f__1(x)
if x > 5
    z = 14;
else
    z = 9;
end
```

However, there are lots of ways of specifying x to produce the above residual code. So long as x is not specified as something which cannot be used in an inequality test, the above code will be produced. This means x includes but not exhaustively sparse arrays, single precision arrays, sparse real arrays, real double precision arrays and even real logical double precision arrays. While the possible classes and traits are finite, there are infinitely many shape specifications which would all produce the same code.

These redundant specialisations are bad for two reasons: they increase the size of the residual program (thus hurting cache performance) and they require redundant computation at specialisation time. Unlike [59], we do not attempt to detect these before creating them, but after specialising a function we compare it against all previously specialised versions of that function. This comparison checks for structural equality of the abstract syntax trees of each of the functions. The variable names must be identical at each point as well as the names of any called functions. This will therefore not catch two functions which are identical but for variable names, but since such functions are unlikely to appear as a result of two specialisations of the same function, this is not a problem. Since called functions must have the same name in each function to be considered identical, it will only mark a function as a duplicate if the called functions were specialised and marked as duplicates.

Debois [16] uses *bisimulation equivalence* to detect duplicate specialisations for a flow chart language. The target there is however more fine-grained than ours as we are only checking for functions, while Debois is looking for duplicate sets of basic blocks leading up to the end of a function. Since we have an AST at this point, a side-by-side comparison of two ASTs is relatively simple, which is not the case in the unstructured flow-chart language.

As with Ruf and Weise [59], we maintain information about the return values of each function. This information is stored in the function call signature, which then links to the specialised function. In the case of duplicates, this return information is still calculated since the function was specialised before the duplication was detected. The function call signature of the duplicate is modified to point to the original specialisation and the extra one is deleted.

A better solution may be to try and avoid creating these specialisations in the first place. However as Ruf and Weise showed, the order of function specialisation can effect whether duplicates are detected prior to specialisation. If there are two specialisations which produce the same result, but one specialisation has a looser specification that includes the other one ($s_1 \sqsubseteq s_2$), then performing the specialisation with the tighter specification (s_1) first will allow the second specialisation (with specification s_2) to be avoided. Reversing the order however requires both specialisations to be performed. An approach which attempted to calculate a further specification, s_m, which was the most precise specification (given the limits of the type system) which led to the same specialisation, in conjunction with the least precise specification s_l, might be feasible. The specialisation can then be reused safely and optimally if $s_m \sqsubseteq s \sqsubseteq s_l$.

Exactly calculating s_m and s_l would not be possible but an approximation could be achieved, which could prevent a significant number of redundant specialisations. Whether or not the cost is justified would have to be examined.

Even with our late function duplication detection, the specialisation time can often be reduced. This is because deleting functions causes the constants stored in the abstract syntax trees to be deleted. Since these use the MATLAB library type `mxarray`, and due to the poor performance of the libraries given large scale array allocation, the larger programs become the slower the partial evaluator becomes. The cost of detecting and deleting functions can then be outweighed by the speed-up achieved by using fewer arrays.

7.5 Summary

In this chapter, we have seen two main methods for reducing the size of the residual code produced by a partial evaluator. Firstly a method for removing all code which neither contributes to the returned values of a function or its side-effects was proposed using ud-chains. Secondly a structural equivalency check was introduced for newly specialised functions, which detects and removes identical functions produced from different call signatures. The need for these methods comes from the partial evaluation technique, which is skewed towards generating more code than necessary, since elimination at a later stage is simpler.

As noted by Knoop [43], it is possible that removing dead assignments or expressions could change the semantics of a program as the dead code could generate a run-time error. Division by zero is not a problem in MATLAB as it evaluates to plus or minus infinity or "Not A Number", all three of which are valid values in MATLAB. However multiplying two matrices with incompatible dimensions will halt execution as will raising a matrix to the power of another matrix. We do not consider this to be an important problem but note it for completeness and discuss it further in Section 9.1.

Another way the semantics of a program could be modified by our dead code elimination algorithm, is if a loop is removed which would never terminate. E.g.

```
function y = f(x)
while x > 1
    x = x + 1;
end
y = 10;
```

In the above example, the function `f(x)` will never terminate unless the initial value of `x` is less than or equal to 1. But our strategy only considers loops which contain statements contributing to the final return value, which this loop does not. As a result the assignment `x = x + 1` is not marked live and consequently neither is the `while` loop, making the resulting function:

```
function y = f(x)
y = 10;
```

Using the assumption that we are only dealing with programs that have been thoroughly tested and work, we will assume that such loops will not feature in the input to our partial evaluator.

With the core partial evaluator functionality built and with an effective post-processor, it is now possible to examine real functions and programs. In the next chapter, we will examine empirically the efficacy of the partial evaluator with and without the post-processor. Our examples range in complexity from the relatively simple functions like Chebyshev approximation and Lagrange interpolation to a large optimisation program used to model aero-engine inlets.

Chapter 8

Results

In this chapter we will evaluate the effectiveness of our partial evaluator on several programs. The code for these tests is a mixture of code developed inside the Computational Engineering and Design research group at the University of Southampton[1], code from partners at other universities and code found in repositories on the Internet.

8.1 Single Function Experiments

These first experiments show the results of applying partial evaluation to small functions, with clear specifications for the parameters.

8.1.1 Chebyshev Series Approximation

The first code tested was a function for the generation of Chebyshev polynomials [24], which, like power series, are used to approximate functions by summing terms. As with power series, using more terms leads to better approximations. This function has two parameters, a `m`-by-`n` matrix, `c`, of coefficients for calculating `m` functions with `n` terms and a vector, `x`, as input to the functions. Table 8.1 shows the relative timings for the chebyshev function (iterated 5000 times to get measurable results). Timings are shown where the function has been partially evaluated where just `n` is fixed, `c` is fixed and lastly where `c` is fixed along with the size of `x`. The timings are further subdivided according to whether post-processing was used. The results show a steady increase in performance as more information is fixed, with the final function running in half the time of the original, when post-processing is used. Listing 8.1 is the implementation of the Chebyshev series approximation algorithm.

[1] `http://www.ses.soton.ac.uk/projects/Comp_Eng_Des/comp_eng_des.html`

	Orig	n fixed		c fixed		c and size of x fixed	
size(c,2)		No p.p.	With p.p.	No p.p.	With p.p.	No p.p.	With p.p.
2	1.00	0.90	0.89	0.81	0.77	0.60	0.48
4	1.00	0.94	0.92	0.84	0.81	0.59	0.50
6	1.00	0.94	0.93	0.85	0.82	0.58	0.51
8	1.00	0.94	0.94	0.85	0.83	0.57	0.52
10	1.00	0.95	0.95	0.86	0.84	0.57	0.52

TABLE 8.1: Relative timings for the Chebyshev functions with $m = 3$ and $p = 3$, relative to original function (p.p. is post-processing).

```
function y = chebseries(c,x)
m = size(c,1);
p = size(x,2);
a = zeros(m,p);
b = a;
xx = ones(m,1) * x;
f = 2 * xx;
for k = size(c,2):-1:1
  d = b;
  b = a;
  a = c(:,k) * ones(1,p) + b .* f - d;
end
y = a - (b .* xx);
```

LISTING 8.1: A simple implementation of Chebyshev series approximation

Listing 8.2 shows the Chebyshev function partially evaluated with `size(c,2)` set to 4. The only difference post-processing makes in this case is the removal of the assignment to k.

```
function y = chebseries__1(c, x)
m = size(c, 1);
p = size(x, 2);
a = zeros(m, p);
b = a;
xx = ones(m, 1) * x;
f = 2 * xx;
d = b;
b = a;
a = c(:, 4) * ones(1, p) + b .* f - d;
d = b;
b = a;
a = c(:, 3) * ones(1, p) + b .* f - d;
d = b;
b = a;
a = c(:, 2) * ones(1, p) + b .* f - d;
d = b;
b = a;
a = c(:, 1) * ones(1, p) + b .* f - d;
k = 1;
y = a - b .* xx;
```

LISTING 8.2: The Chebyshev series approximation code partially evaluated with `size(c,2)` fixed

Listing 8.3 shows the Chebyshev function partially evaluated with c set to a 5-by-4 matrix. Here post-processing would remove the assignments to c, m and k.

```
function y = chebseries__1(x)
c = [0.950129   0.762097   0.615432  0.405706;
     0.231139   0.456468   0.791937  0.93547;
     0.606843   0.0185036  0.921813  0.916904;
     0.485982   0.821407   0.738207  0.41027;
     0.891299   0.444703   0.176266  0.89365];
m = 5;
p = size(x, 2);
a = zeros(5, p);
b = a;
xx = [1; 1; 1; 1; 1] * x;
f = 2 * xx;
d = b;
b = a;
a = [0.405706; 0.93547; 0.916904; 0.41027; 0.89365] * ...
    ones(1, p) + b .* f - d;
d = b;
b = a;
a = [0.615432; 0.791937; 0.921813; 0.738207; 0.176266] *  ...
    ones(1, p) + b .* f - d;
d = b;
b = a;
a = [0.762097; 0.456468; 0.0185036; 0.821407; 0.444703] *  ...
    ones(1, p) + b .* f - d;
d = b;
b = a;
a = [0.950129; 0.231139; 0.606843; 0.485982; 0.891299] *  ...
    ones(1, p) + b .* f - d;
k = 1;
y = a - b .* xx;
```

LISTING 8.3: The Chebyshev series approximation code partially evaluated with c fixed

Listing 8.4 shows the Chebyshev function partially with c set to a 5-by-4 matrix and size(x,2) set to 5. We show here, the post-processed version of the code. While this code runs in only half the time of the original function, it is immediately evident that further improvements could be made. The first assignment to a is effectively a = a + 0 * f - 0. If the expression was reordered, the subtraction would have been performed but a better solution would be to identify binary operations where one operand is all zeros or all ones or the identity matrix and perform the appropriate constant folding.

```
function y = chebseries__1(x)
xx = [1; 1; 1; 1; 1] * x;
f = 2 * xx;
a = [0.405706 0.405706 0.405706 0.405706 0.405706;
     0.93547 0.93547 0.93547 0.93547 0.93547;
     0.916904 0.916904 0.916904 0.916904 0.916904;
     0.41027 0.41027 0.41027 0.41027 0.41027;
     0.89365 0.89365 0.89365 0.89365 0.89365] + ...
     [0 0 0 0 0; 0 0 0 0 0; 0 0 0 0 0; 0 0 0 0 0; 0 0 0 0 0] .* ...
     f - [0 0 0 0 0; 0 0 0 0 0; 0 0 0 0 0; 0 0 0 0 0; 0 0 0 0 0];
b = a;
```

```
a = [0.615432 0.615432 0.615432 0.615432 0.615432;
     0.791937 0.791937 0.791937 0.791937 0.791937;
     0.921813 0.921813 0.921813 0.921813 0.921813;
     0.738207 0.738207 0.738207 0.738207 0.738207;
     0.176266 0.176266 0.176266 0.176266 0.176266] + b .* ...
     f - [0 0 0 0 0; 0 0 0 0 0; 0 0 0 0 0; 0 0 0 0 0; 0 0 0 0 0];
d = b;
b = a;
a = [0.762097 0.762097 0.762097 0.762097 0.762097;
     0.456468 0.456468 0.456468 0.456468 0.456468;
     0.0185036 0.0185036 0.0185036 0.0185036 0.0185036;
     0.821407 0.821407 0.821407 0.821407 0.821407;
     0.444703 0.444703 0.444703 0.444703 0.444703] + b .* f - d;
d = b;
b = a;
a = [0.950129 0.950129 0.950129 0.950129 0.950129;
     0.231139 0.231139 0.231139 0.231139 0.231139;
     0.606843 0.606843 0.606843 0.606843 0.606843;
     0.485982 0.485982 0.485982 0.485982 0.485982;
     0.891299 0.891299 0.891299 0.891299 0.891299] + b .* f - d;
y = a - b .* xx;
```

LISTING 8.4: The Chebyshev series approximation code partially evaluated with c fixed and size(x,2) fixed. Post-processing has also been used

The assignments to d and b before each assignment to a could also be removed. This could be achieved by using copy propagation and variable renaming, but doing this ought to be a compiler optimisation and it is certainly beyond the scope of a partial evaluator.

8.1.2 Lagrange Interpolation

The second test function (Listing B.1), when given a set of points from a function, computes the Lagrange interpolating polynomial [79] that passes through them and returns a set of points on the curve. The MATLAB code is comprised of two nested loops both dependent on the number of points to interpolate followed by a third loop also dependent on the number of points. We specialised this function by first fixing the number of points, n, (and thus the number of x and y coordinates) resulting in all of the loops being unrolled and then further specialised it by fixing the x coordinates. Again, timings were taken with and without post-processing. Table 8.2 shows the improvements we achieved including at least 50% speed increases when the x coordinates are completely fixed.

8.1.3 Solving a Gaussian Hypergeometric Differential Equation

The next example function (Listing B.2) solves the Gaussian Hypergeometric differential equation, $x(1-x)\frac{\mathrm{d}^2y}{\mathrm{d}x^2} + c - (a+b+1)x\frac{\mathrm{d}y}{\mathrm{d}x} - aby = 0$, using a series expansion [19]. The main work is done by a single for loop which calculates the series terms. To get more

		n fixed		x-coordinates fixed	
n	Original	No postproc	Postproc	No postproc	Postproc
2	1.00	0.73	0.67	0.67	0.58
4	1.00	0.82	0.79	0.68	0.64
6	1.00	0.85	0.83	0.67	0.65
8	1.00	0.86	0.84	0.68	0.66
10	1.00	0.87	0.86	0.68	0.66

TABLE 8.2: Relative timings for the Lagrange functions with values of n

n	Original	Partially Evaluated	Post-processed
4	1.17	0.85 (0.73)	0.72 (0.62)
6	1.47	1.07 (0.73)	0.94 (0.64)
8	1.77	1.30 (0.73)	1.17 (0.66)
10	2.08	1.54 (0.74)	1.40 (0.67)

TABLE 8.3: Timings in seconds for the Gaussian Hypergeometric differential equation solver (iterated 8000 times). (Relative times are given in brackets).

accurate results, higher order series terms are required and thus more iterations. The number of terms is the parameter that we have chosen to specialise. As can be seen from Table 8.3, partial evaluation with post-processing is very effective at speeding up the function, showing a 49–62% performance increase over the original function.

8.2 Ordinary Differential Equation Solvers

A common numerical application for which MATLAB is used is the integration of ordinary differential equations (ODEs). MATLAB provides several ordinary ODE solvers, like ode45, which uses a combined 4th and 5th order Runge-Kutta method to solve non-stiff differential equations.

```
[t,y] = ode45(odefun,tspan,y0) with tspan = [t0 tfinal]
[t,y] = ode45(odefun,tspan,y0) with tspan = [t0 t1 t2 ... tfinal]
[t,y] = ode45(odefun,tspan,y0,options)
[t,y] = ode45(odefun,tspan,y0,options,p1,p2...)
[t,y,te,ye,ie] = ode45(odefun,tspan,y0,options...)
sol = ode45(odefun,[t0 tfinal],y0...)
```

There are numerous ways of invoking the highly parameterised function, ode45, ranging from simple invocations which use the default options to the more complicated use of output functions, event functions, extra parameters and so on. We can find a numerical solution to the equation $\frac{dT}{dt} = k(T - T_m)$, which gives Newton's Law of Cooling using ode45. This requires a way of expressing the right hand side of the equation (RHS). E.g.

```
function dTdt = newton(t, T)
dTdt = k * (T - Tm);
```

This function can either be passed to ode45 using a function handle or as a string. The function is then invoked by calling the built-in function feval on the function handle or string. This will be slower than invoking the function directly but allows the solver to be general. Clearly if partial evaluation would just replace the calls to feval with calls to newton, we would achieve a performance increase. In fact we can do much better due to the high level of parameterisation of ode45. When ode45 is invoked it checks its parameters to see how it was called. Assuming it was invoked with a function handle, it checks if tspan and y0 were supplied otherwise it gives an error. It then checks whether tspan has 2 or more elements or it throws an error. It checks whether the differential function returns a column vector. There are too many checks to list here, but it suffices to say that many of these can be precalculated during specialisation.

For instance specialisation of ode45 in its simplest form with respect to the function, newton is given by this (note k and Tm have been hard-coded into the RHS function with the values of -0.0253178 and 100 respectively):

```
function [x, y] = specialise_ode45(xspan, y0)
%# xspan realdouble
%# xspan size [1 2]
%# y0 realscalar
[x, y] = ode45(@newton, xspan, y0);

function dTdt = newton(t, T)
% k = -0.0253178;
% Tm = 100;
dTdt = -0.0253178 * (T - 100);
```

Specialising this with post-processing and inlining enabled gives us a total of 124 lines of code of which 97 lines give the ode45 function and the rest is comprised of the additional non-built-in functions called by ode45. This contrasts with 410 lines for the ode45 function alone (with comments and blank lines stripped).

A second use of ode45 allows us to have one newton function for all values of k and Tm by passing them as parameters.

```
function [x, y] = specialise_ode45(xspan, y0)
%# xspan realdouble
%# xspan size [1 2]
%# y0 realscalar
k = log(39/40);
Tm = 100;
[x, y] = ode45(@newton, xspan, y0, [], k, Tm);

function dTdt = newton(t, T, k, Tm)
dTdt = k * (T - Tm);
```

Specialisation here produces exactly the same code as before as the values of k and Tn are propagated to exactly the same places as before, with post-processing leaving no traces in the residual program.

```
function [t,y] = odeEuler(diffeq, tn, h, y0, varargin)

t = (0:h:tn)';
n = length(t);
y = y0 * ones(n, 1);

for j = 2:n
  y(j) = y(j-1) + h * feval(diffeq, t(j - 1), y(j - 1), varargin{:});
end
```

Listing 8.6: Implementation of Euler's method for solving ordinary differential equations

```
function [t,y] = odeRK4(diffeq, tn, h, y0, varargin)

t = (0:h:tn)';
n = length(t);
y = y0 * ones(n, 1);
h2 = h/2;
h3 = h/3;
h6 = h/6;

for j = 2:n
  k1 = feval(diffeq, t(j - 1), y(j - 1), varargin{:});
  k2 = feval(diffeq, t(j - 1) + h2, y(j - 1) + h2 * k1, varargin{:});
  k3 = feval(diffeq, t(j - 1) + h2, y(j - 1) + h2 * k2, varargin{:});
  k4 = feval(diffeq, t(j - 1) + h, y(j - 1) + h * k3, varargin{:});
  y(j) = y(j-1) + h6 * (k1 + k4) + h3 * (k2 + k3);
end
```

Listing 8.7: Implementation of the 4th Order Runga-Kutta method for solving ordinary differential equations

Due to inlining the function `newton` does not exist at all in the specialised code, but for a more complicated RHS function like `logistic` (given in Listing 8.5), inlining is not always possible as it uses one of its parameters more than once. Since ode45 sometimes passes array subscripts to the RHS function, these calls cannot be inlined. A better inlining strategy would be required to handle this, either by deciding that the number of repeated array accesses is outweighed in cost by the benefit of inlining or by creating temporary variables to store results. In this area, MPE would benefit greatly by being combined with an effective optimising compiler that could make inlining decisions itself.

```
function dPdt = logistic(t, P, options, a, b)
dPdt = P * (a - b * P);

function dydx = normal(x, y)
dydx = -2 * x * y;
```

Listing 8.5: Sample RHS Functions for ODE Solvers

We also demonstrate two other ordinary differential equation solvers, `odeEuler` (Listing 8.6) and odeRK4 (Listing 8.7), which use the Euler and 4th order Runge-Kutta method respectively. These are simple non-adaptive solvers, which are described in [40]. Both of these can pass in additional parameters to RHS functions (useful for specifying constants) but are otherwise unconfigurable.

	Original	mpe	mpe -p	mpe -pi
normal	21.23	12.50 (70%)	12.50 (70%)	5.14 (313%)
logistic	8.40	4.84 (73%)	4.33 (94%)	4.33 (94%)
newton	8.34	4.74 (75%)	4.19 (99%)	1.50 (456%)

TABLE 8.4: Timings (in seconds) for odeEuler with three different RHS functions

	Original	mpe	mpe -p	mpe -pi
normal	8.16	4.76 (71%)	4.76 (71%)	1.80 (353%)
logistic	3.25	1.85 (76%)	1.64 (98%)	1.64 (98%)
newton	6.41	3.62 (77%)	3.18 (102%)	0.96 (568%)

TABLE 8.5: Timings (in seconds) for odeRK4 with three different RHS functions

	Original	mpe	mpe -p	mpe -pi
normal	1.30	1.07 (21%)	0.96 (35%)	0.74 (76%)
logistic	1.02	0.81 (26%)	0.68 (50%)	0.65 (57%)
newton	1.02	0.81 (26%)	0.67 (52%)	0.50 (104%)

TABLE 8.6: Timings (in seconds) for ode45 with three different RHS functions

Table 8.4 and Table 8.5 show the results of specialising odeEuler and odeRK4. We can see that partial evaluation alone produces code that on average runs 70-75% faster, for each of the equations with odeEuler and 71-77% faster for odeRK4. These examples also illustrate where post-processing and inlining can have an effect. For instance normal (given in Listing 8.5) benefits negligibly from post-processing but as the RHS function is very simple, it can be inlined leading to very large speed ups. On the other hand, logistic benefits from post-processing as the constants propagated to the RHS function are assigned to variables that are never referenced (as they expand to their constant values) but it cannot be inlined as it references its parameters more than once. The newton function has both constants and can be inlined leading to the largest speed ups. Berlin and Weise [10] also partially evaluated problems involving Runge-Kutta integration of ODEs. They achieved much larger speed-ups than have been achieved here, but our use of MATLAB means that many of the specialisation opportunities they see are likely hidden inside libraries.

In all cases, MPE took about 0.19 seconds to produce the residual code. In each case the function was executed 100 times to get the timings shown here, although longer running times could also be achieved using longer time intervals for the solvers.

Table 8.6 shows the results of specialisation for the ode45 function provided with MATLAB. The main function is around 400 lines long and it can call many other functions, making this a very complex function. Specialising using the examples from earlier, we only want to use a very limited subset of the ode45 functionality, as we do not require event functions, mass matrices or execution statistics. The residual code produced by

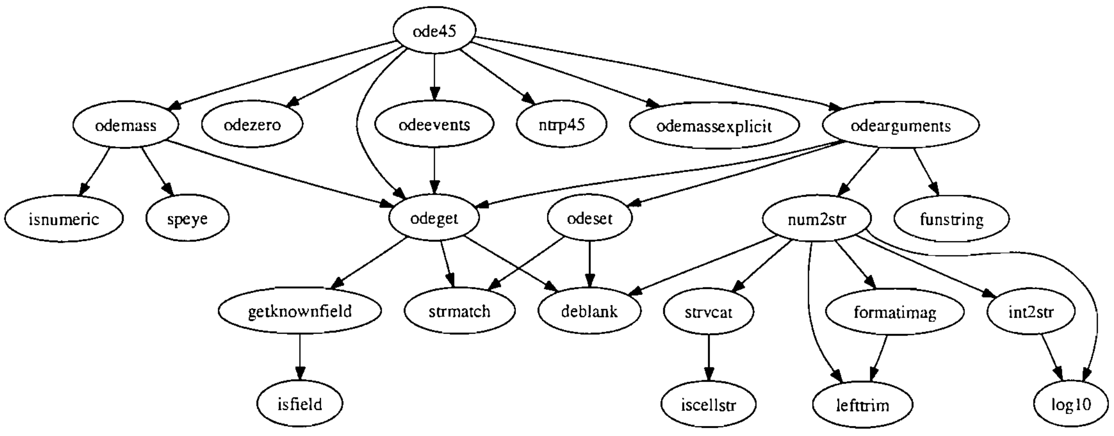


FIGURE 8.1: Function dependency graph for ode45, showing 23 unique functions

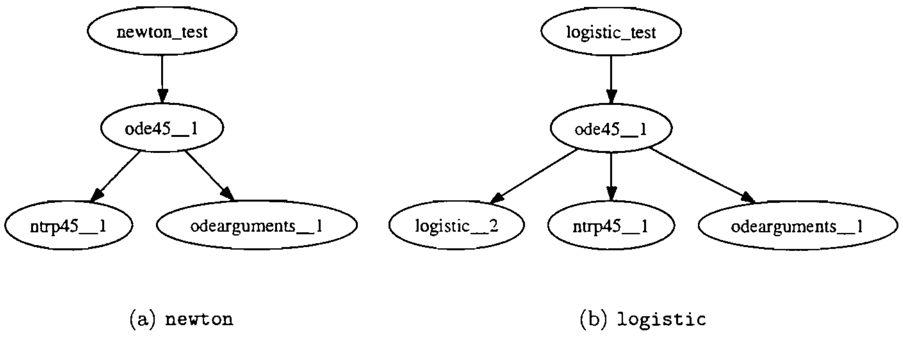


FIGURE 8.2: Function dependency graph for ode45 specialised for different RHS functions

specialising with respect to normal using inlining and post-processing is only 125 lines long and that includes all the functions produced by the specialisation. This specialisation takes around 0.39 seconds and 0.43 seconds with post-processing as well. With the other solvers, execution of the RHS function dominates the computation time and so specialisation gives enormous speed ups. Since ode45 is more complex, the execution of the RHS functions has less of an impact on the performance. However, speed ups of around 50–100% are still significant.

A visual demonstration of the complexity of ode45 is given in Figure 8.1, which shows the main function dependencies (not including any function introduced through function handles).[2] Figure 8.2 shows the dependency graph when ode45 was specialised with respect to newton and logistic. Clearly specialisation can yield much simpler residual programs.

The final results for the ODE solvers come from applying the MATLAB compiler, MCC, to the original code and the residual programs. Tables 8.7 and 8.8, show the timings from

[2]These graphs were produced using a special mode of MPE

the MATLAB interpreter as given before as well timings from the compiled code from the `odeEuler` and `odeRK4` solvers. Figures 8.3 and 8.4 show the effect of partial evaluation and compilation against the original program. In every case with these solvers, compilation more than doubles the performance increase from partial evaluation. With the `ode45` solver, the results in Table 8.9, while not as impressive as the other solvers, show that the performance increases are retained through compilation. The `ode45` results need to be considered carefully, since the MATLAB compiler did not compile the `ode45` solver when it was not partially evaluated. Instead a pre-compiled version, which is part of the run-time libraries, was used. There is no way of knowing whether this version was originally compiled directly from the MATLAB version or whether it has been optimised in some way. Given that compilation does not produce much faster code, in this example, it seems likely that optimisation, if any, was applied sparingly. The performance chart in Figure 8.5 clearly demonstrates how the combination of partial evaluation and compilation can be effective especially given that while the solver is very general, it is a production solver and has no doubt been fairly carefully optimised.

		Original	`mpe`	`mpe -p`	`mpe -pi`
Interpreted	`normal`	21.23	12.50 (70%)	12.50 (70%)	5.14 (313%)
Compiled	`normal`	6.39	2.23 (187%)	2.23 (187%)	0.80 (699%)
Interpreted	`logistic`	8.40	4.84 (73%)	4.33 (94%)	4.33 (94%)
Compiled	`logistic`	2.96	1.12 (164%)	0.78 (279%)	0.76 (289%)
Interpreted	`newton`	8.34	4.74 (75%)	4.19 (99%)	1.50 (456%)
Compiled	`newton`	2.92	1.11 (163%)	0.74 (295%)	0.22 (1227%)

TABLE 8.7: Compilation vs. Interpreter Timings (in seconds) for `odeEuler` with different RHS functions

		Original	`mpe`	`mpe -p`	`mpe -pi`
Interpreted	`normal`	8.16	4.76 (71%)	4.76 (71%)	1.80 (353%)
Compiled	`normal`	2.53	0.89 (184%)	0.89 (184%)	0.30 (743%)
Interpreted	`logistic`	3.25	1.85 (76%)	1.64 (98%)	1.64 (98%)
Compiled	`logistic`	1.16	0.44 (163%)	0.30 (281%)	0.30 (281%)
Interpreted	`newton`	6.41	3.62 (77%)	3.18 (102%)	0.96 (568%)
Compiled	`newton`	2.32	0.87 (167%)	0.58 (300%)	0.15 (1447%)

TABLE 8.8: Compilation vs. Interpreter Timings (in seconds) for `odeRK4` with different RHS functions

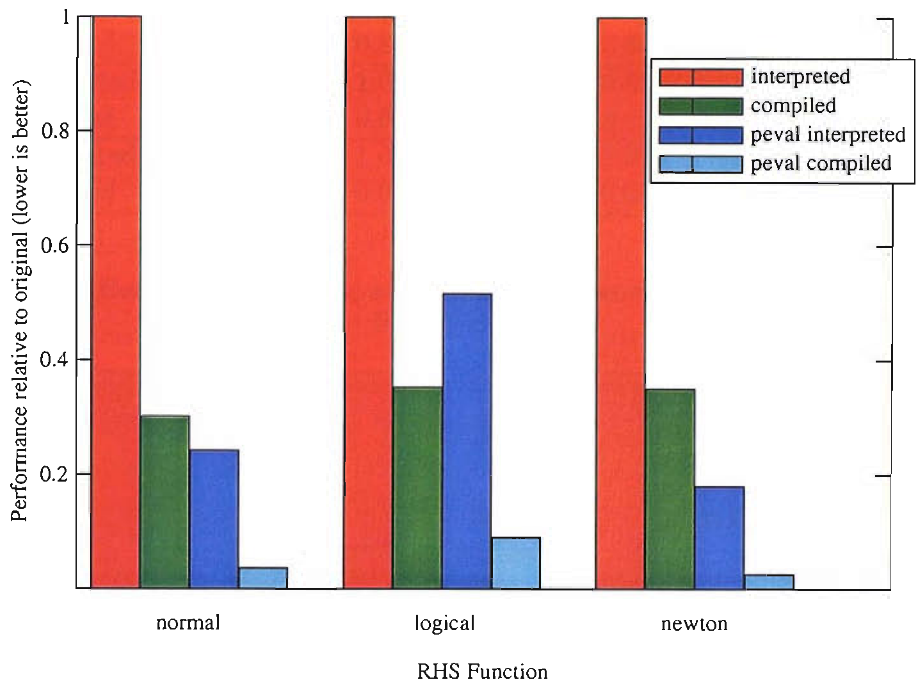


FIGURE 8.3: Relative timings for `odeEuler`. Results normalised with respect to original interpreted program

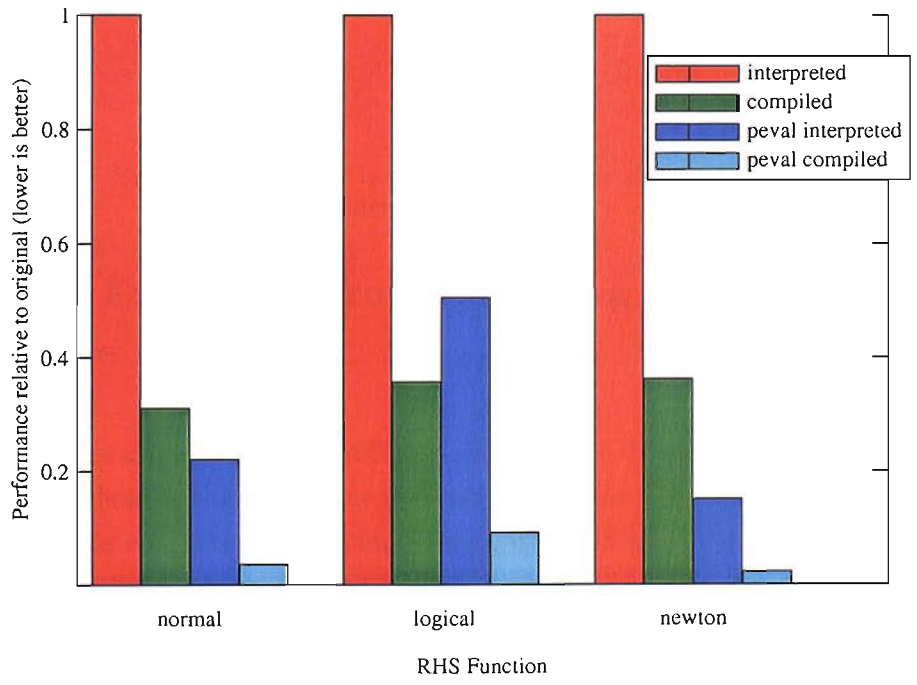


FIGURE 8.4: Relative timings for `odeRK4`. Results normalised with respect to original interpreted program

		Original	`mpe`	`mpe -p`	`mpe -pi`
Interpreted	`normal`	1.30	1.07 (21%)	0.96 (35%)	0.74 (76%)
Compiled	`normal`	0.80	0.65 (23%)	0.50 (60%)	0.46 (72%)
Interpreted	`logistic`	1.02	0.81 (26%)	0.68 (50%)	0.65 (57%)
Compiled	`logistic`	0.69	0.54 (28%)	0.39 (77%)	0.38 (82%)
Interpreted	`newton`	1.02	0.81 (26%)	0.67 (52%)	0.50 (104%)
Compiled	`newton`	0.69	0.54 (29%)	0.39 (78%)	0.36 (92%)

TABLE 8.9: Compilation vs. Interpreter Timings (in seconds) for ode45 with different RHS functions

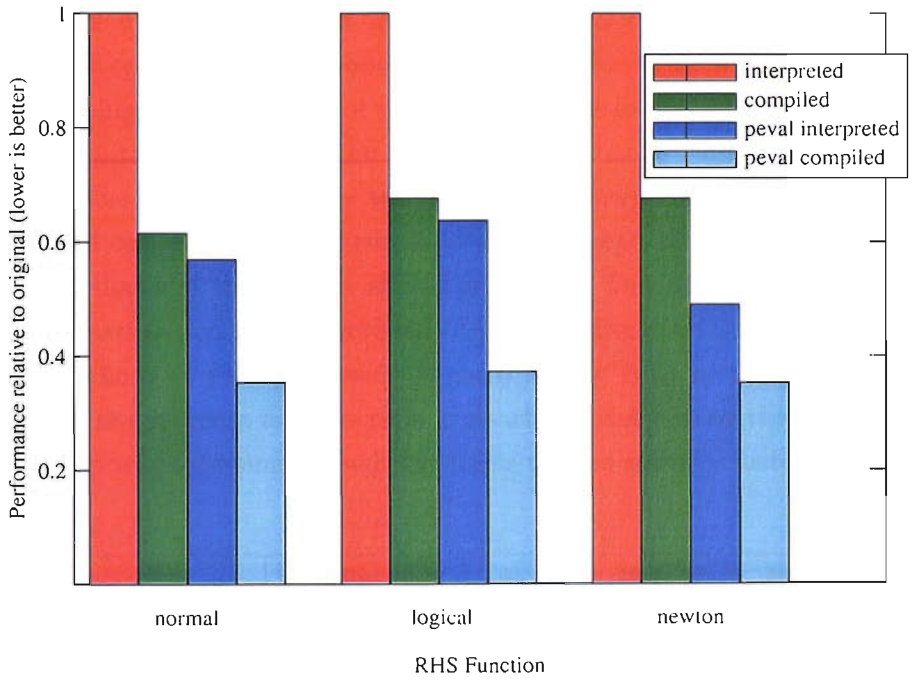


FIGURE 8.5: Relative timings for ode45. Results normalised with respect to original interpreted program

8.3 Dust Erosion of Probes Entering the Martian Atmosphere

This code is an example of where MATLAB really excels. It was written by an aero/astro engineering student at the University of Southampton as a final year project. Since the student has little computer science knowledge, it would not be expected that he could effectively optimise his code himself. With little modification (in the form of *widen* annotations in the ODE solver), the code can be partially evaluated giving a substantial performance boost.

This code investigates what happens to a conical probe using an Aerobraking Manoeuvre to slow down within the Martian atmosphere. One possibility of damage to the cone comes from dust particles hitting the cone while it travels at supersonic speeds. A cone-shaped object travelling at supersonic speeds would create a shock wave in front of it and particles entering this wave would be deflected until they either hit the cone or were travelling parallel to the cone angle (at which point they cannot hit the cone). This code finds the critical height above the cone tip at which particles do not hit the cone with parameters, cone angle and mach speed.

To do this two ordinary differential equations need to be solved. This code (given in Listings B.3 – B.9) extensively uses `ode45`, which was examined in Section 8.2. However the differential equations here are more complicated since they use event functions to terminate the calculation early when no further computation is required. In addition the RHS function of one actually calls `ode45` itself. In Table 8.10, we see the results from partially evaluating this function. For each residual code as well as the original program, two timings were taken. The first run was the time taken to run in a completely clean environment. The intention in this test is to show how much parse time is a factor in executing the code. The second run is identical except now MATLAB should have parsed the function and therefore it should be faster. The difference between the two times is an approximation to the overhead of parsing. Here the time taken to parse the residual code, where no post-processing is used, is very close to the original if slightly lower. However any increase in parse time is clearly outweighed by the overall reduction in time. When post-processing is used, the parse time is actually faster for the residual program.

	Original	`mpe`	`mpe -p`	`mpe -pi`
Partial Evaluation	-	5.1	5.2	5.2
First run	22.7	20.6 (9%)	18.2 (25%)	15.2 (49%)
Second run	22.4	20.3 (10%)	18.0 (25%)	15.0 (49%)

TABLE 8.10: Timings (in seconds) for `length_crit`

Since this program uses nested loops so extensively, it clearly illustrates the effects of caching least upper bounds as described in Section 6.6. Some timings, without least upper bound caching, are shown in Table 8.11. These are considerably longer than the times taken with caching, and are in fact more than we expected. We believe this is an artefact of the bad scaling of the MATLAB libraries in the presence of many arrays. Since the caching reduces the number of calculations at a point when more arrays have been allocated (when code is being generated), the benefits are slightly exaggerated.

	`mpe`	`mpe -p`	`mpe -pi`
Without caching	14.0	13.3	13.5
With caching	5.1	5.2	5.2

TABLE 8.11: Partial Evaluation Timings (in seconds) for `length_crit` with and without least upper bound caching

This is an example, where the time taken to run the partial evaluator and the residual code once is less than the time to execute the original code. This example shows how partial evaluation combined with post-processing and inlining can be very effective. While the speed-up produced is around 50%, no static data was specified initially. The original program, from which `length_crit` is taken, calls it many times, which means that the cost of partial evaluation will be easily eclipsed.

In this example, attempts to give the partial evaluator more information by specifying that all three input parameters are real scalars, produced exactly the same code, but it took 2 seconds longer to produce it. This is presumably because duplicate functions were produced for subtly different call signatures. These were then removed in post-processing, resulting in an identical residual program.

As with the earlier solvers, this program was compiled, producing the results given in Table 8.12. Again the benefits of partial evaluation are retained through the compilation process, although the effect of inlining is less important in the compiled code.

	Original	`mpe`	`mpe -p`	`mpe -pi`
Interpreted	22.4	20.3 (10%)	18.0 (25%)	15.0 (49%)
Compiled	13.4	12.0 (12%)	10.7 (25%)	10.1 (33%)

TABLE 8.12: Compilation vs. Interpreter Timings (in seconds) for `length_crit`

8.4 Arcadia CFD Solver

The 'Arcadia' code is a new type of aeroacoustic Computational Fluid Dynamics (CFD) solver, particularly aimed at the problem of re-designing aero-engine inlets to reduce radiated fan noise [17, 18]. It is particularly targeted for use in an optimisation loop since it returns not only results pertaining to the analysis of a design, but also information about how sensitive the results might be to small changes in the design.

This is an example of an early iteration of a complex system. It has not been fully optimised and so benefits from the constant propagation that is integral to partial evaluation.

This code originally used global variables to store most of the parameters. These global variables are initialised by the calling function and are not altered during execution. Since we have largely ignored global variables, it was necessary to produce a version of this program, which does not use global variables, by passing all of the parameters into the main function and modifying any calls to functions which require global variables to instead take extra parameters. A partial evaluator which handled global variables, used as in this program, would not be too difficult to implement as the global variables remain static throughout the execution of the program.

This code was specialised in two different ways. Firstly the shapes and types of all the parameters were fixed, leading to the results in Table 8.13, which show that partial evaluation was very quick and produced a program that ran in just over half the time of the original program. As with the code from Section 8.3, two timings were taken so that the time taken to load the code could be estimated. No timings were taken with inlining enabled as this program offers no opportunities for this optimisation.

The second specialisation fixed nearly all of the initial parameters, apart from the optimisation point parameter. This is a realistic specialisation since generally the objective function would be run many times with just the optimisation point changing. The results of this are shown in Table 8.14. Specialisation has taken significantly longer and annotations had to be added to prevent a large nested loop being completely unrolled, which would have produced a huge residual program. There is improvement in execution times over Table 8.13, but this difference is not very large. There is also a significant increase in the time taken to parse the residual code, with the second specialisation initially taking longer to execute than the first. This is not surprising given that the first specialisation is 420 lines long and the second is 7203 lines long. This large increase in program size may well have hurt the performance.

	Original	`mpe`	`mpe -p`
Partial Evaluation	-	0.41	0.45
First run	64.8	45.0 (44%)	35.1 (85%)
Second run	65.0	44.7 (45%)	35.0 (86%)

TABLE 8.13: Timings (in seconds) for `arcadia` with parameter types fixed but not their values

	Original	`mpe`	`mpe -p`
Partial Evaluation	-	16.7	13.1
First run	64.7	46.4 (39%)	36.1 (79%)
Second run	64.8	43.4 (49%)	33.7 (92%)

TABLE 8.14: Timings (in seconds) for `arcadia` with most parameter values fixed

The performance improvements for post-processing given in Tables 8.13 and 8.14 are very large and on examination of the residual code, it emerged that `arcadia` performed many computations which did not contribute towards the final returned result. These calculations were a leftover from a previous iteration of the code, and were no longer required. While this does demonstrate that dead-code elimination using ud-chains is very effective at discovering redundancy, it would be fairer to assess the partial evaluator in relation to a version of the code with the redundancy removed first, called `newarcadia`.

As above, `newarcadia` was specialised twice and the results are shown in Tables 8.15 and 8.16. Unsurprisingly, the performance of the partial evaluator has improved since it no longer has to examine these redundant computations. The execution time of the unspecialised program has been reduced by 15 seconds, while the time for the residual code without post-processing is about 8 seconds less. The post-processed version is identical in both cases. Using `newarcadia`, partial evaluation produces around a 50% increase in speed.

	Original	`mpe`	`mpe -p`
Partial Evaluation	-	0.33	0.36
First run	49.8	36.2 (38%)	35.1 (41%)
Second run	50.2	36.1 (40%)	35.0 (43%)

TABLE 8.15: Timings (in seconds) for `newarcadia` with parameter types fixed but not their values

	Original	`mpe`	`mpe -p`
Partial Evaluation	-	13.6	11.5
First run	49.8	37.9 (31%)	36.1 (38%)
Second run	50.2	35.0 (43%)	33.8 (49%)

TABLE 8.16: Timings (in seconds) for `newarcadia` with most parameter values fixed

As with the Mars Lander code, least upper bound caching gives large improvements here. Table 8.17 shows how caching can give a small improvement for programs which already partially evaluate quickly as in the case where only the shapes and types of the parameters were fixed. However when the values of some of the parameters are fixed, more computations are performed inside nested loops and so least upper bound caching saves around 20–25% off the specialisation time.

	`mpe`	`mpe -p`
Types fixed with caching	0.33	0.36
Types fixed without caching	0.37	0.39
Values fixed with caching	13.6	11.5
Values fixed without caching	16.4	15.4

TABLE 8.17: Partial Evaluation Timings (in seconds) for `newarcadia` with and without least upper bound caching

Compiling `arcadia` and `newarcadia` caused some problems, when trying to generate a residual program where most of the parameter values are fixed. With residual MATLAB code of 7203 lines, the C code produced by the compiler was about 10 times as long again. When we tried to compile it, it took around 30 minutes, whereas all the other examples have taken less than 10 seconds. The resulting executable was actually slower than the compiled version of the residual code generated with a much looser specification. This is likely because the executable was 10 times the size (around 2MB). This clearly demonstrates that unrolling needs to be kept in check so that executables do not grow excessively.

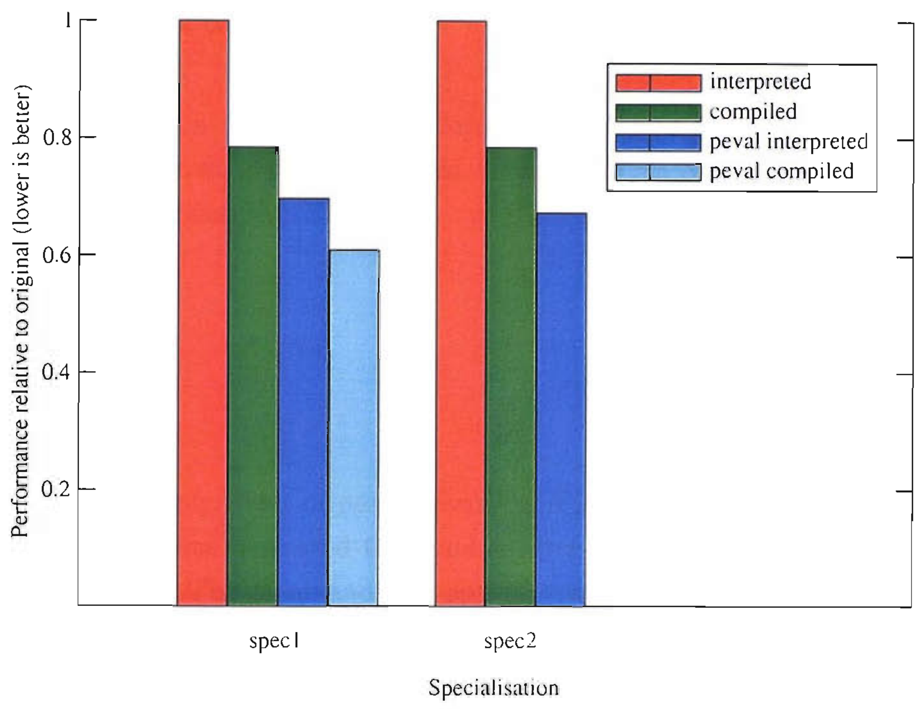


FIGURE 8.6: Relative timings for `newarcadia`, showing effects of partial evaluation and compilation on two specialisations

In Table 8.18 are the results of compiling the less specified residual code generated from both `arcadia` and `newarcadia`. In this example, the benefit of partial evaluation is lessened by compilation. However, since the MATLAB compiler does not perform many traditional optimisations and generates code, which likely hinders the C compiler from optimising itself, this comparison is perhaps unfair. With a better optimising compiler, it is quite likely that partial evaluation would expose many more optimisation opportunities.

	Program	Original	`mpe`	`mpe -p`
Interpreted	`arcadia`	65.0	44.7 (45%)	35.0 (86%)
Compiled	`arcadia`	49.4	38.7 (28%)	30.6 (61%)
Interpreted	`newarcadia`	50.2	36.1 (40%)	35.0 (43%)
Compiled	`newarcadia`	39.4	31.6 (25%)	30.6 (29%)

TABLE 8.18: Compilation vs. Interpreter Timings (in seconds) for `arcadia` and `newarcadia` with parameter types fixed but not their values

The results for `newarcadia` are illustrated in Figure 8.6. Since compiling the second specialisation took unfeasibly long, compilation and partial evaluation are omitted for the second specialisation. One thing to draw from this chart is that specialisation using limited specification produced faster results than ordinary compilation, in a faster time.

8.5 Result Accuracy

While partial evaluation has achieved performance increases for the code tested here. this has come at the cost of a small loss in accuracy. This occurs due to the textual representation of numbers.

```
function s = p(n)

s = 0;
for n = 1:n
    s = s + 1 / 3;
end
```

In this example, the only effect of partial evaluation is to substitute `0.333333` for `1/3`. The partial evaluator has truncated the number after the sixth decimal place as this is its default behaviour. The command line option `--precision` can be used to specify the number of significant figures displayed but this can cause a large increase in the size of the residual program if there are large matrices of numbers. This problem is not easily soluble as we cannot output a binary representation of constants which is exactly equivalent to that which would be stored in memory.[3]

One approach would be to avoid folding operations which lead to constants requiring truncation. In the above example, `1/3` would be retained in the residual code, which would not be detrimental to the performance in either the compiler or interpreter.[4] As the MATLAB compilers produce C code, they too must truncate floating point numbers. In this case it stores the result to 17 significant figures which is the most storeable in a double.

Since the MATLAB compilers perform no constant propagation, they will produce fewer constants than our partial evaluator and so the code will not be unduly bloated. In addition the code is compiled to the machine code where constants will be stored as 8 byte doubles, regardless of the accuracy used.

Going back to the residual code in Listing 8.4 and recreating it by specifying that the output use 17 significant figures produces a program, which is 82% longer, but executes no slower. This is not unduly worrying since the program is an extreme example where the residual program consists almost entirely of matrices. The input values were generated randomly and so any early precision cut-off would have truncated the matrices produced. Most programs would not have grown quite so dramatically given more reasonable input.

[3]This is complicated even more by the fact that floating point registers frequently use a higher precision than can be stored in memory.

[4]Strangely MATLAB 6.5 seems to interpret `1/3` slightly slower than `0.333333`, while 6.1 is not affected.

8.6 Summary

This chapter has demonstrated several applications of partial evaluation with respect to MATLAB code. MPE can be applied both to small functions, with relatively fast execution times, to fine tune the performance of a larger system, as was done with the chebyshev series function from Section 8.1.1, or it can be applied to much larger functions like the Mars Lander code or Arcadia, where the functions involved perform much more intensive calculations.

The smaller functions benefited more from unrolling than many of the larger examples. The unrolling exposed static data within loops thus allowing some pre-computation as well as removing the overhead of checking the loop condition. This generally led to larger programs, which scaled with the size of the data or the accuracy required in the output.

The larger examples gained their benefits from their highly parametric nature. Large complicated functions that could be used in many ways were transformed into highly specialised functions, no longer carrying the overhead of genericity. These programs were often much smaller than the original programs (when assessing all the functions called in the MATLAB libraries).

The results from compilation show that partial evaluation is never a performance hindrance and indeed sometimes works even better with compilation, although some results show less improvement when specialised and compiled over the original compiled program. This is unsurprising since partial evaluation often simplifies programs which means that one of its benefits is the reduction in time handling control structures such as loops, which are usually much better handled by compilation. Since MATLAB 7 reportedly has a much better *Just-In-Time* compiler, it may well be that some of these improvements will be made even less visible. While MATLAB 6.5 also has a *Just-In-Time* compiler, this was not assessed as light testing has shown it to perform very well for some programs ($> 10\times$ improvement), but then modifying those same programs in seemingly inconsequential ways then dropped the performance levels down to that of MATLAB 6.1. This is presumably because it was in a relatively early stage of development. Further work needs to be done to see how partial evaluation can work to bring out the best in *Just-In-Time* compilation.

Chapter 9

Future Work

While we have seen definite performance gains due to partial evaluation, there is undoubtedly room for improvement. The framework is in place to perform many additional optimisations, which are currently not performed. In this chapter we will outline areas where further work can be done.

9.1 Asserting assumptions

In many places, assumptions are made about parameters and the results of functions and operations that cannot be fully evaluated. For instance with `a + b`, if `a` is known not to be a scalar and `b` is known not to be a scalar, it is assumed that both `a` and `b` have the same shape. If this is not the case, a run-time error would be produced.

```
%# a size [3 UNKNOWN]
%# b size [UNKNOWN 4]
y = size(a + b);
```

In the above code, the shape of `a + b` will be assumed to be `[3 4]`. The call to `size` can be fully evaluated as the shape of the expression is known and so the original operation can be removed. In the case where `a` and `b` do not have the same shape, an error would not be produced as the add operation would never be carried out.

In most cases it would be better to assume that errors like this were not made. After all partial evaluation is an optimisation step, usually occurring after testing where it could be assumed that most programmer error had been eliminated.

Mandatory assumption checking would certainly hurt performance, but if assertions could be inserted optionally, for instance when debugging, it would aid both the users and ourselves as it could quickly determine where we make false assumptions.

9.2 Automatic Widening

Currently MPE can loop infinitely on input containing loops which steadily widen shape values. Consider the following code:

```
function y = f(x)
y = 1;
for n = 2:x
  y = [y n];
end
```

This function returns a vector of values from 1 to x with increment 1. The current implementation of MPE will iterate over the loop trying to find the shape of y. It can easily determine that it is a two dimensional matrix with 1 row. However each successive iteration will give it a higher value for the number of columns. Since iteration only ceases when stabilisation of shape information is achieved, the iteration will be infinite. To prevent this problem, the number of columns needs to be *widened*. If the number of columns is set to $\langle 1, \omega \rangle$, no further iteration would be required. This is currently possible using the `widen` annotation but this needs to be determined automatically.

To this end, we need to develop heuristics to determine when there is a possibility of infinite iteration. This would likely be based on looking for incremental shape changes inside loops. This problem is a common one in partial evaluators where recursion is used for all loops. In that case, there is often a counter which is either incremented or decremented and then compared with a dynamic variable. This can lead to the generation of infinitely many functions all specialised with regard to a different counter value, or to the function being unfolded infinitely many times. Ruf [62] describes several heuristics used to detect when recursion will be bounded and when it will be infinite. This includes detecting an induction variable which is reduced on every iteration towards a bound, at which it terminates. Katz and Weise [39] describes a method where increasing parameters which are not actually used at specialisation time (other than in generating further values of the counter) are detected by lazy use-analysis. A similar technique may prove useful to us, since when shape growth is unbounded, the shape itself is generally not used at specialisation time.

9.3 Restructuring expressions for performance benefits

Currently all optimisations on expressions retain the order of expressions. Expressions like $a + 1 + b + 2 - a$ are parsed as $(((a + 1) + b) + 2) - a$. Evaluating strictly using the parse tree form gives no chance for simplification and so the ideal form of $b + 3$ cannot be achieved. Addition is a commutative operation for all kinds of matrices and scalars and so the expression could be reordered to $(a - a) + b + (1 + 2)$, at which point our current scheme can reduce the expression to $(a - a) + b + 3$. Reducing $a - a$ is not so simple; it is not necessarily 0, as a might not be a scalar. In fact it is always equal to `zeros(size(a))`, which is 0 in the case of a scalar and a matrix with the same shape as a if a is a matrix. Unless a is a scalar, the expression can only be reduced if b is has the same shape. If b has a different shape and is not a scalar then a runtime error will occur. If it cannot be determined that b has the same shape as a, then we could assume the absence of programmer error or perhaps generate assertions as described in Section 9.1.

Identity relations like $MI = IM = M$, where I is the identity matrix with the same shape as M, need to be recognised and dealt with. These kinds of expressions can often occur after unrolling. For example:

```
function z = power(x, y)
z = eye(size(x));
for n = 1:y
  z = z * x;
end
```

If `power` is partially evaluated with `y` set to 3, then the residual program would currently be:

```
function z = power(x)
z = eye(size(x));
z = z * x;
z = z * x;
z = z * x;
```

However recognising that `z` is initially the identity matrix would mean that the function can be reduced to:

```
function z = power(x)
z = x;
z = z * x;
z = z * x;
```

With further assignment amalgamation the function just becomes:

```
function z = power(x)
z = x * x * x;
```

As mentioned by Menon and Pengali [49], it is possible to reorder the evaluation of expressions to realise some performance increases. When evaluating $A * B * x$, (where A and B are matrices and x is a scalar), a right to left evaluation would take $O(n^2)$ operations while a left to right evaluation would take $O(n^3)$ operations. Since the natural order of evaluation in MATLAB is left to right, brackets would need to be inserted for efficient evaluation. Clearly in cases where it is unknown whether the operands are matrices or scalars then no such optimisation is possible.

9.4 I/O operations

As mentioned in Section 6.1.1, we currently do not support loading in data with I/O commands like `fopen`, `fscanf` and `fclose`. Unfortunately, problems can easily arise when trying to eliminate I/O operations and embed data directly in the function files. One problem is that data stored externally can often be often be stored more efficiently than as a list of MATLAB assignments. Unless data files are small, this can result in large increases in the residual code size.

It is also problematic because I/O operations must be completed. If a file is opened by `fopen` and the file descriptor returned to a variable, it is not useful to replace this with an assignment to a constant value as file descriptors only have meaning if the `fopen` function is actually called. If all I/O operations have static parameters and can be fully evaluated, this is not a problem. All references to the file handle would be removed during partial evaluation as the calls to I/O functions would evaluate to the values stored in the file.

However if for some reason an I/O operation cannot be evaluated because it is dependent on some dynamic input, then the I/O operation would be left in the code with a file handle that was no longer valid. Even worse, partial evaluation could continue onto another I/O operation that appeared to be fully executable. In the course of normal execution, the file seek pointer would already have moved on, but because we could not execute the previous commands it would be in the old position and the wrong data would be read.

Work needs to be done to assess whether all I/O operations relating to a file handle can be evaluated, so that we can then evaluate them. In some cases we might have to evaluate an I/O operation before we can determine whether further operations can be evaluated, which could mean that we have to backtrack.

9.5 Better Inlining

In Section 6.5, we described a simple strategy for inlining. While its simplicity made its implementation easy, its restrictions greatly limited its applicability.

The restrictions were mostly made to prevent ever creating temporary variables. Part of the reason for this, is that since there is no way of creating a separate scope within a function, the variable will be not be deleted until the function returns or the variable is overwritten. In this way, a naive strategy of creating a new temporary variable for every inlined instance of a function could dramatically increase the memory used by a function. Variables can be explicitly deleted using the `clear` function, but this involves function calls that could hurt performance. Another strategy would be to use a pool of temporary variables which are reused, so that a minimum number of temporary variables would be created. One could still imagine a situation where an inlined function caused the creation of a very large matrix which was not destroyed until the function terminated, as no other inlined functions were called subsequently.

9.6 Context Propagation

When a branch of a conditional is taken, there is implied information, which we currently ignore.

```
%# x ndims 2
if isempty(x)
  for n = x
    ...
  end
else
  ...
end
```

In the above code, if the positive branch is taken, it is implicit that x has no dimensions with size 0. This means that the `for` loop will always iterate at least once and therefore when computing the state after the loop, the least upper bound does not need to be combined with the result for if the loop had not iterated even once.

The difficulty in context propagation is inferring what the condition expression says about its variables. In the above case, x has only 2 dimensions and so since it is not empty it must have a shape of $\left\langle \langle 2,2\rangle, \left\langle \langle 1,\omega\rangle, \langle 1,\omega\rangle \right\rangle \right\rangle$. Inferring this requires inverting the `isempty` function, which would probably require writing code for each function to determine what its context propagation properties were. It is likely that not many functions would provide any useful context information, but this would have to be examined.

9.7 Full Class and Polymorphism Support

Currently functions residing in class directories (e.g. `@inline`), are ignored by MPE. As a first step towards full class support, these directories would have to be searched for m-files. Since the class of an expression is not known at parse time, the loading of the right function file will have to be deferred until the expression is abstractly interpreted. If the expression is static or if the class is known, then the exact function can be found without difficulty, in which case it can be renamed so that there is no ambiguity, which means it no longer has to be in a class-specific directory. If the class is unknown, then there are two options.

Firstly we could choose not to specialise the polymorphic function at all. This is not as simple as it seems: if one of the polymorphic functions are in a *private* directory then they will not be visible to the residual program (unless the residual program is put in the directory above the private directory). We therefore need to ensure that all files that might be called are also callable by the residual function, by if necessary copying m-files.

The second strategy is to speculatively partially evaluate all functions that could be selected. All class-specific functions would need to be stored in separate directories, which would mean they could only call the top-level function of our system, since the output is currently only put in one file. To solve this, every function which is callable by functions in other files need to be stored in a separate file.

Specialising every possible function that could be called, could also cause a performance reduction. This performance reduction might be acceptable if an expression could genuinely have many different classes and fast code is desired for all possibilities, but often the problem will be that the class is really static but it cannot be inferred or that there are only two possible classes. Class annotations could be manually inserted to prevent too much specialisation, but currently a class can either be known completely or not known at all. A way of restricting the set of classes to specialise for would be desirable in this case.

Clearly the work of Schultz et al [65, 64] on Java is relevant here, even though their work uses offline partial evaluation, since they have to tackle similar problems. Recently Andersen and Schultz [4] used a declarative language, called Pesto, to control the specialisation process. This can be used to constrain the possible classes, for which a function can be specialised. Since our reasons for using online specialisation were to save users from requiring a deep understanding of the process, this method may not be readily applicable.

9.8 Extending Types

We have previously stated that our type system can encapsulate more shape information than that of Joisha [32], however much of this information is not directly useable. Knowing that an array has between 2 and 3 columns can rarely translate into any advantage at partial evaluation time using our current system. It is useful to know that there is more than 1 column as certain other shape inference rules can now apply, but the specific information about there being between 2 and 3 columns is no more useful than knowing that there are between 2 and 4 columns.

One use is that if a matrix with between 2 and 3 columns is added to a matrix with between 3 and 4 columns, we can infer that the result if not an error will have exactly 3 columns. With exact information, functions like `size` can be fully evaluated. However, this kind of inference is very unlikely to occur in practice.

The problem arises because when `size` is called and the information is not totally exact, all the information is discarded. This is because our type does not store information about the actual value. If the type was extended to store an interval, in which the actual value was bounded, certain realistic expressions such as inequality tests could be evaluated. E.g.

```
function f(a, b, varargin)
if nargin < 2
  error('Not enough parameters to f')
end
...
```

If `nargin` is at least 2 ($(\langle 2, \omega \rangle)$), with the current partial evaluator, the conditional cannot be checked, since `nargin` will evaluate to a dynamic scalar. However if this dynamic scalar also contained the interval $[2, \infty]$, the binary expression could be checked and the conditional statement removed.

Preliminary work on implementing interval bounds for values known to be scalars has shown encouraging results, but by extending the type in this way, we exacerbate the problem posed by infinitely widening shape components. If bounds are to be realistically introduced, then detection of infinite widening is a must as well as a strategy for testing whether functions signatures differ inconsequentially. The risk here is that adding bounds will cause further redundant specialisations of functions, when the bounds have no impact, causing a large slow down of the partial evaluation process.

Chapter 10

Conclusions

10.1 Review

Chapter 8 showed some impressive results, which demonstrate that partial evaluation can be very effective at improving MATLAB performance. This performance was seen in the small examples as well as large programs such as the Arcadia CFD solver. Performance increases were sometimes due to interprocedural propagation of static values, leading to early computation taking the burden off the residual code, but also due to dynamic values having static properties inferred. While inferring the shape of an array does not yield the same performance benefits as knowing its contents, it can enable some minor precomputations, which can enable loop unrolling, leading to some useful performance boosts as was shown in Section 8.1.

The benefits of compilation were also retained through compilation using the MATLAB compiler, MCC. Especially with the simple ODE solvers, partial evaluation and compilation proved to be a very successful way of boosting performance. Since these solvers were so simple, the use of higher order functions had a very significant effect on the time to execute. Once partially evaluated with respect to fixed RHS functions, this cost was eliminated, and the compiler was able to take advantage of this even more than the interpreter with one example running nearly 15 times faster.

The nature of many MATLAB library functions also makes them candidates for specialisation. Highly parameterised functions which can perform their role in many different ways are prevalent in MATLAB. While their flexibility is enabling and gives users many options, it is also a source of performance loss. MPE is able to produce specialised solvers, with only the minimal of user intervention, giving users the best of both worlds. It is the lack of manual interference by the user that make MPE most attractive to MATLAB users. Often these are users who just want to solve a problem and are not interested in learning any more about their development environments than necessary.

Once programs have been written and tested, it is convenient to leave in place many of the assertion checks that were used during the initial development period as they prove useful later when there is further development. Partial evaluation can statically remove many of these assertions, like the guards on most of the library functions which check that the correct number of parameters were used, allowing the programmer to maintain only one source program for both development and deployment. This is in many ways like C, where `assert` is normally defined as a macro which can be disabled by specifying command line flags to the C compiler. However the compiler will remove all those checks without verifying that they are satisfied statically. It may be that with the assertions still intact, the compiler could optimise away some of them but this would be rarer since most compilers do not propagate information interprocedurally.

Due to the high level of the MATLAB language, traditional areas of partial evaluation gains are not so susceptible to it. This includes unrolling of tight inner loops based on knowledge about the size of arrays. Standard operators like addition are much faster than using loops to iterate over vectors and so unless unrolling exposed a large number of static computations, leaving vectorised operations intact will nearly always be preferable. Since these built-in functions and operators cannot be specialised for particular sized parameters, the inferencing mechanisms often produce data that is not used. However since the partial evaluator has inferred this information, it would be helpful to make it available to tools that can use it. Compilers for other dynamic languages, like Python, can benefit greatly by specifying types. For instance in [56], the Pyrex compiler, with types specified, can create code that runs over 100 times faster. While the MATLAB compilers do try to infer types themselves, adding them to the source code may allow users to tune them where the inference is insufficiently accurate.

Duplication of code and redundant computations both cause performance penalties and unfortunately our partial evaluation technique creates both. It was imperative then that an effective strategy for detecting and removing these barriers was developed. Applying traditional compiler-based techniques such as dead-code elimination using ud-chains allows us to eliminate all code which does not contribute to the final result of a function or to side-effects.

Since our partial evaluator was written in C++, it had to contain an interpreter for MATLAB within it. While much of the complexity of building an interpreter was side-stepped by using the MATLAB run-time libraries, there were still some features which could not be implemented using them and so the interpreter was non-trivial. This introduces scope for bugs in the interpreter leading to imperfect code generation. If a generating extension approach had been taken, the partial evaluator could have been written in C++ but produced MATLAB generating extensions, removing the need for a MATLAB interpreter in the partial evaluator. However, for generating extensions to be useful, they must have many offline decisions taken otherwise the generating extension could have all the complexity of an online partial evaluator built into it. This would ultimately lead to a two-stage approach using binding time analysis, which would move away from our stated aim of producing automated tools that are accessible to non-technical users, since many offline techniques require manual intervention.

Our contributions have been:

- An online partial evaluator for a non-trivial imperative language (MATLAB).
- We have defined abstract domains, which capture useful information about MATLAB arrays, such as their class, type traits and shape characteristics. This used a more complete view of MATLAB types than has been seen in other work on MATLAB.
- We have produced an extensive scheme for inferring type information based on the properties of many built-in functions, operators and array indexing.
- Loops that cannot be unrolled are maintained by calculating the least upper bound of the state on entering the loop. This procedure is recursive and repeated for all nested loops. To avoid repeated computation, we cache least upper bounds for nested loops, even when inside other functions.
- We have produced a post-processing phase which performs dead-code elimination on Abstract Syntax Trees, using ud-chains as well as duplicate function removal by employing structural equivalency detection.
- We have demonstrated the viability of partial evaluation for MATLAB, achieving speed-ups for a variety of codes, with various specifications for the inputs.

10.2 Summary

Progress in computational science and engineering requires a judicious combination of application expertise, algorithm selection and mapping to computational infrastructure. A key underlying technology is the tooling available to computational scientists. High-level languages and advanced problem solving environments have proved to be powerful tools to facilitate rapid and flexible prototyping of new codes in a wide range of application areas.

At their heart of these tools lies the ability to leverage and build upon previously constructed general components and libraries, from which new functionality is composed. In this thesis we have demonstrated how such general components or libraries may be specialised and optimised when used in the context of a particular application or with particular inputs, by exploiting information which is often known in advance but is not usually brought into play. We have shown how to take advantage of this information to deliver improvements in performance. Furthermore this can be done in a way that is transparent and automated from a user's perspective.

We believe that techniques such as partial evaluation and just-in-time compilation/specialisation will allow future generations of computational scientists and engineers to benefit from the productivity gains associated with using high-level generic libraries in advanced problem solving environments, whilst at the same time obtaining high performance.

Appendix A

This appendix describes how to use the MATLAB partial evaluator, MPE. First we give the command line options.

```
Usage: mpe [-a] [-b file] [-c] [-d variable value] [-D] [-f] [-g]
           [-o file] [-p] [-s] file
       mpe -h
       mpe -v
```

If the filename given is '-' then the main source function is read from standard input.

Option	*Explanation*
-a	This option when followed by a filename allows the user to specify a file containing annotations and assignments which will be prepended to the source file before annotation.
-b	This option allows the user to specify a file containing the list of builtins that mpe recognises. If this option is not used the default list is used.
-c	This option suppresses the printing comments describing the attributes of expressions in the output code.
-d	This option causes an assignment to be inserted at the beginning of the initial source file so that parameter values can be specified.
-D	This option causes debug information to printed to standard output.
-f	This option prints a list of functions called by the main function.
-g	This option writes out a file that can be processed by dot to draw a graph of the strongly connected components in the function dependency graph.

Option	*Explanation*
`-h`	This option prints out help text.
`-o`	Specifies a file to which to write the partially evaluated function. Without this option the function is written to standard output.
`-p`	Causes the function to be post-processed before writing it out.
`-s`	This prints out a list of variables used by each function.
`-v`	This prints out the version number of MPE and then exits.
`--precision`	Sets the precision to output floating point numbers.

Now we describe the annotations that can be given to mpe. Annotations always start on a new line and start with '`%#`', which is a MATLAB comment.

There are two types of annotations, ones that are associated with a particular variable and those that relate to the general invocation of a function without being directly tied to a variable. All variable annotations exception for `undefined` automatically set the state of the variable to *defined.*

The variable annotations are of the form `var instruction [value]` and are described in the following table:

Instruction	*Value*	*Explanation*
`size`	Matrix	This annotation sets the shape of the variable in the symbol table. The value field is parsed as an ordinary matrix where the first element gives the number of rows, the second element gives the number of columns and subsequent elements gives the size of any extra dimensions. If any element is `UNKNOWN` then the size of that dimension is left unset.
`ndims`	Scalar	This sets the maximum number of dimensions for a variable.
`widen`	Scalar	This makes the dimension specified by the scalar argument have an unknown value.
`real`	String	If the value string can either be `'yes'` or `'no'` and controls whether the variable is real or complex.
`logical`	String	If the value string can either be `'yes'` or `'no'` and controls whether the variable is logical or not.

Instruction	*Value*	*Explanation*
`class`	String	This declares that the class of a variable. It can be `double`, `char`, `cell`, `struct`, `sparse`, `single`, `int8`, `int16`, `int32`, `uint8`, `uint16`, `uint32` or `function`.
`realdouble`	-	Shorthand way of declaring a variable is real and is of the class double.
`realscalar`	-	Same as `realdouble` but declares the shape to be scalar as well.
`defined`	-	This declares that a variable definitely exists.
`undefined`	-	This declares that a variable definitely does not exist.

The variable annotations are of the form `instruction value` and are described in the following table:

Instruction	*Value*	*Explanation*
`nargin`	Scalar	Sets the value returned by the `nargin` function.
`nargout`	Scalar	Sets the value returned by the `nargout` function.
`unroll`	-	This forces a loop to be unrolled even if it cannot be determined to be possible.
`nounroll`	-	This prevents a loop from being unrolled even if it has been determined to be possible.
`preserve`	String	This prevents the named function from being partially evaluated, but retains it intact in the residual program.
`nointerpret`	String	This prevents the function from being directly interpreted, which is useful for function which produce side-effects.

Appendix B

This appendix gives the source code to some of the functions test in Chapter 8.

```
function y=lagrange(x,pointx,pointy)
%
%LAGRANGE    approx a point-defined function using the Lagrange polynomial
%            interpolation
%
%       LAGRANGE(X,POINTX,POINTY) approx the function defined by the points:
%       P1=(POINTX(1),POINTY(1)),
%       P2=(POINTX(2),POINTY(2)),
%       ...,
%       PN(POINTX(N),POINTY(N))
%       and calculate it in each elements of X
%
%       If POINTX and POINTY have different number of elements the function
%       will return the NaN value
%
%       function wrote by: Calzino
%       7-oct-2001
%
n=size(pointx,2);
L=ones(n,size(x,2));
if (size(pointx,2)~=size(pointy,2))
   fprintf(1,'POINTX and POINTY must have the same number of elements\n');
   y=NaN;
else
   for i=1:n
      for j=1:n
         if (i~=j)
            L(i,:)=L(i,:).*(x-pointx(j))/(pointx(i)-pointx(j));
         end
      end
   end
   y=0;
   for i=1:n
      y=y+pointy(i)*L(i,:);
   end
end
```

LISTING B.1: Langrange Interpolation Code

```
function z=hypergeometric2f1(a,b,c,x,n)
% HYPERGEOMETRIC2F1 Computes the hypergeometric function
% using a series expansion:
%
%    f(a,b;c;x)=
%
%    1 + [ab/1!c]x + [a(a+1)b(b+1)/2!c(c+1)]x^2 +
%    [a(a+1)(a+2)b(b+1)(b+2)/3!c(c+1)(c+2)]x^3 + ...
%
% The series is expanded to n terms
%
% This function solves the Gaussian Hypergeometric Differential Equation:
%
%     x(1-x)y'' + {c-(a+b+1)x}y' - aby = 0
%
% The Hypergeometric function converges only for:
% |x| < 1
% c != 0, -1, -2, -3, ...
%
%
% Comments to:
% Diego Garcia   - d.garcia@ieee.org
% Chuck Mongiovi - mongiovi@fast.net
% June 14, 2002

if nargin ~= 5
    error('Usage: hypergeometric2f1 --> Wrong number of arguments')
end

if (n <= 0 | n ~= floor(n))
    error('Usage: hypergeometric2f1 --> n has to be a positive integer')
end

if (abs(x) > 1)
    error('Usage: hypergeometric2f1 --> |x| has to be less than 1')
end

if (c <= 0 & c == floor(c))
    error('Usage: hypergeometric2f1 --> c != 0, -1, -2, -3, ...')
end

delta = 1;
z = 1;
m = 0;
for m = 1:n-1
   delta = delta .* x .* (a + (m - 1)) .* (b + (m-1)) ./ m ./ (c + (m-1));
   z = z + delta;
end
```

LISTING B.2: Hypergeometric Gaussian Differential Equation Solver Code

```
function clength = length_crit(B, M1, betacrit)

% %# Beta realscalar
% %# M1 realscalar
% %# betacrit realscalar

% Specific heat ratio

G = 1.3;

[c_angle , vrprime_final , VrPrime , dVrPrime] = cone_angle(B, M1);

% Flow properties at surface
T0_T=1+((G-1)/2)*M1^2;
T0=T0_T*147.5;

clength = calc_crit_length(B, M1, c_angle , VrPrime , dVrPrime , ...
                           G, T0, betacrit, 0.00000001);
```

LISTING B.3: Main Function for Mars Lander Code

```
function [theta, s, VrPrime , dVrPrime] = cone_angle(B,M1)

G = 1.3;
MN1 = M1*sin(B);
MN2 = sqrt((MN1^2+(2/(G-1)))/((2*G/(G-1))*MN1^2-1));
Delta = atan(2*cot(B)*((M1^2*(sin(B))^2-1)/(M1^2*(G+cos(2*B))+2)));
M2 = MN2/sin(B-Delta);
VPrime = ((2/((G-1)*M2^2))+1)^(-1/2);
VrPrime = VPrime*cos(B-Delta);
dVrPrime = -VPrime*sin(B-Delta);
thetaspan = [B,pi/180]';

options = odeset('Events', @eventsfun);
[theta, s] = ode45(@rhs, thetaspan, [VrPrime , dVrPrime], options);

theta = theta(end);
s = s(end, 1);
```

LISTING B.4: Auxillary Function for Mars Lander Code (1)

```
function F = rhs(theta, s)

F(1,1) = s(2);
gamma = 1.3;
A = (gamma - 1) / 2;

F(2,1) = (s(2)^2 * s(1) * (1 + 2 * A) + ...
         s(2) * (s(2)^2 * A * cot(theta) - A * cot(theta)) + ...
         s(1) * ( 2 * A * s(1)^2 - 2 * A + A * s(1) * s(2) * cot(theta))) ...
         / (A - A * s(1)^2 - A * s(2)^2 - s(2)^2);
```

LISTING B.5: Auxillary Function for Mars Lander Code (2)

```
function [value, isterminal, direction] = eventsfun(theta, s)

value = s(2);
isterminal = 1;
direction = 0;
```

LISTING B.6: Auxillary Function for Mars Lander Code (3)

```
function clength = calc_crit_length(B, M1, theta, VrPrime, dVrPrime, G, ...
                                  T0, Beta, threshold)

vx0 = M1*((G*191*147.5)^0.5)/((2*860*T0)^0.5);
vy0 = 0;
rybegin = 10^(-6);
t_span = [0 10];

ry1 = rybegin;
vy1 = vx0;
vx1 = vx0;

ry0 = 5*sin(B);
options = odeset('Events', @eventsfun3);

rx1 = ry1 / tan(B);
[t, dfdt] = ode45(@rhs2, t_span, [rx1 ry1 vx1 vy1], options, B, theta, ...
                  VrPrime, dVrPrime, Beta);

rxbegin = dfdt(:,1);
rybegin = dfdt(:,2);

old_clength = ry1;

iterations = 0;
while 1
  fprintf('.')
  iterations = iterations + 1;
  rx0 = ry0 / tan(B);
  [t, dfdt] = ode45(@rhs2, t_span, [rx0 ry0 vx0 vy0], options, B, theta, ...
                    VrPrime, dVrPrime, Beta);

  rx = dfdt(:,1);
  ry = dfdt(:,2);

  lhit = rx(end) * tan(theta);
  lhitbegin = rxbegin(end) * tan(theta);
  h1 = ry(end) - lhit;

  h2 = lhitbegin - rybegin(end);

  clength = ry0 - (ry0 - ry1) / (h1 + h2) * h1;
  if abs(clength - old_clength) < threshold
    break
  end
  ry1 = ry0;
  ry0 = clength;
  rxbegin = rx;
  rybegin = ry;
  old_clength = clength;
end

fprintf('\n')
iterations
```

LISTING B.7: Auxillary Function for Mars Lander Code (4)

```
function [dfdt] = rhs2(t,f, B, cone_angle, VrPrime, dVrPrime, Beta)

rx = f(1);
ry = f(2);
vx = f(3);
vy = f(4);

t = atan(ry / rx);

if B > t + eps
  [theta, s] = ode45(@rhs, linspace(B, t)', [VrPrime dVrPrime]);
  vrprime = s(end, 1);
  vtprime = s(end, 2);
else
  vrprime = VrPrime;
  vtprime = dVrPrime;
end

ux = vrprime * cos(t) + vtprime * sin(t);
uy = vrprime * sin(t) + vtprime * cos(t);

rxprime = vx;
ryprime = vy;
vxprime = -Beta*(vx-ux);
vyprime = -Beta*(vy-uy);

dfdt(1,1) = rxprime;
dfdt(2,1) = ryprime;
dfdt(3,1) = vxprime;
dfdt(4,1) = vyprime;
```

Listing B.8: Auxillary Function for Mars Lander Code (5)

```
function [value, isterminal, direction] = eventsfun3(t, f, B, theta, ...
                                                     VrPrime, dVrPrime, Beta)

value = (f(4)/f(3)) - tan(theta-1e-6);
isterminal = [1];
direction = [0];
```

Listing B.9: Auxillary Function for Mars Lander Code (6)

Bibliography

[1] Alex Aiken and Brian Murphy. Static type inference in a dynamically typed language. In *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 279–290. ACM Press, 1991.

[2] George Almási and David Padua. MaJIC: compiling MATLAB for speed and responsiveness. In *Proceeding of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 294–303, New York, NY, USA, 2002. ACM Press.

[3] Gheorghe Almási. *MaJIC: A MATLAB Just-In-Time Compiler.* PhD thesis, University of Illinois at Urbana-Champaign, 2001.

[4] Helle Markmann Andersen and Ulrik Pagh Schultz. Declarative specialization for object-oriented-program specialization. In *Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 27–38. ACM Press, 2004.

[5] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU, University of Copenhagen, 1994.

[6] Kenichi Asai. Binding-time analysis for both static and dynamic expressions. In *Static Analysis Symposium*, pages 117–133. Springer-Verlag, 1999.

[7] R. Baier, R. Glück, and R. Zöchling. Partial evaluation of numerical programs in Fortran. In *Partial Evaluation and Semantics-Based Program Manipulation, Orlando, Florida, June 1994 (Technical Report 94/9, Department of Computer Science, University of Melbourne)*, pages 119–132, 1994.

[8] Guntis J. Barzdins and Mikhail A. Bulyonkov. Mixed computation and translation: Linearisation and decomposition of compilers, 1988. Preprint 791 from Computing Centre of Siberian divison of USSR Academy of Sciences.

[9] Andrew A. Berlin and Rajeev J. Surati. Partial evaluation for scientific computing: The supercomputer toolkit experience. In *Partial Evaluation and Semantics-Based Program Manipulation, Orlando, Florida, June 1994 (Technical Report 94/9, Department of Computer Science, University of Melbourne)*, pages 133–141, 1994.

[10] Andrew A. Berlin and Daniel Weise. Compiling scientific code using partial evaluation. *IEEE Computer*, 23(12):25–37, 1990.

[11] S. Chirokoff, C. Consel, and R. Marlet. Combining program and data specialization. *Higher-Order and Symbolic Computation*, 12(4):309–335, December 1999.

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

[13] Charles Consel, Luke Hornof, François Noël, Jacques Noyé, and Nicolae Volanschi. A uniform approach for compile-time and run-time specialization. In *Partial Evaluation. International Seminar*, pages 54–72, Dagstuhl Castle, Germany, 12-16 1996. Springer-Verlag, Berlin, Germany.

[14] Patrick Cousot and Rahida Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2-3):103–179, 1992.

[15] Stephen-John Craig and Michael Leuschel. LIX: an effective self-applicable partial evaluator for Prolog. In Yukiyoshi Kameyama and Peter J. Stuckey, editors, *Functional and Logic Programming: 7th International Symposium*, pages 85–99. Springer-Verlag Heidelberg, 2004.

[16] Søren Debois. Imperative program optimization by partial evaluation. In *Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 113–122. ACM Press, 2004.

[17] M.C Duta, M.S Campobasso, M. B Giles, and L.B Lapworth. Adjoint harmonic sensitivities for forced response minimization. In *Proceedings of the ASME International Gas Turbine & Aeroengine Technical Congres, 16-19 June 2003, Atlanta, Georgia*, 2003.

[18] M.C Duta, M. B. Giles, and L. Lafronza. Adjoint sensitivity analysis for aeroacoustic applications. In *Proceedings of the 9th AIAA/CEAS Aeroacoustics Conference and Exhibit 12-14 May, 2003, Hilton Head, South Carolina*, 2003.

[19] eFunda : Hypergeometric Function.
`http://www.efunda.com/math/hypergeometric/hypergeometric.cfm`.

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

[21] Flex – GNU Project. `http://www.gnu.org/software/flex/flex.html`.

[22] Folding@Home Distributed Computing. `http://folding.stanford.edu/`.

[23] Matteo Frigo and Steven G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proc. IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing*, volume 3, pages 1381–1384, Seattle, WA, May 1998.

[24] Carl-Erik Fröberg. *Introduction to Numerical Analysis*. Adison-Wesley, 1965.

[25] Yoshihko Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.

[26] K. Gallivan, B. Marsolf, and E. Gallopoulos. The use of algebraic and structural information in a library prototyping and development environment. In *15th IMACS World Congress on Scientific Computation, Modelling and Applied Mathematics*, volume 4, pages 565–570, 1997.

[27] General Python FAQ. `http://www.python.org/doc/faq/general.html`.

[28] Robert Glück. Jones optimality, binding-time improvements, and the strength of program specializers. In *Proceedings of the ASIAN symposium on Partial evaluation and semantics-based program manipulation*, pages 9–19. ACM Press, 2002.

[29] John Hannan and Patrick Hicks. Higher-order arity raising. In *Proceedings of the third ACM SIGPLAN international conference on Functional programming*, pages 27–38. ACM Press, 1998.

[30] Fritz Henglein. Global tagging optimization by type inference. In *Proceedings of the 1992 ACM conference on LISP and functional programming*, pages 205–215. ACM Press, 1992.

[31] L. Hornof, J. Noyé, and C. Consel. Effective specialization of realistic programs via use sensitivity. In P. Van Hentenryck, editor, *Proceedings of the Fourth International Symposium on Static Analysis, SAS'97*, volume 1302 of *Lecture Notes in Computer Science*, pages 293–314, Paris, France, September 1997. Springer-Verlag.

[32] Pramod G. Joisha. *A Type Inference System for MATLAB with Applications to Code Optimization*. PhD thesis, Electrical and Computer Engineering Department, Northwestern University, 2003.

[33] Pramod G. Joisha and Prithviraj Banerjee. Computing array shapes in MATLAB. In *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, pages 395 – 410. Springer-Verlag, 2001.

[34] Pramod G. Joisha and Prithviraj Banerjee. Correctly detecting intrinsic type errors in typeless languages such as MATLAB. In *Proceedings of the 2001 conference on APL*, pages 7–21. ACM Press, 2001.

[35] Pramod G. Joisha and Prithviraj Banerjee. Static array storage optimization in MATLAB. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 258–268. ACM Press, 2003.

[36] Pramod G. Joisha, Abhay Kanhere, Prithviraj Banerjee, U. Nagaraj Shenoy, and Alok Choudhary. Handling context-sensitive syntactic issues in the design of a front-end for a MATLAB compiler. In *Proceedings of the ACM SIGAPL International Conference on Array Processing Languages (APL)*, Berlin, Germany, 2000. Technischen Universitat.

[37] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, 1993.

[38] Jesper Jørgensen. Similix: A self-applicable partial evaluator for scheme. In *Partial Evaluation - Practice and Theory, DIKU 1998 International Summer School*, pages 83–107. Springer-Verlag, 1999.

[39] Morry Katz and Daniel Weise. Towards a new perspective on partial evaluation. In *Proceedings of the Workshop on Partial Evaluation and Semantics-Based Program Manipulation '92*, pages 29–36, 1992.

[40] David Kincaid and Ward Cheney. *Numerical Analysis.* Brooks/Cole Publishing Company, 1996.

[41] Paul Kleinrubatscher, Albert Kriegshaber, Robert Zöchling, and Robert Glück. Fortran program specialization. In Gregor Snelting and Uwe Meyer, editors, *Semantikgestützte Analyse, Entwicklung und Generierung von Programmen. GI Workshop*, pages 45–54, Schloss Rauischholzhausen, Germany, 1994. Justus-Liebig-Universität Giessen.

[42] Todd B. Knoblock and Erik Ruf. Data specialization. In *Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation*, pages 215–225. ACM Press, 1996.

[43] Jens Knoop, Oliver Ruthing, and Bernhard Steffen. Partial dead code elimination. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 147–158, 1994.

[44] LAPACK – Linear Alegbra PACKage. `http://www.netlib.org/lapack/`.

[45] Michael Leuschel and Maurice Bruynooghe. Logic program specialisation through partial deduction: Control issues. *Theory and Practice of Logic Programming*, 2(4 & 5):461–515, July & September 2002.

[46] Maple Home – Adept Scientific plc.
`http://www.adeptscience.com/products/mathsim/maple/`.

[47] Mathematica: The Way the World Calculates.
`http://www.wolfram.com/products/mathematica/index.html`.

[48] Vijay Menon and Keshav Pingali. A case for source-level transformations in MATLAB. In *Domain-Specific Languages*, pages 53–65, 1999.

[49] Vijay Menon and Keshav Pingali. High-level semantic optimization of numerical codes. In *International Conference on Supercomputing*, pages 434–443, 1999.

[50] Vijay Menon and Anne E. Trefethen. MultiMATLAB: Integrating Matlab with high performance parallel computing. In *Supercomputing '97 ACM SIGARCH and IEEE Computer Society*, pages 1–18, 1997.

[51] Torben Æ. Mogensen and Peter Sestoft. Partial evaluation. In Allen Kent and James G. Williams, editors, *Encyclopedia of Computer Science and Technology*, volume 37, pages 247–279. Marcel Dekker, 270 Madison Avenue, New York, New York 10016, 1997.

[52] Steven Muchnick. *Advanced Compiler Design & Implementation.* Morgan Kaufmann Publishers, 1997.

[53] S. Pawletta, T. Pawletta, W. Drewelow, P. Duenow, and M. Suesse. A MATLAB toolbox for distributed and parallel processing. In Moler C. and S. Little, editors, Proc. of the Matlab Conference 95, Cambridge, MA. MathWorks Inc., October 1995.

[54] M. Quinn, A. Malishevsky, N. Seelam, and Y. Zhao. Preliminary results from a parallel MATLAB compiler. In *International Parallel Processing Symposium*, pages 81–87. IEEE CS Press, 1998.

[55] Thomas W. Reps and Todd Turnidge. Program specialization via program slicing. In *Selected Papers from the International Seminar on Partial Evaluation*, pages 409–429. Springer-Verlag, 1996.

[56] Armin Rigo. Representation-based just-in-time specialization and the Psyco prototype for Python. In *Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 15–26. ACM Press, 2004.

[57] Luiz De Rose. Compiler techniques for MATLAB programs. Technical Report UIUCDCS-R-96-1956, University of Illinois at Urbana-Champaign, 1996.

[58] Luiz De Rose, Kyle Gallivan, Efstratios Gallopoulos, Bret A. Marsolf, and David A. Padua. FALCON: A MATLAB interactive restructuring compiler. In *Languages and Compilers for Parallel Computing*, pages 269–288. Springer, 1995.

[59] Erik Ruf and Daniel Weise. Using types to avoid redundant specialization. In *Proceedings of the 1991 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 321–333. ACM Press, 1991.

[60] Erik Ruf and Daniel Weise. Improving the accuracy of higher-order specialization using control flow analysis. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Directed Program Manipulation*, pages 67–74, 1992.

[61] Erik Ruf and Daniel Weise. Preserving information during online partial evaluation. Technical report, Stanford Computer Science Laboratory, 1992. CSL-TR-92-516.

[62] Erik Steven Ruf. *Topics in online partial evaluation.* PhD thesis, Stanford University, 1993.

[63] D. Sahlin. *An Automatic Partial Evaluator for Full Prolog.* PhD thesis, The Royal Institute of Technology (KTH) Stockholm, Sweden, 1991.

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

[65] Ulrik P. Schultz, Julia L. Lawall, Charles Consel, and Gilles Muller. Towards automatic specialization of Java programs. In *European Conference on Object-oriented Programming*, volume 1628 of *Lecture Notes in Computer Science*, pages 367–390, 1999.

[66] Michael Sperber. Self-applicable online partial evaluation. In *Selected Papers from the Internaltional Seminar on Partial Evaluation*, pages 465–480. Springer-Verlag, 1996.

[67] Michael Sperber, Robert Glück, and Peter Thiemann. Bootstrapping higher-order program transformers from interpreters. In *Proceedings of the 1996 ACM symposium on Applied Computing*, pages 408–413. ACM Press, 1996.

[68] Michael Sperber, Peter Thiemann, and Hervert Klaeren. Distributed partial evaluation. In *Proceedings of the second international symposium on Parallel symbolic computation*, pages 80–87. ACM Press, 1997.

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

[70] Rajeev J. Surati and Andrew A. Berlin. Exploiting the parallelism exposed by partial evaluation. In *IFIP PACT*, pages 181–192, 1994.

[71] The MathWorks – About the MathWorks.
`http://www.mathworks.com/company/aboutus/index.html`.

[72] The MathWorks – About Us.
`http://web.archive.org/web/20000815202330/www.mathworks.com/company/aboutus.shtml`.

[73] The Mathworks – Industries.
`http://www.mathworks.com/industries/`.

[74] The Mathworks - MATLAB Compiler.
`http://www.mathworks.com/products/compiler/`.

[75] The MathWorks, Inc. `http://www.mathworks.com/`.

[76] Peter Thiemann and Michael Sperber. Polyvariant expansion and compiler generators. In *Proceedings of the Second International Andrei Ershov Memorial Conference on Perspectives of System Informatics*, pages 285–296. Springer-Verlag, 1996.

[77] Valentin F. Turchin. The concept of a supercompiler. *ACM Trans. Program. Lang. Syst.*, 8(3):292–325, 1986.

[78] Daniel Weise, Roland Conybeare, Erik Ruf, and Scott Seligman. Automatic online partial evaluation. In *Proceedings of the 5th ACM conference on Functional programming languages and computer architecture*, pages 165–191. Springer-Verlag New York, Inc., 1991.

[79] Eric W. Weisstein. *Lagrange Interpolating Polynomial.* From MathWorld–A Wolfram Web Resource.
`http://mathworld.wolfram.com/LagrangeInterpolatingPolynomial.html`.

[80] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1-2):3–35, January 2001.