Presenting the Lazy Evaluation of Functions

Jonathan Paul Taylor
Queen Mary and Westfield College

Thesis submitted for the degree of Doctor of Philosophy
Abstract

This thesis explores the provision of information about the evaluation of expressions written in a lazy functional language.

The impact of lazy evaluation on both the users of a functional language, and on the design of programmer aids is described.

A taxonomy of user's errors is combined with a review of currently available approaches to the provision of information to users faced with errors. This review indicates that many of the available approaches provide information aimed specifically at certain kinds of error, and that only the rewriting approach seems suitable as a basis for a system which provides many different views of an evaluation. A review of currently available systems is presented, and shows that current rewriting systems lack a number of vital facilities.

Prospero, a system based on rewriting, is described in terms of: an evaluation model; facilities with which to control the information presented; and facilities with which to move through an evaluation, and search for evaluation 'events'.

Prospero's evaluation model is designed to provide low-level information about certain aspects of an evaluation. A description of this model, together with a discussion of its equivalence to other models is provided.

The information presented to the user is controlled through a series of user defined 'filters'. A description of the filtering system together with examples of its use is presented.

The design of facilities which would enable users to search an evaluation for stages with specific properties is provided. The facilities allow the user to eliminate the danger of searches causing 'non lazy' evaluation to take place. Additionally, the user is provided with control over the amount of evaluation carried out by searches.

Finally, conclusions of the work are presented along with a description of work which would naturally follow on from that presented here.
# Table of Contents

1. **Introduction** ................................................................................. 15
   1.1. Users and Their Needs ................................................................. 16
   1.2. Lazy Functional Languages ......................................................... 17
   1.3. Function Definitions and Lazy Evaluation ...................................... 18
   1.4. The Impact of Lazy Evaluation on Debugging ................................. 20
       1.4.1 The 'Pen and Paper' Approach to Debugging .......................... 20
       1.4.2 Impact of Lazy Evaluation on Debugging Systems .................. 23
          Introducing Strictness into an Evaluation .................................. 25
   1.5. Models and Views ...................................................................... 26
   1.6. Prospero ....................................................................................... 29
   1.7. Aims and Structure of the Thesis .................................................. 30
   1.8. State of Implementation .............................................................. 33

2. **Approaches to Providing Information** ............................................ 34
   2.1. Types of Error ............................................................................ 34
       2.1.1 Wrong Answer ...................................................................... 35
       2.1.2 Non-Termination .................................................................. 35
       2.1.3 Application to Arguments Outside a Function’s Domain .......... 35
       2.1.4 Space and Time ................................................................... 35
   2.2. Approaches to Providing Information about Errors ....................... 36
       2.2.1 Rewriting ............................................................................. 36
       2.2.2 Assertions ............................................................................ 36
       2.2.3 Algorithmic Debugging .......................................................... 36
       2.2.4 Profiling ............................................................................... 37
       2.2.5 Abstract Interpretation ............................................................ 37
   2.3. Help with Errors ......................................................................... 37
       2.3.1 Wrong Answer ...................................................................... 37
       Presentation Using Rewriting .......................................................... 37
       An Algorithmic Approach ............................................................... 39
       Assertions .................................................................................... 40
       Summary ..................................................................................... 42
       2.3.2 Non-Termination .................................................................. 42
       Rewriting ..................................................................................... 42
       Assertions .................................................................................... 43
       Summary ..................................................................................... 45
       2.3.3 Application to Arguments Outside a Function’s Domain .......... 45
       Using Rewriting to Show Pattern Matching and Guard Evaluation .... 45
       Using Static Analysis to Show the Domain ...................................... 46
       Summary ..................................................................................... 47
       2.3.4 Space and Time Efficiency ....................................................... 47
       Rewriting ..................................................................................... 47
       Profiling ..................................................................................... 49
       Complexity Analysis ..................................................................... 49
       Summary ..................................................................................... 49
   2.4. Help with Teaching ..................................................................... 50
       2.4.1 Teaching Methods .................................................................. 50
       2.4.2 Range of Teaching Material .................................................... 51
       2.4.3 Explaining Laziness ................................................................. 51
       2.4.4 Explaining Pattern Matching .................................................. 52
5. Filters - Presenting Evaluation Information .........................136
5.1 What is Filtering? ..........................................................137
5.2 How is Filtering Done? ....................................................137
  5.2.1 Augmented Reduction Graphs ..................................137
  5.2.2 Simple Filters .......................................................138
    The Components of a Simple Filter ..................................139
  5.2.3 Composition of Simple Filters ..................................140
  5.2.4 Temporal Filters ...................................................140
    When Simple Filtering is not Enough ..............................140
    The Components of a Temporal Filter ..............................141
5.3 Why is Filtering Necessary? ..........................................142
  5.3.1 Removal of Low Level Evaluation Detail ......................142
  5.3.2 Removal of Value Details ........................................143
    The list-contents filter .............................................143
    Filtering and user-defined constructors .......................144
  5.3.3 Removal of Evaluation Detail ...................................148
    Support for Functional Abstraction ...............................148
    Focusing on Specific Functions .................................154
  5.3.4 Supporting Abstract Data Types and Type Synonyms .......155
    Using Temporal Filters to Support Data .........................156
    Abstraction .........................................................156
    Supporting the Pedagogical Use of ADTs .........................157
5.3.5 Summary ....................................................................160
5.4 Filter Definition and the Filter Mechanisms ......................160
  5.4.1 Simple Filters .......................................................161
    Defining the Operator Filter .....................................162
    Defining op_filter Using Filter Composition ..................164
    Preventing Overwriting .............................................165
  5.4.2 Defining Temporal Filters .......................................165
    The Temporal Filter Mechanism ....................................169
5.5 Summary .......................................................................170
6. Moving Through an Evaluation .........................................168
6.1 Generating the Evaluation History ..................................172
6.2 Single Stepping ............................................................173
6.3 Searching an Evaluation History .....................................174
  6.3.1 Unconditional Breakpoints .....................................175
  6.3.2 Conditional Breakpoints ........................................176
  6.3.3 Post-Conditional Breakpoints .................................176
6.4 Prospero's Searches ....................................................176
  6.4.1 The Goals of the Search System ...............................176
  6.4.2 What is a search? ..................................................177
  6.4.3 The Three Classes of Search ...................................178
    The Eager search ....................................................179
    The Lazy search .....................................................180
    The Patient search ................................................180
  6.4.4 How Searches are Described ....................................180
    All three classes of searches have the same type: ............181
    where the elements of this pair have the types: ..............181
  6.4.5 The Search Mechanism ............................................182
    The Eager Search ....................................................183
    The Lazy Search ...................................................184
    Patient Search .....................................................185
  6.4.6 Worked Example ....................................................186
    The Eager Search ....................................................186
The Lazy Search ........................................... 187
The Patient Search ....................................... 188

6.4.7 Satisfying the Goals of the Search System ...... 190
  Continuation after a Search ................................... 190
  Source of the Expression of Interest ....................... 190
  Termination Strictness Only ................................ 190
  Less Introduction of Strictness than error ................ 190
  Dealing with the Results of Evaluations .................... 191
  No Change of Truth of an Assertion ......................... 191

6.5 Summary .................................................. 191

7. Evaluating Expressions ................................... 192

7.1 Driving a Reduction ................................... 194
  7.1.1 Variables ........................................... 194
  7.1.2 Constants .......................................... 195
  7.1.3 Unary Operators .................................... 195
  7.1.4 Binary Operators ................................... 195
  7.1.5 Function Application ................................ 196
  7.1.6 Constructor Application ............................. 196
  7.1.7 Guards ............................................. 197
  7.1.8 Pattern Matching .................................... 198
    Pattern Matching Variables .............................. 199
    Pattern Matching Constants .............................. 199
    Pattern Matching Sum Types ............................. 200
    Pattern Matching Product Types ........................ 201
    Pattern Matching Tuples ................................ 201
    Pattern Matched Lambda (l) ............................. 202
    Pattern Failed Lambda (--) ............................ 202
    The ThinBar Operator: Representing Functions with a Number of Definitions .......................... 202
  7.1.9 Design of the Pattern Matching Mechanism ......... 204
    Fatbar .................................................. 204
    Equivalence of Prospero and Peyton Jones’ Mechanism ...................................................... 205
    An Alternative Thinbar .................................... 206
    Case Statements .......................................... 208

7.2 Summary .................................................. 209

8. Prospero’s Interface ................................... 210

8.1 Interface Implementation Details ..................... 210
  8.1.1 Interaction using Lazily Evaluated Lists .......... 211
  8.1.2 The Virtual Display ................................ 213
    What is the Virtual Display? ............................. 213
  8.1.3 The Smalltalk Interface ............................. 215
    The View ................................................ 215
    The Model .............................................. 216
  8.1.4 An Alternative Graph Display Technique .......... 218

8.2 Classifying Prospero using a Taxonomy of Software Visualisation ........................ 219
  8.2.1 Classification Summary .............................. 225
    Scope .................................................... 226
    Content ................................................ 226
    Form ..................................................... 226
    Method .................................................. 226
    Interaction .............................................. 226
    Effectiveness ......................................... 227

8.3 Summary .................................................. 227
# Table of Contents

1. **Introduction** .......................................................... 15  
   1.1. Users and Their Needs ............................................. 16  
   1.2. Lazy Functional Languages ....................................... 17  
   1.3. Function Definitions and Lazy Evaluation .................... 18  
   1.4. The Impact of Lazy Evaluation on Debugging .................. 20  
       1.4.1 The ‘Pen and Paper’ Approach to Debugging .............. 20  
       1.4.2 Impact of Lazy Evaluation on Debugging Systems ....... 23  
       Introducing Strictness into an Evaluation .................... 25  
   1.5. Models and Views ............................................... 26  
   1.6. Prospero ...................................................... 29  
   1.7. Aims and Structure of the Thesis ............................. 30  
   1.8. State of Implementation ....................................... 33  

2. **Approaches to Providing Information** ............................ 34  
   2.1. Types of Error .................................................. 34  
       2.1.1 Wrong Answer .............................................. 35  
       2.1.2 Non-Termination .......................................... 35  
       2.1.3 Application to Arguments Outside a Function’s Domain . 35  
       2.1.4 Space and Time ............................................ 35  
   2.2. Approaches to Providing Information about Errors ........... 36  
       2.2.1 Rewriting .................................................. 36  
       2.2.2 Assertions ................................................ 36  
       2.2.3 Algorithmic Debugging .................................... 36  
       2.2.4 Profiling .................................................. 37  
       2.2.5 Abstract Interpretation ................................... 37  
   2.3. Help with Errors ............................................... 37  
       2.3.1 Wrong Answer .............................................. 37  
       Presentation Using Rewriting ................................... 37  
       An Algorithmic Approach ....................................... 39  
       Assertions ...................................................... 40  
       Summary ........................................................ 42  
       2.3.2 Non-Termination .......................................... 42  
       Rewriting ....................................................... 42  
       Assertions ...................................................... 45  
       Summary ........................................................ 45  
   2.3.3 Application to Arguments Outside a Function’s Domain .... 45  
       Using Rewriting to Show Pattern Matching and ....... 45  
       Guard Evaluation ............................................ 45  
       Using Static Analysis to Show the Domain .................. 46  
       Summary ........................................................ 47  
   2.3.4 Space and Time Efficiency ................................... 47  
       Rewriting ....................................................... 47  
       Profiling ....................................................... 49  
       Complexity Analysis .......................................... 49  
       Summary ........................................................ 49  
   2.4. Help with Teaching ............................................. 50  
       2.4.1 Teaching Methods .......................................... 50  
       2.4.2 Range of Teaching Material ................................ 51  
       2.4.3 Explaining Laziness ....................................... 51  
       2.4.4 Explaining Pattern Matching ............................. 52
# Table of Contents

1. Introduction .................................................. 15
   1.1. Users and Their Needs ..................................... 16
   1.2. Lazy Functional Languages ................................. 17
   1.3. Function Definitions and Lazy Evaluation ............... 18
   1.4. The Impact of Lazy Evaluation on Debugging ............ 20
       1.4.1 The ‘Pen and Paper’ Approach to Debugging ......... 20
       1.4.2 Impact of Lazy Evaluation on Debugging Systems ... 23
       Introducing Strictness into an Evaluation ................. 25
   1.5. Models and Views ........................................... 26
   1.6. Prospero .................................................... 29
   1.7. Aims and Structure of the Thesis .......................... 30
   1.8. State of Implementation ................................... 33

2. Approaches to Providing Information .......................... 34
   2.1. Types of Error ............................................. 34
       2.1.1 Wrong Answer ......................................... 35
       2.1.2 Non-Termination ....................................... 35
       2.1.3 Application to Arguments Outside a Function’s Domain 35
       2.1.4 Space and Time ....................................... 35
   2.2. Approaches to Providing Information about Errors ...... 36
       2.2.1 Rewriting .............................................. 36
       2.2.2 Assertions ............................................ 36
       2.2.3 Algorithmic Debugging ................................ 36
       2.2.4 Profiling ............................................. 37
       2.2.5 Abstract Interpretation ................................ 37
   2.3. Help with Errors ........................................... 37
       2.3.1 Wrong Answer ......................................... 37
           Presentation Using Rewriting ............................ 37
           An Algorithmic Approach ............................... 39
           Assertions ............................................. 40
           Summary ............................................... 42
       2.3.2 Non-Termination ....................................... 42
           Rewriting ............................................. 42
           Assertions ............................................. 45
           Summary ............................................... 45
       2.3.3 Application to Arguments Outside a Function’s Domain 45
           Using Rewriting to Show Pattern Matching and Guard Evaluation ..................... 45
           Using Static Analysis to Show the Domain ............. 46
           Summary ............................................... 47
       2.3.4 Space and Time Efficiency .............................. 47
           Rewriting ............................................. 47
           Profiling ............................................. 49
           Complexity Analysis ................................. 49
           Summary ............................................... 49
   2.4. Help with Teaching ......................................... 50
       2.4.1 Teaching Methods ..................................... 50
       2.4.2 Range of Teaching Material ............................ 51
       2.4.3 Explaining Laziness .................................. 51
       2.4.4 Explaining Pattern Matching .......................... 52
3. **Existing Techniques and Systems** ............................................. 57
   3.1 Print Statements ........................................................................ 57
   3.2 Shadow Parameters .................................................................. 59
   3.3 Problems of Using the error Function ................................. 63
      3.3.1 The error Function Terminates an Evaluation ............ 63
      3.3.2 Introduction of Strictness .............................................. 64
      3.3.3 Finding the Source of Errors ......................................... 65
      3.3.4 Post-Conditional Use of error ......................................... 65
   3.4 Information Through Snapshots .............................................. 67
      3.4.1 Fly ............................................................................. 68
      3.4.2 Glide .......................................................................... 69
   3.5 Efficiency Information Through Profilers ............................ 69
   3.6 Algorithmic Debuggers ............................................................ 71
   3.7 Dataflow .............................................................................. 71
   3.8 Information Through Rewriting ............................................. 71
      3.8.1 Potentially High Levels of Information ....................... 72
      3.8.2 Too Many Steps ................................................................ 72
      3.8.3 Presenting Rewriting Information ................................. 75
      3.8.4 The Miranda Calculator .............................................. 77
      3.8.5 Hint ........................................................................... 79
   3.9 Strengths and Weaknesses ....................................................... 79
      3.9.1 Print Statements ............................................................. 80
      3.9.2 Shadow Parameters ....................................................... 80
      3.9.3 The error Function ......................................................... 81
      3.9.4 Snapshots ..................................................................... 81
      3.9.5 Profiling ...................................................................... 81
      3.9.6 Algorithmic Debuggers ............................................... 81
      3.9.7 Dataflow ..................................................................... 82
      3.9.8 Rewriting Systems ....................................................... 82
   3.10 Summary .............................................................................. 83

4. **A Tour of Prospero** ............................................................... 85
   4.1 Interface Description ............................................................... 86
   4.2 Scenarios where Prospero would Help with Errors ............. 88
      4.2.1 Wrong Answer ............................................................... 88
      4.2.2 Controlling Step Granularity - Filters and Searches .... 93
      4.2.3 Assertions using Searches ............................................ 99
      4.2.4 Non-Termination .......................................................... 101
      4.2.5 Application to Arguments Outside a Function's Domain 105
      4.2.6 Space/Time Efficiency .................................................. 118
   4.3 Scenarios where Prospero would Help with Teaching ........ 123
      4.3.1 Explaining Laziness ...................................................... 124
      4.3.2 Explaining Pattern Matching ...................................... 131
      4.3.3 Explaining Higher-order Functions ............................. 131
      4.3.4 Illustrating Algorithms .............................................. 134
   4.4 Summary .............................................................................. 134
5. Filters - Presenting Evaluation Information .................. 136

5.1 What is Filtering? ................................................. 137
5.2 How is Filtering Done? .......................................... 137
  5.2.1 Augmented Reduction Graphs ............................. 137
  5.2.2 Simple Filters .............................................. 138
      The Components of a Simple Filter ...................... 139
  5.2.3 Composition of Simple Filters ......................... 140
  5.2.4 Temporal Filters .......................................... 140
      When Simple Filtering is not Enough .................. 140
      The Components of a Temporal Filter ................. 141

5.3 Why is Filtering Necessary? .................................. 142
  5.3.1 Removal of Low Level Evaluation Detail .............. 142
  5.3.2 Removal of Value Details ................................ 143
      The list-contents filter ................................ 143
      Filtering and user-defined constructors ............. 144
  5.3.3 Removal of Evaluation Detail ......................... 148
      Support for Functional Abstraction .................... 148
      Focusing on Specific Functions ......................... 154
  5.3.4 Supporting Abstract Data Types and Type Synonyms ... 155
      Using Temporal Filters to Support Data Abstraction 156
      Supporting the Pedagogical Use of ADTs ............... 157

5.3.5 Summary ................................................... 160

5.4 Filter Definition and the Filter Mechanisms ................. 160
  5.4.1 Simple Filters ............................................ 161
      Defining the Operator Filter ............................. 162
      Defining operator Using Filter Composition ........... 164
      Preventing Overwriting ................................ 165
  5.4.2 Defining Temporal Filters ............................... 165
      The Temporal Filter Mechanism ......................... 169

5.5 Summary ...................................................... 170

6. Moving Through an Evaluation .................................. 168

6.1 Generating the Evaluation History ......................... 172
6.2 Single Stepping ............................................. 173

6.3 Searching an Evaluation History ............................ 174
  6.3.1 Unconditional Breakpoints ............................. 175
  6.3.2 Conditional Breakpoints ............................... 176
  6.3.3 Post-Conditional Breakpoints ......................... 176

6.4 Prospero's Searches ........................................ 176
  6.4.1 The Goals of the Search System ...................... 176
  6.4.2 What is a search? ...................................... 177
  6.4.3 The Three Classes of Search .......................... 178
      The Eager search ....................................... 179
      The Lazy search ....................................... 180
      The Patient search .................................. 180
  6.4.4 How Searches are Described ........................... 180
      All three classes of searches have the same type: .. 181
      where the elements of this pair have the types: .. 181

6.4.5 The Search Mechanism .................................... 182
  The Eager Search ........................................... 183
  The Lazy Search ............................................ 184
  Patient Search ............................................. 185

6.4.6 Worked Example .......................................... 186
  The Eager Search ........................................... 186
The Lazy Search .................................................. 187
The Patient Search .............................................. 188
6.4.7 Satisfying the Goals of the Search System ............. 190
  Continuation after a Search .................................. 190
  Source of the Expression of Interest ......................... 190
  Termination Strictness Only .................................. 190
  Less Introduction of Strictness than error .................... 190
  Dealing with the Results of Evaluations ...................... 191
  No Change of Truth of an Assertion ......................... 191
6.5 Summary ...................................................... 191

7. Evaluating Expressions ........................................ 192
  7.1 Driving a Reduction ........................................ 194
  7.1.1 Variables ................................................. 194
  7.1.2 Constants ................................................ 195
  7.1.3 Unary Operators ........................................ 195
  7.1.4 Binary Operators ....................................... 195
  7.1.5 Function Application .................................... 196
  7.1.6 Constructor Application ................................ 196
  7.1.7 Guards .................................................. 197
  7.1.8 Pattern Matching ........................................ 198
    Pattern Matching Variables ................................ 199
    Pattern Matching Constants ................................ 199
    Pattern Matching Sum Types ................................. 200
    Pattern Matching Product Types ............................ 201
    Pattern Matching Tuples .................................... 201
    Pattern Matched Lambda (l) ................................ 202
    Pattern Failed Lambda (→) .................................. 202
    The ThinBar Operator: Representing Functions with a Number of Definitions 202
  7.1.9 Design of the Pattern Matching Mechanism ............... 204
    Fatbar ....................................................... 204
    Equivalence of Prospero and Peyton Jones’ Mechanism .... 205
    An Alternative Thinbar ...................................... 206
    Case Statements ............................................. 208
7.2 Summary ...................................................... 209

8. Prospero’s Interface .......................................... 210
  8.1 Interface Implementation Details ........................... 210
  8.1.1 Interaction using Lazily Evaluated Lists ............... 211
  8.1.2 The Virtual Display ..................................... 213
    What is the Virtual Display? ................................. 213
  8.1.3 The Smalltalk Interface ................................ 215
    The View ................................................... 215
    The Model .................................................. 216
  8.1.4 An Alternative Graph Display Technique ............... 218
  8.2 Classifying Prospero using a Taxonomy of Software Visualisation 219
  8.2.1 Classification Summary ................................ 225
    Scope ....................................................... 226
    Content ................................................... 226
    Form ....................................................... 226
    Method .................................................... 226
    Interaction ................................................ 226
    Effectiveness .............................................. 227
8.3 Summary ...................................................... 227
9. Conclusions and Future Work ........................................ 228

9.1 Future Work .......................................................... 230

9.1.1 Experimental Evaluation ........................................ 231
   Evaluation of Prospero as a Teaching Tool ..................... 231
   Evaluation of Prospero as an Error Finding Tool ............. 232

9.1.2 Filters .................................................................. 232
   Defining Filters ...................................................... 233
   User-Definition of Filters ......................................... 233

9.1.3 Searches ................................................................ 235

9.1.4 Interface .............................................................. 235
   More Adventurous Graphics ....................................... 235
   Explanation of redux and reduction rules ..................... 236
   Spatial Coherence .................................................... 236
   Evaluation Stages in Context ...................................... 236

9.1.5 Improvements to the Implementation of the Interpreter .... 236
   Compilation of Hidden Evaluation .............................. 237
   Implementation Using a Functional Language Compiler ... 237

9.1.6 Scalability ............................................................ 237
   Filtering ................................................................. 237
   Searches and Evaluation History ................................. 238

9.1.7 Future Work Summary ............................................ 238

9.2 Concluding Remarks .................................................. 239

References ................................................................. 240

APPENDIX A ............................................................... 245

APPENDIX B ............................................................... 247

Current Implementation State .......................................... 247

1. Language Implementation ........................................... 247
2. Filter System .......................................................... 247
3. Navigation .............................................................. 247

APPENDIX C ............................................................... 248

Introduction to Miranda .................................................. 248

Writing Definitions ....................................................... 248
Defining Functions ....................................................... 249
Evaluating Expressions in an Environment ......................... 250
Defining Types ................................................................ 250
List of Figures

Figure 2.1(a) Definition of function tree_member containing an error; (b) a trace of the function highlighting the error; (c) the correct definition of tree_member ................................................................. p38

Figure 2.2 (a) the definition of a tree insertion function containing the erroneous definition of tree_member defined in Figure 2.1(a); (b) the use of an algorithmic debugger to locate the error ......................................................... p39

Figure 2.3 (a) definition of the function funny_reverse containing an error; (b) transcript of Miranda session involving evaluation of the function; (c) an assertion used to locate the error in the function; (d) the corrected version of the function ................................................................. p41

Figure 2.4(a) definition of $\xi$ the evaluation of which may not terminate; (b) an example non-terminating evaluation......................................................... p42

Figure 2.5 (a) the non-terminating function $\varrho$; (b) a non-termination evaluation involving the function ................................................................. p43

Figure 2.6 The evaluation of a function annotated with distance from base case ................................................................. p43

Figure 2.7 Extract from a real-life non terminating set of definitions............ p44

Figure 2.8 (a) definition of a function $\xi$ and its domain; (b) definition and domain of a second function $\xi$ ................................................................. p46

Figure 2.9 (a) the definition of the function slow_reverse; (b) the rewriting of an application of slow_reverse; (c) the definition of the function quick_reverse; (b) the rewriting of an application of quick_reverse ................................................................. p48

Figure 2.10 (a) definitions of functions used to demonstrate lazy evaluation; (b) a trace of an evaluation involving these functions which illustrates lazy evaluation in progress .................................................................. p51

Figure 2.11 definition of function $\xi$ which is strict in both of its arguments ................................................................................. p52
Figure 2.12 (a) definitions of functions used to demonstrate the use of higher-order function; (b) a trace of an evaluation involving these functions which illustrates the use of higher-order functions....................... p53

Figure 3.1 Session of a Miranda system enriched with a side-effecting print function................................................................. p59

Figure 3.2 (a) definition of the $\text{hd}$ function; (b) the function incorporating a print function......................................................... p59

Figure 3.3 (a) definition of the function $\text{negate}$; (b) new definition with a shadow parameter; (c) example evaluation with shadow parameter. ........ p60

Figure 3.4 (a) definition of the function $\text{last}$; (b) new definition with a shadow parameter; (c) example evaluation with shadow parameter. ...... p61

Figure 3.5 (a) definition of the function $\text{negate}_\text{last}$; (b) new definition with a shadow parameter; (c) example evaluation with shadow parameter. .... p62

Figure 3.6 (a) definition of the function $\text{first}$; (b) new definition with a shadow parameter................................................................. p62

Figure 3.7 Definition of function $\text{hd}$ using Miranda's $\text{error}$ function........ p63

Figure 3.8 Definition of recursive function $f$................................................. p64

Figure 3.9 Definition of function $f$ showing adaptation necessary to catch non-recursive applications of $f$ to the empty list......................... p64

Figure 3.10 Introduction of strictness into definition of function $g$ when $\text{error}$ is used................................................................. p65

Figure 3.11 Introduction of strictness into definition of function $f$ when $\text{error}$ is used to display arguments of the function ............................... p65

Figure 3.12 Change of truth of assertion made using $\text{error}$......................... p66

Figure 3.13 Example of the use of $\text{Trace}$ in the Fly system....................... p68

Figure 3.14 Example of the use of $\text{Force}$ in the Fly system....................... p69
Figure 4.12 Evaluation of an application of \( \varepsilon \) defined in Figure 4.10........p102

Figure 4.13 Evaluation of an application of \( \varepsilon \) defined in Figure 4.10,
shown using a distance function .............................................p103

Figure 4.14 A non- terminating function (duplication of Figure 2.5(a))........p105

Figure 4.15 Definition of the function \textit{isA} ................................p106

Figure 4.16 Evaluation of expression involving function \textit{isA} ..........p106

Figure 4.17 Definition of function \textit{factorial}..............................p113

Figure 4.18 Evaluation of \textit{factorial} illustrating the representation of
guards..........................................................................................p114

Figure 4.19 Definition of the functions \textit{slowReverse} and \textit{quickReverse}
(duplication of Figure 2.9(a) and (b)).........................................p119

Figure 4.20 Evaluation of expression involving the function
\textit{slowReverse}........................................................................p119

Figure 4.21 Evaluation of expression involving the function
\textit{quickReverse}......................................................................p122

Figure 4.22 Definitions used to illustrate the presentation of lazy
evaluation (duplication of Figure 2.10(a))....................................p124

Figure 4.23 Evaluation of the expression \textit{double2 bottom v}, illustrating
the explanation of laziness.............................................................p124

Figure 4.24 Definitions used to illustrate the presentation of higher order
functions (duplication of Figure 2.11(a))......................................p131

Figure 4.25 Evaluation illustrating the presentation of higher order
functions ......................................................................................p132

Figure 5.1 Overview of Prospero..................................................p137

Figure 5.2 The Prospero viewing pipeline......................................p138

Figure 5.3 Unfiltered representation of \([1, 2, 3]\).............................p143

Figure 5.4 Example of the list \([1,2,3]\) filtered by the list-contents filter ....p144
Acknowledgements

Thanks to Richard Bornat and Keith Clarke for their work as supervisors of this Ph.D.

I would like to thank all of the staff in the Computer Science department at Queen Mary and Westfield for their help during my time there and for making that time enjoyable both academically and socially. In particular I would like to thank Adam Billyard and Paddy Toal for their support, energy, friendship and bar-table debates; I also thank Eliot Miranda for all of those things together with his Smalltalk help and advice. A very special thanks to Doug Goldson, a special friend who’s work as a critic and proof reader went far beyond the call of duty.

I would also like to thank the members of the Computer Science Department at Manchester University, who keep finding money to allow me to pay the bills, and who encouraged me to see this work through. Special thanks to Martyn Spink for his understanding and honesty as my manager, and his encouragement as a friend.

Comments and encouragement from Colin Runciman and Sandra Foubister at York were always available and always useful.

Finally, thanks to Jo and my family for their love and patience.
Chapter 1 - Introduction

1 Introduction

PROSPERO Canst thou remember
    A time before we came unto this cell?
    I do not think thou canst

MIRANDA 'Tis far off,
    And rather like a dream than an assurance
    That my remembrance warrants.

PROSPERO What seest thou else
    In the dark backward and abysm of time?
    If thou rememb'rest aught ere thou cam'st here,
    How thou cam'st here thou mayst.

MIRANDA But that I do not.

Shakespeare's The Tempest (Act 1 Scene 2) - abridged

This thesis is an investigation into the provision of information about the lazy evaluation of expressions written in a functional language. The central goal of the thesis is to demonstrate by example, that a system based on a single underlying evaluation model can provide useful information to users faced with a wide range of errors, and also to students of lazy functional programming.

Functional languages are a form of declarative programming language in which every definition is a pure mathematical function. The user executes a program by demanding the evaluation of an expression written in the language. This 'top-level' expression is usually defined in terms of functions written by the user. At the lowest level, functions are defined in terms of operations primitive to the language.

A large amount of research has been carried out into the theoretical aspects of functional programming, and the practical aspects of producing language implementations. At the same time, the subject of development environments for functional programmers is relatively unexplored; in particular, environments which
help with the debugging of functional programs are few and far between. One such environment is presented in this thesis.

The use of a functional language enables users to formulate mathematical proofs about properties of their definitions. However, it is still the case that programmers write function definitions containing errors. The thesis will present an investigation into, and taxonomy of the problems faced by users, and will review various possible approaches to the presentation of information to help deal with these problems. Many of these approaches will be shown to be narrowly focused on specific needs of the user, rather than attempting to provide information which is more generally useful, or put another way, useful to the user in a wide range of circumstances.

I will propose that one particular approach, that of rewriting, provides information which is useful in a wide range of circumstances. In fact I will show it to be useful in all of the cases laid out by investigation of the needs of users faced with errors, and of students using a lazy functional language. Several systems exist which are based upon a rewriting approach. The thesis will highlight the shortfalls of these systems, and go on to describe a new system which overcomes these shortfalls using various novel techniques.

A close investigation of systems based upon rewriting, shows that there are a number of inherent problems which must be overcome in order to produce a system which successfully provides information about expression rewriting. The system must be based upon a rewriting model which provides sufficiently low-level information, such that the scope of the problems which the system is applied to is not limited by abstractions inherent in the model. The system must provide a means by which to control the potentially large volume of information produced by a rewriting system. Finally, the system must provide facilities which allow the user to move backwards and forwards through an evaluation. All currently available systems which are based upon rewriting will be shown to fall short on at least one of these fronts. At the heart of this thesis is a detailed description of these problems, together with the description of Prospero, a rewriting system which provides novel solutions to them.

1.1. Users and Their Needs

For the purposes of this thesis, I will divide the users of a system which provides information about the lazy evaluation of expressions into two groups: developers of functional programs who are faced with errors, and students of functional programming.
Chapter 1 - Introduction

There is an argument that developers of functional programs do not need tools with which to find their errors; that the tools which are traditionally provided for imperative programmers are made necessary by the imperative, state-based, side-effecting nature of the languages, and that functional languages remove these problems, thus allowing the replacement of these tools with more formal mathematically based techniques.

The research described in this thesis is based upon the belief that, it is frequently the case that users write definitions containing errors, and that once such a definition has been written, it becomes very difficult for the user to behave in a systematic manner when attempting to find this error. Giving the user a graphic focused account of the failure of their definitions is a sure-fire way of guiding the user to a better understanding of their definitions.

The study carried out as part of the Calculator project (Goldson '93a, Fung & O’Shea '92) suggests that there is a genuine need for ‘symbolic calculators’ to help with the teaching of functional programming and the teaching of logic at an undergraduate level. Support for the teaching of functional programming involves the provision of support to both the teacher and the student. The teacher requires a system which can be used to support and reflect their material, and which will facilitate the graded exposure of students to the detail of the material. The student requires information about the evaluation of expressions at a level which matches the material with which they are familiar.

1.2. Lazy Functional Languages

A functional program may be viewed as a set of rewrite rules. The evaluation of an expression can be seen as a process which repeatedly rewrites the current expression, using the right hand side of an appropriate rewrite rule, until a form is reached where the expression is said to be fully evaluated (i.e. no more rewriting is necessary). The expression resulting from the full evaluation of an expression E, is called the ‘value of E’.

For all but the most trivial expressions, there are essentially two different ways of defining the order in which the rewriting of an expression may take place. These evaluation orders are known as lazy and eager evaluation. A detailed description of lazy evaluation, and the impact that it has on functional language users, and on the development of debugging tools is given in a later section of this chapter. For now it is sufficient to say that a functional language which employs lazy evaluation is known as a lazy functional language.
Lazy functional languages, such as Miranda\textsuperscript{1} (Turner '85), LML (Augustsson & Johnsson '89) and Haskell (Hudak & Wadler '89) are becoming popular. The main reason for this is the increased expressiveness that lazily evaluated languages offer over other languages; a classic example of this is the possibility of handling infinite data structures.

The reader of this thesis should be familiar with functional programming languages. Knowledge of a lazily evaluated functional language would be an advantage but is not essential. There are a number of comprehensive texts on functional programming including (Bird & Wadler '88, Holyer '91). A brief introduction to Miranda may be found in the appendix of (Peyton Jones '87), and in Appendix C of this thesis.

1.3. Function Definitions and Lazy Evaluation

The general principle of lazy evaluation is that no expression should be evaluated until its value is needed, and that no expression should be evaluated more than once. The best way of describing lazy evaluation is through an example which contrasts it with eager evaluation. Given the following definition:

\[
\text{double\_first } a \ b = a \times 2
\]

an eager evaluation involving this function would progress as follows:

\[
\begin{align*}
\text{double\_first } (3+4) & \ (2\times8) \\
& \rightarrow \text{double\_first } 7 \ (2\times8) \\
& \rightarrow \text{double\_first } 7 \ 16 \\
& \rightarrow 7 \times 2 \\
& \rightarrow 14
\end{align*}
\]

a lazy evaluation of the same expression would progress as follows:

\[
\begin{align*}
\text{double\_first } (3+4) & \ (2\times8) \\
& \rightarrow (3+4) \times 2 \\
& \rightarrow 7 \times 2 \\
& \rightarrow 14
\end{align*}
\]

In the first example, the arguments to the function are \textit{eagerly} evaluated before the function is applied to them. In the second example, the arguments are not evaluated

\textsuperscript{1} Miranda is a trademark of Research Software Limited.
Chapter 1 - Introduction

until their value is needed. In this case the value of the second argument is never needed, and the value of the first argument is not needed until it becomes the argument of an application of the multiplication operator.

In order to be able to discuss lazy evaluation in more detail it is necessary to define a number of terms. Some expressions do not have a well-defined value. For example, the result of the division of a number by zero is undefined, as is the value of an expression whose evaluation never terminates. It is common practice to refer to all such undefined values using the symbol \( \bot \) (pronounced \textit{bottom}).

A \textit{strict} function is a function whose value is undefined whenever its argument is undefined. An example of such a function is:

\[
succ \ a \ = \ a + 1
\]

The application of this function to \((1/0)\) is \( \bot \), and so \textit{succ} is strict.

Lazily evaluated languages facilitate the definition of non-strict functions (i.e. functions which may return a defined value, even when their argument is \( \bot \)). The function \textit{double\_first} defined above is an example of a non-strict function, since its second argument could be \( \bot \) and an evaluation of the function would still be well defined. For example in the following evaluation, involving \textit{double\_first}, we see that the value of the second argument is never calculated:

\[
\begin{align*}
\text{double\_first} \ (3+4) \ (1/0) & \quad \text{(using \textit{double\_first})} \\
\Rightarrow (3+4) \ast 2 & \quad \text{(using \textit{+})} \\
\Rightarrow 7 \ast 2 & \quad \text{(using \textit{\ast})} \\
\Rightarrow 14 & \\
\end{align*}
\]

Thus, it is clear that a lazily evaluated functional language allows the user to express things which they would not be able to express using an eager language. Two concrete examples of the usefulness of such an approach are given below. The first shows the ability to define infinite data structures without causing a non-terminating evaluation, the second shows how a conditional (or if-then-else) construct can be define as an ordinary function, which in an eager language would cause both 'branches' to be evaluated strictly:

\[
\text{all\_nats\_after} \ n \ = \ n : \ (\text{all\_nats\_after} \ (n + 1))
\]

\[
\text{cond} \ \text{True} \ b \ c = b
\]
\[
\text{cond} \ \text{False} \ b \ c = c
\]
Examples of evaluations involving these definitions are:

\[
\begin{align*}
\text{head} & \ (\text{all\_nats\_after} \ 0) \\
& \Rightarrow \text{head} \ (0 : (\text{all\_nats\_after} \ 0+1)) \\
& \Rightarrow 0 \\
\text{cond} & \ (1<2) \ 3 \ \text{loop} \\
& \Rightarrow \text{cond} \ \text{True} \ 3 \ \text{loop} \\
& \Rightarrow 3
\end{align*}
\]

It is important at this point to introduce an interesting relationship between lazy and eager evaluation of expressions. It has been shown (Barendregt '84) that if the eager evaluation of an expression produces a defined value, lazy evaluation is guaranteed to produce the same value. Furthermore, if an expression has a defined value, lazy evaluation guarantees to produce that value, while the same can not be said for eager evaluation, as we have seen in the above examples. The impact of these findings on the work presented here is that any possibility of a system deviating from lazy evaluation may result in the failure to produced a defined value where one exists.

1.4. The Impact of Lazy Evaluation on Debugging

Lazy evaluation has a large impact on users when they are faced with errors in their definitions, and on designers of systems aimed at assisting such users. This is borne out by Milner, who when writing about his design of the language ML said:

"ML does not use lazy evaluation; it calls by value. This was decided for no other reason than our inability to see the consequences of lazy evaluation for debugging"

(Milner '84)

In this section I will investigate these ‘consequences of lazy evaluation’.

1.4.1 The ‘Pen and Paper’ Approach to Debugging

Let us first explore the problems faced by users who have errors in their definitions, and no system to provide them with information about their definitions, or evaluations involving these definitions. In order to reason about the failure of their definitions, observation has shown that users frequently resort to a ‘pen and paper’ approach to debugging, i.e. they will attempt to recreate, by hand, the conditions which have caused their definitions to fail by reproducing evaluations of expressions. Note that I am not ruling out other approaches to debugging, such as attempting to prove that
definitions have certain properties, rather I am suggesting that the ‘pen and paper’ approach is one commonly adopted by experienced and novice users alike.

In (Sinclair '91) it is argued that:

"in the functional language paradigm, and especially in the presence of lazy semantics, not even the more sophisticated programmer would be able to predict the evaluation order of anything but the simplest programs."

Self observation, and the observation of others confirms this to be the case. Users of lazy functional languages seem to have problems when trying to reliably reproduce the lazy evaluation of all but the simplest of evaluations. The main reason for this seems to be the need for users to keep a mental record of all sub-evaluations which are waiting upon the evaluation of other sub-expressions. Compared to eager evaluation where the user finds the correct sub-expression to reduce, and fully reduces it, at which point the next sub-expression is found, lazy evaluation requires the user to find the correct sub-expression to reduce, and then to reduce it up to a point where the evaluation of other sub-expressions may proceed. Thus the user must remember why they have started to evaluate a sub-expression, spot when they have carried out sufficient evaluation, and record the exact point that they have reached in the evaluation of the sub-expression.

A second reason for users having problems with a ‘pen and paper’ approach is the tendency of users to follow ‘leads’ when tracking down errors, and to evaluate sub-expressions suspected as being the source of their errors, rather than selecting the correct sub-expression using the rules of lazy evaluation, thus users are too eager in their evaluation scheme. In the worst case this may lead users to a sub-expression with an undefined value, when the evaluation of that sub-expression would not normally have taken place; only slightly better is the possibility that users may discover an error in their definitions, which turns out not to be the error that they had set out to find. In some cases this is a worse scenario, since the user becomes confused as to why removing the error hasn’t fixed their definitions.
In (O'Donnell & Hall '87) a debugging system for the language Daisy is described. One of the main areas in which the system differs from other systems is that it provides users with the ability to carry out ‘lead following’ when using the debugger. O'Donnel and Hall claim that this is:

"preferable in most cases because the user can reduce the amount of irrelevant output by directing the debugging package to the parts of the program that are causing trouble."

This approach is a dangerous one to take since lazy evaluation presents users with the opportunity to manipulate expressions whose value is undefined. Allowing users to steer an evaluation allows them to change the termination properties of that evaluation, perhaps making a terminating evaluation fail to terminate.

An added complication faced by users is the effect that pattern matching has on laziness. In many cases it is the use of pattern matching that governs the order in which expressions are evaluated. Let us examine an example using the following definition of a function twins, which when applied to a list, will evaluate to True if the list contains two adjacent elements which are equal, and will evaluate to False otherwise:

\[
\begin{align*}
twins \ [] &= \text{False} \\
twins \ (x:x:xs) &= \text{True} \\
twins \ (x:xs) &= \text{twins} \ xs
\end{align*}
\]

Here the patterns on the left hand side of the definition will govern the amount of evaluation carried out on the argument to the function before a recursive call, if any, is made. The first clause of the function will cause the argument to be evaluated to the point where it is known that the list is empty or not. If the argument is the empty list, no more evaluation takes place. The second clause will cause the argument to be evaluated to the point where it is known that the list has at least two elements, these elements will then both be evaluated to the point where they can be compared for equality. The third clause will not cause any evaluation to be carried out, since the previous clause would have carried out sufficient evaluation in order to be able to bind the tail of the list to the free variable (xs) on the right hand side of the clause.

Thus we can see that lazy evaluation poses problems to users, since they have to mentally track large amounts of information in order to be able to produce ‘pen and paper’ examples. Furthermore being tempted away from truly lazy evaluation may lead to the value of an expression appearing to be undefined when in reality it is not.
Finally, pattern matching introduces a further level of complexity when attempting to reproduce an evaluation. We can conclude from this discussion that lazy evaluation makes life intolerably difficult for unsupported users faced with errors.

1.4.2 Impact of Lazy Evaluation on Debugging Systems

Given that lazy evaluation makes 'pen and paper' approaches to finding errors in definitions difficult, let us look at the impact of lazy evaluation on the implementors of systems designed to help users when faced with errors.

The first question which must be answered is whether the 'confusing' qualities of lazy evaluation which caused Milner ('83) to elect not to use it, will render systems which present lazy evaluation unusable. Sinclair follows up the statement quoted on page 21 by saying that the inability of programmers to predict the evaluation order "suggests that the average programmer would quickly get lost within the debugging session presented according to evaluation order".

Sinclair goes on to show that this claim does not rule-out all forms of system, rather, he says, it only rules out systems which explicitly present the user with evaluation stages ordered according to the rules of lazy evaluation. Other systems, which present information independent of the order of evaluation are acceptable.

It is open to debate whether a programmer who was introduced to programming using a lazy functional programming language would indeed have difficulties with a system which presented information as evaluation stages ordered according to lazy evaluation. More importantly, while I agree with Sinclair, that the hand reproduction of lazy evaluation is beyond the abilities of most programmers, I believe that when the evaluation is presented to a user, it is possible for the user to follow its course. Systems in which evaluations are presented "according to evaluation order" require only that the user follow, rather than predict, the evaluation order and so are not flawed in the way suggested by Sinclair.

Another way of viewing this debate over the pros and cons of presenting information in terms of the lazy evaluation of expressions is in terms of declarative and imperative approaches to providing information.

Whether a system should provide declarative or imperative information is a point of real contention between workers in the field of lazy functional programming. The argument against imperative information is simply that declarative information should be provided for a declarative language. This is backed up with the argument outlined
above, stating that an imperative explanation of lazy evaluation would be too confusing.

There are certainly cases where declarative information about an evaluation, or indeed about definitions independent of any evaluation that they may be involved in, can provide information which answers questions about the definitions/evaluation in question. For example, one may ask if a certain function is ever applied to an argument with a specific value. This question can be answered without having to resort to any imperative information about the evaluation in question.

On the other side of the argument, there are certainly cases where imperative information is the only way to answer some of the questions that a user may ask. For example a user asking to be shown the pattern matching which caused one clause of a definition to be chosen over another in a particular case, must be shown imperative information about the pattern matching that has taken place. Similarly, once a user has asked if a certain function is ever applied to an argument with a specific value, a logical next question is: how did the function come to be applied to that argument? Once again imperative information is essential in order to be able to show the user the stages in the evaluation that led to the function in question being applied to a specific argument.

A second argument for an imperative approach is along pedagogical lines. Preliminary results from the Calculator project (Goldson '93a, Fung & O'Shea '92) suggest that there are distinct advantages to be gained from teaching functional programming using a lazily evaluated functional language, together with tools which show an imperative view of the progress of an evaluation. It seems perfectly reasonable to explain lazy evaluation by showing lazy evaluation in progress.

Imperative information about an evaluation relies on the definition of an evaluation model on which to base this information. The problem faced by the developers of systems which provide this information is to produce a system with two important properties. The first property is that the chosen system should faithfully represent the evaluation scheme of the language in question, i.e. for all possible expressions, the evaluation of the expression using the system will produce the same result as that produced by the original evaluation system.

The second property is far more subtle, if not unattainable: it is that the system produces information at a suitable level of abstraction to meet the needs of the system's users. As we shall see in later chapters of this thesis, there have been a number of attempts at selecting such a 'perfect' level of abstraction, or at least there
are a number of systems which have a fixed level of abstraction at which all
information is presented to users.

An alternative approach, and the one that has been chosen for the work presented in
this thesis, is to design a system which, rather than fixing the level of abstraction,
produces large amounts of low-level information about an evaluation, and provides
tools which allow for the construction of new levels of abstraction as they are
required. This approach avoids the trap of fixing the abstraction at the wrong level,
and at the same time allows for new abstractions to be created specifically for the
definitions in question. For example a new abstraction can be created to reflect new
data structures involved in a set of definitions.

*Introducing Strictness into an Evaluation*

There are a number of additional considerations which must be taken into account
when producing a system which presents information about the lazy evaluation of
functions. The most important of these is the possibility that a system could change
the order of evaluation. Lazily evaluated functional languages allow the user to define
non-strict functions, and involve these functions in evaluations, therefore any
possibility of introducing strictness into an evaluation (i.e. evaluating anything in a
non-lazy manner) may cause an expression to be undefined when it would normally
be defined.

There are a myriad of ways in which a system may introduce strictness into an
otherwise lazy evaluation. Two of them - displaying and searching - are introduced
here, since they are problems which are dealt with in detail in later sections of this
thesis. In particular the work described in Chapter 6 has at its heart the problem of
introducing strictness when searching an evaluation.

Displaying (in many cases synonymous with printing) is normally a strict process. In
order to be able to display the value of an expression a system will evaluate it until it
is in a displayable form. In fact the strictness of displaying is normally at the root of
all evaluation in a lazy functional system, since it is the demand by the user to be
shown the value of an expression that causes the system to do any work. When
designing a system which provides information about an evaluation it is vital not to
carry out any extra evaluation when trying to present the user with information. Take
as an example the evaluation of the following expression, which involves the function
double_first defined above:

double_first 1 (1/0)
If at some point in the evaluation of this expression the user asks to be shown the value of the second argument, and the system naively evaluated the expression, then an error would occur when under a lazy evaluation regime it should not have. Similar problems arise from the attempt to print sub-expressions which represent infinitely large data structures.

Thus it is essential that a system meets the user’s needs to be shown information, while satisfying the condition that any displayed information is not the result of any extra evaluation having taken place.

In a similar way, allowing the user to search for the stage in an evaluation where a certain condition holds can introduce strictness. Using the same expression as above: if the user asked to be shown stages in an evaluation where double_first was applied to arguments which were equal to each other, then an apparent error would once again be introduced into the evaluation. This thesis will describe a compromise between the provision of searches and avoiding the introduction of strictness into an evaluation.

1.5. Models and Views

Before moving on to introduce Prospero, and to lay out the structure of this thesis, I will discuss another design aspect which must be considered by the designer of a system which produces information about expressions and evaluations. First, it is necessary to introduce some definitions regarding the basis of this information and its presentation:

- A model, in this context, is the basis of all of the information that is provided to the user. The model is essentially the representation of the evaluation of expressions that users are expected to accept. For example the Byrd box (Byrd ’80) is a model of evaluation for Prolog;

- A view is a way of presenting an evaluation based on a model to the user. It may involve the presentation of the information in a particular way, or may involve presenting a part or summary of the information available.

Ideally a system would be able to support the needs of a wide range of users and be able to fulfil a number of roles. A step towards this is the ability to present evaluations using a number of models each with a number of different views.

Users will want to investigate a number of properties of their definitions. In order to be able to support this, a single model with a single view, while not necessarily
Chapter 1 - Introduction

prohibitive, is certainly not ideal. For example, presenting an evaluation using string rewriting with information at the level of the user’s definitions, could be used to investigate the space efficiency properties of a set of definitions, but a model which could explicitly show space properties in terms of an abstract store would be more useful.

The increased cognitive load placed on the user to learn and understand a number of models and the relationship between them is an obvious problem. A system which uses a number of models in order to illustrate a number of attributes of a set of definitions may produce better illustrations with respect to their content, but is not necessarily superior to a system which uses a single model to illustrate a number of properties. The illustrations of the former may turn out to be superior in some cases, but the learning load on the user will certainly be larger and there is a risk of the user misunderstanding the relationships between the different models.

Using a single model places a number of demands on the model in question. It must be possible to use the model as a basis for a wide range of views of an evaluation. It is not satisfactory to decide on a model which is highly suitable for the illustration of a particular aspect of an evaluation, but which cannot be used as a basis for any other view. For example a heap profiling system, such as that described in (Runciman & Wakeling ’92), is excellent for illustrating the space efficiency of functional language definitions, but is of no use to users wishing to investigate other properties of their definitions.

Whether a single, flexible model has been chosen for a system, or a number of models are chosen, it is preferable to provide a number of views of an evaluation for each model. Better still, the system should allow the user to specify new views of their own. This approach was taken to its extreme in the BALSA II system, an algorithm animator for Pascal (Brown ’88) where no default view was available to the user, instead an animation had to be ‘authored’ for each example. This ‘authoring’ was carried out either by the user or by a specialist author.

In (du Boulay, O’Shea & Monk ’81) it is proposed that an ideal system for the presentation of computing concepts, such as the execution of a declarative program, is one which consists of

"black boxes" whose own internal workings do not need to be explained

and in which
Chapter 1 - Introduction

languages for novices can be implemented in such a way that some form of commentary is available. This commentary is the “glass box” through which the novice can see the “black boxes” working.

The authors then go on to state that

There is no reason why a high level language such as PROLOG, with its pattern-matching and backtracking facilities, could not be implemented to present the novice with a simple visible notional machine.

The ideas presented in (Pain & Bundy ’87) can be seen as a natural progression from the ideas put forward in (du Boulay, O’Shea & Monk ’81). In their paper, Pain and Bundy search for the ideal Prolog ‘story’ or model with which to explain the execution of a program. They examine a number of potential ‘stories’ with regard to their appropriateness for use in teaching material. In part, the conclusion of that paper is:

The complete model of Prolog to be represented in any story is:

- the AND/OR tree story of the search space;
- the database of clauses;
- the information for controlling search.

Presenting all this information in one story leads to a confusing complexity for all but the simplest examples....We present some parts of the story automatically and give access to the remainder. The actual story that we present will be determined by the aims and background of the users.

The natural progression of these ideas ends with their expansion and the subsequent implementation of the Transparent Prolog Machine (TPM) (Eisenstadt & Brayshaw ’87) developed at the Open University to support the University’s Intensive Prolog course (Eisenstadt ’88).

The question which must be faced here is whether a story suitable for explaining lazy functional programs exists and how one would go about defining it.

What are the properties that a lazy functional language story should exhibit? Like the Prolog story, it should be a story which can be used to explain evaluations to users with a wide range of programming experience. The story should never have to be
changed as a user becomes more experienced; instead different aspects and new views of the story would be presented. The story should be suitable for use in the classroom as well as in the computing laboratory.

1.6. Prospero

There are very few systems currently available which satisfy the needs of the users described at the beginning of this chapter, by providing information about the lazy evaluation of functional programs. Those systems which are available either provide information about a particular aspect of an evaluation (e.g. profiling information), or they provide general information about an evaluation but pay little attention to the way that information is presented.

Prospero is a system based upon an implementation of a lazy functional language. This language implementation is specifically designed to provide users with information about the evaluation of expressions. It is by describing the design and implementation of Prospero, that I aim to show that it is possible to create a system, based upon a single evaluation model, which provides suitable information to users faced with errors in their definitions, and information to students of lazy functional languages.

A major part of the work on Prospero has involved research into ways of controlling the presentation of the potentially vast amount of information available when using a imperative, rewriting based model of an evaluation. At a basic level, failure to provide control over the information presented quickly leads to the user becoming overwhelmed by the volume of information available. The approach to providing control over information described here will be shown to support, amongst other things: the focusing of the display on particular aspects of an evaluation; experimentation with novel views of evaluation stages; elision of large data structures and highlighting of specific qualities of this data. At the same time, no strictness is introduced as a result of displaying information to users.

In the main, because of the problems of introducing strictness outlined earlier in this chapter, currently available systems fail to provide users with facilities which allow them to find and focus upon a specific stage in an evaluation. Instead the user is forced to step through the evaluation until they find a particular stage. Chapter 6 of this thesis describes a design for an extension to the current implementation of Prospero which would offer facilities with which to move through the evaluation history, and to search for stages in the evaluation with specific properties. This extension would allow users to experiment with the amount of strictness introduced
into an evaluation by searches. I will defer a discussion of this design, and why any
strictness should be allowed at all, until Chapter 6.

1.7. Aims and Structure of the Thesis

This thesis is an investigation into the provision of information about the lazy
evaluation of expressions written in a functional language. In the main, this
investigation is carried out through the description of a system with which to tell
'stories' about the evaluations to users of a lazy functional language.

The thesis describes a range of different errors that users of lazy functional languages
can introduce into their definitions, and shows that tools can be created which provide
users with information about these errors. In Chapter 2 I will present a taxonomy of
these errors, and outline the software support required by students and teachers of
functional languages.

Given that these users require information about properties of their definitions, I go
on to outline the possible approaches available to an implementor of a system
designed to meet these needs. A survey of these approaches and an assessment of
their suitability for the task in hand, is presented.

Many of the presented approaches will be shown to be specialised in order to create
information specific to a certain aspect of a set of definitions, or of an evaluation
involving these definitions. I will show that it is possible to build a system which
provides more general information, and that while this information will not be specific
to a certain class of problems faced by users, it will be shown to be useful in a wide
range of scenarios.

To reinforce the findings of Chapter 2, Chapter 3 surveys currently available facilities
and systems, and aims to show three things: that the basic facilities provided by
Miranda are inadequate; that systems based upon non-rewriting approaches are flawed
in the way described in Chapter 2, and that the currently available rewriting systems,
while being strongly influenced by the work described in this thesis, fail to provide
facilities which overcome the inherent problems of a rewriting approach. Another
central theme of this thesis is to define what these inherent problems are, and to
present tools which overcome them.

Chapters 2 and 3 set the scene for Prospero. They survey the approaches available to
implementors, and the systems/facilities currently available to users of lazy functional
languages. The general conclusion of these chapters is that rewriting is an all-round
approach to providing information about an evaluation, and that a rewriting system must overcome a number of problems in order to be a success.

Chapter 4 moves on from a discussion of suitable approaches, to the description of Prospero, a system designed to demonstrate that rewriting is a suitable approach, to illustrate the problems of rewriting, and to demonstrate that these problems can be overcome by providing suitable tools.

Chapter 4 provides a ‘whistle-stop’ tour of Prospero, the structure of which is based upon that of Chapter 2. By adopting this structure, I show that a system based upon rewriting, can be used to provide information to the user faced with all of the errors discussed in Chapter 2, and that the same system can provide information to support the teaching of different aspects of functional programming. At the same time, Chapter 4 introduces the different aspects of Prospero which are discussed in detail in later chapters.

A system based upon a rewriting approach must be based upon three things. The system must provide an evaluation model; the system must provide facilities with which to control the information presented to the user; and the system must provide facilities with which to move through an evaluation, and search for evaluation ‘events’. One claim of this thesis is that existing systems based on rewriting, provide inadequate models and facilities. Chapters 5, 6 and 7 describe how these needs are met in Prospero.

It should be noted that another pre-requisite for a successful system is the provision of a well designed user interface. While Prospero does provide an implementation of a user interface, the design and development of that interface was not seen as an aim of the research described here, rather the current interface is a response to the need for a display of some kind. Further discussion of the user interface, and the scope of this thesis is provided below.

Chapter 5 describes the facilities provided by Prospero, in order to allow the user to control the information presented in each stage of an evaluation. The aim of the work described in Chapter 5 is to show that by using ‘filters’ the user can present information which is pertinent to their needs, and can also control the potentially large volume of information produced by a rewriting system.

The work described in Chapter 6 is concerned with allowing the user to ‘navigate’ through an evaluation. The problems of navigation and lazy evaluation are highlighted, showing that a naive approach to the problem of searching an evaluation
can alter the strictness of an evaluation, thus changing the evaluation order. The chapter goes on to describe a number of search strategies which allow the user to control the amount of strictness introduced into an evaluation by the search.

A major failing of all currently available systems is their inability to show certain ‘low level’ aspects of an evaluation; pattern-matching for example. Prospero’s evaluation model has been designed to enable the provision of this information in a way which is as illuminating as possible. Chapter 7 provides a detailed description of this evaluation model. This description is a complete one (i.e. it describes the evaluation scheme for all well formed expressions), and focuses on the aspects designed specifically to provide the low-level information unavailable in other systems. Earlier in this chapter it was stated that any evaluation model must be faithful to the original evaluation model of the language in question. Chapter 7 presents an informal proof that the Prospero’s evaluation model is equivalent to a recognised evaluation model.

While this thesis is entitled ‘The Presentation of Lazy Evaluation’, the ‘presentation’ discussed here is presentation in an abstract sense. This thesis is not about the implementation of a concrete user interface; it discusses the presentation of information at a level which is independent of the choice, or style of user interface. Despite this, it has been necessary to develop an interface to Prospero, in order to demonstrate the ‘real’ contributions of this thesis. Sufficient interest has been shown in the implementation of the current interface to Prospero, to warrant the writing of Chapter 8, where the more novel aspects of the interface are discussed.

All research must have a limited scope. The research described here is no exception, and has deliberately set out to avoid certain areas which would nevertheless be essential to the translation of the ideas presented here into a useful tool. Chapter 9 presents the thesis’ conclusions, and describes work which is beyond the scope of this thesis, and suggests future work directions. The evaluation of the current system is discussed, along with suggested improvements to the current implementation.

In conclusion, the aims of the work described here are:

- to present a taxonomy of the errors faced by users of a lazy functional language, and the needs of students and teachers using such a language;

- to review approaches to the provision of information about lazy function definitions;
• to show that while rewriting does not produce information specific to a certain class of problems faced by users, it is a good all-round approach to providing information;

• to highlight the short-falls of existing systems based upon rewriting;

• to describe and implement an evaluation scheme which forms a suitable foundation upon which to build presentations of lazy evaluation;

• to describe and implement a filtering system with which to control the information presented by a system based upon rewriting;

• to illustrate the problems faced when providing facilities with which to search for events in an evaluation;

• to describe a search system which allows the user to control the strictness introduced by a search system;

• to suggest future work which, while being outside the scope of the research presented here, would be essential in order to use this work as the foundation for a useful tool.

1.8. State of Implementation

At this point it is worth making clear the current state of the implementation of Prospero, since the time available for the research was not sufficient to provide an implementation of all of the facilities described here. As it stands, Prospero provides a complete implementation of the evaluation model as described in Chapter 7. A partial implementation of the filtering system described in Chapter 5 is available, and further details of the implementation state will be given in that chapter. None of the search system described in Chapter 6 is available.

It is also worth drawing the readers attention to the different styles of illustration in the thesis, since these differences reflect the current implementation state of the system. Figures produced in a style such as that of Figure 4.3(a) are produce using a drawing package, since they depend on parts of Prospero which are not implemented. Figures produced in the style of 4.18(a) are screen-dumps taken from a live Prospero session.
2 Approaches to Providing Information

In this chapter I introduce the reader to the problem of finding errors in functional language definitions, and to the requirements of software support for the teaching of lazy functional programming. The chapter presents a taxonomy of the errors frequently encountered by users of a functional programming system, and presents a number of approaches which could be used to provide information to users about these errors.

This thesis is in part an exploration of the possible ‘stories’ which can be presented to users about the lazy evaluation of expressions. This chapter sets out some of the issues which must be considered when producing such a story and goes on to explore a number of candidate ‘stories’ which could be used as the basis for a system such as Prospero. The suitability of each of the ‘stories’ to each of the errors in the taxonomy is then assessed.

In Chapter 3 I will go on to describe existing systems which implement some of these approaches.

In Chapter 4 I will return to the examples presented here and show how Prospero provides information about the errors discussed in this chapter.

2.1. Types of Error

In this section I will present some of the types of errors that occur in users’ definitions. In the following section I will go on to describe some of the approaches to providing the user with information which could help the user to find these errors. Following on from this I will tie these two sections together, by showing which of these approaches is suited to which type of error.
The list of errors discussed here, has come from three main sources: personal experiences following extensive use of the lazy language Miranda; the comments of the staff of the Department of Computer Science, Queen Mary and Westfield College; and my observations of novices learning to use the strict functional language ML. As such, it would be incorrect to present this as a complete taxonomy of errors.

2.1.1 Wrong Answer

One of the most common types of error introduced by users into their definitions, is that a function may produce a value which is not the one expected by the user. There are three main causes for this type of error:

- the algorithm contains an error;
- the translation of the algorithm into Miranda contains an error;
- the user misunderstands the semantics of some other functions - in particular, those functions defined in the Miranda standard environment.

The fact that Miranda is a strongly-typed language helps reduce the number of errors caused by the second and third of the causes listed above, but experience shows that these errors still occur frequently.

2.1.2 Non-Termination

Recursion enters into all but the simplest of Miranda definitions, and once it has been introduced, there is the possibility of defining an expression whose evaluation will not terminate. Again the cause of a non-terminating evaluation could be a flaw in the algorithm, or a flaw in the translation of the algorithm into Miranda.

2.1.3 Application to Arguments Outside a Function's Domain

Another common type of error results from the user defining a partial function and then applying the function to an argument whose value is outside the domain of the function.

2.1.4 Space and Time

The final type of error considered here occurs when an evaluation requires more memory, or takes longer to execute than was expected. It could be argued that this is not actually an error. But at its most extreme a program that takes all day rather than a few minutes to run is not a useful one. Even if problems of this type are not perceived
as errors, it is undoubtedly true that useful information can be provided to the user about the efficiency of their definitions.

2.2. Approaches to Providing Information about Errors

Following on from the taxonomy of errors given above, I will describe some of the approaches to providing information which may help the user to find these errors.

2.2.1 Rewriting

The first approach is to show the rewriting of expressions. This approach involves showing the user the step-by-step transformation of their original expression, using function definitions as rewrite rules, to its fully evaluated form. This transformation process involves the rewriting of an expression using a set of simplification rules (e.g. $3+4$ simplifies to $7$), and the equations defined by the user.

2.2.2 Assertions

The second approach is the provision of program annotations with which the user may make assertions about properties of their definitions. Typically these assertions will concern the range and domain of functions (e.g. asserting that the argument to a function will always be a string whose length is greater than 1 character).

2.2.3 Algorithmic Debugging

The next alternative is an 'algorithmic' approach. Algorithmic debugging was pioneered by Shapiro for the language Prolog. In (Shapiro '82) a technique of program debugging is described; this technique can be summarised as follows.

The program to be debugged is executed and a complete execution space is built in the form of a call graph for the evaluation of the program; this trace holds information about the arguments and results of all function calls. Once this evaluation trace has been built, the user is walked through the execution space, and is shown the values of arguments and results at each point. At each of these points, the user is asked whether the result is correct for the given argument. If the answer is no, then the error must be either in the function which the walk has reached, or in one of the functions called by the current function. In order to establish this, the system explores all of the function calls made by the current function, and either one of these returns an incorrect result, in which case the walk continues by examining that function call, or all of the functions return the correct result, in which case the current function is at fault and the error has been located.
2.2.4 Profiling

Profiling systems provide information to users in terms of the time and space usage of their definitions during an evaluation. As such these systems will only be considered in the discussion of efficiency problems faced by users.

2.2.5 Abstract Interpretation

The field of abstract interpretation can provide various types of information about functional language definitions. For example it can provide information about the strictness of a definition, or the order of complexity of a definition. There are a number of drawbacks to these approaches which will be explored later in this chapter.

2.3. Help with Errors

In this section I will examine how each of the types of error introduced earlier in this chapter may be tackled using, where appropriate, each of the approaches described above.

2.3.1 Wrong Answer

Presentation Using Rewriting

A system that presents the rewriting of an expression can provide information to a user about why an expression evaluated to an unexpected value. For example, Figure 2.1(a) shows a definition of a tree-membership function, for a tree which is defined such that values on the left of a branch are always less than the values at the branch, or on the right of that branch. This tree-membership function’s recursive clause is defined in terms of the wrong branch of the tree (i.e. the condition \( \text{if } v < v' \) should read \( \text{if } v \geq v' \), as shown in Figure 2.1(c)). Figure 2.1(b) shows how a trace based on rewriting, could be used to provide information about the error.

The rewriting shown in Figure 2.1(b) gives a step-by-step view of the evaluation of the expression `tree_member treel 1`. Each of the steps in the evaluation is produced by replacing sub-expressions with their appropriate definition taken from those given in Figure 2.1 (a).
Figure 2.1(a) Definition of function \( \text{tree\_member} \) containing an error; (b) a trace of the function highlighting the error; (c) the correct definition of \( \text{tree\_member} \).

The example in Figure 2.1 shows that rewriting can be used to produce a step-by-step picture of the evaluation of an expression which results in an unexpected answer. In this example, the level at which the information is produced is equal to that of the user’s definitions, i.e. each stage shows only information from the user’s definitions and give no information about the underlying evaluation model, for example no information about the selection of the correct clause of \( \text{tree\_member} \) is given. Later sections in this chapter will demonstrate that the ability to show this low-level information is a major strength of the rewriting approach.
Chapter 2 - Approaches to Providing Information

(a)

\[
\text{tree } * ::= \text{Branch (tree *) } * \text{ (tree *) } | \text{Leaf}
\]

\[
\text{insert Leaf } v = \text{Branch Leaf } v \text{ Leaf}
\]

\[
\text{insert } t \ v = \begin{cases} t, & \text{tree_member } t \ v \\ \text{insert } l \ v', \ v' < v \\ \text{insert } r \ v', & \text{otherwise} \end{cases}
\]

||N.B. tree_member defined as in Figure 2.1(a)||

\[
\text{treel } = \text{Branch}
\]

\[
\begin{array}{l}
(\text{Branch Leaf } 1 \ \text{Leaf}) \\
2 \\
(\text{Branch Leaf } 3 \ (\text{Branch Leaf } 4 \ \text{Leaf}))
\end{array}
\]

\[
\text{main } = \text{insert treel } 2
\]

(b)

\[
\begin{array}{l}
\text{main } \Rightarrow \text{Branch (Branch Leaf } 1 \ \text{Leaf}) \\
\quad 2 \\
\quad (\text{Branch (Branch Leaf } 2 \ \text{Leaf}) \\
\quad \quad 3 \\
\quad (\text{Branch Leaf } 4 \ \text{Leaf}))
\end{array}
\]

\[
(y/n)? \ n
\]

\[
\begin{array}{l}
\text{tree_member Branch ((Branch Leaf } 1 \ \text{Leaf}) \\
\quad 2 \\
\quad (\text{Branch Leaf } 3 \ (\text{Branch Leaf } 4 \ \text{Leaf})) \ 2 \Rightarrow \text{False}
\end{array}
\]

\[
(y/n)? \ n
\]

Error is in function "tree_member"

Figure 2.2 (a) the definition of a tree insertion function containing the erroneous definition of tree_member defined in Figure 2.1(a); (b) the use of an algorithmic debugger to locate the error

An Algorithmic Approach

An illustration of the use of an algorithmic debugger to find the error in the definitions given in Figure 2.2(a) is shown in Figure 2.2(b)\(^1\). While the tree insertion function in

\(^1\)Throughout this thesis I shall use the following convention for representing interactions between a system and a user. Any text which is printed by the system with which the user is interacting will appear in courier. Any text which is entered by the user will appear in courier bold
this figure is correct, it makes use of the function tree\_member (defined in Figure 2.1(a)) which contains an error.

In this example the user is asked whether particular steps of the evaluation are correct. The first question that the user is asked is “Should main evaluate to this particular tree?” to which the user answers no. The next question is about the result of searching for the value 2 in the tree tree1, and leads the user to the location of the error. Of course the length of this session is not representative and the user would not expect to find the error so quickly. More complex and realistic examples are described in (Nilsson & Fritzson '92).

A system which offers an algorithmic approach to finding errors allows the user to work at the declarative level at which they wrote their definitions. This means that in order to find errors in their definitions, users need only know what values expressions should evaluate to. However, an algorithmic system does not provide all of the information that may be required about an evaluation. For example, it may be the case that the user has an interest in the order in which expressions are being evaluated.

**Assertions**

Figure 2.3 shows the use of assertions to locate an error in the definition of the function funny\_reverse given in Figure 2.3(a). This function takes four arguments: an index into the list to be reversed; the maximum value of the index (i.e. the length of the list to be reversed); the list to be reversed; and an accumulator into which the result of the function is placed, one list element at a time.

\[ (\text{list } | n) \text{ returns the n-th element of list, where the head of a list is the 0-th element.} \]
\[ \text{== denotes the inequality operator.} \]
Chapter 2 - Approaches to Providing Information

(a)\nfunny_reverse index limit l1 l2
  = funny_reverse (index + 1) limit (tl l1) ((l1 ! index) : l2)
  , index >= limit
  = l2 , otherwise

(b)\nMiranda funny_reverse 0 3 [1,2,3] []
SUBSCRIPT ERROR
Miranda

(c)\nassert index + limit = (length l1) + (length l2)
in funny_reverse

funny_reverse 0 3 [1,2,3] []
assertion failed for:
funny_reverse 1 3 [2,3] [1]

(d)\nfunny_reverse index limit l1 l2
  = funny_reverse (index + 1) limit l1 ((l1 ! index) : l2)
  , index >= limit
  = l2 , otherwise

Figure 2.3 (a) definition of the function funny_reverse containing an error; (b) transcript of Miranda session involving evaluation of the function; (c) an assertion used to locate the error in the function; (d) the corrected version of the function

Figure 2.3(b) shows the error message resulting from the evaluation of an expression involving this definition. Figure 2.3(c) shows the use of an assertion to check that an expected property holds for the definition. Finally, Figure 2.3(d) shows the correct definition of funny_reverse with the correction to the definition underlined. Note that this function definition is being used to illustrate the use of assertions, and is not being put forward as a realistic definition of a list-reversal function.

Combining assertions with a lazily evaluated language should be done with care; it is possible that the use of assertions will change the termination properties of an evaluation by causing extra evaluation to take place in order to check the validity of the assertion. This problem is discussed in detail in Chapter 3.

Like the algorithmic approach described above, assertions provide one particular kind of the information that may be required about an evaluation. As such, the approach must be viewed as a specialised, rather than general, approach to providing information.
Rewriting, algorithmic debugging and assertions can all be used in different ways to provide information about an evaluation which produces the wrong value. All of these approaches provide a very different, and valid approach to dealing with such problems. Rewriting leads the user through the evaluation showing how the value came to be what it was. The algorithmic system takes a much more declarative approach and allows the user to narrow down the definitions at fault by asking questions about the values produced by the definitions. Assertions allow the user to express their beliefs about a set of definitions in order to track down a definition which is behaving incorrectly.

2.3.2 Non-Termination

Rewriting

Figure 2.4(a) shows a definition whose application will sometimes fail to terminate. In fact it will only terminate when applied to a list with an odd number of elements. Figure 2.4(b) illustrates how a rewriting approach can be used to provide information about a non-terminating calculation involving this function.

(a)
\[
\begin{align*}
  f \ (x:y:xs) &= f \ xs \\
  f \ [] &= f \ [1,2] \\
  f \ [x] &= \text{True}
\end{align*}
\]

(b)
\[
\begin{align*}
  f \ [5,6,7,8] \\
  => f \ [7,8] \\
  => f \ [] \\
  => f \ [1,2] \\
  => f \ [] \\
  => f \ [1,2] \\
  => f \ [] \\
  => f \ [1,2] \\
  \ldots \text{interrupted}
\end{align*}
\]

Figure 2.4(a) definition of \( f \) the evaluation of which may not terminate; (b) an example non-terminating evaluation.
Chapter 2 - Approaches to Providing Information

Figure 2.5(b) shows a different kind of non-terminating evaluation. In this case the evaluation of the definition given in Figure 2.5(a) does not terminate because the argument to the function is not approaching the base case. Again a rewriting approach would provide the user with basic information about the evaluation of this expression.

(a)
\[
\begin{align*}
g \ (x : xs) &= g \ ((x + 1) : xs) \\
g \ [x] &= [] \\
\end{align*}
\]

(b)
\[
\begin{align*}
g \ [7, 8] \\
&\Rightarrow g \ [8, 8] \\
&\Rightarrow g \ [9, 8] \\
&\Rightarrow g \ [10, 8] \\
&\ldots \text{interrupted}
\end{align*}
\]

Figure 2.5 (a) the non-terminating function \( g \); (b) a non-termination evaluation involving the function.

An alternative rewriting approach is to indicate the distance of a function’s argument(s) from the base case of that function. This requires the user to provide the system with a function which could be used to measure this distance (in the example of Figure 2.5, this function would be based upon the length of the argument list). The system would then show the user the ‘distance’ of the argument from the base case. An example of a distance function used on the evaluation shown in Figure 2.4(b) is given in Figure 2.6 (note that in this case the distance function is \( \text{mod} \left( \text{length \ arg} \right) - 1 \)).

\[
\begin{align*}
f \ [5, 6, 7, 8] \\
&\Rightarrow f \ [7, 8] \quad \quad \quad \quad \quad \text{(distance 1)} \\
&\Rightarrow f \ [1, 2] \quad \quad \quad \quad \quad \text{(distance 1)} \\
&\Rightarrow f \ [1, 2] \quad \quad \quad \quad \quad \text{(distance 1)} \\
&\Rightarrow f \ [1, 2] \quad \quad \quad \quad \quad \text{(distance 1)} \\
&\Rightarrow f \ [1, 2] \quad \quad \quad \quad \quad \text{(distance 1)} \\
&\ldots \text{interrupted}
\end{align*}
\]

Figure 2.6 The evaluation of a function annotated with distance from base case.
A more complex example taken from my own programming experiences is based on the code extracts shown in Figure 2.7 (The complete set of definitions for this example are given in Appendix A). In this example the evaluation of the expression

```
send test_graph ([],[])```

does not terminate.

Because the graph in the example is represented using a list of vertices and edges and most of the recursion is over lists, it is straightforward to show that most of the functions are terminating. The exception is the function search, shown in Figure 2.7, which will only terminate if the number of vertices marked new in the graph $g$ is greater than the number in the graph $g'$ which is the result of an application of searchv. The distance function provided by the user in this example would be the difference between the number of new nodes in the graphs $g$ and $g'$.

```
graph == ([vertex],[edge])
vertex == ([char],bool)
edge == ([char],[char],num)
tree == graph
is_old == True

search::graph->tree->tree

search g t
    = t , marked_new g = []
    = search g' t' , otherwise
    where
        (g',t') = searchv (v,is_old) g t
        || marked_new returns singleton list containing
        || name of first "new" vertex in
        || graph or empty list if there are none
        marked_new ([],edges) = []
        marked_new (((v,is_old):vertices),edges) = [v]
        marked_new ((v:vs),edges) = marked_new (vs,edges)

test_graph = {(*"b*,True),
              (*"a*,True),
              (*"c*,True)),
              [[(*"a","b",0),
                (*"a","c",0),
                (*"b","c",0)]]
```

Figure 2.7 Extract from a real-life non terminating set of definitions
Chapter 2 - Approaches to Providing Information

Assertions

With a system which offers an assertion facility the user could make assertions about the argument(s) to and the result of a recursive call of a function. Taking the example given in Figure 2.7, an assertion could be made about the difference between the number of 'new' nodes in the argument to search and the result of the application involving searchv', this would give the following assertion:

\[
\text{assert (new_in \ g < new_in \ g')} \text{ in search}
\]

\[
\text{where new_in \ g = \ldots..}
\]

Summary

To summarise, the simplest approach to providing users with information about non-terminating evaluations is to show the evaluation in progress using a rewriting approach. This approach can be augmented by allowing the user to provide a 'distance' function which is used to indicate the distance of a function call from the function's base case. Assertions may also be used to check that the result of a recursive call has the properties necessary for termination to occur.

2.3.3 Application to Arguments Outside a Function's Domain

The domain of a function is specified using patterns and guards. It follows then that in order for users to be given information as to why an evaluation has failed, they must be shown either the domain of the function and the argument in question, or pattern matching and guard checking in progress.

Using Rewriting to Show Pattern Matching and Guard Evaluation

By showing the rewriting of an expression in a way which includes details of pattern matching and the evaluation of guards, it is possible to give information to the user about how an argument came to be outside the domain of a function. The challenge is to produce an explanation of guard evaluation and pattern matching which is simple to understand and at the same time is faithful to the semantics of the language in question. Rather than describing such an explanation system here, the reader is referred to the description presented in Chapter 7.

There is an important observation which can be made here about the provision of such information. Many contemporary systems produce information to help the user to find errors in their definitions. Predominantly these systems provide information at the level of the user's definitions. Taking the example of an evaluation which results in a
function being applied to an argument which is outside its domain, these systems merely serve to confirm what the user already knows (i.e. that the argument is outside the domain) and do not provide information about why the function has been applied to this particular argument, nor do they provide information about why the argument is outside the domain of the function. It may be the case that the user believes the argument value is in the domain of the function, in which case they need to be given information about the failure of their definition to match their beliefs. This information can only be supplied by a rewriting system if the system's model is designed to support views which show pattern-matching and guard evaluation.

Using Static Analysis to Show the Domain

Showing users the domain of their functions is most usefully done by explaining which parts of the domain are covered by which parts of the function definition - Figure 2.8 gives two examples of this. Figure 2.8(a) shows an example where the user has not defined a clause of $e$ which deals with lists containing a single element, or a clause which deals with longer lists whose first two elements are not equal.

```
(a)

f [x:x:xs] = 1
f [] = 2

domain_{def1} = \{ a \mid \text{hd a} = (\text{hd(tl a))}\}
domain_{def2} = \{[]\}
domain = domain_{def1} \cup domain_{def2}

(b)

f a = 1 , p a
f a = 2 , q a

domain_{def1} = \{a \mid \text{p a}\}
domain_{def2} = \{a \mid \text{q a}\}
domain = domain_{def1} \cup domain_{def2}
```

Figure 2.8 (a) definition of a function $f$ and its domain; (b) definition and domain of a second function $f$

An error of this kind will result from the user either inadvertently defining the domain to be smaller than was intended, or from the function being applied to an argument which was not expected and so is outside the defined domain. However, showing the user the domain of the defined functions does not provide an explanation of how the argument came to fall outside the domain of the function.
Chapter 2 - Approaches to Providing Information

Summary

To summarise, a system must provide the following information to a user who is faced with an argument which is outside the domain of a function:

- the argument;
- a derivation of the argument value;
- the domain of the function;
- the derivation of the domain of the function.

While Static Analysis can show the user the domain of a function, and the derivation of the domain in terms of sub-expressions in the clauses of the function, it provides none of the other categories of information.

Rewriting shows the user the derivation and value of a specific argument. By providing information about pattern-matching and guard evaluation, rewriting also provides information about why the domain of a function is not what the user expected it to be.

While these approaches provide very different views of a set of definitions, neither of them could be said to be generally superior to the other in the area of application of a function to an argument outside its domain.

2.3.4 Space and Time Efficiency

The goal of the programmer is to write programs that are

(i) (absolutely) correct, i.e. they should meet their specification;
(ii) (reasonably) efficient, i.e. they should consume as few machine resources as possible.

Peyton Jones '87

All of the facilities discussed so far in this chapter have been aimed at helping with errors which are related to the correctness of the program. It is also necessary to provide facilities to help users to find and correct efficiency errors in their code.

Rewriting

One approach to the provision of information about the efficiency of expressions involves information based on rewriting. Figure 2.9 shows the rewriting of
Figure 2.9 (a) the definition of the function slow_reverse; (b) the rewriting of an application of slow_reverse; (c) the definition of the function quick_reverse; (b) the rewriting of an application of quick_reverse.

expressions involving slow_reverse and quick_reverse (defined in Figures 2.9(a) and 2.9(b) respectively). By comparing the trace of slow_reverse (Figure 2.9(c)) with that of quick_reverse (Figure 2.9(d)) it is clear that quick_reverse is the more efficient of the two.

While a raw count of the number of reductions taken to evaluate the two expressions would have answered the question “Which of these two definitions is the most efficient?”, it would have failed to have given any information about “Why is one more efficient than the other?”.
Chapter 2 - Approaches to Providing Information

Profiling

Another approach is to provide information in terms of the usage of machine resources (space and time) during an evaluation. The machine in question can be either the physical machine on which the evaluation is taking place, or an abstract machine. Several existing systems which provide this type of information are discussed in Chapter 3.

The information provided by a profiling system is basically a set of numbers which give the user a breakdown of the machine resources used in order to evaluate specific sub-expressions. The work described in (Runciman & Wakeling '92) is an excellent demonstration of the usefulness of graphics to convey this information to the user. It seems certain that any approach which provided this kind of information would certainly provide a graphical view of the information it produced.

Complexity Analysis

An alternative to the provision of profiling information is to carry out static analysis of definitions. Complexity Analysis, based upon Abstract Interpretation techniques, has started to make inroads in this area. Such systems are attractive because of their ability to provide general information about functions, rather than the information provided by profilers which is execution specific. However, much of the work in this area (for example Rosendahl '89) is currently limited in terms of its scope (e.g. limited to first order functions). It must also be kept in mind that these techniques produce approximate answers, and will sometimes fail to give informative ones.

Summary

The efficiency of a definition is measured by the amount of memory used during the evaluation of its applications, and the amount of 'work' required to evaluate that expression. The approaches I have described here provide the user with information about both of these properties. There is no clearly superior way of presenting this information, and a combination of the approaches discussed above would be preferable; i.e. a combination of information about the usage of heap storage and about work done in evaluating sub-expressions, together with the more direct approach of providing information through rewriting. The limitations and nature of the information provided using abstract interpretation techniques, make this approach unsuitable at present.
Chapter 2 - Approaches to Providing Information

2.4. Help with Teaching

Any system which provides users with information about the errors in their code, can be said to be educating those users about their errors. With careful design it should be possible to widen the range of possible users of a system to allow the system to be used to educate novices as well as helping them to find their errors.

The Transparent Prolog Machine (TPM) has demonstrated that such ambitious goals can be met by a system aimed at users of a declarative programming language (Eisenstadt & Brayshaw '87). TPM is used extensively in the teaching of Prolog courses; it is used by students when they are developing programs; and it is used by the staff of the Open University to develop and debug large, real-world Prolog programs.

2.4.1 Teaching Methods

When designing a system to support teaching it is important to take stock of existing alternative approaches to teaching functional programming, and to see if and how these approaches can be supported. Of course teaching techniques vary widely between teachers. There is no fixed set of techniques used to teach functional programming using a lazy functional language. However, one can attempt to draw up a set of requirements for a system used to support teaching.

When teaching a computer language it is necessary to gradually reveal different aspects of the language to the students, rather than make any attempt to introduce everything at once. This means that the material used to teach, including any software that is used, must support a gradual exposure of the students to the material. At the same time, it is preferable to be able to keep all material consistent with that which has gone before it.

In order to support this step-by-step unfolding of the facts, a system must support the concealment of detail and its subsequent disclosure. It must be possible to support the 'story so far' without providing information about 'the next exciting instalment'. This technique of controlled disclosure needs to be supported in two areas. First, the system must allow the generation of examples by the teacher which can then be used as part of the content of lectures. Second, the system must be able to support the execution of students' programs. By using the system in this way, the teacher can produce teaching material which presents the same disclosed information as that which will be available to the students when they evaluate expressions using the
system. Furthermore, this information disclosure is directly under the control of the teacher.

2.4.2 Range of Teaching Material

A system must be able to support the teaching of many aspects of functional programming. It is common in texts on lazy functional programming (Bird & Wadler ’88, Field & Harrison ’88, Holyer ’91) for material on declarative aspects of functional programming to be taught alongside material covering operational areas such as graph reduction, lazy evaluation, and time and space optimisation. Ideally, a system used to teach from such texts must support these multiple perspectives.

In the next sections I will examine a number of problem areas which frequently occur when teaching functional programming, and I will examine approaches to providing information which could be used to support the teaching of this material.

2.4.3 Explaining Laziness

The use of a lazy evaluation scheme provides users with a number of interesting possibilities not available under a strict scheme. In order to appreciate these possibilities, students need to be given insight into the technique and consequences of lazy evaluation.

The laziness of a set of definitions can be demonstrated by carrying out Strictness Analysis on the definitions, and using the analysis to produce information about the laziness of the functions. Any such system would have to produce explanations about the laziness information that it produced. The problems of using Abstract Interpretation techniques (described in Section 2.3.4) also apply here.

```
(a)
double2 a b = b + b
bottom = bottom
e = 3 * 4

(b)
double2 bottom e
=> e + e where e = 3 * 4
=> e + e where e = 12
=> 12 + 12
=> 24
```

Figure 2.10 (a) definitions of functions used to demonstrate lazy evaluation; (b) a trace of an evaluation involving these functions which illustrates lazy evaluation in progress
Chapter 2 - Approaches to Providing Information

An alternative approach is to use a system which shows the rewriting of expressions. Figure 2.10(b) demonstrates this by showing that the first argument to the function `double2` (defined in Figure 2.10(a)) remains unevaluated, and using a `where` clause, shows that the second argument is evaluated only once.

2.4.4 Explaining Pattern Matching

Explaining pattern matching can involve two things. Students can be shown how patterns govern the selection of the right hand sides of their definitions, and they can be shown how pattern matching changes the order in which expressions are evaluated.

Both of these needs can be satisfied by providing the student with an operational model of pattern matching which can be used to illustrate the pattern matching involved in the evaluation of their definitions. Such operational models are presented in more depth in Chapter 3.

It is true that a non-operational approach could provide users with information about the strictness of their definitions due to the patterns used, but this approach depends on abstract interpretation techniques, and tends towards the provision of information about what properties the definitions display rather than why the definitions display those properties. For example given the definitions given in Figure 2.11, a system would be able to tell the user that the function `f` was strict in both of its arguments. However, such a system would be unlikely to explain that this was due to the patterns used in the definition of `g` and the guard used in the definition of `h`.

```
f a b = g a + h b
where
  g l = 10
  g n = n + 100
  h b = 2, if b / 2 = 0
       = b, otherwise
```

Figure 2.11 definition of function `f` which is strict in both of its arguments.

2.4.5 Explaining Higher-order Functions and Partial Application

One of the problems encountered by students new to functional programming is understanding the concept of higher order functions. One way in which this could be overcome would be to have a system which was capable of illustrating evaluations involving higher order values. This means that a system must be capable of incorporating higher order and partially evaluated values into the information that it
supplies. An evaluation which involves the display of higher order values is shown in Figure 2.12(b) using the definitions given in 2.12(a).

(a)
\[
\begin{align*}
\text{inc} &= (+ 1) \\
\text{my}_\text{last} [x] &= x \\
\text{my}_\text{last} (x:xs) &= \text{my}_\text{last} xs
\end{align*}
\]

(b)
\[
\begin{align*}
\text{my}_\text{last} (\text{map inc [1,2,3]}) \\
&\Rightarrow \text{my}_\text{last} ((\text{inc} 1) : (\text{map inc [2,3]))) \\
&\Rightarrow \text{my}_\text{last} (\text{inc} 1 : (\text{inc} 2 : \text{map inc [3]})) \\
&\Rightarrow \text{my}_\text{last} (\text{inc} 2 : \text{map inc [3]}) \\
&\Rightarrow \text{my}_\text{last} (\text{inc} 2 : (\text{inc} 3 : \text{map inc []})) \\
&\Rightarrow \text{my}_\text{last} ((\text{inc} 3) : (\text{map inc []})) \\
&\Rightarrow \text{my}_\text{last} ((\text{inc} 3) : []) \\
&\Rightarrow (\text{inc} 3) \\
&\Rightarrow (3 + 1) \\
&\Rightarrow 4
\end{align*}
\]

Figure 2.12 (a) definitions of functions used to demonstrate the use of higher-order function; (b) a trace of an evaluation involving these functions which illustrates the use of higher-order functions

2.4.6 Illustrating Algorithms

Figure 2.9 demonstrates another possible use for a system which presents the evaluation of expressions. The figure illustrates the difference between two algorithms to reverse a list. It is clear that any system which can provide information about the evaluation of different algorithms can be used to compare those algorithms in terms of the information provided.

Ideally these illustrations would be visible to the user at the same time, to allow easy comparison, and the system would allow the designer of the illustration to customise the presentation in such a way as to highlight the relevant differences between the algorithms. Essentially the system I am describing here would have the facilities of the algorithm animation system described in (Brown '88) which was designed to illustrate algorithms written in imperative languages.

2.5. Summary

In order to be able to provide information to users faced with errors in their definitions, or users learning about aspects of lazy functional languages, it is necessary to build a system based on a suitable evaluation model, and to provide users
Chapter 2 - Approaches to Providing Information

with a variety of views of evaluations. This chapter has presented a number of the possible evaluation models and views which could be made available to users of Miranda.

A taxonomy of errors frequently encountered by users of a lazy functional programming system has been presented, together with a number of approaches to providing information about an evaluation.

Each of the approaches discussed here provides information which would be of use to a user faced with a particular type of error. Very few of the approaches provide useful information for all of the errors.

Assertions appear to be useful when faced with an evaluation which produces the 'wrong' answer, a non-terminating evaluation, or the application of a function to an argument which is outside the functions domain.

Looking more closely at the information provided by assertions, it can be seen that they provide information about whether specific properties of the user's definitions hold or not, rather than providing information about why this might be. For example assertions can confirm that a function is applied to an argument outside the function's domain, and can tell the user the value of the argument in question. What assertions fail to do is to provide the user with the events leading up to this failure. This seems only partially satisfactory in an error-finding context as well as in a teaching context, as the approach can only provide information about what properties the definitions have, rather than why the definitions exhibit these properties. In order for a user to be able to see why a particular assertion has failed or held, more and more assertions must be added to the system to glean information about the definitions in question.

A further problem associated with assertions is the possibility of assertion-checking changing the properties of an evaluation by carrying out extra evaluation necessary to check an assertion. This is a particular problem in lazily evaluated languages where undefined values can be easily expressed. The subsequent evaluation of these values by an assertion checking mechanism would be unacceptable. A detailed description of these problems is given in Chapter 3.

Like assertions, the algorithmic approach to providing information about evaluations has the advantage that a user can track down errors while working at the level of their definitions; no information about the underlying evaluation mechanism is disclosed to the user. Also, the algorithmic approach is a declarative one; the user is asked to validate the results of expression rewrites.
Chapter 2 - Approaches to Providing Information

The algorithmic approach goes some way towards steering the user towards the source of their error, as long as that error is either a non-terminating evaluation or a 'wrong' answer. The approach becomes less useful when operational information is required, for example in order to be able to see where definitions are failing to behave as expected, it may be necessary to be given fine-grain information about pattern matching. This lack of operational information leads to this approach telling the user what is going wrong with their definitions but only providing partial information about why. The main disadvantages of the algorithmic approach are its limited scope and the lack of fine-grain operational information.

Narrowness of scope is also the main short-fall of the profiling approach. By definition this approach only provides efficiency information. As one would expect from such a focused approach, the profiling information that it produces is superior to profiling information produced by other approaches.

Abstract Interpretation is a developing area. Some of the problems of the approach which I have outlined here are surmountable. For example, while current systems are not designed to explain the basis of their declarations about the strictness of definitions, systems which provided such explanations could be developed. However, the nature of systems based on Abstract Interpretation is that they are designed to provide information based on approximations, and that in some cases it will be impossible to produce any useful information.

Of all of the approaches presented in this chapter, the rewriting approach has been shown to be the only approach which can provide information to the user faced with all of the kinds of errors listed here. While in some cases this information may be inferior to that produced by other approaches (e.g. a well designed profiling systems will provide superior profiling information), it will always be possible to provide some useful information.

A system based on rewriting offers the possibility of providing a large number of different views of an evaluation. While such an approach explicitly presents the laziness of the evaluation scheme, I predict that users will be able to use such systems, since they present the evaluation, and do not demand that the user predict it.

This chapter has demonstrated the need for the provision of low-level information to users. For example, in order to provide detailed information about the application of a function to an argument outside its domain, a low-level view of the pattern-matching and guard evaluation mechanisms must be provided. Rewriting is ideally suited to providing such a view of an evaluation, provided that the system is based upon a
model designed to support such views. One such model is presented in detail in Chapter 7.

Before useful views of an evaluation can be provided to the user it is also necessary to overcome a number of problems associated with the rewriting approach; these problems are discussed in detail in the following chapter. Tools must be provided with which to manage the potentially large volumes of information which this approach can generate, and with which to allow the user to navigate through the potentially large number of reduction steps produced by such an approach. Chapters 5 and 6 describe such tools.
2 Approaches to Providing Information

In this chapter I introduce the reader to the problem of finding errors in functional language definitions, and to the requirements of software support for the teaching of lazy functional programming. The chapter presents a taxonomy of the errors frequently encountered by users of a functional programming system, and presents a number of approaches which could be used to provide information to users about these errors.

This thesis is in part an exploration of the possible 'stories' which can be presented to users about the lazy evaluation of expressions. This chapter sets out some of the issues which must be considered when producing such a story and goes on to explore a number of candidate 'stories' which could be used as the basis for a system such as Prospero. The suitability of each of the 'stories' to each of the errors in the taxonomy is then assessed.

In Chapter 3 I will go on to describe existing systems which implement some of these approaches.

In Chapter 4 I will return to the examples presented here and show how Prospero provides information about the errors discussed in this chapter.

2.1. Types of Error

In this section I will present some of the types of errors that occur in users' definitions. In the following section I will go on to describe some of the approaches to providing the user with information which could help the user to find these errors. Following on from this I will tie these two sections together, by showing which of these approaches is suited to which type of error.
Chapter 2 - Approaches to Providing Information

The list of errors discussed here, has come from three main sources: personal experiences following extensive use of the lazy language Miranda; the comments of the staff of the Department of Computer Science, Queen Mary and Westfield College; and my observations of novices learning to use the strict functional language ML. As such, it would be incorrect to present this as a complete taxonomy of errors.

2.1.1 Wrong Answer

One of the most common types of error introduced by users into their definitions, is that a function may produce a value which is not the one expected by the user. There are three main causes for this type of error:

- the algorithm contains an error;
- the translation of the algorithm into Miranda contains an error;
- the user misunderstands the semantics of some other functions - in particular, those functions defined in the Miranda standard environment.

The fact that Miranda is a strongly-typed language helps reduce the number of errors caused by the second and third of the causes listed above, but experience shows that these errors still occur frequently.

2.1.2 Non-Termination

Recursion enters into all but the simplest of Miranda definitions, and once it has been introduced, there is the possibility of defining an expression whose evaluation will not terminate. Again the cause of a non-terminating evaluation could be a flaw in the algorithm, or a flaw in the translation of the algorithm into Miranda.

2.1.3 Application to Arguments Outside a Function’s Domain

Another common type of error results from the user defining a partial function and then applying the function to an argument whose value is outside the domain of the function.

2.1.4 Space and Time

The final type of error considered here occurs when an evaluation requires more memory, or takes longer to execute than was expected. It could be argued that this is not actually an error. But at its most extreme a program that takes all day rather than a few minutes to run is not a useful one. Even if problems of this type are not perceived
as errors, it is undoubtedly true that useful information can be provided to the user about the efficiency of their definitions.

2.2. Approaches to Providing Information about Errors

Following on from the taxonomy of errors given above, I will describe some of the approaches to providing information which may help the user to find these errors.

2.2.1 Rewriting

The first approach is to show the rewriting of expressions. This approach involves showing the user the step-by-step transformation of their original expression, using function definitions as rewrite rules, to its fully evaluated form. This transformation process involves the rewriting of an expression using a set of simplification rules (e.g. \(3+4\) simplifies to \(7\)), and the equations defined by the user.

2.2.2 Assertions

The second approach is the provision of program annotations with which the user may make assertions about properties of their definitions. Typically these assertions will concern the range and domain of functions (e.g. asserting that the argument to a function will always be a string whose length is greater than 1 character).

2.2.3 Algorithmic Debugging

The next alternative is an 'algorithmic' approach. Algorithmic debugging was pioneered by Shapiro for the language Prolog. In (Shapiro '82) a technique of program debugging is described; this technique can be summarised as follows.

The program to be debugged is executed and a complete execution space is built in the form of a call graph for the evaluation of the program; this trace holds information about the arguments and results of all function calls. Once this evaluation trace has been built, the user is walked through the execution space, and is shown the values of arguments and results at each point. At each of these points, the user is asked whether the result is correct for the given argument. If the answer is no, then the error must be either in the function which the walk has reached, or in one of the functions called by the current function. In order to establish this, the system explores all of the function calls made by the current function, and either one of these returns an incorrect result, in which case the walk continues by examining that function call, or all of the functions return the correct result, in which case the current function is at fault and the error has been located.
2.2.4 Profiling

Profiling systems provide information to users in terms of the time and space usage of their definitions during an evaluation. As such these systems will only be considered in the discussion of efficiency problems faced by users.

2.2.5 Abstract Interpretation

The field of abstract interpretation can provide various types of information about functional language definitions. For example it can provide information about the strictness of a definition, or the order of complexity of a definition. There are a number of drawbacks to these approaches which will be explored later in this chapter.

2.3. Help with Errors

In this section I will examine how each of the types of error introduced earlier in this chapter may be tackled using, where appropriate, each of the approaches described above.

2.3.1 Wrong Answer

Presentation Using Rewriting

A system that presents the rewriting of an expression can provide information to a user about why an expression evaluated to an unexpected value. For example, Figure 2.1(a) shows a definition of a tree-membership function, for a tree which is defined such that values on the left of a branch are always less than the values at the branch, or on the right of that branch. This tree-membership function’s recursive clause is defined in terms of the wrong branch of the tree (i.e. the condition $v < v'$ should read $v \geq v'$, as shown in Figure 2.1(c)). Figure 2.1(b) shows how a trace based on rewriting, could be used to provide information about the error.

The rewriting shown in Figure 2.1(b) gives a step-by-step view of the evaluation of the expression `tree_member treel 1`. Each of the steps in the evaluation is produced by replacing sub-expressions with their appropriate definition taken from those given in Figure 2.1 (a).
Chapter 2 - Approaches to Providing Information

(a)

\[
\text{tree} * ::= \text{Branch} \ (\text{tree} *) \ * \ (\text{tree} *) \mid \text{Leaf}
\]

\[
\text{tree\_member} \ (\text{Branch} \ l \ v \ r) \ v' = \begin{cases} 
\text{True} & , v = v' \\
\text{tree\_member} \ l \ v', & \text{if } v < v' \\
\text{tree\_member} \ r \ v', & \text{otherwise}
\end{cases}
\]

\[
\text{tree\_member} \ \text{Leaf} \ v = \text{False}
\]

\[
\text{treel} = \text{Branch} \\
\hspace{1em} (\text{Branch} \ \text{Leaf} \ 1 \ \text{Leaf}) \\
\hspace{2em} 2 \\
\hspace{2em} (\text{Branch} \ \text{Leaf} \ 3 \ (\text{Branch} \ \text{Leaf} \ 4 \ \text{Leaf}))
\]

(b)

\[
\text{tree\_member} \ \text{treel} \ 1 \\
\Rightarrow \text{tree\_member} \ (\text{Branch} \\
\hspace{1em} (\text{Branch} \ \text{Leaf} \ 1 \ \text{Leaf}) \\
\hspace{2em} 2 \\
\hspace{2em} (\text{Branch} \ \text{Leaf} \ 3 \ (\text{Branch} \ \text{Leaf} \ 4 \ \text{Leaf})) \\
\hspace{3em} 1 \\
\Rightarrow \text{tree\_member} \ (\text{Branch} \ \text{Leaf} \ 3 \ (\text{Branch} \ \text{Leaf} \ 4 \ \text{Leaf})) \ 1 \\
\Rightarrow \text{tree\_member} \ (\text{Branch} \ \text{Leaf} \ 4 \ \text{Leaf}) \ 1 \\
\Rightarrow \text{tree\_member} \ \text{Leaf} \ 1 \\
\Rightarrow \text{False}
\]

(c)

\[
\text{tree} * ::= \text{Branch} \ (\text{tree} *) \ * \ (\text{tree} *) \mid \text{Leaf}
\]

\[
\text{tree\_member} \ (\text{Branch} \ l \ v \ r) \ v' = \begin{cases} 
\text{True} & , v = v' \\
\text{tree\_member} \ l \ v', & \text{if } v >= v' \\
\text{tree\_member} \ r \ v', & \text{otherwise}
\end{cases}
\]

\[
\text{tree\_member} \ \text{Leaf} \ v = \text{False}
\]

---

Figure 2.1(a) Definition of function \text{tree\_member} containing an error; (b) a trace of the function highlighting the error; (c) the correct definition of \text{tree\_member}.

The example in Figure 2.1 shows that rewriting can be used to produce a step-by-step picture of the evaluation of an expression which results in an unexpected answer. In this example, the level at which the information is produced is equal to that of the user's definitions, i.e. each stage shows only information from the user's definitions and give no information about the underlying evaluation model, for example no information about the selection of the correct clause of \text{tree\_member} is given. Later sections in this chapter will demonstrate that the ability to show this low-level information is a major strength of the rewriting approach.
(a)

\[
\begin{align*}
tree^* & ::= \text{Branch (tree *) * (tree *) | Leaf} \\
\text{insert Leaf } v &= \text{Branch Leaf } v \text{ Leaf} \\
\text{insert } t \ v &= t, \text{ tree_member } t \ v \\
\text{insert (Branch } l \ v \ r) \ v' &= \text{insert } l \ v' \ , \ v' < v \\
&= \text{insert } r \ v' \ , \ \text{otherwise}
\end{align*}
\]

| N.B. tree_member defined as in Figure 2.1(a)

\[
\begin{align*}
treel &= \text{Branch} \\
& \quad (\text{Branch Leaf } 1 \text{ Leaf}) \\
& \quad 2 \\
& \quad (\text{Branch Leaf } 3 \ (\text{Branch Leaf } 4 \text{ Leaf}))
\end{align*}
\]

\[
\begin{align*}
\text{main} &= \text{insert treel } 2 \\
\end{align*}
\]

(b)

\[
\begin{align*}
\text{main} & \Rightarrow \text{Branch (Branch Leaf } 1 \text{ Leaf)} \\
& \quad 2 \\
& \quad (\text{Branch (Branch Leaf } 2 \text{ Leaf)} \\
& \quad \quad 3 \\
& \quad (\text{Branch Leaf } 4 \text{ Leaf}))
\end{align*}
\]

\[
\begin{align*}
(y/n)? \ n
\end{align*}
\]

\[
\begin{align*}
\text{tree_member Branch ((Branch Leaf } 1 \text{ Leaf)} \\
& \quad 2 \\
& \quad (\text{Branch Leaf } 3 \ (\text{Branch Leaf } 4 \text{ Leaf}))) \ 2 \Rightarrow \text{False}
\end{align*}
\]

\[
\begin{align*}
(y/n)? \ n
\end{align*}
\]

Error is in function "tree_member"

Figure 2.2 (a) the definition of a tree insertion function containing the erroneous definition of tree_member defined in Figure 2.1(a); (b) the use of an algorithmic debugger to locate the error

An Algorithmic Approach

An illustration of the use of an algorithmic debugger to find the error in the definitions given in Figure 2.2(a) is shown in Figure 2.2(b). While the tree insertion function in

\[1\] Throughout this thesis I shall use the following convention for representing interactions between a system and a user. Any text which is printed by the system with which the user is interacting will appear in courier. Any text which is entered by the user will appear in courier bold

- 39 -
this figure is correct, it makes use of the function \texttt{tree\_member} (defined in Figure 2.1(a)) which contains an error.

In this example the user is asked whether particular steps of the evaluation are correct. The first question that the user is asked is "Should \texttt{main} evaluate to this particular tree?" to which the user answers \texttt{no}. The next question is about the result of searching for the value 2 in the tree \texttt{tree1}, and leads the user to the location of the error. Of course the length of this session is not representative and the user would not expect to find the error so quickly. More complex and realistic examples are described in (Nilsson & Fritzson '92).

A system which offers an algorithmic approach to finding errors allows the user to work at the declarative level at which they wrote their definitions. This means that in order to find errors in their definitions, users need only know what values expressions should evaluate to. However, an algorithmic system does not provide all of the information that may be required about an evaluation. For example, it may be the case that the user has an interest in the order in which expressions are being evaluated.

\textit{Assertions}

Figure 2.3 shows the use of assertions to locate an error in the definition of the function \texttt{funny\_reverse} given in Figure 2.3(a)\textsuperscript{1}. This function takes four arguments: an index into the list to be reversed; the maximum value of the index (i.e. the length of the list to be reversed); the list to be reversed; and an accumulator into which the result of the function is placed, one list element at a time.

\begin{footnotesize}
\footnotesize
\begin{itemize}
\item \texttt{(list \_! n)} returns the \texttt{n}th element of \texttt{list}, where the head of a list is the 0th element.
\item \texttt{\sim{=}} denotes the inequality operator.
\end{itemize}
\end{footnotesize}
Chapter 2 - Approaches to Providing Information

(a) 
funny_reverse index limit 11 12
    = funny_reverse (index + 1) limit (t1 11) ((11 ! index) : 12)
    , index ~ limit
    = 12 , otherwise

(b) 
Miranda funny_reverse 0 3 [1,2,3] []
SUBSCRIPT ERROR
Miranda

(c) 
assert index + limit = (length 11) + (length 12)
in funny_reverse
funny_reverse 0 3 [1,2,3] []
assertion failed for:
funny_reverse 1 3 [2,3] [1]

(d) 
funny_reverse index limit 11 12
    = funny_reverse (index + 1) limit 11 ((11 ! index) : 12)
    , index ~ limit
    = 12 , otherwise

Figure 2.3 (a) definition of the function funny_reverse containing an error; (b) transcript of Miranda session involving evaluation of the function; (c) an assertion used to locate the error in the function; (d) the corrected version of the function

Figure 2.3(b) shows the error message resulting from the evaluation of an expression involving this definition. Figure 2.3(c) shows the use of an assertion to check that an expected property holds for the definition. Finally, Figure 2.3(d) shows the correct definition of funny_reverse with the correction to the definition underlined. Note that this function definition is being used to illustrate the use of assertions, and is not being put forward as a realistic definition of a list-reversal function.

Combining assertions with a lazily evaluated language should be done with care; it is possible that the use of assertions will change the termination properties of an evaluation by causing extra evaluation to take place in order to check the validity of the assertion. This problem is discussed in detail in Chapter 3.

Like the algorithmic approach described above, assertions provide one particular kind of the information that may be required about an evaluation. As such, the approach must be viewed as a specialised, rather than general, approach to providing information.
Chapter 2 - Approaches to Providing Information

Summary

Rewriting, algorithmic debugging and assertions can all be used in different ways to provide information about an evaluation which produces the wrong value. All of these approaches provide a very different, and valid approach to dealing with such problems. Rewriting leads the user through the evaluation showing how the value came to be what it was. The algorithmic system takes a much more declarative approach and allows the user to narrow down the definitions at fault by asking questions about the values produced by the definitions. Assertions allow the user to express their beliefs about a set of definitions in order to track down a definition which is behaving incorrectly.

2.3.2 Non-Termination

Rewriting

Figure 2.4(a) shows a definition whose application will sometimes fail to terminate. In fact it will only terminate when applied to a list with an odd number of elements. Figure 2.4(b) illustrates how a rewriting approach can be used to provide information about a non-terminating calculation involving this function.

(a)
\[
\begin{align*}
& f \hspace{1em} (x\hspace{1em} y\hspace{1em} xs) = f \hspace{1em} xs \\
& f \hspace{1em} [] = f \hspace{1em} [1,2] \\
& f \hspace{1em} [x] = \text{True}
\end{align*}
\]

(b)
\[
\begin{align*}
& f \hspace{1em} [5,6,7,8] \\
& => f \hspace{1em} [7,8] \\
& => f \hspace{1em} [] \\
& => f \hspace{1em} [1,2] \\
& => f \hspace{1em} [] \\
& => f \hspace{1em} [1,2] \\
& => f \hspace{1em} [] \\
& => f \hspace{1em} [1,2] \\
& ... \text{interrupted}
\end{align*}
\]

Figure 2.4(a) definition of $f$ the evaluation of which may not terminate; (b) an example non-terminating evaluation.
Figure 2.5(b) shows a different kind of non-terminating evaluation. In this case the evaluation of the definition given in Figure 2.5(a) does not terminate because the argument to the function is not approaching the base case. Again a rewriting approach would provide the user with basic information about the evaluation of this expression.

(a)
\[
g \ (x:xs) = g \ ((x+1) : xs) \\
g \ [] = []
\]

(b)
\[
g \ [7,8] \\
=> g \ [8,8] \\
=> g \ [9,8] \\
=> g \ [10,8] \\
... interrupted
\]

Figure 2.5 (a) the non-terminating function \(g\); (b) a non-termination evaluation involving the function.

An alternative rewriting approach is to indicate the distance of a function’s argument(s) from the base case of that function. This requires the user to provide the system with a function which could be used to measure this distance (in the example of Figure 2.5, this function would be based upon the length of the argument list). The system would then show the user the ‘distance’ of the argument from the base case. An example of a distance function used on the evaluation shown in Figure 2.4(b) is given in Figure 2.6 (note that in this case the distance function is \(\text{mod} \ ((\text{length } \text{arg}) - 1)\).

\[
f \ [5,6,7,8] \\
=> f \ [7,8] \quad \text{(distance 1)} \\
=> f \ [] \quad \text{(distance 1)} \\
=> f \ [1,2] \quad \text{(distance 1)} \\
=> f \ [] \quad \text{(distance 1)} \\
=> f \ [1,2] \quad \text{(distance 1)} \\
=> f \ [] \quad \text{(distance 1)} \\
=> f \ [1,2] \quad \text{(distance 1)} \\
... interrupted
\]

Figure 2.6 The evaluation of a function annotated with distance from base case.
A more complex example taken from my own programming experiences is based on the code extracts shown in Figure 2.7 (The complete set of definitions for this example are given in Appendix A). In this example the evaluation of the expression

```
search test_graph([[],[[]])
```

does not terminate.

Because the graph in the example is represented using a list of vertices and edges and most of the recursion is over lists, it is straightforward to show that most of the functions are terminating. The exception is the function `search`, shown in Figure 2.7, which will only terminate if the number of vertices marked `new` in the graph $g$ is greater than the number in the graph $g'$, which is the result of an application of `searchv`. The distance function provided by the user in this example would be the difference between the number of `new` nodes in the graphs $g$ and $g'$.

```
graph == ([:vertex],[:edge])
vertex == ([:char],[bool])
edge == ([:char],[:char],[num])
tree == graph
is_old == True

search::graph->tree->tree

search $g$ $t$
  = $t$, marked_new $g$ = []
  = search $g'$ $t'$, otherwise
  where
  ($g'$,$t'$) = searchv ($v$,is_old) $g$ $t$
  || marked_new returns singleton list containing
  || name of first "new" vertex in
  || graph or empty list if there are none
  marked_new ([[],edges]) = []
  marked_new (((v,is_old):vertices),edges) = [v]
  marked_new ((v:vs),edges) = marked_new (vs,edges)

test_graph = ([("b",True),
  ("a",True),
  ("c",True)],
  [{"a","b",0},
  {"a","c",0},
  {"b","c",0}])
```

Figure 2.7 Extract from a real-life non terminating set of definitions
Chapter 2 - Approaches to Providing Information

Assertions

With a system which offers an assertion facility the user could make assertions about the argument(s) to and the result of a recursive call of a function. Taking the example given in Figure 2.7, an assertion could be made about the difference between the number of 'new' nodes in the argument to search and the result of the application involving searchV', this would give the following assertion:

\[
\text{assert (new_in } g < \text{new_in } g') \text{ in search}
\]

where new_in \( g = \ldots \ldots \)

Summary

To summarise, the simplest approach to providing users with information about non-terminating evaluations is to show the evaluation in progress using a rewriting approach. This approach can be augmented by allowing the user to provide a 'distance' function which is used to indicate the distance of a function call from the function's base case.Assertions may also be used to check that the result of a recursive call has the properties necessary for termination to occur.

2.3.3 Application to Arguments Outside a Function's Domain

The domain of a function is specified using patterns and guards. It follows then that in order for users to be given information as to why an evaluation has failed, they must be shown either the domain of the function and the argument in question, or pattern matching and guard checking in progress.

Using Rewriting to Show Pattern Matching and Guard Evaluation

By showing the rewriting of an expression in a way which includes details of pattern matching and the evaluation of guards, it is possible to give information to the user about how an argument came to be outside the domain of a function. The challenge is to produce an explanation of guard evaluation and pattern matching which is simple to understand and at the same time is faithful to the semantics of the language in question. Rather than describing such an explanation system here, the reader is referred to the description presented in Chapter 7.

There is an important observation which can be made here about the provision of such information. Many contemporary systems produce information to help the user to find errors in their definitions. Predominantly these systems provide information at the level of the user's definitions. Taking the example of an evaluation which results in a
function being applied to an argument which is outside its domain, these systems merely serve to confirm what the user already knows (i.e. that the argument is outside the domain) and do not provide information about why the function has been applied to this particular argument, nor do they provide information about why the argument is outside the domain of the function. It may be the case that the user believes the argument value is in the domain of the function, in which case they need to be given information about the failure of their definition to match their beliefs. This information can only be supplied by a rewriting system if the system's model is designed to support views which show pattern-matching and guard evaluation.

Using Static Analysis to Show the Domain

Showing users the domain of their functions is most usefully done by explaining which parts of the domain are covered by which parts of the function definition - Figure 2.8 gives two examples of this. Figure 2.8(a) shows an example where the user has not defined a clause of \( f \) which deals with lists containing a single element, or a clause which deals with longer lists whose first two elements are not equal.

![Code snippet](image)

Figure 2.8 (a) definition of a function \( f \) and its domain; (b) definition and domain of a second function \( f \)

An error of this kind will result from the user either inadvertently defining the domain to be smaller than was intended, or from the function being applied to an argument which was not expected and so is outside the defined domain. However, showing the user the domain of the defined functions does not provide an explanation of how the argument came to fall outside the domain of the function.
Chapter 2 - Approaches to Providing Information

Summary

To summarise, a system must provide the following information to a user who is faced with an argument which is outside the domain of a function:

- the argument;
- a derivation of the argument value;
- the domain of the function;
- the derivation of the domain of the function.

While Static Analysis can show the user the domain of a function, and the derivation of the domain in terms of sub-expressions in the clauses of the function, it provides none of the other categories of information.

Rewriting shows the user the derivation and value of a specific argument. By providing information about pattern-matching and guard evaluation, rewriting also provides information about why the domain of a function is not what the user expected it to be.

While these approaches provide very different views of a set of definitions, neither of them could be said to be generally superior to the other in the area of application of a function to an argument outside its domain.

2.3.4 Space and Time Efficiency

The goal of the programmer is to write programs that are

(i) (absolutely) correct, i.e. they should meet their specification;
(ii) (reasonably) efficient, i.e. they should consume as few machine resources as possible.

Peyton Jones '87

All of the facilities discussed so far in this chapter have been aimed at helping with errors which are related to the correctness of the program. It is also necessary to provide facilities to help users to find and correct efficiency errors in their code.

Rewriting

One approach to the provision of information about the efficiency of expressions involves information based on rewriting. Figure 2.9 shows the rewriting of
Figure 2.9 (a) the definition of the function slow_reverse; (b) the rewriting of an application of slow_reverse; (c) the definition of the function quick_reverse; (b) the rewriting of an application of quick_reverse.

expressions involving slow_reverse and quick_reverse (defined in Figures 2.9(a) and 2.9(b) respectively). By comparing the trace of slow_reverse (Figure 2.9(c)) with that of quick_reverse (Figure 2.9(d)) it is clear that quick_reverse is the more efficient of the two.

While a raw count of the number of reductions taken to evaluate the two expressions would have answered the question “Which of these two definitions is the most efficient?”, it would have failed to have given any information about “Why is one more efficient than the other?”.
Chapter 2 - Approaches to Providing Information

Profiling

Another approach is to provide information in terms of the usage of machine resources (space and time) during an evaluation. The machine in question can be either the physical machine on which the evaluation is taking place, or an abstract machine. Several existing systems which provide this type of information are discussed in Chapter 3.

The information provided by a profiling system is basically a set of numbers which give the user a breakdown of the machine resources used in order to evaluate specific sub-expressions. The work described in (Runciman & Wakeling '92) is an excellent demonstration of the usefulness of graphics to convey this information to the user. It seems certain that any approach which provided this kind of information would certainly provide a graphical view of the information it produced.

Complexity Analysis

An alternative to the provision of profiling information is to carry out static analysis of definitions. Complexity Analysis, based upon Abstract Interpretation techniques, has started to make inroads in this area. Such systems are attractive because of their ability to provide general information about functions, rather than the information provided by profilers which is execution specific. However, much of the work in this area (for example Rosendahl '89) is currently limited in terms of its scope (e.g. limited to first order functions). It must also be kept in mind that these techniques produce approximate answers, and will sometimes fail to give informative ones.

Summary

The efficiency of a definition is measured by the amount of memory used during the evaluation of its applications, and the amount of ‘work’ required to evaluate that expression. The approaches I have described here provide the user with information about both of these properties. There is no clearly superior way of presenting this information, and a combination of the approaches discussed above would be preferable; i.e. a combination of information about the usage of heap storage and about work done in evaluating sub-expressions, together with the more direct approach of providing information through rewriting. The limitations and nature of the information provided using abstract interpretation techniques, make this approach unsuitable at present.
2.4. Help with Teaching

Any system which provides users with information about the errors in their code, can be said to be educating those users about their errors. With careful design it should be possible to widen the range of possible users of a system to allow the system to be used to educate novices as well as helping them to find their errors.

The Transparent Prolog Machine (TPM) has demonstrated that such ambitious goals can be met by a system aimed at users of a declarative programming language (Eisenstadt & Brayshaw '87). TPM is used extensively in the teaching of Prolog courses; it is used by students when they are developing programs; and it is used by the staff of the Open University to develop and debug large, real-world Prolog programs.

2.4.1 Teaching Methods

When designing a system to support teaching it is important to take stock of existing alternative approaches to teaching functional programming, and to see if and how these approaches can be supported. Of course teaching techniques vary widely between teachers. There is no fixed set of techniques used to teach functional programming using a lazy functional language. However, one can attempt to draw up a set of requirements for a system used to support teaching.

When teaching a computer language it is necessary to gradually reveal different aspects of the language to the students, rather than make any attempt to introduce everything at once. This means that the material used to teach, including any software that is used, must support a gradual exposure of the students to the material. At the same time, it is preferable to be able to keep all material consistent with that which has gone before it.

In order to support this step-by-step unfolding of the facts, a system must support the concealment of detail and its subsequent disclosure. It must be possible to support the 'story so far' without providing information about 'the next exciting instalment'. This technique of controlled disclosure needs to be supported in two areas. First, the system must allow the generation of examples by the teacher which can then be used as part of the content of lectures. Second, the system must be able to support the execution of students' programs. By using the system in this way, the teacher can produce teaching material which presents the same disclosed information as that which will be available to the students when they evaluate expressions using the
system. Furthermore, this information disclosure is directly under the control of the teacher.

2.4.2 Range of Teaching Material

A system must be able to support the teaching of many aspects of functional programming. It is common in texts on lazy functional programming (Bird & Wadler '88, Field & Harrison '88, Holyer '91) for material on declarative aspects of functional programming to be taught alongside material covering operational areas such as graph reduction, lazy evaluation, and time and space optimisation. Ideally, a system used to teach from such texts must support these multiple perspectives.

In the next sections I will examine a number of problem areas which frequently occur when teaching functional programming, and I will examine approaches to providing information which could be used to support the teaching of this material.

2.4.3 Explaining Laziness

The use of a lazy evaluation scheme provides users with a number of interesting possibilities not available under a strict scheme. In order to appreciate these possibilities, students need to be given insight into the technique and consequences of lazy evaluation.

The laziness of a set of definitions can be demonstrated by carrying out Strictness Analysis on the definitions, and using the analysis to produce information about the laziness of the functions. Any such system would have to produce explanations about the laziness information that it produced. The problems of using Abstract Interpretation techniques (described in Section 2.3.4) also apply here.

<table>
<thead>
<tr>
<th>(a)</th>
<th>(b)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>double2 a b = b+b</code></td>
<td><code>double2 bottom e</code></td>
</tr>
<tr>
<td><code>bottom = bottom</code></td>
<td><code>=&gt; e + e where e = 3 * 4</code></td>
</tr>
<tr>
<td><code>e = 3 * 4</code></td>
<td><code>=&gt; e + e where e = 12</code></td>
</tr>
<tr>
<td></td>
<td><code>=&gt; 12 + 12</code></td>
</tr>
<tr>
<td></td>
<td><code>=&gt; 24</code></td>
</tr>
</tbody>
</table>

Figure 2.10 (a) definitions of functions used to demonstrate lazy evaluation; (b) a trace of an evaluation involving these functions which illustrates lazy evaluation in progress
Chapter 2 - Approaches to Providing Information

An alternative approach is to use a system which shows the rewriting of expressions. Figure 2.10(b) demonstrates this by showing that the first argument to the function `double2` (defined in Figure 2.10(a)) remains unevaluated, and using a `where` clause, shows that the second argument is evaluated only once.

### 2.4.4 Explaining Pattern Matching

Explaining pattern matching can involve two things. Students can be shown how patterns govern the selection of the right hand sides of their definitions, and they can be shown how pattern matching changes the order in which expressions are evaluated.

Both of these needs can be satisfied by providing the student with an operational model of pattern matching which can be used to illustrate the pattern matching involved in the evaluation of their definitions. Such operational models are presented in more depth in Chapter 3.

It is true that a non-operational approach could provide users with information about the strictness of their definitions due to the patterns used, but this approach depends on abstract interpretation techniques, and tends towards the provision of information about what properties the definitions display rather than why the definitions display those properties. For example given the definitions given in Figure 2.11, a system would be able to tell the user that the function \( f \) was strict in both of its arguments. However, such a system would be unlikely to explain that this was due to the patterns used in the definition of \( g \) and the guard used in the definition of \( h \).

```plaintext
\[
f \ a \ b = g \ a + h \ b
\]

where

\[
g \ l = 10
\]
\[
g \ n = n + 100
\]
\[
h \ b = 2 , \text{ if } b / 2 = 0
\]
\[
= b , \text{ otherwise}
\]

Figure 2.11 definition of function \( f \) which is strict in both of its arguments.

### 2.4.5 Explaining Higher-order Functions and Partial Application

One of the problems encountered by students new to functional programming is understanding the concept of higher order functions. One way in which this could be overcome would be to have a system which was capable of illustrating evaluations involving higher order values. This means that a system must be capable of incorporating higher order and partially evaluated values into the information that it
supplies. An evaluation which involves the display of higher order values is shown in Figure 2.12(b) using the definitions given in 2.12(a).

(a)
inc = (+ 1)
my_last [x] = x
my_last (x:xs) = my_last xs

(b)
my_last (map inc [1,2,3])
=> my_last ((inc 1) : (map inc [2,3]))
=> my_last (inc 1 : (inc 2 : map inc [3]))
=> my_last (inc 2 : map inc [3])
=> my_last (inc 2 : (inc 3 : map inc []))
=> my_last ((inc 3) : [])
=> (inc 3)
=> (3 + 1)
=> 4

Figure 2.12 (a) definitions of functions used to demonstrate the use of higher-order function; (b) a trace of an evaluation involving these functions which illustrates the use of higher-order functions

2.4.6 Illustrating Algorithms

Figure 2.9 demonstrates another possible use for a system which presents the evaluation of expressions. The figure illustrates the difference between two algorithms to reverse a list. It is clear that any system which can provide information about the evaluation of different algorithms can be used to compare those algorithms in terms of the information provided.

Ideally these illustrations would be visible to the user at the same time, to allow easy comparison, and the system would allow the designer of the illustration to customise the presentation in such a way as to highlight the relevant differences between the algorithms. Essentially the system I am describing here would have the facilities of the algorithm animation system described in (Brown '88) which was designed to illustrate algorithms written in imperative languages.

2.5. Summary

In order to be able to provide information to users faced with errors in their definitions, or users learning about aspects of lazy functional languages, it is necessary to build a system based on a suitable evaluation model, and to provide users
Chapter 2 - Approaches to Providing Information

with a variety of views of evaluations. This chapter has presented a number of the possible evaluation models and views which could be made available to users of Miranda.

A taxonomy of errors frequently encountered by users of a lazy functional programming system has been presented, together with a number of approaches to providing information about an evaluation.

Each of the approaches discussed here provides information which would be of use to a user faced with a particular type of error. Very few of the approaches provide useful information for all of the errors.

Assertions appear to be useful when faced with an evaluation which produces the ‘wrong’ answer, a non-terminating evaluation, or the application of a function to an argument which is outside the functions domain.

Looking more closely at the information provided by assertions, it can be seen that they provide information about whether specific properties of the user’s definitions hold or not, rather than providing information about why this might be. For example assertions can confirm that a function is applied to an argument outside the function’s domain, and can tell the user the value of the argument in question. What assertions fail to do is to provide the user with the events leading up to this failure. This seems only partially satisfactory in an error-finding context as well as in a teaching context, as the approach can only provide information about what properties the definitions have, rather than why the definitions exhibit these properties. In order for a user to be able to see why a particular assertion has failed or held, more and more assertions must be added to the system to glean information about the definitions in question.

A further problem associated with assertions is the possibility of assertion-checking changing the properties of an evaluation by carrying out extra evaluation necessary to check an assertion. This is a particular problem in lazily evaluated languages where undefined values can be easily expressed. The subsequent evaluation of these values by an assertion checking mechanism would be unacceptable. A detailed description of these problems is given in Chapter 3.

Like assertions, the algorithmic approach to providing information about evaluations has the advantage that a user can track down errors while working at the level of their definitions; no information about the underlying evaluation mechanism is disclosed to the user. Also, the algorithmic approach is a declarative one; the user is asked to validate the results of expression rewrites.
Chapter 2 - Approaches to Providing Information

The algorithmic approach goes some way towards steering the user towards the source of their error, as long as that error is either a non-terminating evaluation or a ‘wrong’ answer. The approach becomes less useful when operational information is required, for example in order to be able to see where definitions are failing to behave as expected, it may be necessary to be given fine-grain information about pattern matching. This lack of operational information leads to this approach telling the user what is going wrong with their definitions but only providing partial information about why. The main disadvantages of the algorithmic approach are its limited scope and the lack of fine-grain operational information.

Narrowness of scope is also the main short-fall of the profiling approach. By definition this approach only provides efficiency information. As one would expect from such a focused approach, the profiling information that it produces is superior to profiling information produced by other approaches.

Abstract Interpretation is a developing area. Some of the problems of the approach which I have outlined here are surmountable. For example, while current systems are not designed to explain the basis of their declarations about the strictness of definitions, systems which provided such explanations could be developed. However, the nature of systems based on Abstract Interpretation is that they are designed to provide information based on approximations, and that in some cases it will be impossible to produce any useful information.

Of all of the approaches presented in this chapter, the rewriting approach has been shown to be the only approach which can provide information to the user faced with all of the kinds of errors listed here. While in some cases this information may be inferior to that produced by other approaches (e.g. a well designed profiling systems will provide superior profiling information), it will always be possible to provide some useful information.

A system based on rewriting offers the possibility of providing a large number of different views of an evaluation. While such an approach explicitly presents the laziness of the evaluation scheme, I predict that users will be able to use such systems, since they present the evaluation, and do not demand that the user predict it.

This chapter has demonstrated the need for the provision of low-level information to users. For example, in order to provide detailed information about the application of a function to an argument outside its domain, a low-level view of the pattern-matching and guard evaluation mechanisms must be provided. Rewriting is ideally suited to providing such a view of an evaluation, provided that the system is based upon a
model designed to support such views. One such model is presented in detail in Chapter 7.

Before useful views of an evaluation can be provided to the user it is also necessary to overcome a number of problems associated with the rewriting approach; these problems are discussed in detail in the following chapter. Tools must be provided with which to manage the potentially large volumes of information which this approach can generate, and with which to allow the user to navigate through the potentially large number of reduction steps produced by such an approach. Chapters 5 and 6 describe such tools.
3 Existing Techniques and Systems

In this chapter I review existing approaches to providing information about the evaluation of expressions written in a lazy functional language. I will describe facilities which are available to users of existing lazy functional programming systems, together with systems designed specifically to provide information about the evaluation of expressions.

3.1 Print Statements

The use of print statements is one of the most common techniques adopted by programmers using an imperative language, when faced with a program whose behaviour they do not understand. By using print statements programmers can achieve a number of goals: they can confirm that a piece of code is being executed; they can trace the state of a number of variables; they can trace when a procedure is being called and with what arguments, and so on.

One of the properties of purely functional languages is their lack of side effects. One of the consequences of this is the impossibility of providing the user with a side-effecting print facility. In (Hughes '89) this kind of property is said to give the user the air of a “medieval monk, denying himself the pleasures of life in the hope that it will make him virtuous. To those involved in material gains, these ‘advantages’ are not convincing”. To a programmer faced with a set of function definitions containing one or more errors, such ‘advantages’ seem positively disadvantageous. The lack of printing facilities leaves functional programmers without the most primitive facilities available to their imperative counterparts.

There is an important distinction to be drawn between the provision of a printing statement in an eagerly evaluated functional programming language, and the provision of one for a lazily evaluated language. One of the most popular eager functional languages is SML, the core language of which is described in (Milner '84). In the
section describing Standard functions and constants, Milner describes a print function for the language in the following way:

Two multi-typed functions are included as quick debugging aids. The function print is an identity function, which as a side-effect prints its argument exactly as it would be printed at top-level. The printing caused by "print(exp)" will depend upon the type ascribed to this particular occurrence of exp.....

(Milner '84)

So, it is possible to provide a print function for an eager functional language, as long as one is prepared to accept that it is side-effecting. However, the situation for users of a lazy functional language is quite different. Rather than freeing these users from 'medieval' misery, supplying them with a print function would only serve to transport them into the Dark Ages.

As I commented in Chapter 1, one of the consequences of lazy evaluation is that programmers are rarely able to predict the order in which expressions are being evaluated; this means that the output generated by a print function would rarely appear in the expected order, making it difficult for the programmer to decide whether the output was correct or was due to an error in the function definitions. This is borne out by Augustsson and Johnsson who, frustrated by the lack of a print function during their work on the language LML, took the opportunity of introducing one into their language implementation. They called the function trace. The result was described by them as:

"generally an indecipherable mish-mash of output from different instances of trace"

(Augustsson & Johnsson '89).

When introducing a print function into a lazy language, other factors also have to be considered. It will frequently be the case that the arguments to a print function will not have been fully evaluated when the time comes to print them. What should the solution to this be? Either the implementor of the print function must provide a means of printing a partially evaluated expression, or the expression must be evaluated until it reaches a printable form. The danger of the second approach is that the extra evaluation may change the termination properties of the evaluation.

To illustrate this problem we must imagine that Miranda has been extended by adding a print function. Rather than adopting the scheme used in ML (i.e. the print function is
Chapter 3 - Existing Techniques and Systems

the identity function with the side effect of printing its argument on the user's terminal), I will use Augustsson & Johnsson's scheme, whereby the print function takes a pair of arguments, the first of which is printed and the second is returned. Figure 3.1 shows an imaginary Miranda session which involves the evaluation of this new function, which I shall call print. The evaluation results in the first argument being printed (the first line of Miranda output), and the argument being returned, (the second line of Miranda output).

<table>
<thead>
<tr>
<th>Miranda print</th>
<th>&quot;A message&quot;,1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>message</td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Miranda</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.1 Session of a Miranda system enriched with a side-effecting print function.

If we now use print to provide more information about the hd function defined in Figure 3.2(a), we get the function defined in 3.2(b). Now, because the printing mechanism will evaluate the whole list in order to print it, the application of hd to the list [1,⊥] will be undefined, rather than evaluating to 1. This type of problem will repeat itself throughout this chapter, as various attempts to provide the user with information, change the evaluation order, and so risk changing an evaluation that would terminate with a well defined value into one that does not.

(a)

\[
hd \ (x:xs) = x
\]

(b)

\[
hd \ (x:xs) = \text{print} \ ((x:xs),x)
\]

Figure 3.2 (a) definition of the hd function; (b) the function incorporating a print function

To summarise, lazy functional languages and print functions do not sit well alongside each other. Printing with lazy languages is far worse than with eager functional languages where the only problem with printing is that the purely functional nature of the language is violated. Print statements in lazy languages may change the termination properties of a definition, and even if they do not, the output of these statements is likely to be incomprehensible.

3.2 Shadow Parameters

One solution to the printing problem described above is to add a trace parameter to function definitions - a technique described in (O'Donnell & Hall '87). To use this
technique, users add an extra parameter to the definitions that they require information about. The extra parameter is called a “shadow parameter” and is used to return a string which contains trace information. So, to trace a function \( g \) of type \( \alpha \rightarrow \beta \), a list of characters is added to the result type, changing the type of \( g \) to \( \alpha \rightarrow (\beta, \{\text{char}\}) \), where the \{char\} is the type of the trace information. An example of this transformation is given in Figure 3.3 where the definition in 3.3(a) has a shadow parameter added to give the definition in 3.3(b). A Miranda session showing the result of an application of this transformed definition is shown in 3.3(c).

(a)
```haskell
negate \( n = n \times (-1) \)
```

(b)
```haskell
negate \( n = (\text{result, trace}) \)
where
\( \text{result} = n \times (-1) \)
\( \text{trace} = \text{\textasciitilde}\text{\textasciitilde}_{\text{negate}} \rightarrow \text{\textasciitilde} \) \( + \) (shownum \( n \))
```

(c)
```
Miranda negate \( 9 \)
\((-9, \text{\textasciitilde}_{\text{negate}} \rightarrow 9\))
```

Figure 3.3 (a) definition of the function \text{\textasciitilde}_{\text{negate}}; (b) new definition with a shadow parameter; (c) example evaluation with shadow parameter.

If a function is recursive then both its argument and result type must be changed so that the function can take the current trace and append its own trace information to it. This changes a function of type \( \alpha \rightarrow \beta \), to one of type \( \alpha \rightarrow [\text{char}] \rightarrow (\beta, [\text{char}]) \). An example of this transformation is given in Figure 3.4 where the function \text{\textasciitilde}_{\text{last}}, defined in Figure 3.4(a), is shown with a shadow parameter in Figure 3.4(b). Notice that the definition in Figure 3.4(b) has an extra case which has been used to add debugging information to the trace. An example application of this new function is shown in Figure 3.4(c).
Chapter 3 - Existing Techniques and Systems

(a)
last [x] = x
last (x:xs) = last xs

(b)
last [] trace = ([], "last called with empty list!!")
last [x] trace = (x, new_trace)
   where
       new_trace
           = trace ++ "last called with -> (show [x])
                         + returns ->" ++ (show x) ++ "\n"
last (x:xs) trace = last x new_trace
   where
       new_trace
           = trace ++ "last called with"
                         ++ (show (x:xs)) ++ "\n"

(c)
Miranda last [1,2,3] "\n"
(3,"
last called with [1,2,3]
last called with [2,3]
last called with [3] returns 3
"

Figure 3.4 (a) definition of the function last; (b) new definition with a shadow parameter; (c) example evaluation with shadow parameter.

Finally, if the function which is to be traced occurs in the definitions of other functions, then these functions too must be transformed to accommodate the new argument and result of the traced function. An example of this is given in Figure 3.5 where the type of the function negate_last is changed from \[\textit{num} \to \textit{num}\] to \[\textit{num} \to (\textit{num}, \textit{char})\] even though it provides no tracing information.

It is easy to see that the tracing of a large program containing many definitions would require a large number of changes to be made to the code. This large set of changes makes the shadow parameter technique far less convenient than the insertion of a single print function. It is also likely that a large number of manual changes to function definitions will introduce mistakes. The solution to these problems is to provide a program transformation system which, given a set of definitions together with information about which definitions to trace and what trace information to provide, could produce an equivalent set of definitions with the necessary shadow parameter transformations carried out.
Chapter 3 - Existing Techniques and Systems

(a)
```
negate_last = negate_last
```

(b)
```
negate_last l = (negated,trace)
  where
      (negated,neg_trace) = negate last_el
      (last_el,last_trace) = last l "\n"
      trace = last_trace ++ neg_trace
```

(c)
```
Miranda negate_last [1,2,3]
(-3,"
last called with [1,2,3]
last called with [2,3]
last called with [3] returns 3
negate -> 3"
```

Figure 3.5 (a) definition of the function negate_last; (b) new definition with a shadow parameter; (c) example evaluation with shadow parameter.

The tracing information in all of these examples is explicitly ordered in the definitions which are being traced, and so appears in the expected order regardless of the actual order in which the expressions are evaluated. Thus the problem of 'mish-mash' output, described above, has been overcome. However, the problem of forced evaluation of an expression still remains. Figure 3.6(a) gives the definition of first which returns the first element of a pair. Figure 3.6(b) gives the definition modified to incorporate a shadow parameter.

(a)
```
first (a,b) = a
```

(b)
```
first (a,b) = (a , (trace ++ "first called with 
  ++ (show (a,b))
  ++ " returns " ++ (show a)))
```

Figure 3.6 (a) definition of the function first; (b) new definition with a shadow parameter

Because show fully evaluates its argument in order to 'print' it, the result of examining the trace of the evaluation of first (1,1) is undefined.
3.3 Problems of Using the error Function

The Miranda system provides an error function to users. This function is defined in the Miranda manual as follows:

'error' applied to a string creates an error value with the associated message. Error values are all equivalent to the undefined value - any attempt to access the value causes the program to terminate and print the string as a diagnostic.

>error :: [char] -> *    || defined internally

So, the error function applied to a string can be placed on the right hand side of any definition; any attempt to evaluate this right hand side will result in an error value being returned and the string being printed as a diagnostic. The function can be used in a number of ways to obtain information about an evaluation which is behaving erroneously1.

An example of the use of the error function is shown in Figure 3.7. Here the evaluation will result in an error value, if the function hd is applied to the empty list.

```
hd [] = error "hd of empty list"
hd (x:xs) = x
```

Figure 3.7 Definition of function hd using Miranda’s error function

There are a number of problems associated with the use of the error function. These problems are described in the following sections.

3.3.1 The error Function Terminates an Evaluation

One of the main drawbacks of the error function is that in order to respect the semantics of Miranda, the error function must return an error value and that value will be the result of the current evaluation. This means that an error value is returned and the evaluation is terminated. There is no way to continue with an evaluation if the error function is being used and the application of error that has caused the termination is not the one of interest to the user.

1 During the development of Prospero the technique of using the error function was essentially the only one used.
For example, given the definition of \( \varepsilon \) defined in Figure 3.8, suppose the user wishes to spot all applications of \( \varepsilon \) to the empty list which are not the result of a recursive call of \( \varepsilon \). Put another way, \( \varepsilon \) can only be applied to the empty list by itself. There is no simple way, using only the \texttt{error} function, to check for erroneous applications of \( \varepsilon \). Instead the definition of \( \varepsilon \) has to be changed to be the one given in Figure 3.9, and all calls of \( \varepsilon \) have to be changed so that \( \varepsilon \) is applied to an extra parameter which is originally \texttt{True} and is subsequently set to \texttt{False} by the recursive case.\(^1\)

\[
\begin{align*}
\varepsilon \ [\ ] & = \texttt{error "\varepsilon applied to [\][]"} \\
\varepsilon \ [\ ] \texttt{False} & = [\ ] \\
\varepsilon \ (x:xs) \texttt{flag} & = \varepsilon \ xs \texttt{False}
\end{align*}
\]

Figure 3.9 Definition of function \( \varepsilon \) showing adaptation necessary to catch non-recursive applications of \( \varepsilon \) to the empty list.

### 3.3.2 Introduction of Strictness

The \texttt{error} function can introduce strictness into an evaluation in two ways. First, if the \texttt{error} function is introduced as an extra clause in a function definition, then it must be separated from other clauses of the function using a pattern or a guard. Either of these approaches may introduce strictness. The result of this added strictness is that an evaluation that previously terminated with the ‘wrong’ value, may now fail to terminate. For example the definition given in Figure 3.10(b) is the definition given in Figure 3.10(a) but with the \texttt{error} function added. The addition of patterns to deal with the \texttt{error} function has made \( g \) strict in its second argument.

---

\(^1\) This example is a simplification of a problem encountered during the implementation of Prospero using Miranda.
(a)
\[ g \ n \ m = f \ n \]

(b)
\[ g \ n \ 0 = \text{error} \ "\text{error in } g" \]
\[ g \ n \ m = f \ n \]

Figure 3.10 Introduction of strictness into definition of function \( g \) when \text{error} is used

The second source of strictness is brought about when a programmer uses the \text{error} function in combination with the \text{show} function. In this way, the diagnostic associated with the \text{error} function can be used to print information based upon the value of the definition containing it. This use of \text{show} may introduce strictness when its argument is evaluated so that it can be printed. For example, the user may change the definition given in Figure 3.11(a) to be that given in 3.11(b). The use of \text{show} has made the definition of \( f \) strict in both of its arguments and has once again introduced the potential for non-termination.

(a)
\[ f \ 1 \ m = [] \]

(b)
\[ f \ 1 \ m = [] \]
\[ f \ n \ m = \text{error} \ ("f \ \text{applied to } -\to -" ++ \ \text{show } n ++ \ "and } \]
\[ ++ \ \text{show } m) \]

Figure 3.11 Introduction of strictness into definition of function \( f \) when \text{error} is used to display arguments of the function

3.3.3 Finding the Source of Errors

Anyone who has used Miranda and has seen the error message "attempted \text{hd} of []" will know that what is required is not only the information that this message contains, but also information about which of the many possible applications of \text{hd} has caused the error. There may be many applications of \text{hd} in a program and the user is offered no information about which application the message is referring to.

3.3.4 Post-Conditional Use of \text{error}

By placing applications of the \text{error} function into a set of definitions, and then evaluating an expression which involves these definitions, it is possible to make assertions about an expression. Figure 3.12 shows the way in which a post-
conditional assertion (i.e. an assertion about the result of the evaluation of an expression) can be simulated using the `error` function.

Each of the sub-figures of Figure 3.12 are split into two sections. The section above the dashed line contains the expression which is to be evaluated. The definitions below the line are the definitions in the environment in which this expression is to be evaluated.

In Figure 3.12(b) a post-conditional assertion of the form "`(f 2)` does not evaluate to `[42]" has been inserted into the definitions given in Figure 3.12(a).

The introduction of the guards necessary to carry out such a test changes the strictness properties of the expression.

In Figure 3.12(a) the expression `(f 2)` need only be evaluated to a point where it is in constructor form, whereas in Figure 3.12(b) the expression `(f 2)` will be fully
evaluated. This introduction of extra strictness is directly related to the guard that is added to the definition of \( e \) in Figure 3.12(b). Had a different assertion been required, leading for example to the definition shown in Figure 3.12(c), then no extra strictness would have been introduced; in this case the evaluation necessary in order to evaluate the guard is the same evaluation necessary for the pattern matching of \( \text{len} \) to take place, and so no strictness has been introduced compared to the definition in Figure 3.12(a).

An evaluation of the expression defined in Figure 3.12(b) would result in an error value, indicating to the user that their assertion had not held. However, this error value is a direct consequence of the extra strictness introduced by the guards. If the additional strictness had not been introduced, then the evaluation of \( (\mathbb{N} \ 2) \) would never have taken place.

This means that by using \texttt{error} to make an assertion, a user may change the strictness properties of the expression and thereby change the truth of the assertion.

In general, the \texttt{error} function can be seen as an unsatisfactory facility with which to produce information about errors in a set of definitions. While this section has been specific to the problems associated with the \texttt{error} function, many of these problems carry through to other approaches. In particular the problem of introducing strictness into an evaluation, as a result of attempting to provide information, is a serious one, and must be avoided.

In the following sections I will present a number of existing systems, which go beyond the primitive information provided by languages such as Miranda. All of the systems described here are based upon the approaches to providing information, presented in Chapter 2.

3.4 Information Through Snapshots

A snapshot is a finite source-level textual representation of an interrupted functional computation. A snapshot tool is most useful for computations that fail by computing undefined values...

(Toyn '87)

Two systems which provide information about an evaluation based upon snapshots are Fly and Glide. These systems are described in (Toyn '87).
3.4.1 Fly

Fly is an eager functional programming system which includes a number of tools to provide information about an evaluation. Fly is implemented in terms of an SECD machine (Landin '66). This implementation is augmented with Mycroft's error values (Mycroft '81) which are used to provide information about non-terminating evaluations, evaluations which return an undefined result, and evaluations which return an unexpected result.

All of the information provided by Fly is in terms of source level text. Fly was not designed as a debugging system, rather it is a development system which incorporates certain debugging features. Information about an evaluation is not provided to the user as a matter of course, instead users must state their interest in part of an evaluation, and information about it is then provided.

Users are offered a number of commands which provide information about the history and current state of an evaluation. For example the Trace command applied to an interrupted evaluation (i.e. the user has stopped the evaluation using Control-C) will give a syntax level description of the evaluation to date including variable bindings. An example of the use of Trace (taken from Runciman '87) is given in Figure 3.13.

```
fly> Define from(n) -> n : from(n+1) [from]
fly> from(1).
^C interrupt
fly> Trace.
from(n->1) ->
   1 : from(n->2) ->
      2 : from(n->3) ->
         3 : from(n->4)
fly>
```

Figure 3.13 Example of the use of Trace in the Fly system.

By using Mycroft's error values, Fly allows users to be shown traces of evaluations which include undefined results. For example, should a user attempt to evaluate an expression which included the application of head to an empty list, then a trace of the evaluation would substitute "head []" in the trace in place of the undefined expression resulting from this application. An extension of this facility is also offered to the user in the form of the Force command which allows the user to specify the value of an undefined result. This is illustrated in Figure 3.14 (also taken from Runciman '87).
Chapter 3 - Existing Techniques and Systems

```haskell
fly> head('h') : "it".
undefined result:
head('h')
fly> Trace.
[head('h'),'i','t']
fly> Force('h').
hit
fly>
```

Figure 3.14 Example of the use of Force in the Fly system.

Information about evaluations which terminate with an unexpected, but defined result, is provided by Fly using the Suspect command which results in the reporting of the name, arguments and result of applications involving suspected functions. (A command Halfsuspect also exists which reports only the name and arguments of a function, allowing reporting to take place before an application has been evaluated).

3.4.2 Glide

Glide is a lazy functional programming system which includes a number of tools to provide information about evaluations. It is implemented using a combinator reduction machine (Turner '79). This implementation is adapted to provide information about the state and history of an evaluation in the same way as Fly. As far as the provision of information about an evaluation is concerned, Glide is essentially the lazy equivalent of Fly¹.

3.5 Efficiency Information Through Profilers

Programmers using conventional languages such as C and Pascal are likely to have profiling tools which provide information about time and storage usage in terms of their source code.

Providing such systems for functional languages raises an interesting problem for the system's implementors. The difficulty is the provision of meaningful information about the efficiency of a user's definitions, when the run-time representation of these definitions is in a form far removed from the user's text. Recently a number of

¹ Glide offers a number of extensions in other areas such as an incremental type checker and a library manager (see Toyn '87 for more detail).
Chapter 3 - Existing Techniques and Systems

systems have been developed (Sansom & Peyton Jones '92), (Clack, Clayman & Parrot '95), (Runciman & Wakeling '92) which overcome this problem and provide users of lazy functional languages with profiling information about the evaluation of their definitions.

These three systems can be divided into invasive and non-invasive systems, and can also be divided in terms of whether they provide information about space usage and/or information about time taken by an evaluation. The table below summarises the differences between the systems.

<table>
<thead>
<tr>
<th></th>
<th>Space</th>
<th>Time</th>
<th>Space and Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invasive</td>
<td>-</td>
<td>-</td>
<td>SPJ</td>
</tr>
<tr>
<td>Non-Invasive</td>
<td>RW</td>
<td>-</td>
<td>CCP</td>
</tr>
</tbody>
</table>

Key: SPJ - (Sansom & Peyton Jones '92)
     CCP - (Clack, Clayman & Parrot '95)
     RW - (Runciman & Wakeling '92)

The system described in (Sansom & Peyton Jones '92) is invasive; it requires users to mark 'cost centres' in their code about which they require profiling information. Cost centres are marked using a cost centre function which is semantically identical to the identity function \( \text{id} \) but which operationally attributes the cost of evaluating its argument to a cost centre which is given a label by the user.

The system described in (Clayman, Parrot & Clack '91) is non-invasive. It shows the user a graph of the space used by each definition involved in the evaluation of an expression. The user is also given a textual breakdown of the time taken up by each definition, and the time spent garbage collecting.

Finally, the system described in (Runciman & Wakeling '92) is non-invasive and gives a graphical breakdown of store usage against time. The information provided by this system is more advanced than that provided by the other two: the user may select a number of ways of viewing the space usage; it may be viewed in terms of the definition which produced the store locations; or it may be viewed in terms of the constructors which occupy them. As an example, if a single definition which produced a large list and then terminated was profiled, the 'producer' view would show heap used by that definition increasing as the expression was evaluated, and the 'constructor' view would show an increasing number of cons cells present in the
heap. Definitions which use less than 1 percent of the heap are grouped together under the label ‘others’ in any display, and users may also selectively omit definitions from the display.

This system provides the user with information about the time taken to carry out an evaluation, but this information is simply the total time taken, and is not broken down in any way.

3.6 Algorithmic Debuggers

Algorithmic debugging was introduced and described in Chapter 2. A development of this technique, which allows users to answer yes/no/maybe to questions about their evaluation, and allows the use of primitive assertions, has been implemented for the lazy functional language Freja\(^1\) by Nilsson & Fritzon (’92). Algorithmic debugging of lazy functional programming is also discussed in (Hall, Hammond & O’Donnell ’90).

3.7 Dataflow

In (Sinclair ’91) a debugging system based on dataflow is described. The underlying model of this system is that of functions as producers and consumers of data. The flow of data between occurrences of functions can be used to provide information about the evaluation of an expression. This dataflow would be shown graphically as a series of nodes (representing the functions) and arcs (representing the dataflows). Rather than showing the complete dataflow graph for an evaluation, the proposed system would allow users to focus on particular parts of the dataflow graph and to expand and move this focus as necessary.

3.8 Information Through Rewriting

The systems described in the following sections provide information to the user which is based upon the step-by-step rewriting of expressions from their initial to a canonical form. Before describing these systems, I will outline some of the potential problems of rewriting systems, and suggest ways in which a system could deal with them.

---

\(^1\) The language Freja is described in (Nilsson & Fritzon ’92) as "essentially a subset of Miranda ... implemented using a G-machine approach". A description of the G-machine can be found in (Peyton Jones ’87)".
3.8.1 Potentially High Levels of Information

One potential problem with rewriting systems is the volume of information produced by a trace of an evaluation involving a large number of definitions and complex arguments. This problem would be amplified by the inclusion of information about pattern matching and the checking of guards. A system based on rewriting would have to provide the user with facilities with which to reduce the amount of detail being presented.

3.8.2 Too Many Steps

Rewriting systems produce a potentially large number of evaluation stages for anything but the simplest of evaluations. It would be unsatisfactory for the user to be shown the entire evaluation when that evaluation involved a large number of steps. Instead a system should allow the user to control the granularity of the trace.

The control of granularity could be carried out in a number of ways:

- the user could remove detail specific to the evaluation of a particular sub-expression;
- the user could change the step size to be a number of rewrite steps;
- the user could set a breakpoint and then allow the evaluation to continue until the condition associated with the breakpoint was satisfied.

Figure 3.15 illustrates all three of these techniques using the evaluation of an expression involving the functions defined in Figure 3.15(a).
Figure 3.15 (a) the definitions of functions used to demonstrate techniques which can be used to remove some steps from the trace of an evaluation; (b), (c) & (d) a trace with various such techniques applied.\(^1\)

\(^1\)The functions map and filter are defined in the Miranda standard environment as follows:

\[
\begin{align*}
\text{filter } p \text{ [ ] } &= [ ] \\
\text{filter } f \text{ (x:xs) } &= x : \text{ filter } p \text{ xs . if } f \ x \\
&= \text{ filter } f \text{ xs, if otherwise} \\
\text{map } f \text{ [ ] } &= [ ] \\
\text{map } f \text{ (x:xs) } &= f \ x : \text{ map } f \text{ xs}
\end{align*}
\]
The evaluation of the expression (double_odds [1,2,3]) is shown in Figure 3.15(b). This figure involves the removal of detail related to the evaluation of a number of sub-expressions. For example, the trace shows no detail of the evaluation of filter, apart from the application of filter to its arguments (the second stage of the trace), and the steps where a list element is odd and so is added to the result of the application of filter (stages 3 and 7), or when the argument list is empty (stage 11). An evaluation of filter which did not involve the removal of detail about the filter function is shown in Figure 3.16.

The evaluation in Figure 3.15(b) is shown with a step size of 3 in Figure 3.15(c), giving stages 1, 4, 7, 10 and the final stage in the evaluation.

Finally Figure 3.15(d) shows the first and last steps of the evaluation together with the steps resulting from a breakpoint which stops evaluation when the next reduction step involves the rewriting of an application of double.
Chapter 3 - Existing Techniques and Systems

All of these techniques could be supplied by a system providing information about the rewriting of expressions. The simplest technique, that of carrying out multiple steps, presents the user with a quick and easy way of jumping to a later stage in an evaluation. The technique of removing information about a particular sub-expression allows the user to remove information concerning definitions which they believe to be error free, for example functions provided by the system itself, or functions which they have used before without encountering problems. The final technique involving the use of breakpoints allows the user to rapidly locate the evaluation steps involving a definition which they strongly suspect as being the source of an error, or which they believe will lead them to the source of an error. A detailed presentation of techniques such as these is presented in Chapter 5 and Chapter 6.

3.8.3 Presenting Rewriting Information

When presenting rewriting information, there are a number of approaches which can be used to present information; these include purely textual information, the use of graphics to augment textual information, and the use of labelled graphical information.

Looking first at the use of textual information; this information has the clear advantage that the user is presented with information which is in the format used to enter definitions and to interact with the functional language system. In (Patel, du Boulay & Taylor '91) it is also observed that the provision of textual rather than graphical information by a debugger seems to improve the system’s ability to provide sequential information (such as an execution trace).

```latex
\begin{verbatim}
double (double (3*4))
    \text{where}
    double x = x + x
\end{verbatim}
```

Figure 3.17 The definition of the function `double`.

The use of graphics to augment textual presentation of an evaluation can offer advantages when it is important to show the sharing present in the evaluation. For example Figure 3.18 shows a textual representation of the evaluation of the expression defined in Figure 3.17.
The use of labelled graphics to present rewriting information about the evaluation of expressions is an approach which is used by Prospero and by other debugging systems such as TPM (Eisenstadt & Brayshaw '87). This approach has a number of advantages and disadvantages when compared to a textual approach.

It is suggested in (Eisenstadt & Brayshaw '87) that the use of graphical overviews of evaluation spaces (which may be too large to be displayed in detail using either a graphical or textual approach) allows users to spot errors by recognising patterns in the displayed information. Without proper experimental evidence\(^1\) it is difficult to know whether this is a true claim, although it is my own experience that after working with graphical trace information for some time one does become aware of patterns in the displayed information and become able to recognise errors in evaluation before the details of the information displayed has been studied. My own experience therefore supports Eisenstadt & Brayshaw's claim.

---

\(^1\) To my knowledge, no such evidence exists.
Chapter 3 - Existing Techniques and Systems

The possibility of keeping the display location of sub-expressions as static as possible throughout the evaluation, may prove to help the user to follow the evaluation more easily and to better understand the evaluation order and the state of the evaluation of sub-expressions. Again without proper experimental evidence it is difficult to know whether this is true or not.

3.8.4 The Miranda Calculator

The Miranda Calculator has been specifically written to provide information about the evaluation of Miranda definitions. It is an interpreter of a sub-set of Miranda which can be used to display a single-step evaluation history based upon textual rewriting. Part of such an evaluation is shown in Figure 3.19.

The Calculator allows the user to step forwards and backwards (over a limited number of reduction steps) through an evaluation. A user may also skip the evaluation of a sub-expressions once that sub-expression has become the redex of the evaluation.

The Miranda Calculator was aimed at supporting the teaching of lazy functional programming using (Bird & Wadler ’88). Because of this the evaluation was presented to the user in a way which was as close to the presentation used in the text book.

The Calculator also provides a facility where the user can ask to be shown the type of any sub-expression. In reverse, the user may also attempt to state the type of an expression and the system will inform the user whether they have correctly identified the type.

---

1 The Miranda Calculator is a part of the Calculator Project; a collaborative project between the Department of Computer Science, Queen Mary and Westfield College, and the Institute of Educational Technology, Open University. The purpose of the project is to empirically evaluate the effectiveness of program calculators in the teaching of logic and functional programming (Bornat, Fung, Goldson, O’Shea, Reeves ’91)
Looking at the facilities provided by the Miranda Calculator in order to deal with the potentially high levels of information, and large number of evaluation steps, very little support is offered. At any point in the evaluation, the user may choose to skip all of the evaluation steps of the sub-expression which is due to be evaluated next, but other
than this, the system offers no facilities with which to control the evaluation steps which should be presented, or the way in which these steps should appear.

3.8.5 Hint

Hint (Foubister '93) is a program visualisation system based on the language $h$ a subset of the Haskell lazy functional language. In order to simplify the implementation of Hint the user is required to carry out translations on their definitions before they are used, these include the translation of pattern matching to case statements, and manual lambda lifting.

Hint evaluates $h$ expressions using a simple graph reducer, and it is the reduction graphs produced during the reduction process which are displayed to the user. Rather than display the reduction graphs as graphs, Hint displays a spanning tree of the graph, with node labelling used to indicate the sharing in the original graph. Hint provides an overview of the reduction graph together with a close-up view of parts of the graph.

Hint also provides a 'clustering' facility which allows sets of nodes to be grouped together to form a single node, thus reducing the number of nodes displayed to the user. A facility with which to search for points in a reduction with specific properties is proposed.

The aim of Hint is not to provide a visualisation system which can be used to display arbitrary views of an evaluation, rather Foubister proposes users:

> will make use of the information to write “better “ programs. i.e. more efficient in terms of space and time usage...

In other words, Hint is a profiling system which provides information based on graph rewriting. Alternative approaches to the provision of profiling information have been described in Section 3.5.

3.9 Strengths and Weaknesses

In this chapter I have presented a number of techniques and systems aimed at providing users of lazy functional programming systems with information about their definitions and about evaluations involving those definitions.

Here I will summarise these systems and techniques, highlighting their strengths and weaknesses.
3.9.1 Print Statements

While print statements are a common technique adopted by programmers faced with an error, they are unsuitable to users of lazy functional languages for a number of reasons:

- print statements are side-effecting and so can not be provided by a 'pure' functional language;

- the insertion of side-effecting print statements into a set of function definitions will produce output which can be difficult to follow;

- the function which provides a printing facility will be strict. This strictness may result in expressions being evaluated in order to print their value. This may lead the user to believe that some expressions are being evaluated when this would not have been the case were it not for the print statements. Additionally, this may lead to an evaluation failing to terminate when the evaluation would normally terminate without print statements.

It is important that the tendency of print statements to produce output which is difficult to follow, should not lead one to conclude that providing information about the lazy evaluation of expressions is a lost cause, or that it is impossible to understand the order in which lazy evaluation takes place. Print statements produce a partial picture of an evaluation. They will only produce information about selected steps in an evaluation, and these steps will be provided without any context. It is this lack of context which makes the evaluation difficult to follow. An equivalent situation would be to try to follow a novel by reading one page out of every 20, or to read only the pages which concerned a particular character.

3.9.2 Shadow Parameters

Shadow parameters go part of the way towards solving the problems of print statements. If we suppose that a program transformation system is produced which inserts and removes shadow parameters, then the problem of having to rewrite a large number of definitions is overcome. The order in which information is produced by the system is explicitly defined, thus removing the possibility of information being produced in a confusing order. However, shadow parameters can introduce strictness and so still fall foul of the possibility of changing the order of an evaluation.

Interestingly, the linear, textual nature of the output of both print statements and shadow parameters allow for the simple provision of facilities which allow the user to
search the text for points at which specific ‘events’ occurred. The granularity of the output can also be changed easily by the user, as they have direct control over the information produced by the print statements. However, the production of low-level information related to the evaluation mechanism can not be provided by shadow parameters or print statements, as the user has no access to the low-level evaluation mechanism.

3.9.3 The error Function

Whilst the use of the error function is the only facility available to Miranda users today, it has been shown to be the worst of all of the techniques described here. The error function has all of the strictness problems of print statements without even providing the user with any kind of tracing information such as that provided by print statements and shadow parameters. Instead the error function, terminates an evaluation, and provides a single message with no contextual information.

3.9.4 Snapshots

The two systems based on Snapshots offer a useful set of facilities with which to examine evaluations. Users are provided with the execution trace of an interrupted evaluation, and may request information about ‘suspected’ functions as an evaluation progresses. These systems also provide the user with views of partially evaluated expressions, thus overcoming the problems of strictness being introduced by a printing mechanism.

Snapshots provide information at a level equal to that of the user’s definitions, this rules out the provision of lower level information, such as that about the pattern-matching mechanism.

3.9.5 Profiling

The systems described in this chapter illustrate the points made in the summary of Chapter 2. While profiling systems provide superior information about the space/time efficiency of an evaluation, they fail to go beyond this.

3.9.6 Algorithmic Debuggers

The system described here illustrates the points made in the previous chapter. Algorithmic debugging has a limited scope; its only use is to track down errors which fall into the ‘semantic’ class described in (Hall, Hammond & O'Donnell '90), that is, algorithmic debugging can only be used to deal with errors which manifest themselves
in the form of an unexpected value, an undefined value or a non-terminating evaluation. Algorithmic debuggers provide declarative information in the form of argument-result pairs and nothing more. This means that they cannot be used to investigate operational bugs, such as efficiency problems.

Algorithmic debuggers have the advantage of not requiring the user to understand lazy evaluation or any other operational aspects of the system they are using. Users simply have to provide answers to questions about the results of applying the functions that they have defined, and they will be led to the error, as long as they answer the questions correctly. The price of this simplicity is of course the system’s lack of scope.

3.9.7 Dataflow

The dataflow approach to providing information about an evaluation is centred on the belief that a radically different approach has to be taken in order to overcome the ‘difficulties’ of following a lazy evaluation. As I stated earlier, I believe that these difficulties occur only when the user is asked to predict the evaluation order, and not when a user is being shown the evaluation.

In (Sinclair '91) the author points out a number of shortcomings of the dataflow model, particularly the representation of higher-order values and the display of partially evaluated values.

Interestingly, Sinclair also states that in some cases “simple dataflow is not enough, and some sort of visualisation of execution order is required”. The example given by Sinclair is that of an error related to the strictness of a definition. Sinclair then proposes that watching the development of the dataflow graph will imply the execution order. As no dataflow systems are currently available it is difficult to know whether the user really will be able to bridge this gap. Even if this is possible, it raises questions about the usefulness of this approach when the user is faced with errors which require information about the order of evaluation.

3.9.8 Rewriting Systems

Of the two rewriting systems described here, Hint must be viewed as a research platform rather than a real-world system. Hint requires the user to perform transformations on their definitions before information about evaluation involving these definitions can be provided. Curiously, the system produces rewriting information in the form of evaluation graphs, but is limited in scope to the provision of profiling information; again, this may be due to the research nature of the tool, and
the need to limit the scope of the research carried out. The work described in this thesis will show that the information available when using a rewriting system has a far wider scope than the one explored by Hint.

The Miranda Calculator provides step-by-step string rewriting information. This project illustrates that the production of a useful system based on rewriting is possible; at the same time the system has a number of short-falls:

- the level of the information provided is limited to that of the users definitions, i.e. no information about pattern-matching and guard evaluation is provided;

- no tools are provided to control the potentially large volume of information produced by a rewrite system. For example there is no way to elide large data structures, or to avoid showing all parts of a deeply nested evaluation step;

- beyond the ability to ‘skip’ the evaluation of a particular expression, no facilities are provided to allow the user to move quickly to a point in an evaluation, or to specify the evaluation steps in which they are interested in seeing;

- sharing of common sub-expressions is not shown. In fact, the implementation of the Miranda Calculator is not truly lazy, as sharing of common sub-expressions is not carried out by the evaluator.

3.10 Summary

After examining the proposed and existing systems and techniques a number of points are clear.

The facilities provided by the Miranda system are limited to the use of the error function, and manual addition of shadow parameters. Both of these techniques have been shown to be unsatisfactory for a range of reasons.

Profiling systems have been shown to provide superior profiling information, but to be extremely limited in the scope of errors which they can provide information about.

Apart from rewriting systems, the other systems described here have been deliberately raised above the abstract-machine description of an evaluation, and aim to provide a declarative description of an evaluation. Again, this reduces the scope of the information which the system can provide. One of the aims of the research described here is to examine the usefulness of information which is provided at an
abstract-machine level, and to show that the direct presentation of lazy evaluation does not hinder the understanding of the users of a system.

While the rewriting systems described here (i.e. Hint and The Miranda Calculator) go part of the way towards demonstrating the usefulness of the abstract-machine level of description, they do not take full advantage of the information available when rewriting is adopted. The systems also fail to provide facilities which will be shown to enhance the presentation of the lazy evaluation of expressions. The Prospero system is a precursor of both of the systems described here, and as I will show goes far beyond the facilities that they provide.
4 A Tour of Prospero

The aim of this chapter is to provide the reader with a ‘whistle-stop’ tour of Prospero. The chapter will take the form of a number of examples of Prospero in action, where the subjects of the examples and the overall structure of the chapter are drawn from Chapter 2. This provides the opportunity to compare Prospero with the systems presented in previous chapters, and provides an introduction to many of the features of Prospero, which will be explored in greater depth in later chapters.

In Chapter 2, I put forward the idea that rewriting was an ‘all-round’ approach to providing information about lazy evaluation of expressions. That is, of all of the approaches considered in that chapter, rewriting provides information about the greatest number of aspects of an evaluation. This chapter will reinforce this idea by showing that Prospero, an implementation of a system based on rewriting, is capable of satisfying all of the claims put forward in Chapter 2. Additionally this chapter will introduce the facilities provided by Prospero in order to overcome the difficulties faced by any system based upon rewriting. These facilities are described in more depth in future chapters.

Not all of the facilities described and illustrated in this chapter are available in the current version of Prospero. A full description of the facilities currently available can be found in Appendix B. A number of the figures presented here have been produced using a drawing package, rather than by capturing pictures of the running system. The two different kinds of figures can easily be identified as they have noticeably different appearances. Figures such as 4.3(a) are produced using a drawing package. Figures such as 4.18(a) have been produced using screen-capture. Additionally, some of the screen-captured figures have been retouched to improve their clarity.
4.1 Interface Description

I will start this chapter with a description of the current interface to Prospero. A screen-dump of the current interface is shown in Figure 4.1. The interface makes it possible to zoom in and out of any part of a graph by using the zoom buttons (small and large circles) in the bottom left of the window. Additionally the user may zoom into part of a displayed graph using a click-and-drag technique to selecting a bounding box around the area to be magnified. The user is also offered the home view menu option, which sets the zoom factor to be 100% of the original view.

![Prospero window](image)

Figure 4.1 Example Prospero window

The user may pan horizontally and vertically over a graph. This panning is carried out using the scroll bars on the bottom and left sides of the window.
Chapter 4 - A Tour of Prospero

The interface involves no type of direct manipulation as far as the graph content is concerned. The appearance and positioning of graph nodes is not controllable by the user, instead the graph layout is carried out automatically, using a layout algorithm which is outlined later in this chapter.

Using a pop-up menu, and the buttons on the right of the window, the user is offered a number of commands, with which to control an evaluation. These menu options and buttons are described below:

**Evaluate:** This menu option produces a pop-up form which prompts the user for the expression which is to be evaluated next by the system. The current evaluation history is lost and a fresh one, with the expression at its start, is created.

**Home View:** This menu option resets the magnification to 100%, i.e. leaving the whole graph displayed in the window.

**Open File:** This menu option produces a pop-up form which prompts the user for the name of the file which holds a set of Miranda definitions defined by the user. These definitions along with the definitions which form the Miranda standard environment (Turner '89) form the environment in which any expressions are evaluated.

**Goto:** This menu option produces a pop-up form which prompts the user for the position (in terms of a number of evaluation steps) in the evaluation history which the system should jump to.

**Quit:** Exits Prospero.

The buttons described below are those on the right hand side of the window and will be described from top to bottom.

**Step Forwards (>):** This button causes the next step in the evaluation, relative to the present position in the history, to be displayed. The definition of step as it is used here, is that which is described in Chapter 5 (i.e. the smallest number of atomic reduction steps necessary for a change to be seen in the display).

**Step Backwards (<):** This button causes the previous step in the evaluation, relative to the present position in the history, to be displayed.
Multiple Step Forwards (>>): This button produces a pop-up form which prompts the user for a number of steps. The current position in the evaluation history is then moved on by that amount, and the expression at that stage is displayed. Again the definition of step here is that which is described in Chapter 5.

Multiple Step Backwards (<<): This button is as above but moves the current position backwards through the evaluation history.

Search Forwards: This button uses a search condition to search through the evaluation history for the first stage in the history which satisfies the condition. If the condition is met, then that stage of the evaluation history is displayed, otherwise the final stage of the evaluation is displayed and the user is notified that the search failed.

Search Backwards: This button moves backwards through the evaluation history to the last stage where the current search condition was met. If the condition was not met, then the first stage of the evaluation history is displayed.

Fully Reduce: This button causes the entire evaluation history to be produced and the final stage in the history to be displayed.

4.2 Scenarios where Prospero would Help with Errors

In Chapter 2 of this thesis I described approaches which could be used to present information to users about the evaluation of expressions. I will use some of the examples from that chapter to present the facilities that are offered by Prospero. Many of the figures in the following sections are duplications of figures used in Chapter 2.

By reproducing the examples presented in Chapter 2, I aim to demonstrate that a system based on rewriting is capable of producing information in all of the areas described in that chapter. This is the first goal of the chapter. A second goal is to introduce the reader to the way evaluations are represented in Prospero (i.e. Prospero’s evaluation scheme), the way in which they are presented to the user, and the tools offered to the user with which to control the evaluation of expressions.

4.2.1 Wrong Answer

In Chapter 2, I showed that rewriting information can be used to provide information about evaluations which have resulted in the ‘wrong’ answer. The example used there is presented in terms of the rewriting of an expression involving the membership function tree_member, which is defined over polymorphic binary trees. The definition of tree_member is reproduced in Figure 4.2. Figure 4.3 (a)-(f) shows the
presentation of the rewriting of the expression \texttt{tree\_member \ treel \ 1}. A description of each of the evaluation steps represented in this figure is given below.

\[
\begin{array}{l}
tree \ast : = \text{Branch} \ (tree \ast) \ast \ (tree \ast) \mid \text{Leaf} \\
tree\_member \ (\text{Branch} \ l \ v \ r) \ v' \\
\quad = \text{True} , \ v = v' \\
\quad = \text{tree\_member} \ l \ v' , \ \text{if} \ v < v' \\
\quad = \text{tree\_member} \ r \ v' , \ \text{otherwise} \\
\text{tree\_member} \ \text{Leaf} \ v = \text{False} \\
\text{treel} = \text{Branch} \\
\quad \quad (\text{Branch} \ \text{Leaf} \ 1 \ \text{Leaf}) \\
\quad \quad 2 \\
\quad \quad (\text{Branch} \ \text{Leaf} \ 3 \ (\text{Branch} \ \text{Leaf} \ 4 \ \text{Leaf}))
\end{array}
\]

Figure 4.2 Definition of the function \texttt{tree\_member} containing an error (duplication of Figure 2.1(a))

In order to understand the 'picture' presented in Figure 4.3, one must realise that all functions in Miranda are 'curried' (Curry & Feys '58). The implication of this fact is that all functions take a single argument, thus any function which takes $n$ arguments is represented as $n$ applications of a single-argument function. This is illustrated in Figure 4.3(a), where the function \texttt{tree\_member} is applied to \texttt{treel}, and the resulting function is applied to the constant 1. Hence there are two \texttt{apply} nodes in the graph each one representing the application of a function to its argument.

In Figure 4.3(b) \texttt{treel} has been replaced by its definition. Figure 4.3(c) shows the evaluation following the replacement of the application of \texttt{tree\_member} with an application of \texttt{tree\_member} to the right branch of \texttt{treel}. Figure 4.3(d) shows the result of the next replacement of the application of \texttt{tree\_member}, again to the right branch. Figure 4.3(e) shows the final recursive call of \texttt{tree\_member}. Finally, Figure 4.3(f) shows the result of the evaluation.
Chapter 4 - A Tour of Prospero

Figure 4.3(a)

Figure 4.3(b)

Figure 4.3(c)

Figure 4.3(d)
As well as giving a first taste of the appearance of Prospero, the example introduces the idea of filtering. Prospero offers a filtering facility whereby users can control the information which is presented to them at each stage of an evaluation. The figures used throughout this chapter will almost all involve the use of such filters to simplify the contents of each evaluation stage, or to highlight certain aspects of a stage. Figure 4.3 uses filters in order to present evaluation stages in such a way that all information is presented at the same level as that of the users’ definitions, rather than at the default low-level at which the complete evaluation is represented. Examples of evaluations displayed using this low-level information will be given later in this chapter.

Prospero’s filters can be split into two main types. The first type are known as simple filters; these filters can be thought of as functions which take as their argument a representation of an expression and return a new representation as their result. An example of such a filter is illustrated in the presentation of the tree data structures in the Figure 4.3; these data structures are presented in a way close to that of the function definitions in Figure 4.2. The appearance of these data structures is the result of the use of a simple filter.

For example, Figure 4.4 shows an unfiltered presentation of the data structure Branch Leaf 3 (Branch Leaf 4 Leaf). The same data structures can be seen as the argument to tree_member in Figure 4.3(c). Note that value constructors such as Branch are applied to expressions in order to create data structures. Figure 4.4
shows the curried application of `Branch` to three arguments in order to create an expression of type `tree`.

```
Prospero

apply
apply
apply
Branch
Leaf

apply
5
apply
Leaf

apply
4
Branch
Leaf
```

Figure 4.4 Unfiltered view of a binary tree.

The second type of filters are known as `temporal` filters. These filters are similar to `simple` filters in that they change the representation of expressions. The difference is that `temporal` filters change representations over a number of evaluation steps. Simple filters have a ‘lifetime’ of a single reduction step, a `temporal` filter has a ‘lifetime’ which is defined by the user.

`Temporal` filters have 3 components, a mask filter, a start condition and a stop condition. The mask filter determines how the expression being filtered will appear while the filter is active. The start condition determines when the filter becomes active, and the stop condition determines when it should stop being active. For example, the evaluation presented in Figure 4.3 shows no detail of the evaluation of `tree_member` applied to a tree, other than the recursive call of `tree_member`. This is achieved using a temporal filter which:

- becomes active when `tree_member` is applied to an argument;
- uses a mask filter which removes all detail of the application of `tree_member`;
- stops being active when the current expression becomes either another application of `tree_member`, or a boolean.
This section has shown that Prospero is capable of providing information about an evaluation, the result of which was an unexpected value, and has introduced filters. Using filters it is possible to change the way in which an evaluation stage is presented. It is also possible to remove the stages of the evaluation of specific sub-expressions from a presentation. Further details of the use, definition and implementation of filters will be given in Chapter 5.

4.2.2 Controlling Step Granularity - Filters and Searches

When presenting the problems related to rewriting systems in Chapter 2, I described the following three techniques for controlling the granularity of an evaluation:

- the user can remove detail specific to the evaluation of a particular set of expressions;
- the user can change the step size to be an arbitrary number of the rewrite steps;
- the user can set one or more breakpoints and then skip all steps up to a point in the evaluation where a condition associated with a breakpoint is satisfied.

All of these techniques are available to the Prospero user. Figures 4.6, 4.7 and 4.8 illustrate these techniques using the evaluation of an expression which involves the functions defined in Figure 4.5.

```plaintext
double_odds l = map double (filter odd l)
odd n = (n mod 2) == 0
double n = n + n
```

Figure 4.5 Definitions of functions used to illustrate techniques with which to control the granularity of the presentation of an evaluation. (duplication of Figure 3.15(a)).

---

1 The functions map and filter are defined in the Miranda standard environment as follows:

```plaintext
filter p [] = []
filter f (x:xs) = x : filter p xs , if f x
                = filter f xs,if otherwise
map f [] = []
map f (x:xs) = f x : map f xs
```
Figure 4.6 Filtered presentation of evaluation of `double_odds [1, 2, 3]`

The Prospero evaluation of `double_odds [1, 2, 3]` presented at a similar level to that of the evaluation in Chapter 2, is shown in Figure 4.6(a)-(h). This presentation involves the use of one `simple` filter and four `temporal` filters. The `simple` filter is used to change the presentation of a list data structure in much the same way as the presentation of trees was changed in Figure 4.3. The temporal filters are described in the following table.
<table>
<thead>
<tr>
<th></th>
<th>Start Condition</th>
<th>Mask Filter</th>
<th>Stop Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>double_odds</code></td>
<td>application of <code>double_odds</code> to an argument</td>
<td>expression as it appeared when filter became active</td>
<td>expression is an application of <code>map</code> to two arguments</td>
</tr>
<tr>
<td><code>map</code></td>
<td>application of <code>map</code> to two arguments</td>
<td>expression as it appeared when filter became active</td>
<td>expression is a <code>cons</code></td>
</tr>
<tr>
<td><code>filter</code></td>
<td>application of <code>filter</code> to two arguments</td>
<td>expression as it appeared when filter became active</td>
<td>expression is a <code>cons</code></td>
</tr>
<tr>
<td><code>double</code></td>
<td>application of <code>double_odds</code> to an argument</td>
<td>expression as it appeared when filter became active</td>
<td>expression is a <code>number</code></td>
</tr>
</tbody>
</table>

Figure 4.6(a) shows the application of the function `double_odds`. In Figure 4.6(b) `double_odds` has been replaced by its definition: `map double (filter odd l)` where `l` is the argument to `double_odds`.

The second argument to `map` must now be evaluated to constructor form (i.e. the application of a list constructor) to allow pattern-matching to take place. Consequently the evaluation of the application of `filter` is carried out until the constructor form is reached. Note that these evaluation steps have been removed by the `temporal` filter which removes all details of the evaluation of `filter` until the evaluated expression becomes the application of a list constructor (see above table). The resulting expression is shown in Figure 4.6(c).

The application of `map` can now proceed; once again the evaluation steps are removed by the `temporal` filter until the evaluation stage shown in Figure 4.6(d) is reached (i.e. until the expressions resulting from the application of `map` has become the application of a list constructor). The steps showing the subsequent applications of `map` and `filter` are shown in Figures 4.6(e)-(g). With the result of the application shown in Figure 4.6(h).

Figure 4.7 shows the evaluation using the same filters as that of Figure 4.6 but this time the user has used a multiple-step facility to jump three steps at a time through the evaluation. Note that a ‘step’ here is defined to be a step at the level of abstraction.
imposed by the filters. In other words, a step in this context is a change in the displayed expression, rather than a graph-reduction step.

Figure 4.7 Presentation of evaluation of `double_odds [1, 2, 3]` with a step size of three evaluation steps

Finally, Figure 4.8 shows the first and last steps of the evaluation together with the steps found using a breakpoint which stops evaluation when the next evaluation step involves the application of `double` to an argument.
Figure 4.8 Presentation of evaluation of `double_odds [1, 2, 3]` showing the first and last steps of the evaluation together with the steps involving the application of `double` to an argument.

The breakpoint illustrated in Figure 4.8 is specified using Prospero's search system\(^1\) which allows a user to define a search in the form of a condition defined over expressions, and then to search the reduction history for an evaluation stage which satisfies this condition. In this example, the condition is that the next reduction of the

\(^1\)Note that at present this search system is not implemented. A detailed description and design of the system can be found in Chapter 6.
expression will be the application of \textit{double} to any argument (the user is provided with a function which determines which sub-expression will be evaluated next).

It is important to note that the use of searches here has not changed the order in which the evaluation has taken place. A naive implementation of a search system may carry out extra evaluation in order to check the search condition.

Prospero’s search system never presents the user with an evaluation whose order has been changed due to the use of searches. However, as an experiment in the usefulness of different search evaluation regimes, Prospero allows the user to specify how much extra evaluation of expressions is carried out by the search system. The reader is referred to Chapter 6, where the search system is described in detail.

The example in this section has shown that Prospero provides all three of the techniques (filtering, breakpoints/searches and stepping), outlined in Chapter 3, as ways of overcoming the problems of a large number of evaluation stages. The section has also introduced Prospero’s search system, with which it is possible to find stages is an evaluation with specific properties, without the risk of carrying out evaluation in order to check the search condition.

\subsection{4.2.3 Assertions using Searches}

One of the approaches to finding errors, described in Chapter 2, suggests that users should be supplied with an assertion system. Prospero does not support assertions as such, but does supply the search facility outlined above; this facility can be used to simulate assertions.

To simulate an assertion the user uses the negation of the condition associated with the assertion to search the evaluation history. If the search fails and the end of the evaluation is reached, then the assertion must have held. If the search succeeds then the evaluation is halted and the stage in the evaluation is displayed to the user, thus the user is immediately directed to the point in the evaluation at which the assertion failed.

Below is an example of the use of searches to simulate assertions (the example is the same as that used to illustrate assertions in Chapter 2).

Figure 4.9(a) gives a definition of the function \texttt{funny\_reverse} which contains an error. Figure 4.9(c) shows the correct definition of this function. Figure 4.9(b) gives a textual version of an assertion about \texttt{funny\_reverse}. The result of simulating this assertion using searches, and applying this search to the evaluation of the expression \texttt{funny\_reverse 0 3 [1,2,3] []} is shown in Figure 4.10.
Chapter 4 - A Tour of Prospero

Figure 4.10(b) introduces an aspect of Prospero which has not been shown so far in this chapter. This figure shows that sharing has been introduced into the reduction by the definition of funny_reverse. This sharing represents the implicit sharing of the right hand side of the first clause of funny_reverse, where both \texttt{l1} and \texttt{index}, appear more than once in the definition, and so can be shared by the sub-expressions in which they appear. It is worth explaining the way arcs to shared nodes are drawn in this and other figures in this thesis. When a node is shared, the node is shown only once in the graph, and sharing is represented by drawing a number of arcs to the shared node. The arcs are drawn first to the position where the node would have been had it not been shared, and then extended to the actual location of the node. Hence, arcs to shared nodes may appear 'kinked' when displayed. There are two examples of kinked arcs in Figure 4.10. The first is the left arc coming from the + node at the bottom of Figure 4.9 (b). This arc eventually leads to the shared 0 node. The second kinked arc is the right arc coming from the apply node in the centre of Figure 4.10 (b). This arc eventually leads to the shared list node on the right of the figure.

(a)

\begin{verbatim}
funny_reverse index limit l1 l2
    = funny_reverse (index + 1) limit
      (tl l1) ((l1 ! index) : l2), index /= limit
    = l2 , otherwise
\end{verbatim}

(b)

\begin{verbatim}
assert index + limit = (length l1) +
                     (length l2) in funny_reverse
\end{verbatim}

(c)

\begin{verbatim}
funny_reverse index limit l1 l2
    = funny_reverse (index + 1) limit l1 ((l1 ! index) : l2)
      , index /= limit
    = l2 , otherwise
\end{verbatim}

Figure 4.9 (a) definition of \texttt{funny_reverse} containing an error; (b) assertion user to locate error; (c) correct version of \texttt{funny_reverse}. (Duplications of definitions given in Figure 2.3)
4.2.4 Non-Termination

In Chapter 2 I showed that a system based on rewriting could be used to provide information to a user faced with a non-terminating evaluation.

Because Prospero presents information in the form of rewrites, the most basic help that it can provide when a user is faced with a non-terminating evaluation, is to show the rewriting of that evaluation. Because this approach is similar to all the examples of rewriting given above, it is not necessary to give further examples here.

\[
\begin{align*}
\text{\textit{f}}(\text{x:y:xs}) &= \text{\textit{f}} \text{ xs} \\
\text{\textit{f}}[\ ] &= \text{\textit{f}}[1,2] \\
\text{\textit{f}}[\text{x}] &= \text{\text{True}} \quad || \quad \text{base case}
\end{align*}
\]

Figure 4.11 A non-terminating function \textit{f}. (Duplication of definitions given in Figure 2.4(a))

In Chapter 2 I introduced the concept of a ‘distance’ function which can be used to measure the distance of the argument of a function from its base case. Using Prospero’s filters it is possible to achieve the effect of a distance function. Figure 4.12 shows the evaluation of an expression involving the definition of \textit{f} given in Figure 4.11. Figure 4.13 shows the same evaluation using a distance function. The effect is possible because filters are functions which take the representation of an evaluation stage as their input and produce another representation as their output. The filter can transform the contents of the representation in a way which allows the removal of
information or the summary of the original information. In this example the filter produces information about the length of lists.

Figure 4.12(a)  Figure 4.12(b)

Figure 4.12(c)  Figure 4.12(d)

Figure 4.12 Evaluation of an application of \( \ell \) defined in Figure 4.10.
Chapter 4 - A Tour of Prospero

Figure 4.13(a)

Propero

apply

f
list length > 4

Figure 4.13(b)

Propero

apply

f
list length >= 2

Figure 4.13(c)

Propero

apply

f
list length > 0

Figure 4.13(d)

Propero

apply

f
list length > 2
Figure 4.13 Evaluation of an application of \( f \) defined in Figure 4.11, shown using a distance function

**Assertions and non-termination**

In Chapter 2 I showed that assertions could be made about the argument(s) and result of recursive functions in order to track down the cause of a non-terminating evaluation. In this section I have demonstrated that Prospero’s searches may be used to simulate assertions; these searches may also be used to detect errors which involve the failure of an evaluation to terminate. Thus it is possible to use Prospero’s searches when faced with a non-terminating evaluation.

A simple example of this is given below. In this example an assertion is made that all recursive calls to the function \( g \), defined in Figure 4.14, will move the argument of \( g \) closer to its base case, which in this case is the empty list. If we were to use a distance function to relay information to the user about the behaviour of \( g \), it would be based upon the length of the argument to \( g \). The assertion which we wish to make will also be based upon this ‘distance’: we wish to assert that any recursive call to \( g \) will reduce the length of \( g \)’s argument. It is possible to simulate this assertion using a search for a stage in the evaluation where this is not true. In this example the search will succeed (i.e. the assertion will be shown to be false) as soon as \( g \) is applied to any non-empty list.
4.2.5 Application to Arguments Outside a Function’s Domain

In Chapter 2 I showed that rewriting information can be used to give information about why a function has been applied to an argument which is outside that function’s domain. Because the domain of a function is defined using patterns and guards, any rewriting information about a domain must include information about the pattern-matching mechanism and the evaluation of guards. Prospero provides low-level information about an evaluation so that information such as that about pattern-matching and guard evaluation is available to the user.

The challenge is to produce an explanation of guard evaluation and pattern matching which is simple to understand and faithful to the lazy semantics of the language. One solution to this problem is illustrated here and is presented in detail in Chapter 7.

It is important to remember when looking at the following examples, that the presentation of this low-level information is by no means the norm in Prospero. Using such low-level information is unlikely to help the user except in the case where the user needs to know why a specific clause of a function definition was selected, or why pattern-matching/guard evaluation failed. It is envisaged that higher level views created using filters will be selected most of the time by Prospero users.

Presenting Pattern Matching

Pattern matching in Prospero involves showing the user a detailed account of the matching of a function’s patterns against its arguments. The user is shown the checking of each pattern and is shown at exactly which point pattern matching succeeds or fails. This process requires three representations of a pattern: there must be a way of showing that a pattern has not yet been checked against an argument; there must be a way of showing that a pattern has been successfully checked against an argument; and there must be a way of showing that a pattern has been unsuccessfully checked against an argument. This information is represented in a Prospero display using nodes labelled lambda, jolly and grumpy respectively.
Chapter 4 - A Tour of Prospero

Figure 4.16 shows a simple example of pattern matching which involves all three types of pattern matching node in an evaluation. The evaluation is of the expression

\[ \text{isA "B"} \]

where the function \text{isA} is defined in Figure 4.14.

\[
\begin{align*}
\text{isA "A"} &= \text{True} \\
\text{isA c} &= \text{False}
\end{align*}
\]

Figure 4.15 Definition of the function \text{isA}

This figure is the first in this chapter to be generated by Prospero. The figure also introduces a number of new graph nodes. Note that the nodes labelled \text{ap} are equivalent to those labelled \text{apply} in the figures generated using a drawing package shown earlier in this chapter.
Chapter 4 - A Tour of Prospero

Figure 4.16 (g)

Figure 4.16 (f)
Figure 4.16 Evaluation of expression involving function isA
Figure 4.16(a) shows the first stage in the evaluation. Figure 4.16(b) shows the expansion of the definition of \texttt{isa}. This representation shows the two clauses in the definition of \texttt{isa} grouped under the node labelled \texttt{Def. Union}. This node is used to represent a function definition written in terms of a number of clauses distinguished using pattern matching.

The \texttt{Def. Union} node always takes two arguments. In this figure, the left branch of the topmost \texttt{Def. Union} is the first clause of \texttt{isa}; the right branch is a further \texttt{Def. Union} node. The left branch of this node is the second clause of \texttt{isa}, while the right branch is a single \texttt{ERROR} node.

\texttt{ERROR} nodes are used to represent the value returned from an evaluation where pattern matching has failed for all clauses of a function. Note that pattern matching can not fail in this example, as the pattern of the second clause of \texttt{isa} will match any argument.

In summary, a \texttt{Def. Union} node is created for every clause in a function definition, and the final node of the union will be an \texttt{ERROR} node, used to represent pattern matching failure.

Looking at the representation of the first clause of the definition of \texttt{isa}: the node labelled \texttt{lambda} represents an unchecked pattern match. The left hand branch of the node is the constant pattern \texttt{"A"}, and the right hand branch represents the right-hand-side of the first clause of \texttt{isa}, which will be used if pattern matching succeeds. This right-hand-side is represented as a \texttt{body} node.

A \texttt{body} node is used to represent the evaluation of an expression within an environment (i.e. a set of bindings from variable names to expressions). The left hand branch of a \texttt{body} node represents the expression to be evaluated (in this case the first clause of \texttt{isa}). The right hand branch of a \texttt{body} node represents the environment in which the expression is to be evaluated; this environment is represented using a node labelled \texttt{where}, reflecting the fact that such environments are created using the keyword \texttt{where} in Miranda. Note that in this implementation of Prospero the bindings of the environment are not shown. Note also that \texttt{body} nodes are omitted from the earlier figures in this chapter.

In the second definition of \texttt{isa}; the node labelled \texttt{lambda} represents an unchecked pattern match with its two branches representing the variable pattern \texttt{c} and the right hand side of the second clause.
Chapter 4 - A Tour of Prospero

The next stage in the pattern matching process is to attempt to match the first pattern against the argument. In order to do this the \texttt{lambda} node of the first pattern is 'applied' to the argument; this is shown in Figure 4.16(c).

The attempt to match the constant pattern "A" against the constant "b" fails, and so the \texttt{lambda} node is rewritten as a \texttt{grumpy} node; this is shown in Figure 4.16(d).

The presence of a \texttt{grumpy} node in a clause causes the whole clause to be rewritten as \texttt{FAIL} (Figure 4.16(e)) and the clause is then rejected and pattern matching with the next definition is carried out (Figure 4.16(f)).

Again the pattern matching is carried out by applying the \texttt{lambda} of the clause to the argument (Figure 4.16(g)). In this case the matching of the variable pattern \texttt{c} against the argument succeeds and the \texttt{lambda} is rewritten as \texttt{jolly} (Figure 4.16(h)).

Once pattern matching has succeeded for a clause, all other possible clauses can be rejected, in this case this simply means the rejection of the \texttt{ERROR} branch of the \texttt{Def Union} node (Figure 4.16(i)).

Finally the right hand side of the successful clause is instantiated, replacing all free occurrences of \texttt{c} with "b". In this case there are no free occurrences and so the evaluation results in the expression shown in Figure 4.16(k).

\textit{Presenting Guard Evaluation}

Figure 4.17 gives a definition of the factorial function, written using guards.

\begin{verbatim}
factorial x
    = 1 , x = 0 ∨ x = 1
    = x * (factorial (x - 1)) , x > 0
    = error ("factorial applied to negative number ->"
              ++ (show x)) , otherwise
\end{verbatim}

Figure 4.17 Definition of function \texttt{factorial}.

Figure 4.18 illustrates Prospero's representation of guards, and shows how the evaluation of the guard statements and the subsequent elimination of the clauses of the definition can be presented.
Figure 4.18 (a)
Figure 4.18 (c)
Figure 4.18 Evaluation of factorial illustrating the representation of guards.

Figure 4.18(a) shows the application of the right-hand-side of the definition of factorial to 3. The right-hand-side of factorial is a single Def. Union with the definition of factorial as the left branch and an ERROR node as the right branch. The ERROR node will be the result of the evaluation should pattern matching or guard evaluation fail.

The definition of factorial is a series of guard nodes, each of the nodes having three branches: the graph on the left branch is the condition associated with the guard; the centre branch is the body of the definition, which will be evaluated if the guard evaluates to True; the right branch is the graph which will be evaluated if the guard evaluates to False. The node labelled with ellipses is the result of a simple filter which converts long lists into an elided representation showing only the first and last element of a list. This particular example is the elided version of a lengthy string argument to the error function.

Figure 4.18(b) shows an instantiation of the right-hand-side following an application of factorial to 3. The instantiation involving the variable $x$ has resulted in a large
amount of sharing in the expression. This sharing is reflected by the large number of references to the 3 node in this evaluation stage.

Figure 4.18(c) shows the failure of the first guard in terms of the evaluation of the guard’s condition to False.

After the failure of the guard, the Guard node is replaced by its right branch. This is shown in Figure 4.18(d).

Figure 4.18(e) shows the second guard evaluating to True, and Figure 4.18(f) shows the subsequent selection of the right-hand-side associated with this guard (i.e. the middle branch of the Guard node). Once a guard has succeeded, then all other possible clauses of a function may be rejected. In this case only the ERROR definition remains, this is rejected, leaving the expression shown in Figure 4.18(g).

The examples in the section have illustrated the way in which Prospero enables the user to view pattern-matching and guard evaluation. These views provide information essential to a good understanding of an evaluation involving the application of a function to an argument outside its domain.

4.2.6 Space/Time Efficiency

In Chapter 2 I showed that one approach to providing information about the efficiency of a set of definitions was to show rewriting of the expressions taking place. Figure 4.19 shows the evaluation of expressions involving slowReverse and quickReverse (defined in Figure 4.19(a) and 4.19(b) respectively). Simply by comparing the trace of slowReverse (Figure 4.20) and quickReverse (Figure 4.21) it is clear that quickReverse is the most efficient of the two.

---

1 Note that this sharing could have been shown earlier in Figure 4.17(a). Showing the ‘pre-instantiation’ sharing has the advantage of indicating the sharing in a definition, without having to apply the definition to an argument. This approach is potentially confusing for the user, since the same variable name can appear a number of times in a pattern and across patterns, and thus some variables would appear to be shared, while others would not. Neither approach is wrong, and experimentation with both approaches would be the only conclusive way of deciding which approach is most appropriate.
Chapter 4 - A Tour of Prospero

(a)
slowReverse [] = []
slowReverse (x:xs) = conc (slowReverse xs) [x]
conc [] ys = ys
conc (x:xs) ys = x : (conc xs ys)

(b)
quickReverse [] acc = []
quickReverse (x:xs) acc = quickReverse xs (x:acc)

Figure 4.19 Definition of the functions slowReverse and quickReverse (duplication of Figure 2.9(a) and (b))

Figure 4.20(a)

Figure 4.20(b)

Figure 4.20(c)

Figure 4.20(d)
Figure 4.20 Evaluation of expression involving the function slowReverse
Figure 4.21 Evaluation of expression involving the function quickReverse

4.3 Scenarios where Prospero would Help with Teaching

In Chapter 2 I stated that it should be possible to widen the range of possible users of a system aimed at providing information about the errors in a set of definitions, in order to allow the system to be used to educate novices as well as helping them with errors.

When teaching, it is necessary to gradually reveal detail to students while preserving the consistency of any material which has been previously presented. Prospero’s filters facilitate the presentation of material at different levels of abstraction. Thus, it is possible to mirror the controlled disclosure of information in the classroom with the information disclosed at the interface.

Prospero can be used to support the teaching of some of the concepts frequently covered in a course on lazy functional programming. In the following section I will outline the way in which Prospero can be used to illustrate the concepts of laziness, pattern-matching and higher-order functions.
4.3.1 Explaining Laziness

In Chapter 2 I proposed that the laziness of a set of definitions can clearly be demonstrated using a system which shows the rewriting of expressions as they are evaluated. Figure 4.23 demonstrates this using the expression `double2 bottom v`. In this figure the first argument to the function `double2` (defined in Figure 4.22) can be seen to remain unevaluated, and it is clear that the second argument is evaluated only once.

```
double2 a b = b + b
bottom = bottom
v = 3 + 4
```

Figure 4.22 Definitions used to illustrate the presentation of lazy evaluation (duplication of Figure 2.10(a))
Figure 4.23 (d)
Figure 4.23 Evaluation of the expression `double2 bottom v`, illustrating the explanation of laziness.
4.3.2 Explaining Pattern Matching

In Chapter 2 I called for a system which would help to explain pattern matching in two ways: students would be shown how patterns govern the selection of the right-hand-sides of their definitions; and they would be shown how pattern matching can change the order in which expressions are evaluated.

Both of these needs can be satisfied by providing the user with an operational model of pattern matching which can be used to illustrate how pattern matching is involved in the evaluation of their definitions. As can be seen from the earlier section on pattern matching and guard evaluation, Prospero offers such an operational viewpoint, and so could be used to teach this material.

4.3.3 Explaining Higher-order Functions

In Chapter 2 I observed that one of the problems that students new to functional programming encounter, is understanding the concept of higher order functions. I then went on to suggest that one way in which this barrier could be overcome would be to have a system capable of including higher-order functions in the information that it presented about an evaluation. In fact, higher-order functions are central to functional programming, and thus it is essential that a system is capable of presenting them to users.

Figure 4.25 illustrates the presentation of higher-order values in Prospero. It shows the evaluation of the expression `my_last (map inc [1,2,3])`. The definitions of the relevant functions are given in Figure 4.24.

```
inc x = x + 1
my_last [x] = x
my_last (x:xs) = my_last xs
```

Figure 4.24 Definitions used to illustrate the presentation of higher order functions (duplication of Figure 2.12(a))
4.3.4 Illustrating Algorithms

Figure 4.20 demonstrates another possible use for a system which presents the stepwise evaluation of expressions. The figure shows the difference between two algorithms for reversing a list. It is clear that such a system can be used to compare different algorithms. Ideally both evaluations would be visible to the user at the same time, to allow easy comparison, and the system would allow the designer of the illustration to customise the presentation in such a way as to highlight the salient differences between the algorithms. Essentially the system I envisage would have the similar facilities of the algorithm animation system described in (Brown '88); designed to illustrate algorithms written in imperative languages.

4.4 Summary

In this chapter I have demonstrated that Prospero is capable of providing information about all of the aspects of an evaluation suggested in Chapter 2. I have shown that a system which provides information based on the rewriting of expressions can be used to help users when faced with a variety of errors, and can also be used to support the teaching of many of the concepts which would lie at the heart of a course on lazy functional programming.

Alongside this demonstration I have introduced a number of Prospero’s core concepts. I have shown that Prospero goes beyond the provision of basic rewriting information by providing filter and search systems. The filter system allows users to
view evaluations at different levels of abstraction, and allows the novel presentation of
the information in a reduction graph (e.g. the display of the distance of a function's
argument from its base case). A detailed description of the definition of filters, and the
implementation of the filter system is given in the following chapter.

I have also introduced Prospero's search system. This system can be used by users to
specify a point in the evaluation in which they have an interest. The system has also
be shown to be capable of simulating assertions. The search system does not take the
naive approach of allowing searches to change the order in which evaluation stages
are presented to the user. However, the search system does allow the user to define
and experiment with the amount of evaluation the search system is allowed to carry
out in order to test whether the search condition is satisfied. The search system is
described in detail in Chapter 6.

Finally, aspects of the lazy-evaluation scheme adopted by Prospero have been
described. This chapter has shown that while the display of such low-level
information is not the norm, it is essential when the display of information about the
pattern-matching and guard evaluation mechanism would be fruitful for debugging or
pedagogical reasons.
Chapter 5 - Filters

5 Presenting Evaluation Information using Filters

So far in this thesis I have shown that rewriting is a suitable basis upon which to base views of the lazy evaluation of expressions. I have also shown that there is a trade-off to be made between the advantages and disadvantages of providing low-level rewriting information. The advantage of providing low-level information is that the system is not tied to a level of abstraction too high to satisfy the needs of some users, since certain views of an evaluation can be provided once low-level information is made available. The disadvantage is that this low-level information is not always appropriate, and frequently occurs in high volumes.

In this chapter, I will present Prospero's filter system which is used to specify the parts of the available information that will be displayed to the user, and how that information will appear. The filter system allows users to define their own levels of abstraction, or to have a one or more levels made available to them.

The filter system could be seen as little more than a counter-measure for the problems associated with low-level information. I will show that while this is a crucial role for the filter system, it is by no means the only one. The filter system goes far beyond this and provides a powerful means by which to produce views of an evaluation stage which contain information pertinent to the needs of the user. A number of uses of the filter system will be demonstrated in this chapter.

This chapter is structured as follows: An introduction is given to the different kinds of filters available, together with an overview of how these filters are defined by the user; this is followed by a description of why filtering is necessary in Prospero, together with a number of illustrations of filtering; finally, a more detailed account is given of the way in which filters are defined, and the implementation of the filter mechanism.
5.1 What is Filtering?

In photography, a filter is used either to prevent certain frequencies of light from passing into a camera from a scene, or to change the appearance of a scene. In electronics, filters are used to remove noise from a signal, or to remove part of the signal. Both of these types of filter can be said to remove information passing between a source and a sink.

Prospero generates information about the evaluation of expressions written in a lazy functional language. As we shall see in this chapter, the usefulness of this information can be greatly enhanced if it is filtered before it reaches the user. In Prospero’s case the source of information is the internal representation of an evaluation, and the sink is the display on which this information is presented.

An overview of Prospero is shown in Figure 5.1. An evaluation is represented by a sequence of Reduction Graphs. The user is able to view an evaluation on a display. Using filters, which are interposed between the internal representation of the evaluation and the display, the user is able to select which parts of the available information about an evaluation are to be displayed, and how that information is to be presented.

![Figure 5.1 Overview of Prospero](image)

5.2 How is Filtering Done?

In this section I will introduce the basic concepts involved in Prospero’s filtering system.

5.2.1 Augmented Reduction Graphs

Figure 5.2 gives an outline of the processing that takes place in order for an expression which is represented as a reduction graph (i.e. the internal representation
of a stage in an evaluation) to be displayed to the user. The first stage of this process is the transformation of the reduction graph into an Augmented Reduction Graph (ARG).

![Diagram of the Prospero viewing pipeline]

Figure 5.2 The Prospero viewing pipeline

An ARG has the same structure and contains the same information as the reduction graph, together with additional fields at each graph node. It is the job of Prospero's filters to set the information in these fields, and it is this information that is subsequently used to generate graphics commands when displaying an expression. These additional fields, called 'filter fields', contain:

- The text which will be displayed at the node;
- A list of edges to other nodes of the graph.

The initial augmentation of the graph simple involves the addition of these two extra fields at each node of the graph, and setting the fields to initial dummy values.

In the subsequent stages of the display process the ARG is passed through a filter, the filtered ARG is transformed into a series of graphics commands using a plot function, and these commands are then used to display the expression.

5.2.2 Simple Filters

Prospero offers the user two different kinds of filter. The most straightforward of these is a simple filter. A number of examples of simple filters have been presented in earlier chapters. For example, the filtering of binary trees in Figure 4.3 would have been achieved in a fully implemented Prospero using a simple filter. In this section I will describe how simple filters are defined. In order to understand the details of the
definition process, it is necessary to understand how *simple* filters are used in the Prospero system.

Looking back at Figure 5.2, the second stage in the display process involves the transformation of an ARG to a ‘Filtered Reduction Graph’ (i.e. an ARG which has been through the filtering process). All of the Prospero system is written in Miranda (with the exception of the user interface described in Chapter 8). Filtering involves the application of Miranda functions to each ARG. Thus, each *simple* filter can be defined by the user as a Miranda function from ARG to ARG.

In order to define a new filter, the user must write a Miranda script which is then incorporated into Prospero’s display mechanism using the Miranda compiler. This means that users are adding their own functions to Prospero whenever they define and use a *simple* filter. An example of the definitions which users are required to write is given at the end of this chapter. For now *simple* filters will be described in more abstract terms.

*The Components of a Simple Filter*

A *simple* filter is a function which takes an ARG and returns an ARG with some of the information in its filter fields set to new values. A filter has three main components:

- a search condition which, when applied to a graph, evaluates to *True* or *False* depending on whether the graph is one which the filter should be applied to;

- a label generator which, when applied to a graph, returns a string which will be used to label the root of the graph when it is displayed;

- an offspring generator which, when applied to a graph, returns references to the offspring of the root of the graph.

A number of examples of the definition and use of filters define in this way, are given later in this chapter. One such example is shown in Figure 5.17. In this example, the search condition evaluates to *True* whenever it is applied to a graph representing the application of an infix operator, the label generator returns a string representing the operator, and the offspring generator returns references to the arguments of the operator.
5.2.3 Composition of Simple Filters

Filters take an ARG as their argument, and return one as their result. This allows them to be composed with each other to form a new filter. Using function composition, the user may define a filter which deals solely with a particular type of expression and then compose it with more ‘general’ filters which will deal with other elements of an expression.

This technique allows users to build complex filters simple by re-using existing ones. If a filter is required which can not be built using composition, the user need only define the novel parts of their filter, and compose these definitions with existing filters to achieve the required effect. Examples of the composition of simple filters are given throughout this chapter. One such example would be the combination of the list_contents filter shown in Figure 5.4 and the point filter shown in Figure 5.12, to give a simple filter which handled lists of points.

5.2.4 Temporal Filters

A simple filter is a function which is applied to the ARG every time a stage in the evaluation of an expression is displayed to the user. This type of filtering can be used to produce a number of useful filtering effects. However, many of the examples of filtering in this thesis cannot be carried out using simple filtering.

When Simple Filtering is not Enough

In Figure 4.6 all of the stages in the evaluation of the application of map have been filtered out of the evaluation. The evaluation of the application of map involves a number of reduction steps, and so filtering must take place over these steps. This will involve the filtering of a number of different ARGs (one for each step). Using simple filters this would require one filter for each step in the evaluation.

However, even the use of a set of simple filters would not necessarily achieve the desired result, since the sub-expressions that represent the evaluation of the application of a function may not be unique to the evaluation of that function.

For a trivial example of this, imagine that a user wishes to filter the evaluation of map, and attempts to do this using a set of simple filters, one for each step in the evaluation of an application of map to its arguments. The user also has a function map' which is identical to map. If the user should attempt to use this set of filters on an evaluation which involved the application of both map and map', there would be no way to
differentiate between a stage in the application of \texttt{map} and a stage in the application of \texttt{map}'. Thus the user would be inadvertently filtering both functions.

Temporal filters overcome this problem by providing the user with two facilities. The first facility is a matter of convenience, it provides the user with the ability to filter an expression for more than a single step, without having to write a simple filter for each of the steps involved. The second facility is necessary in order to provide the ability to filter an expression through a number of evaluation stages, even when those evaluation stages may be identical to the stages in the evaluation of other expressions.

Temporal filters allow the user to define a filter which detects an expression of interest when the expression is created and can then be applied to the representation of that expression for more than a single step. The filter can also detect when to stop filtering an expression, since it may not be the case that filtering is required for the whole of the lifetime of the expression.

At the time of writing, temporal filters have not been incorporated into the implementation of Prospero, this is for no other reason than the time available to carry out the research described in this thesis.

\textit{The Components of a Temporal Filter}

A temporal filter has three components:

- \textit{a start condition:} a function which evaluates to \texttt{True} when applied to an evaluation stage which contains an expression which the filter should be applied to;

- \textit{a mask:} a function which determines the appearance of the expression throughout the life of the filter. In fact \texttt{masks} are specialised \texttt{simple} filters. The way in which this specialisation takes place is detailed below;

- \textit{an end condition:} a condition which evaluates to \texttt{True} when the expression being filtered becomes a value which marks the end of the lifetime of the filter.

Using these three components it is possible to define a filter which presents all expressions which satisfy the \textit{start condition}, using the \texttt{mask}, until the \textit{end condition} is satisfied. Again, details of the definition of temporal filters, and the implementation of the mechanism which uses them is presented later in this chapter.
Chapter 5 - Filters

5.3 Why is Filtering Necessary?

There are many reasons for using filters when viewing information about an evaluation, in this section I will discuss some of these reasons. In the following section I show how this filtering is achieved.

5.3.1 Removal of Low Level Evaluation Detail

In the previous chapter, I demonstrated that in order to provide the widest possible range of information about the evaluation of expressions, it is necessary for Prospero to generate information which is at a very low level. For example, detailed information about pattern matching is generated every time a function is applied to an argument.

There are a number of reasons why the presentation of this information to users would not always be appropriate:

- Low level information is far removed from the syntactic level at which the user writes function definitions; it contains internal representational details which would be initially unfamiliar to users.

This problem is analogous to providing a Pascal programmer with a debugger, such as Unix’s adb, which presents the execution of a program in terms of machine code instructions and the contents of a machine’s memory. Of course such tools have their uses, but this kind of view of an evaluation should be available rather than mandatory.

- Low level presentation produces a large volume of information when presenting steps in the evaluation of even the most trivial expressions, leaving users overwhelmed by the volume of information that they are faced with.

- A low level presentation would not reflect any data and functional abstraction present in a user’s definitions.

- The representation would promote a very low level view of a high level language. It therefore runs the risk of encouraging the user to concentrate on how functions are evaluated rather than writing functions in a clear, concise and elegant way.

- The representation does not accurately represent the way in which optimised functional programs are executed.
• In some circumstances, the provision of low-level detail does not provide the user with information about an evaluation, rather it presents the user with data about an evaluation. It is not until this data is reproduced in a more abstract form, that it becomes informative.

In order to overcome these problems, it must be possible to present evaluations in a form which does not contain all of the information available. One of the main uses of filters is to facilitate this removal of information.

5.3.2 Removal of Value Details

A filter system can be used to present values in a number of ways. The most basic advantage offered by filtering is to remove low level information about values. Just as it is likely for a user to be more comfortable writing \(\{1, 2, 3\}\) rather than \(1 : (2 : (3 : []))\), it is reasonable to expect that a system which presented lists in the first form rather than the second would be a more successful one. A filter which achieves this effect is the list-contents filter.

![Figure 5.3 Unfiltered representation of \(\{1, 2, 3\}\)](image)

The list-contents filter

Figure 5.3 shows an unfiltered representation of the expression \(\{1, 2, 3\}\). Figure 5.4 is an example of filtering, it shows the same expression after it has passed through the list-contents filter. All of the low level information concerning the representation of lists has been removed, giving a representation which is more familiar to the user. This kind of filtering is essential to the success of the Prospero interface; without it the
presentation of all but the simplest expression would be too cumbersome, and too far
removed from the level of concrete syntax at which the user works.

Figure 5.4 Example of the list [1,2,3] filtered by the list-contents filter

The filtering shown in Figure 5.4 is achieved using a simple filter with the following components:

• a search condition which, when applied to a graph representing the
  application of the list constructor \texttt{cons}, evaluates to \texttt{True};

• a label generator which, when applied to a graph, returns a string \texttt{list};

• an offspring generator which, when applied to a graph, returns references to
  all of the operands of \texttt{cons} in the graph.

Filtering and user-defined constructors

Filtering extends to user defined value constructors. Figure 5.5 is an example of an
algebraic type and Figure 5.6 shows a filtered expression of this type, where the
representation of the constructor and its arguments has been changed in order to bring
it closer to the level of concrete syntax.
**Figure 5.5** A user defined type

\[
\text{circus\_Act} ::= \text{Juggler bean\_bags fire\_sticks | Trapeze height troupe\_members safety\_net | Clowns troupe\_members exploding\_car | Sword\_swallower}
\]

**Figure 5.6** Filtering of a user defined data type

The filtering shown in Figure 5.6 is achieved using a *simple* filter with the following components:

- a search condition which, when applied to any graph representing the application of any user defined value constructor, evaluates to True;

- a label generator which, when applied to a graph, returns a name of the value constructor;

- an offspring generator which, when applied to a graph, returns references to all of the operands of the value constructor.

Note that while the above filter will work for all user defined value constructors. It is possible to write filters which are specialised in respect of a specific set of constructors. It is also possible to compose the specialised filter with the more general one, to create a filter which combines the filtering effects of both filters.
Filters also allow users to highlight certain attributes of values, or to summarise information about them. A simple example of the highlighting of attributes of a value is shown in Figure 5.7. This shows a number of different views of a binary tree all of which can be generated using filters.

Figure 5.7 Various views of a balanced tree
Chapter 5 - Filters

The filtering shown in Figure 5.7 is achieved using a number of simple filters. All of these filters have the same search conditions and offspring generators:

- a search condition which, when applied to a graph representing the application of the list constructor \texttt{cons}, evaluates to \texttt{True};
- an offspring generator which, returns references to all of the operands of value constructors in question.

It is the different label generators which govern the appearance of the trees. In Figure 5.7(a), the label generator \texttt{simple} returns the name of the value constructor involved.

In Figure 5.7(b) we see the first example of the use of the fact that the label generator is a function defined over ARGs. Here the label generator traverses the data structure and calculates the depth of the tree below the node for which it is generating a label.

Similarly, the label generators in Figure 5.7(c) and (d) are functions which traverses the data structure representation and return a string representation of the information that they find there.

Another way of changing the presentation of data structures is to summarise the contents of large data structures (e.g. long lists or large trees). Figure 5.8 shows how filters can be used to carry out this summary. Here the list containing the numbers 4 to 40 is presented using elision.

![Figure 5.8 Elision of a list](image)
5.3.3 Removal of Evaluation Detail

Filters allow the user to remove information about the evaluation of an expression. That is, filters can be used to remove stages in the presentation of an evaluation and to specify how the remaining stages are to be presented.

Support for Functional Abstraction

One of the great strengths of functional languages is the availability of higher order functions, which supply the user with a very powerful means of abstraction and so a powerful tool for writing programs (Bird & Wadler '88, Hughes '89).

A classic example of this technique is the development and subsequent reuse of the foldr function, given in (Hughes '89). Here the foldr function is first discovered while defining the function \textit{sum}, which sums the elements of a list of numbers.

The first definition given for \textit{sum} is:

\[
\begin{align*}
\text{sum} \; [] &= 0 \\
\text{sum} \; (\text{num:} \text{list}) &= \text{num} + \text{sum list}
\end{align*}
\]

Hughes then observes that only the boxed parts of this definition given above are specific to computing a sum and that a \textit{foldr} function defined as:

\[
\begin{align*}
\text{foldr} \; \text{op} \; x \; [] &= x \\
\text{foldr} \; \text{op} \; x \; (\text{el:} \text{list}) &= \text{op} \; \text{el} \; \text{foldr} \; \text{op} \; x \; \text{list}
\end{align*}
\]

could be used to express \textit{sum} in the following way:

\[
\text{sum} = \text{foldr} \; (+) \; 0
\]

Hughes then goes on to use the \textit{foldr} function to define functions such as:

\[
\begin{align*}
\text{product} &= \text{foldr} \; (\times) \; 1 \\
\text{anytrue} &= \text{foldr} \; (\lor) \; \text{False} \\
\text{alltrue} &= \text{foldr} \; (\&) \; \text{True}
\end{align*}
\]
Chapter 5 - Filters

Such a derivation and subsequent re-use of higher-order functions is common place in Miranda programming, and is a fundamental part of the syllabus represented in texts such as (Bird & Wadler '88). Because of this, Prospero’s interface must support this type of abstraction in order to simplify the presentation of evaluations which involve such functional abstractions. At the same time it must be possible, in certain circumstances, for the user to be able to see through the abstraction.

For example, it will not be necessary to show the reduction of foldr to a user who is using it as an abstraction, but it must be possible to investigate the detailed reduction of an expression involving an application of foldr if necessary. Such detailed information may be required when the user is checking that an application of the function is behaving as expected, or when the user is learning how the function works by watching the details of an application. These varied demands on the kind of presentation required of a system can all be met using the filtering system provided by Prospero.

The following example (Taylor '91) demonstrates the usefulness of this approach over one where the user is committed to a particular view of an evaluation. The example follows from my own observations of a set of novice programmers\(^1\). Exercises set for the students were often aimed at illustrating the properties of in-built functions. The cause of this particular error was a misconception about the meaning of the in-built foldr function.

Students were set the problem of writing a function listOr which evaluates the logical or of a list of booleans. Thus:

\[
\begin{align*}
\text{listOr } [\text{True, False, False}] &= \text{True} \\
\text{listOr } [\text{False, False, False}] &= \text{False}
\end{align*}
\]

\(^1\)The programmers studied were taking a Masters conversion course in Computer Science. Few of them had done any programming before taking the course. All of them had just completed an 11 week course in Modula-2 and Prolog programming.
Many of the students appeared to be confused regarding the second argument of foldr and seemed to guess at a value. This resulted in a number of students producing the following definition of listOr:

\[ \text{listOr aList} = \text{foldr} (\lor) \text{ True aList} \]

rather than:

\[ \text{listOr aList} = \text{foldr} (\lor) \text{ False aList} \]

When asked to explain how their definition of listOr worked, the students would disregard the second argument to foldr, and answer that the foldr function “places the \( \lor \) operator between each element of the list”. When asked to write down the stages in the evaluation of:

\[ \text{listOr [False, False, False]} \]

a student, who had made an error, would respond with:

\[ \text{False} \lor \text{False} \lor \text{False} \]

rather than:

\[ \text{False} \lor \text{False} \lor \text{False} \lor \text{True}. \]

Clearly, if the evaluation of listOr [False, False, False] were to be viewed with a filter which removed the detail of any evaluation involving an ‘in-built’ function (Figure 5.9), then no insight would be gained from the display. In fact the display would be no better, with regard to understanding the error, than that given by the Miranda system itself.
Figure 5.9 An evaluation involving the filtering the in-built function foldr

This filtering would be achieved using two *temporal* filters. The first of these filters would have the following components:

- the *start condition* would be a function which evaluated to True when an application of foldr to three arguments appeared in the evaluation;

- the *mask* would generated the ARG representing the application of foldr to its arguments;
Chapter 5 - Filters

- the end condition would be True when the application of foldr reached weak head normal form.

The second filter would have the following components:

- the start condition would be a function which evaluated to True when an application of listOr to an argument appeared in the evaluation;

- the mask would be a function which generated the ARG representing the application of listOr to its arguments;

- the end condition would be True when the application of listOr reached the point where it had been rewritten to become an application of foldr.

In order to properly investigate the student's error, it is necessary to remove any filtering involving the foldr function. In this way, a display such as that of Figure 5.10 would illustrate the mistake that had been made.

![Figure 5.10(a)](image1)

![Figure 5.10(b)](image2)
Chapter 5 - Filters

Figure 5.10 An evaluation with filtering of in-built functions removed

*Focusing on Specific Functions*

When faced with an error, it is common for a user to attribute the error to a particular function, or conversely to believe that most of the functions do not contain errors. The user may express these beliefs by focusing the presentation of the evaluation on the suspect functions. This focusing can be achieved by carrying out removal of evaluation detail using filters.
This facility is equally useful when teaching, where it is often desirable to focus the student’s attention on a certain aspect of an evaluation, either because the evaluation detail of certain functions will confuse students to whom the functions are not familiar, or because the evaluation detail of other functions will only serve to distract the student. Again, using filters it is possible to remove unnecessary detail of the evaluation of certain functions.

The users of imperative debugging systems are usually presented with the option of seeing the evaluation of a procedure call, or skipping it. Prospero’s filter system is more flexible than this approach in that it allows users to choose a middle ground and to see part of the evaluation of a function. For example the user may not want to view the details of pattern matching following a function application, but may wish to see the evaluation of the function’s body following pattern matching. By using filters to view an evaluation, the user is allowed to take a position between the monolithic reduction step of the Miranda system, and the single graph-reduction step of an unfiltered evaluation in Prospero.

5.3.4 Supporting Abstract Data Types and Type Synonyms

Examples of a type synonym and an abstract data type (ADT) are shown in Figure 5.11(a) and Figure 5.11(b) respectively.

(a)

point == (num,num)

(b)

abstype point with
  x::point->num
  y::point->num
  mkPoint::num->num->point
  showpoint::point->[char]
point == (num,num)
x (a,b) = a
y (a,b) = b
mkPoint a b = (a,b)
showpoint (a,b)
  = '<' : (show a) ++
     "", ++ (show b) ++ ">"
Chapter 5 - Filters

By using filters to present evaluations which involve such data abstraction, users can maintain the abstraction when required, and at the same time are able to remove the abstraction when necessary. In this way it is possible to use the data abstraction to simplify the presentation of an expression and so highlight the features of the expression which are most salient. Figure 5.12 shows a point which is presented using a filter to support the data abstraction of Figure 5.11(b). Figure 5.13 shows the same expression without this filter, leading to the presentation of the low level representation of pairs (2-tuples).

This representation takes the following form. Each n-tuple is represented as a Tuple node. The left branch of this node is the arity of the tuple, the right branch will be n-1 instances of an In-Tuple node, each of which has a left branch representing a value in the tuple, and a right branch which is either another In-Tuple node, or the final element in the tuple.

Figure 5.12 A filter used to support an abstract data type
Using Temporal Filters to Support Data Abstraction

Without temporal filtering it is impossible to single out specific occurrences of an expression for special filtering while ignoring other occurrences of the same kind. For example, it could be that a user has chosen to represent a point as a pair of integers as shown in Figure 5.11. An attempt to support this abstraction using a simple filter which searches for the pair constructor in a graph and checks if the members of the pair are integers, will fail. This is because there may be a number of pairs of integers in a graph and only some of these pairs should be thought of as points.

The filter used in Figure 5.12 is a temporal filter with the following components:

- the start condition would be a function which evaluated to True when an application of mkpoint to two arguments appeared in the evaluation;

- the mask would generate the ARG representing a point;

- the end condition would always be False, since we require that points are filtered for as long as they are part of the expression.
Chapter 5 - Filters

An alternative approach to the support of ADTs is to create a filtering system which is provided with type information, and so would be able to differentiate between an instance of the ADT point, and any other pair of numbers. However, this solution is a less powerful one, since temporal filters allow the user to differentiate between different kinds of point, for example a different way of presenting points may be required for those introduced by a create_rectangle function, where the rectangle was represented in terms of two points (the origin and corner). The components of this temporal filter would be:

• the *start condition* which evaluated to True when an application of create_rectangle appeared in the evaluation;

• the *mask* would generate the specialised representation of points in this sub-expression;

• the *end condition* would always be False, since we require that points are filtered for as long as they are part of the expression.

*Supporting the Pedagogical Use of ADTs*

ADTs allow programmers to think about their definitions in abstract terms; this means that they will be freed from the implementation detail hidden by ADTs and will hopefully be able to find their errors more quickly by working at the higher level of abstraction. It is therefore desirable for Prospero to support these abstractions, failure to provide this support would negate many of the advantages that ADTs offer in both a development and teaching environment.

In a pedagogical context, if certain complexities of an exercise are hidden from the students using an ADT, it is possible for students to be set problems which are valuable but which involve aspects of the subject which the students are not familiar with. An example of the use of filtering to support this practice is given below.
Chapter 5 - Filters

This example, taken from (Sethi '89), involves an exercise set for a group of first-year undergraduate Computer Science students. The problem is to define a set of functions which create patch-work quilt patterns on the screen. The students were given a set of pre-defined ‘patches’ and four functions which are defined by the ADT quilt.

```
abstract quilt
  with
    rotate :: quilt -> quilt
    reflect :: quilt -> quilt
    sew :: quilt -> quilt -> quilt
    showQuilt :: quilt->[char]
```

The exercise (an example of which is shown in Figure 5.14) was the first programming problem set to the students. It was intended to introduce them to function composition; abstraction was used to hide large amounts of implementation detail. The students had no debugging or development aids available to them, and soon faced the problem of having the wrong result on the screen and no intermediate steps in the evaluation of their expressions. By the end of the practical session the laboratory was strewn with hastily drawn paper ‘patches’, which the students had used to hand evaluate their expressions. These patches frequently reflected the conceptual errors that the students were making, but rather than helping to correct these errors, the patches reinforced them, making the students more certain than ever that their solution was correct: after all they could see it with their own eyes.

```
given the quilt:
  ***---
  **---
  *-----
produce the quilt:
  ---------
  ---------
  ---------
  ---------
  ---------
  ---------
```

Figure 5.14 An example exercise involving quilts

Figure 5.15 shows the presentation of an error-free evaluation using filters to support the abstraction of quilts. If such presentations had been available to the students, they would have replaced the use of hand-drawn patches, and would have given the
Chapter 5 - Filters

students a step-by-step presentation of why the result they expected was not the result they got.

Filters are used in this example to raise the information presentation to the level of abstraction at which the students were working, and at which the problem was expressed. All of the information hidden using functional and data abstraction is also removed from the Prospero presentation.

Figure 5.15(a)

Figure 5.15(b)

Figure 5.15(c)

Figure 5.15(d)

Figure 5.15 An evaluation involving filtering of quilts
5.3.5 Summary

In this section I have shown that filtering goes far beyond the removal of low-level information from evaluation stages. Filtering allows users to re-pesent sub-expression in an evaluation stage in a range of ways. Examples have been shown where filters have been used to support functional and data abstraction in a users definitions, and have been used to produce different views of data structures ranging from simple elision, to the summarisation of a data structure using functions defined over the data structures content.

This section, in demonstrating the usefulness of a number of views of filtering, has shown one of the main differences between Prospero and other systems which produce rewriting information. Prospero’s model of an evaluation, combined with its filter system, allows many different views of an evaluation to be created, rather than the single view provided by other systems.

5.4 Filter Definition and the Filter Mechanisms

In the final part of this chapter, I will provide more detailed information about the way in which users define simple and temporal filters. I will also discuss the implementation of the mechanisms which take these filter definitions and put them in to effect.

5.4.1 Simple Filters

I will illustrate the way in which a simple filter is defined using an example based on a commonly used filter: namely the operator filter. Figure 5.16 shows an unfiltered representation of the expression 2+3. The operator filter will be defined such that all applications of operators are represented in the way shown in Figure 5.17. The three components of the operator filter are defined as follows:

- a search condition which evaluates to True when it is applied to a graph which is the application of an operator to its argument(s);

- a label generator which returns a string representation of the operator;

- an offspring generator which returns a reference to each of the operator’s arguments.
Figure 5.16 Unfiltered view of 2+3

Figure 5.17 Filtered view of 2+3

Defining the Operator Filter

Since defining a filter involves the writing of a set of Miranda functions, it is possible to provide users with a number of definitions to simplify the creation of a new filter. The use of these definitions to produce the operator filter illustrated above is shown in
Figure 5.18 Definition of an operator filter

Figure 5.18. These definitions show that, in the present version of Prospero, many of the arguments to the functions involved in the filter are in terms of ARGs.

The filter itself is called op_filter and is defined on the third line of the figure. This definition states that op_filter is a filter made up of the three components: is_op, op_label and op_paths. These components are the search condition, label generator and path generator described above. The function create_filter is provided to the user in order to compose the three components of a filter.

The first component of the simple filter is the search condition. This function should return True when applied to an ARG representing a sub-expression which we wish to filter. For the operator filter, the search condition is is_op, which returns True if it is applied to an ARG representing either the application of an infix operator to two arguments, or the application of a unary function to a single argument.

---

1While this is sufficient in order to experiment with the creation of filters, it would not be acceptable to require 'real' users of the system to understand the ARG representation of expressions in order to write filters. This point is expanded upon in the Chapter 9.
Chapter 5 - Filters

The second component of the *simple* filter is the label generator. This function should return a string which will be used to label the graph node at the root of the sub-expression that we are filtering. For the *operator* filter, the label generator is the function `op_label`, which simply converts an expression which is an operator into a list of characters using the pre-defined function `show_op`.

The third component of the *simple* filter is the offspring generator. The offspring generator `op_paths` returns a list of paths leading from the root of the graph representing the application of the operator, to the operands. Each path is a list of directions. Because each internal node of the ARG can only have edges to two other nodes, the directions are in terms of taking the left or right edge leading from a node. So the directions to the operands of an infix operator are `[Left, Right]` to reach the first operand, and simply `[Right]` to reach the second. These paths are illustrated in Figure 5.19.

![Figure 5.19 Paths leading to operands of infix operator](image)

*Defining op_filter Using Filter Composition*

The most basic filter of all is one which when used to display a stage in an evaluation will result in the display of all of the information in that stage. This filter is pre-defined for the Prospero user, and is called the 'all-pass' filter. It sets the filter fields of each node of the ARG to be values which will result in the display of the complete graph.

The all-pass filter is implemented as a case statement defined over the complete set of primitive graph nodes, i.e. the full set of nodes used by Prospero’s interpreter to represent all possible Miranda expressions, and all stages in the evaluation of these expressions. The all-pass filter sets the text field of each node to be a textual representation of the node, for example the text field of a `λ` node will be "LAMBDA"
and the text field of a + node will be "+", and sets the list of edges to be the edges already present at the node.

The definition of op_filter given above only describes the filtering that should take place when an operator applied to its argument(s) is found in the representation of an evaluation stage. It says nothing about how the operands of the operator should be filtered, nor does it define the filtering for any other sub-expression. This is typical of Prospero’s filter system: each filter is specific to a particular kind of expression and leaves all other expressions unchanged. In the case of the expression shown in Figure 5.17, the filter acts upon the application of an operator to its arguments, and nothing else; even the arguments to the operator must be dealt with by a separate filter.

How then are the arguments of the infix operator, and any other sub-expression, not dealt with by the operator filter, to be handled? Examining Figure 5.17, we see that the numbers which form the arguments to the + operator appear exactly as they did in Figure 5.16. Earlier I stated that Figure 5.16 was an unfiltered representation of an application of the + operator. In fact the figure is not unfiltered in the strictest sense of the word. Figure 5.16 is the result of using a single filter called the all-pass filter. Figure 5.17 shows the result of composing the operator filter with the all-pass filter.

Preventing Overwriting

A simple rule is used to prevent filters ‘overwriting’ filtering that has been carried out by the filters they are composed with. This states that no filter may change the fields of a node once they have been set by another filter. An implication of this rule is that composition of filters that are not mutually exclusive with respect to the nodes that they effect, will produce a different displayed graph depending on the order in which they are composed. For example, if a filter was defined to be the operator filter composed with the all-pass filter, then the result would be the filter demonstrated in Figure 5.17. However, if the filter was defined as the all-pass filter composed with the operator filter then the result would be exactly the same as the all-pass filter on its own (Figure 5.16). This is because the all-pass filter would change the filter fields of all nodes in the ARG, making any changes by the operator filter impossible.

To summarise, simple filters are defined by users in terms of a Miranda script. The script defines the filter as a function from ARG to ARG. The definition of the filter function is simplified by the provision of a set of pre-defined functions. The filters defined in this way will usually be specific to a particular kind of sub-expressions, and the remainder of the filtering necessary before an expression can be display is defined by composing the filter with other more general ones.
5.4.2 Defining Temporal Filters

Like simple filters, temporal filters would be defined as Miranda scripts; making use of a set of pre-defined functions to ease the definition process.

A temporal filter’s start condition would be a function which takes an ARG as its argument and returns True if the ARG is a sub-expression which should be filtered.

Similarly, the stop condition would be a function which takes an ARG as its argument and returns True if the ARG has reached a point where filtering should stop.

The mask associated with a temporal filter is a specialised simple filter. As we saw in the previous section, simple filters use a path generator to refer to the siblings of the root of the filtered ARG. Because a temporal filter may exist for more than a single evaluation step, it is impossible to refer to the siblings of an ARG using literal paths. This is because the evaluation of the filtered expressions will result in changes to the graph representing the sub-expression, and so change the position of the siblings referenced through the paths.

To overcome this problem, a facility is provided by the temporal filter mechanism allowing a filter to refer to particular sub-graphs of the ARG throughout the filters ‘life’. These references allow the mask to keep track of sub-graphs as rewriting takes place. The sub-graphs are referenced using paths which lead to the sub-graphs at the point at which the filter becomes active (i.e. when the start condition is met), from then on, the references are automatically updated by the filter mechanism, and the mask is provided with the list of siblings as if they had been located by paths defined for each of the reduction stages for which the filter is active.

An example of the way that masks can follow sub-graphs as they move around an expression is shown in Figure 5.20. This figure shows the evaluation of the expression f (1+2). Where the function f is defined as:

\[ f \ 3 \ = \ True \]

The left of the figure shows the reduction stages in the evaluation as if they had passed through the all-pass filter. The right of the figure shows a sketch of the appearance of the expression to the user when using a filter with the following components:

- the start condition would be a function which evaluated to True when an application of f to an argument appeared in the evaluation;
• the mask would generate the ARG representing the application of \( \varepsilon \) to its arguments. In order for it to be able to do this the mask has a reference to \( \varepsilon \)'s argument, allowing the filter access to the argument even as it moves around the expression;

• the end condition would be True when the application of \( \varepsilon \) has evaluated to True or False.

In the first stage of the evaluation \( \varepsilon \) is applied to \((1 + 2)\), thus the start condition holds and the filter becomes active. The expression is presented as the application of \( \varepsilon \) to its argument, and a reference to the argument in the expression is created (shown shaded in the figure) in order to allow the tracking of this argument during the evaluation.

In the second stage, \( \varepsilon \) is replaced by its right hand side. The expression is still represented as it was in the first stage.

In the third stage the argument is pulled into the right hand side of the definition of \( \varepsilon \). The reference to the argument becomes necessary at this point since the position of the argument has changed, and the explicit reference to it using a path would have failed to reach the correct sub-graph.

In the fourth stage the argument is reduced in order for pattern matching to be carried out. The reference to the argument means that this reduction is automatically reflected by the filter.

In the fifth stage pattern matching succeeds meaning that in the sixth stage the ERROR clause can be thrown away. Both of these stages are represented in the same way as in the third stage.

Finally, in the seventh stage the expression is reduced to True, the end condition holds and the filter is no longer active.
Chapter 5 - Filters

Figure 5.20 Example of sub-graph tracking by a temporal filter

In the case of the `foldr` example shown in Figure 5.14:
Chapter 5 - Filters

• the start condition would be a function which evaluated to True when an application of foldr to three arguments appeared in the evaluation;

• the mask would generate the ARG representing the application of foldr to its arguments. In order for it to be able to do this the mask has a reference to foldr’s arguments, allowing the filter access to the arguments even as they move around the graph;

• the end condition would be True when the application of foldr reached weak head normal form. This filter is illustrated along with another temporal filter in Figure 5.14.

The second temporal filter in Figure 5.14 is one where:

• the start condition would be a function which evaluated to True when an application of listOr to an argument appeared in the evaluation;

• the mask would be a function which generated the ARG representing the application of listOr to its arguments. Again the mask would have references to these arguments, allowing the filter to refer to them as they move around the graph;

• the end condition would be True when the application of listOr reached the point where it had been rewritten to become an application of foldr.

The Temporal Filter Mechanism

At the time of writing, the temporal filter mechanism is not incorporated into Prospero. This section provides an outlined of the envisaged mechanism.

Because temporal filters are essentially constructed from two conditions and a simple filter the temporal filter mechanism is a simple extension of the mechanism used to apply simple filters to an ARG. The temporal filter mechanism will maintain two tables of filters: the first containing an entry for every temporal filter available (the filter table); the second (the active table) keeping a record of all temporal filters which are currently active (i.e. their start condition has been met and their end condition has not), together with a pointer to the sub-expression which met the start condition (the target), and pointers to any sub-expressions which need to be tracked by the mask (e.g. the argument to f in Figure 5.20).
Before any reduction stage is displayed to the user the following actions will be carried out:

- the *start* condition of all filters in the *filter table* are applied to the ARG; any filters whose *start* condition is met are copied into the *active table* and any references to sub-graphs required by the filter are instantiated;

- the *end* condition of all filters in the *active table* are applied to their *target*; any filters whose end condition is met is removed from the active table;

- the mask of all of the filters in the *active table* is used to replace the filter’s *target* in the ARG.

Note that temporal filtering would be carried out before *simple* filtering in order to avoid simple filters changing an ARG in such a way that it interferes with the checking of the condition associated with a *temporal* filter, and to allow *simple* filtering of parts of an ARG produced by a *mask*.

5.5 Summary

Without filters Prospero would be nothing more than an implementation of a lazy functional language which produced information about the step-by-step graph reduction of expressions. In this form, Prospero could only be described as a tool which provided large volumes of potentially useful data.

All of the systems which currently provide users with information about the evaluation of expressions in terms of rewriting information, concentrate on providing information at a single level of abstraction. By providing a filtering mechanism, Prospero goes beyond this limited approach and allows the user to choose the level of abstraction at which they wish to work, and to define new levels. The filter system described here, transforms Prospero into a system which provides the user with a mechanism to control the information that is presented to them, and the way in which this information is presented.

In their simplest form, filters can be seen as a mechanism with which to remove low-level information about an evaluation. While this information is necessary in order to allow the presentation of low-level mechanisms such as pattern-matching, it is often preferable to have this information removed from the presentation of evaluations. I have shown that filters go beyond the removal of low-level information, and can be used to remove evaluation detail at a ‘higher’ level. For example, all of detail about a specific function, or about the application of a function to a set of
specific values can be removed. I have also shown that an approach is possible, whereby a selection of the low-level information is available (for example the presentation of pattern matching information only when pattern matching fails).

Because filters are simply functions from ARG to ARG, the possible uses of functions is endless and would be impossible to fully explore in this thesis. Instead, I have provided a description of the way filters are described, and a number of examples of these functions in use.

Filters have been shown to support a number of every-day techniques used when writing function definitions, and when tracking down errors in these definitions. Both data and functional abstraction can be mirrored in the presentation of an evaluation, and filters are capable of providing simplifications and abstractions of data structures.

Filters are described using Miranda and are ‘compiled’ into the Prospero system at run time. While this is sufficient at this experimental stage, the approach has a number of problems. These problems and possible solutions are described in Chapter 9.
Moving Through an Evaluation

Prospero presents evaluations in terms of the steps taken to move from the expression to be evaluated, to that expression’s fully evaluated form (if such a form exists). This collection of evaluation stages is referred to as the evaluation history, and it is the facilities provided to explore this history that are presented in this chapter.

Prospero’s single stepping mechanism is presented, together with an investigation of the meaning of ‘step’ in this context. Following on from this, a facility which would allow users to search an evaluation history for a specific stage is presented.

The combination of search facilities and lazy evaluation is not a simple one to make. A naive implementation of a search system would allow the user to unknowingly introduce strictness into the evaluation. This chapter will investigate whether a search system which guarantees to introduce no strictness is sufficient, or whether there are cases where it is preferable to allow searches to introduce a limited amount of strictness, in exchange for the provision of more information about the result of a search.

6.1 Generating the Evaluation History

The evaluation history contains all of the stages involved in the evaluation of an expression. In conventional lazy functional programming systems the evaluation of expressions is driven by the system’s printing mechanism. When a user requests the evaluation of an expression, the printing mechanism calls the system’s evaluator with the expression.

This model is considerably different to the way an evaluation is driven in Prospero, where the evaluator is not driven by a printing mechanism, instead it is driven by a history-making mechanism. Rather than producing a stream of characters, the evaluator produces a list of evaluation stages representing the evaluation of an expression.

In Prospero the user’s demand to single step to the next point in the evaluation history,
or to search forward through the evaluation history, results in the history-making mechanism calling the evaluator with the current expression and appending the resulting expression to the evaluation history. This process is repeated until the user does not require to see any more of the history, or until the expression is in a fully reduced form, i.e. the form which would normally be returned by a conventional evaluation system.

While it would have been possible to implement Prospero in a way such that the complete evaluation history was generated before the user interacted with the system, there are a number of reasons for avoiding this strategy. Because Prospero makes no attempt to optimise the evaluation of expressions, the generation of an evaluation stage is a expensive process relative to the evaluation of expressions by the Miranda system, thus, the generation of the complete evaluation history would take a substantial amount of time for anything but the simplest evaluations. Furthermore, it may often be the case that the user need only study the first section of an evaluation history, in order to understand how the evaluation is progressing, thus it was felt a better approach, to generate the evaluation history on demand. Finally, it is frequently the case that errors in a user's definitions lead to the non-termination of evaluations, and thus an infinitely long evaluation history. This problem could be overcome by allowing the user to interrupt an evaluation, and to inspect the evaluation history up to the point where the evaluation was interrupted.

6.2 Single Stepping

A basic facility provided by systems which present the execution of a program is the single step. The granularity of an execution step is dependent on the language in question, but is usually defined in terms of the program code being displayed to the user. For example, if the user is being shown the program in terms of machine-code, then a step is defined to be the execution of a single machine-code instruction. Similarly, if the user is being shown the execution in terms of Pascal code, then it makes no sense for a step to be defined as a machine-code instruction, as this would mean that many steps would have to occur before a Pascal instruction was executed. Instead, the execution is shown in terms of Pascal instructions, each of which may represent a number of machine code instructions.

How then is the granularity of a Prospero step to be defined? In (Bird & Wadler '88 p5), it is shown that the evaluation of expressions written in a lazy functional language can be represented using "a basically simple process of substitution and simplification, using both primitive rules and rules supplied by the programmer". The evaluation of an application of square (Figure 6.1) illustrates this point.
(a)
\textit{square} \ n = n \times n

(b)
\begin{align*}
\textit{square} \ (3+4) & \Rightarrow (3+4) \times (3+4) \\
& \Rightarrow 7 \times (3+4) \\
& \Rightarrow 7 \times 7 \\
& \Rightarrow 49
\end{align*}

Figure 6.1 (a) Definition of the function \textit{square}; (b) evaluation of an application of \textit{square}

Each line in the evaluation can be thought of as a single evaluation step. The granularity here is a string rewrite, where terms are rewritten using: the right-hand-sides of the user's definitions; the right-hand-sides of Miranda's standard functions; and by applications of Miranda's built-in operators.

In this example all of the information presented is at the same level of abstraction as the user's definitions. There are no steps which represent the pattern matching and instantiation which would take place if this expression were to be evaluated using Prospero. It is this removal of detail which gives the clue to the definition of a step.

Just as a Pascal step is a number of machine code instructions, a step can be thought of as a number of atomic steps of the graph reducer at the heart of Prospero (described in Chapter 7). The number of graph reduction steps that go to make a step is equal to the minimum number of reductions which need to be made before a change would be seen in the display of a filtered expression. The definition of a step is dependent on the current filter being applied to an evaluation.

Given this definition we can see that the response produced by Prospero when requested to carry out a single step is to perform the minimum number of reduction steps required in order for a change to occur in the user interface. All of the evaluation stages created by this evaluation are added to the evaluation history. Thus it is the user's requests to be shown the next step in the evaluation that is driving the evaluation of expressions.

\section*{6.3 Searching an Evaluation History}

While stepping allows the user to move freely through the evaluation history, it is a poor substitute for more advanced facilities which allow users to specify stages in an evaluation in which they have an interest. Without this facility, users are forced to
single-step through possibly lengthy evaluation histories searching for a particular evaluation stage. At best, this is a process which takes time. At worst, users may miss the fact that they have reached the point of interest, either through inattentiveness or through a misunderstanding of the information being presented to them: it is quite possible that the misconceptions that caused an error in the first place may be mirrored in their failure to correctly interpret the facts being presented to them by Prospero.

A second use of a search facility is outlined in Chapters 4, where I demonstrate that assertions can be simulated using a search system.

Just as stepping through an evaluation has its analogy in systems for imperative languages (i.e. single stepping), so does the idea of searching an evaluation history. Imperative programmers are offered the breakpoint with which to search for pre-defined machine states during an execution, and so avoid having to step through the execution to reach these points. The different kinds of breakpoint commonly available are described below.

6.3.1 Unconditional Breakpoints

Unconditional breakpoints are a common facility in most imperative debugging systems. In systems such as Unix’s dbx, the user can select a line of source code and insert a breakpoint which will stop the execution if that piece of code is executed. In the Smalltalk programming system (Goldberg & Robson ’83), the user may explicitly place a breakpoint (in the form of a self halt message) into the code of an object, the effect of which is to activate the Smalltalk debugger.

The explicit representation of an execution sequence would seem to be a prerequisite for having a breakpoint facility. Inserting a breakpoint is literally marking a point in the sequence of execution, explicitly defined by the program code. There is no explicit execution sequence in a functional language. How then is it possible to have a breakpoint? To answer this question it is necessary to think of breakpoints not as breaks in a sequence of instructions, but as conditions which refer to a specific instruction without referring to the instruction sequence; for example, “if this piece of code is executed then halt the execution”.

Now it is possible for breakpoints to fit into a functional programming system where they can exist in the form: “if this sub-expression is evaluated then halt the evaluation”. In fact, the error function provided by Miranda can be thought of in exactly this way.
6.3.2 Conditional Breakpoints

An extension of unconditional breakpoints is to allow the user to associate a condition with a breakpoint. In imperative languages these conditions usually take the form of a boolean function which examines the value of a piece of state. When the execution sequence reaches a conditional breakpoint, the boolean function is applied to the piece of program state and the execution continues or is halted depending on the result.

6.3.3 Post-Conditional Breakpoints

In imperative systems, post-conditional breakpoints inspect the machine state following the execution of a piece of code, rather than the machine state immediately prior to it. A standard breakpoint which tests the relevant piece of state, is placed following the piece of code of interest.

6.4 Prospero’s Searches

In this section I will describe Prospero’s search facilities, which would allow users to search through evaluation histories for stages in an evaluation with specific properties. The search mechanism described here sets out to achieve the goals described below.

Note that none of the search facilities described here have yet been implemented. The reader is referred to Appendix B for a full description of the status of the implementation of Prospero at the time of writing.

6.4.1 The Goals of the Search System.

Before describing a proposed search system, I will lay out a number of goals which can be set out for a search system. At the end of this chapter I will show that, by providing a number of different search techniques, it is possible to produce a search system which satisfies all of these goals.

In Chapter 3 I showed that using Miranda’s error function as a primitive type of search, had a number of flaws. One of these was that the evaluation of the error function marked the end of an evaluation (i.e. a search expressed in this way could only succeed once). This makes it impossible for the user to express searches such as “show me all of the points at which \( f \) is applied to \( 10 \)”, since the first time this search is satisfied the evaluation is terminated. One of the goals of the Prospero search system is that a successful search will not terminate an evaluation, it will allow the user to continue searching or to continue interacting with the system in other ways.
Chapter 6 - Moving Through an Evaluation

Because a search will result in a 'jump' to the next point in the evaluation history where
the search condition is satisfied, it is important to provide the user with context
regarding the evaluation step which they have reached. Of particular relevance is
information about the source of values involved in the search itself. For example,
rather than simply being told that $f$ is applied to 10 at a particular evaluation stage,
information about the point in the evaluation history where the application took place
should also be provided. One effect of lazy evaluation is that this information may not
be present in the evaluation stage where the search succeeded, instead the application
may have appeared in the expression a number of stages prior to the current one.

The search system must provide searches which guarantee not to introduce strictness of
any kind into the evaluation. While this need not be the only kind of search provided, it
is necessary in order for users to be sure that they are witnessing the real strictness
properties of their expressions.

In Chapter 4 I showed that searches could be used to check assertions about a set of
definitions. It is vital that a search system avoids changing the strictness properties of
an evaluation in order to avoid possibly changing the truth of an assertion. Failure to
avoid this would lead to problems similar to those illustrated in Section 3.2.1.

If the search system does introduce any strictness into an evaluation, this strictness
must only be termination strictness. That is to say, an evaluation history which is
generated by the search system should use the same number of graph nodes and the
same number of reduction steps as a reduction history generated without the use of a
search.

Some searches may be usefully expressed in a 'post-conditional' way. For example
rather than a user searching for all applications of $f$ to an argument, the user may wish
to search for applications which result in an expression with specific properties (e.g.
search for an application of $f$ which evaluates to 10). A search system should facilitate
the writing of such searches.

6.4.2 What is a search?

The search mechanism of Prospero would allow users to search an evaluation history
for a stage with a specific property. The result of any search would be that the history
of the evaluation of an expression is generated up to the point where, either the search
is satisfied, or the expression is fully reduced.
Searches would be divided into three different classes. The main difference between these classes being the strictness that they introduce into an evaluation. This strictness is introduced in order to determine whether a search condition holds or not. None of these searches satisfy all of the goals set out above. The reason for presenting the different types of search here is to explore how different approaches can be used in order to satisfy different goals.

All searches would have two components, a start expression, and a search condition. I will introduce these components using a simple example of a search which will be used to determine whether a function $\xi$ is ever applied to an expression whose value is 42.

The start expression describes an expression which must appear in the evaluation. In the example above, the search condition should be tested when $\xi$ is applied to any expression, so the start expression is any application of $\xi$. In order to be able to describe such an expression, start expressions may contain wild cards. In this example, the start expression would use a wild card to match any application of $\xi$.

Once a start expression has appeared in an evaluation stage, the search is said to be ‘active’. The activity of the search depends on the search condition, and the kind of search that we have defined.

The search condition is a boolean expression, the value of which determines whether the search should stop the evaluation or not. The search condition may directly reference parts of the expression which matched the start expression. In the example, the argument to $\xi$ would be referenced and compared to 42.

In the following sections, I provide a detailed account of the way that searches would be described by the user and how these descriptions would be handled by Prospero. I will provide a worked example involving the searching of an evaluation using all three classes of search, and I will examine how the search system satisfies the goals set out in Section 6.3.1.

6.4.3 The Three Classes of Search

I will introduce the three classes of search, by applying the search outlined above to the evaluation of the expression shown in Figure 6.2 (a). Part of the evaluation of this expression is given in Figure 6.2 (b).
Figure 6.2 (a) function definitions used to illustrate searches; (b) part of an evaluation involving these functions, with underlining on the third line showing the start expression of an Eager search, and underlining on the sixth line showing the point at which a Patient search's start expression becomes the redex.

The Eager search

The Eager search checks the search condition as soon as possible (i.e. as soon as the start expression appears in the evaluation history). Once the start expression has appeared, the search is said to be ‘active’ and any evaluation necessary in order to test the value of the search condition is carried out immediately. So in the example above, once the start expression (ε applied to any expression) has appeared (shown underlined) in the third step, the Eager search evaluates the argument to ε in order to check the search condition. This evaluation allows the search condition to be shown to be True, and the evaluation will stop.

Any evaluation which is carried out by the Eager search is thrown away once it has been established whether the search condition holds or not. This means that if the search condition holds, the user is not shown the value of the argument of ε in its evaluated form, and if the condition does not hold, the evaluation will continue as if the extra evaluation had not been carried out.
Chapter 6 - Moving Through an Evaluation

The Lazy search

Whereas the Eager search carries out extra evaluation in order to determine the value of the search condition, the Lazy search carries out no extra evaluation at all. Instead once the start expression has appeared, and the search has become active, the Lazy search tracks the expressions referenced by the search condition until such time as enough evaluation has been carried out to determine the value of the search condition. In the example given above the start expression (the application of \( \varepsilon \)) appears in the second stage of the evaluation. At each evaluation step after this point the expression which was the argument to \( \varepsilon \) is examined to see if it has been evaluated to a stage where the search condition can be checked. This is not the case until the seventh step of the example, at which point the comparison can be carried out and the search condition can be shown to have the value True.

The Patient search

The Patient search lies between the Lazy and the Eager search in terms of the amount of evaluation carried out in order to check the search condition. A patient search will carry out evaluation in the same way as the Eager search, but this evaluation may only take place once the expression involved has become the next expression to be evaluated (know as the redex) in the current stage in the evaluation.

In the example, the start expression appears in the second stage of the evaluation, but does not become the redex until the sixth step (where it is shown underlined). At this point the Patient search may evaluate the argument to \( \varepsilon \) and the search can be shown to hold. In the same way as the Eager search any evaluation carried out by the search is thrown away.

6.4.4 How Searches are Described

In this section I will suggest how Prospero’s searches might be described by a user. While the definitions here resemble Miranda syntax, they should not be seen as Miranda expressions. Unlike the filters described in the previous chapter, searches are not Miranda definitions which are incorporated into Prospero at run-time. Searches are definitions which would have to be parsed by the search system. At present no such parser exists and therefore, the definitions given below should be viewed as illustrations of the possible way in which searches could be described rather than concrete examples of search definitions.
All three classes of searches have the same type:

\[
\text{search} ::= (\text{start_expression}, \text{search_condition})
\]

where the elements of this pair have the types:

\[
\begin{align*}
\text{start_expression} & ::= \text{expression} \\
\text{search_condition} & ::= (\text{expression}, \text{bool})
\end{align*}
\]

The \textit{start expression} is an expression which may contain a number of constants and free variables. The free variables are identified using the value constructor \texttt{Wild} which takes a string as its argument. Values constructed in this way are viewed as wild-cards by the search mechanism, i.e. they can map onto any expression. The argument to the \texttt{Wild} value constructor enables multiple wild-cards to be used in a single \textit{start expression}. Examples of \textit{start expressions} are:

\[
\begin{align*}
&f \; \texttt{(Wild 'x')} \rightarrow \text{the application of } f \text{ to any argument} \\
&3 \rightarrow \text{the constant } 3 \\
&[] \rightarrow \text{the empty list} \\
&\texttt{hd} \; [] \rightarrow \text{the application of } \texttt{hd} \text{ to the empty list} \\
&(\texttt{Wild 'x'}) \; [] \rightarrow \text{the application of any function to the empty list} \\
&(4, \; \texttt{(Wild 'x')}) \rightarrow \text{a pair whose first element is } 4
\end{align*}
\]

The \textit{search condition} is a pair. The first element of the pair, the \textit{search expression} must, through the use of wild-cards if necessary, map on to the expression which matched the \textit{start expression}. The second element of the pair is a boolean expression which may contain references to any free variables occurring in the search expression.

Examples of a \textit{search conditions} are:

\[
\begin{align*}
&(i) \quad (f \; \texttt{(Wild 'x')'), \; (\texttt{Wild 'x'}) = 42) \\
&(ii) \quad (\texttt{(Wild 'e'), \; (\texttt{Wild 'e'})} \; = \; [] \} \\
&(iii) \quad (((\texttt{Wild 'a'), \; (\texttt{Wild 'b')})), \; (\texttt{Wild 'a'}) + (\texttt{Wild 'b'}) \; = \; 10) \\
&(iv) \quad ((\texttt{Wild 'tree'}), \; \texttt{balanced} \; (\texttt{Wild 'tree'}) \\
&\text{where } \texttt{balanced} \; = \; ....)
\end{align*}
\]

The first \textit{search condition} is the example used earlier in this chapter. The search expression maps on to any application of \texttt{f} and the \textit{condition} will be \texttt{True} if the argument has the value \texttt{42}.

The second \textit{search condition} has a search expression which maps on to any expression, and a \textit{condition} which will be \texttt{True} if the expression is the empty list. In order to understand the usefulness of this example it is important to understand that the \textit{search expression} and the \textit{start expression} are not necessarily the same expression,
they only need to map onto each other using wild-cards. Using search and start expressions in this way allows the user to define searches in terms of the result of an application. For example the following search:

```lisp
{
    (g (Wild 'x')),  || start expression
    ((Wild 'e'), (Wild 'e') = [])  || search expression
}
```

will check every application of the function \( g \) to see whether the application evaluates to the empty list.

The third search above demonstrates that search conditions do not have to be simple equality tests. Here the elements of a pair are summed and then compared to 10.

The fourth example shows how a search condition can make use of predicates which are defined locally using a where clause.

### 6.4.5 The Search Mechanism

Evaluation of expressions in Prospero is driven by the user's requests to see parts of the evaluation history, rather than being driven by the printing mechanism of the programming system as described in (Peyton Jones '87)\(^1\). When users instigate a search, they are asking to see the part of the evaluation that satisfies the conditions laid out by that search. Two things can happen after a search has started, either the search succeeds, in which case the evaluation stage at that point is displayed to the user, or the search fails, in which case the final reduction stage is displayed.

In this section I shall describe how the evaluation is driven and what happens at each evaluation stage, for each of the search classes.

\(^1\)Although it could be said that the printing mechanism is simply responding to the users request to see the final stage in the evaluation history of an expression.
Chapter 6 - Moving Through an Evaluation

The Eager Search

The outline of a definition of the Eager search mechanism is given in Figure 6.3. The function `eager_search` when applied to a start expression, search condition and evaluation history, returns an evaluation history. The first clause of the function definition states that the evaluation history which `eager_search` was applied to will be returned, if the expression at the head of the history is in normal form, or if the search has succeeded.

A search is said to have succeeded if the start expression is found in the expression at the head of the history, and if checking the search condition returns `True`.

The checking of the search condition is carried out by the function `check`. This function uses two auxiliary functions: `cond_apply` and `single_step`. The function `cond_apply` when applied to a search condition and an expression returns a `Bool`. If it is possible to test the value of the condition then `check` returns `True` or `False` depending on the result of the test. If more evaluation is required before the test can be carried out then `cond_apply` returns `Unknown`. Using these functions `check` repeatedly single-steps the expression until it is possible to check the condition.

```plaintext
Bool' ::= True' | False' | Unknown

eager_search start_e search_c (exp:hist)
  = exp : hist , if normal_form exp \ search_success
  = eager_search start_e search_c (exp' : exp : hist)
    , otherwise
  where
    exp' = single_step exp
    search_success
      = contains start_e exp & check search_c exp
    check exp cond
      = True , cond_apply cond exp = True'
      = False , cond_apply cond exp = False'
      = check (single_step exp) cond , otherwise
```

Figure 6.3 Definition of the eager search mechanism

The second clause of `eager_search` is a recursive call with the evaluation history extended by one step.
Chapter 6 - Moving Through an Evaluation

The Lazy Search

The outline of a definition of the Lazy search mechanism is given in Figure 6.4. The function lazy_search when applied to a start condition, search condition, evaluation history and condition table, returns a new evaluation history.

```haskell
lazy_search start_e search_c (exp:hist) c_tab
    = exp : hist , if normal_form exp \/
        foldr False (\/
            (map (test_cond exp) c_tab)
        )
    = lazy_search start_e search_c (exp' : exp : hist) c_tab',
        otherwise

where

    exp' = single_step exp
    c_tab' = insert (bind search_c exp) c_tab
        , if contains start_e exp
    = c_tab , otherwise
```

Figure 6.4 Definition of the lazy search mechanism

The first obvious difference between this search and the Eager search is the use of a condition table. This table holds instances of the search condition for which the start expression has appeared in the expression being evaluated, but the expression has not been sufficiently evaluated for the search condition to be tested.

The first clause of lazy_search states that the evaluation history is returned if either the expression at the head of the history is in normal form, or if any of the conditions held in condition table hold.

The function test_cond applies the function cond_apply, described above, to the condition and the current expression, and returns True only if the condition can be tested and the result of the test is True'. If the condition can not be tested, or if the test is carried out and the result is False', test_cond returns False.'
If the head of the evaluation history is not in normal form, and if none of the checks of the conditions in the condition table return True, then a recursive call of lazy_search is made, with the evaluation history extended by a single step and with a new condition table containing a new entry if the start expression has appeared in the expression at the head of the evaluation history\(^1\). The new entry contains the search condition with any wild cards bound to their matching expressions.

\[
\text{Bool' = True' | False' | Unknown}
\]

\[
\text{patient_search start_e search_c (exp:hist) c_tab}
= \begin{cases} 
\text{exp : hist}, & \text{if normal_form exp \hspace{1em} \cap \hspace{1em} search_success} \\
\text{patient_search start_e search_c (exp' : exp : hist) c_tab'}, & \text{otherwise}
\end{cases}
\]

where
\[
\begin{align*}
\text{exp'} &= \text{single_step exp} \\
\text{search_success} &= \text{foldr False (\hspace{1em} \cap \hspace{1em})} \\
&\quad \text{map (check exp)} \\
&\quad \text{(filter {exp_compare (redex exp)})} \\
&\quad \text{c_tab}
\end{align*}
\]

\[
\text{c_tab'} = \text{insert (bind search_c exp) c_tab}, & \text{if contains start_e exp} \\
&\quad = \text{c_tab}, \text{otherwise}
\]

\[
\text{check exp cond}
= \begin{cases} 
\text{True}, & \text{cond_apply cond exp = True'} \\
\text{False}, & \text{cond_apply cond exp = False'} \\
\text{check (single_step exp) cond}, & \text{otherwise}
\end{cases}
\]

Figure 6.5 Definition of the patient search mechanism

**Patient Search**

The outline of a definition of the Patient search mechanism is given in Figure 6.5. The function `patient_search` when applied to a start condition, search condition, evaluation history and condition tables, returns a new evaluation history.

\(^1\) Note that in order to keep the outline of the patient and lazy search mechanism simple, garbage collection of the condition table is not shown in the definition.
The first clause of the function `patient_search` returns the original evaluation history if either the expression at the head of the history is in normal form, or if a search succeeds.

In order to check whether a search has succeeded, the condition table has all entries whose `search conditions` do not involve the current redex, removed from it. The function `check`, described above, is then mapped on to the remaining table. If this map results in any of the `search conditions` holding, then the search is said to have succeeded.

If the head of the evaluation history is not in normal form, and if none of the checks of the conditions in the condition table return `True`, then a recursive call of `lazy_search` is made, with the evaluation history extended by a single step, and if the start expression has appeared in the expression at the head of the evaluation history, with a new condition table containing a new entry.

### 6.4.6 Worked Example

Following on from the description of the three classes of search given above, I will present a simple worked example which illustrates the way in which these different classes search through an evaluation.

The example uses a search which is defined to succeed when the function `hd` is applied to an empty list. The search would be defined as follows:

```
  start : hd (Wild 'x')
  search : (hd (Wild 'x')), (Wild 'x') = []
```

That is to say, the search will become active every time `hd` is applied to an expression, and the search will succeed if the argument to `hd` is `[]`.

The equality operator used here can be thought of as equivalent to the equality used in pattern matching. So, in this example, the equality operator need only evaluate `x` to constructor form.

The search will be applied to the evaluation of the expression:

```
  hd (filter even [2,4,6]) + hd (filter odd [2,4,6])
```

### The Eager Search

The first example given in Figure 6.6 shows the use of an Eager search.
The first stage in the evaluation contains two applications of \texttt{hd}. Both of these applications match the \textit{start expression} and two \textit{search conditions} are created. Each \textit{search condition} has the single free variable \( x \) which is bound to the argument of \texttt{hd}.

Any evaluation necessary in order to test the \textit{search conditions} is now carried out. The first condition to be tested, the condition on the left of the figure, evaluates to \texttt{False} after two reductions, when the argument to \texttt{hd} becomes an application of \texttt{cons} and so can not be the empty list. The search mechanism now turns to the second condition and carries out the evaluation necessary in order to test it. This results in the complete evaluation of the application of \texttt{filter} showing the argument to \texttt{hd} to be \([\,]\). At this point a \textit{search condition} has become \texttt{True} and the search succeeds, the extra evaluation carried out is thrown away, and an evaluation history containing only the first evaluation stage is returned.

\textit{The Lazy Search}

The second example given in Figure 6.7 shows the use of a Lazy search. Once again, both of the applications of \texttt{hd} match the search expression and so two start conditions are created with different bindings for the variable \( x \).

The Lazy search mechanism now carries out no evaluation, instead it monitors the arguments of \texttt{hd}. As the evaluation progresses the first (leftmost) binding of \( x \) becomes a \texttt{cons} cell and so can be seen not to be the empty list. It is not until the seventh evaluation stage that the second binding of \( x \) becomes the empty list, at this point a \textit{search condition} has become \texttt{True}, the search succeeds and the evaluation history up to this stage is returned to the user.
The Patient Search

The final example given in Figure 6.8 shows the use of a Patient search. Again, both of the applications of `hd` match the search expression and so two start conditions are created with different bindings for the variable $x$. 
In the figure rounded shadow boxes are used to mark the redex at each step in the evaluation. The Patient search mechanism will not carry out any evaluation necessary in order to check a search condition, until the expression involved in the search has become the redex.

In the first evaluation step, the expression bound to $x$ is the redex, and so this expression can be evaluated. The evaluation need only proceed until the expression becomes an application of cons before the condition can be shown to be False. At the third stage in the evaluation the second search condition can be tested and shown to be True. At this point the search succeeds and the evaluation history up to this stage is returned to the user.
6.4.7 Satisfying the Goals of the Search System

In this section I will examine whether the searches described here satisfy the goals set out in Section 6.3.1.

Continuation after a Search

A successful search returns the evaluation history up to the point where the search succeeded. The user may continue with an evaluation from that point, or may continue the search for the next point in the history where the search will succeed. This is irrespective of the class of search that the user has used.

Source of the Expression of Interest

The different classes of search allow the user to find the source of an expression in different ways. An eager search which succeeds will do so as soon as an expression which satisfies the search condition appears anywhere in the reduction graph, thus giving a clear indication of the source of the expression.

The patient search will not stop a search until the expression in question has become the redex. However, it may be that by the time an expression has become the redex, the context in which it first appeared is no longer clear. The information provided by the eager search may have been lost at the expense of not stopping a search prematurely.

The lazy search would appear to be the least beneficial in terms of finding the source of an expression. This type of search will spot an expression later than any of the other types of search.

Termination Strictness Only

Any reductions that are carried out by a search are thrown away after it has been established whether a search condition has or has not been satisfied; this means that work done by the search mechanism may have to be repeated by the graph reducer, but ensures that a search mechanism will not effect the space (i.e. number of graph nodes used at each stage of an evaluation) and time (i.e. number of reduction steps necessary to complete an evaluation) characteristics of any evaluation.

Less Introduction of Strictness than error

It is clear that eager searches introduce more strictness (in terms of termination) than the equivalent use of the error function. It is impossible for the error value to be returned as the result of an evaluation, unless the expression which evaluated to that
error value had been the redex of the expression. This is not the case for eager searches, which may find an expression which would never have become the redex.

Patient searches, because they wait for an expression to become the redex, introduce an equivalent amount of strictness to that introduced by error. Once an expression has become the redex, any subsequent evaluation carried out by the search mechanism, is equivalent to the reduction that would be necessary to evaluate any patterns or guards used to incorporate the error function into a definition.

Lazy searches introduce no strictness at all to an evaluation. Using these searches an evaluation that would normally terminate without any searches in place, is guaranteed to terminate.

Dealing with the Results of Evaluations

By using start and search expressions in the way demonstrated in Section 6.3.4 it is possible to design searches in terms of the result of evaluations. For example it is possible to check that all applications of a tree balancing function return balanced trees by setting the start condition of a search to be the application of the balancing function, and setting the search condition to be a tree which is not balanced. All of the classes of search are suitable for this type of search.

No Change of Truth of an Assertion

In Section 3.3.4 an example was given of the use of error to make an assertion about a set of definitions changing the truth of that assertion. This change was due to strictness added by the insertion of the error function into the definitions in question. Because lazy searches do not add any strictness to an evaluation, they cannot change the truth of any assertion.

6.5 Summary

In this chapter a number of facilities which allow the user to move through an evaluation history have been introduced.

Single stepping is provided to the user, where a step is defined as the smallest number of graph reductions required to change the expression being presented to the user.

A search system has been described which presents the user with three classes of searches which differ with respect to the amount of strictness they may introduce into an evaluation. These searches satisfy all of the goals set out earlier in this chapter. In many cases the choice of search class involves a trade off between the possible addition
of strictness into an evaluation, and the possibility that a large number of evaluation steps will take place between the time when the start expression appears in an evaluation, and the time when the search condition can be shown to be true.
7 Evaluating Expressions

In the introduction of this thesis, I stated that a system aimed at providing information about evaluations, must provide a set of views to the user and a model upon which to base these views. Early chapters also showed that an approach based on rewriting provided a wide range of views, suitable for users faced with errors in a set of definitions, and for a number of teaching requirements. I have also shown that in order to get the most out of a rewriting approach, low-level information must be available.

In this chapter I will describe the evaluation model at the heart of Prospero, showing how certain aspects of the evaluation were designed with provision of information about an evaluation in mind.

The main requirements of the evaluation model described here were:

- the faithful reproduction of the results of all evaluations as if they had been evaluated by the Miranda system;

- the ability to represent pattern matching;

- the ability to represent guard evaluation;

- the ability to represent partial evaluation of functions;

- the ability to represent higher order functions.

The model described here, is not being proposed as a mechanism for producing meaningful information, rather it is put forward as a mechanism capable of producing data to form a suitable foundation for the production of views. At the same time, the model has been designed to follow as closely as possible the standard top-to-bottom, left-to-right presentation of pattern-matching, classically used in texts (e.g. Bird & Wadler '88) and teaching material. Various other possible models of pattern-matching
are discussed and rejected on the grounds that while they faithfully represent pattern-matching in progress, they do not follow this *classical* presentation scheme.

It is one of the many roles of the filter system (described in Chapter 5) to transform the data produced by the model into informative views. However, the filter system should not be seen as an artifact of the choice of model. The filter system is a flexible tool with which to control the presentation of rewriting information. While filters are necessary in order to control the volume of data produced by the evaluation model, the filter system would have been equally useful had a higher-level evaluation model been selected.

### 7.1 Driving a Reduction

In the previous chapter I showed that the generation of the evaluation history is a result of the users demands to be shown more of that history. The evaluation history creation process may be represented using the following code:

```haskell
make_history E
    = E : make_history E' , if -printable E
    = [E] , otherwise
    where
        E' = single_step E
```

Each call to `single_step` carries out an atomic reduction step on the current stage of the evaluation history. It is the description of this function which is at the heart of this chapter.

#### 7.1.1 Variables

If an expression is a variable bound to a value, then the variable is the redex, and the result of reducing it, is to replace it with its binding, giving us the following definition of `single_step`.

```haskell
single_step ([v] env = lookup env v

where
    v is a variable
    lookup env v returns the binding of v in the environment env
```
7.1.2 Constants

Constants such as \((\text{True}, 1, 'a', "a string")\) are in a form which require no more reduction in order for them to be printed, therefore no definitions of \text{single\_step} exists for constants.

7.1.3 Unary Operators

The treatment of the application of one of Miranda’s in-built unary operators to its argument depends on whether the operator is strict or not. If the operator is strict then the argument must be in weak head normal form before the application can take place. If the operator is not strict then the application is the redex and is reduced. The rule for \text{single\_step} is as follows:

\[
\text{single\_step} \ [u \ e] \ \text{env} \\
= \ \text{eval} \ [u \ e] , \ (\neg \text{strict} \ u) \ \lor \ \text{whnf} \ e \\
= \ [u \ \text{(single\_step} \ e)] \ , \ \text{otherwise}
\]

where
\(u\) is a unary operator
\(e\) is an expression
(\text{strict} \ u) evaluates to \text{True} if \(u\) is strict in its argument
(\text{whnf} \ e) evaluates to \text{True} if \(e\) is in weak head normal form

7.1.4 Binary Operators

Like unary operators, the treatment of Miranda’s in-built binary operators depends on their strictness qualities. Evaluation of the arguments of the operator will be necessary if the arguments are not fully reduced and the operator is strict in that argument.

\[
\text{single\_step} \ [e_1 \ \text{op} \ e_2] \ \text{env} \\
= \ \text{eval} \ [e_1 \ \text{op} \ e_2] \\
, \ (\neg \text{strict\_in\_1 op}) \ & \ (\neg \text{strict\_in\_2 op}) \ \lor \\
(\text{whnf} \ e_1) \ & \ (\text{whnf} \ e_2) \ \lor \\
(\text{strict\_in\_1 op}) \ & \ (\neg \text{strict\_in\_2 op}) \ & \ (\text{whnf} \ e_1) \ \lor \\
(\text{strict\_in\_2 op}) \ & \ (\neg \text{strict\_in\_1 op}) \ & \ (\text{whnf} \ e_2)) \\
= \ [(\text{single\_step} \ e_1) \ \text{op} \ e_2] \\
, \ (\text{strict\_in\_1 op}) \ & \ (\neg \text{whnf} \ e_1) \\
= \ [e_1 \ \text{op} \ \text{(single\_step} \ e_2)] \\
, \ (\text{strict\_in\_2 op}) \ & \ (\neg \text{whnf} \ e_2)
\]

where
\(\text{op}\) is a binary operator
\(e_1, e_2\) are expressions
(\text{strict\_in\_1 op}) evaluates to \text{True} if the operator is strict in first argument
(\text{strict\_in\_2 op}) evaluates to \text{True} if the operator is strict in second argument
(\text{whnf} \ e) evaluates to \text{True} if \(e\) is in weak head normal form
Chapter 7 - Evaluating Expressions

7.1.5 Function Application

At first sight the rule for function application may appear counterintuitive, this is because there is no definition of single_step which deals with applications in the form \( f \ a \ b \ c \) where \( f \) is a function and \( a \ b \ c \) are its arguments. The explanation for the absence of this rule lies in the definition given in Section 7.1.1 which deals with variables, and the definitions below, which deal with the pattern matching mechanism.

\[
\text{single_step } \left[ f \ a_1...a_n \right] \\
= \left[ (\text{single_step } f) \ a_1...a_n \right]
\]

where

\( f \) is a function name

\( a_1...a_n \) are expressions

To deal with a function application single_step is applied immediately to the function name. The definition given in 7.1.1 is then used and the definition of the function is 'looked up' in the current environment. The definitions of single_step used to deal with the evaluation of the application of a function body to its arguments are given in later sections.

7.1.6 Constructor Application

Because no printing is carried out by Prospero - the expression which would normally be printed by the Miranda system is shown as a reduction graph - it is necessary to be able to reflect the fact that evaluation of an application of a constructor would have been carried out so that the expression containing the application can be printed. In order for printing to take place, the arguments to a constructor must be reduced to weak head normal form. If this is the case then single_step can mark the application of the constructor as printed, otherwise single_step is used to further evaluate the argument of the constructor. In the following definitions, an expression which has been printed is shown in bold.

\[
\text{single_step } \left[ \text{C e}_1...\text{e}_n \ e_{(n+1)} \right] \text{ env} \\
= \left[ \text{C e}_1...\text{e}_n \ e_{(n+1)} \right] , \text{ wnhf e}_{(n+1)} \\
= \left[ \text{C e}_1...\text{e}_n \ \text{(single_step e}_{(n+1)}) \right] , \text{ otherwise}
\]

where

\( C \) is a constructor

\( e_1...e_{n+1} \) are expressions
7.1.7 Guards

Once all pattern matching has taken place, any guards in a definition must be evaluated, before the other clauses of a function can be discarded. It may be that no guard condition evaluates to True and so the definition will be rewritten as FAIL. This FAIL will then be handled by thinbar (see p202) and the clause of the function will be discarded.

Guards are represented in Prospero using the function COND which takes three arguments.

The translation of guards is carried out in such a way that the definition:

\[
\begin{align*}
  f &= \text{rhs1}, \text{g1} \\
  &= \text{rhs2}, \text{g2} \\
  &= \text{rhs3}, \text{otherwise}
\end{align*}
\]

is translated into the form:

\[
 f = \text{COND g1 rhs1 (COND g2 rhs2 rhs3)}. 
\]

(COND a b c) is equivalent to (IF a THEN b ELSE c).

The rules for single_step associated with COND are given below:

\[
\begin{align*}
\text{single_step} \left[ \text{COND } c \text{ e1 e2 } \right] \text{ env} &= \left[ \text{COND (single_step c env) e1 e2 } \right], \sim\text{whnf c} \\
\text{single_step} \left[ \text{COND True e1 e2 } \right] \text{ env} &= \left[ e1 \right] \\
\text{single_step} \left[ \text{COND False e1 e2 } \right] \text{ env} &= \left[ e2 \right] \\
\end{align*}
\]

where

- \( p_1...p_n \) are variables or constants
- \( c, e_1, e_2 \) are expressions
Pattern matching must be carried out whenever a function is applied to an argument. As pointed out in the introduction to this Chapter, pattern matching is classically described as being carried out in a top-to-bottom, left-to-right order. So in the definition:

\[
\begin{align*}
f 0 1 &= \text{rhs1} \\
f 1 0 &= \text{rhs2}
\end{align*}
\]

the patterns of the first definition will be checked before those of the second (top-to-bottom), and within the definitions the pattern matching necessary for the first argument will be checked before the second (left-to-right). It is vital, from the point of view of communicating what is going on to the user, that any presentation of pattern matching should follow this ordering.

Pattern matching is carried out by Prospero using a pattern-matching lambda, for which I will use the symbol "\(\Lambda\)". The appearance of \(\Lambda\) in an expression signifies that some pattern matching involving a definition is still to be carried out. The \(\Lambda\) symbol is introduced into the expression once for each pattern on the left hand side of a definition. So the first part of the definition of \(f\) given above would appear as:

\[
\Lambda 0. \Lambda 1. \text{rhs1}
\]

Once pattern matching has been carried out, \(\Lambda\) is rewritten to become either \(\lambda\), or \(\nabla\). Where \(\lambda\) is used to signify that pattern matching of a pattern against a variable has been carried out and has succeeded, and \(\nabla\) is used to signify that pattern matching has failed. The use of symbols to explicitly represent the success or failure of each stage in the matching of a pattern, allows the user to see exactly where pattern matching is failing. So pattern matching the above definition against the arguments 0 and 1 would mean that pattern matching would succeed for both patterns and would lead to the definition being rewritten as:

\[
\lambda 0. \lambda 1. \text{rhs1}
\]

Pattern matching of the definition against 0 and 0 would mean that pattern matching would fail for the first argument, at which point pattern matching for this definition would stop, leaving:

\[
\nabla 0. \Lambda 1. \text{rhs1}
\]
Chapter 7 - Evaluating Expressions

The following sections give details first of all of the rules which deal with a \( \Lambda \) in a definition (i.e. how \( \Lambda \) is rewritten to become either a \( \lambda \) or a \( \forall \)). Following this the rules involving \( \lambda \) and \( \forall \) are given. Finally details are given of the way in which functions with multiple clauses are represented and what happens when all of the clauses for a function fail to pattern match.

**Pattern Matching Variables**

Pattern matching with variables is relatively straightforward. Pattern matching of a variable against any argument will always succeed. It is clear that given the definition:

\[
\text{f x = rhs}
\]

pattern matching will immediately succeed; the \( \Lambda \) in the representation of this definition applied to any argument can be rewritten as \( \lambda \), signifying that pattern matching of the variable against the argument has succeeded. That is:

\[
(\Lambda \text{rhs}) \text{arg} \Rightarrow (\lambda \text{rhs}) \text{arg}
\]

The rule for single_step is:

\[
\text{single_step } [[(\lambda \text{v}. \text{b}) \text{a}]] \text{env} = [[(\lambda \text{v}. \text{b}) \text{a}]]
\]

where
- \( v \) is a variable
- \( a, b \) are expressions

**Pattern Matching Constants**

Pattern matching with a constant pattern, requires that the argument is fully evaluated in order to allow an equality test to be made between the pattern and the argument; if they are equal then \( \Lambda \) becomes \( \lambda \) signifying that pattern matching has succeeded; if they are not equal then \( \Lambda \) becomes \( \forall \) signifying that pattern matching has failed.

\[
\text{single_step } [[(\lambda k_1 \cdot \text{b}) \text{k}_2 ]] \text{env} \\
= [[(\lambda k_1 \cdot \text{b}) \text{ (single_step k}_2 \text{) } ]] \text{, \simwhnf k}_2 \\
= [[(\lambda k_1 \cdot \text{b}) \text{ k}_2 ]] \text{, k}_1 = \text{k}_2 \\
= [[(\forall k_1 \cdot \text{b}) \text{ k}_2 ]] \text{, k}_1 \neq \text{k}_2
\]

where
- \( a, b \) are expressions
- \( k, k_1, k_2 \) are constants
Pattern Matching Sum Types

A sum type is a structured type with more than one constructor. For example num_tree defined below has the two constructors $\text{BRANCH}$ and $\text{LEAF}$.

\[
\text{num_tree ::= BRANCH \ num\_tree \ num \ num\_tree} \\
\quad | \ \text{LEAF}
\]

Pattern matching involving a sum type pattern takes a number of steps. First of all the pattern matching mechanism must check that the constructor of the pattern and the argument are the same. So given the definition of $f$:

\[
f (\text{BRANCH lt n rt}) = \text{rhs}
\]

pattern matching would fail for the application:

\[
f \ \text{LEAF}
\]

and would succeed for the application:

\[
f (\text{BRANCH left v right})
\]

In order for this pattern matching to be carried out, the argument of an application must be sufficiently reduced in order for the comparison of constructors to be carried out. If this comparison shows that the constructors are not equal then $\Lambda$ is replaced by $\lor$, if the constructors are equal then pattern matching must be carried out between the arguments of the constructor in the pattern and those in the argument.

So the rewriting for the first example above where pattern matching fails would be:

\[(\Lambda(\text{BRANCH lt n rt}).\text{rhs}) \ \text{LEAF} \Rightarrow (\lor(\text{BRANCH lt n rt}).\text{rhs}) \ \text{LEAF}\]

and the second would be

\[(\Lambda(\text{BRANCH lt n rt}).\text{rhs}) (\text{BRANCH left v right}) \Rightarrow (\Lambda(\text{lt}.\text{An}.\text{Art}.\text{rhs}) \ \text{left v right})\]
The rules for single_step are given below

\[
\text{single_step} \quad \text{[[}\Lambda(S_1 \ e_1\ldots e_n) \ . \ f\ ]\ (S_2 \ g_1\ldots g_n)\ \text{]} \ \text{env}
\]
\[
= \ [\ [\ \text{[\text{(f)}]}\ (S_2 \ g_1\ldots g_n)\ ]\ , \ S_1 \neq \ S_2
\]
\[
= \ [\ [\ \text{[\text{(f)}]}\ g_1\ldots g_n\ ]\ , \ S_1 = \ S_2
\]

\[
\text{single_step} \quad \text{[[}\Lambda(S_1 \ e_1\ldots e_n) \ . \ f\ ]\ h\ ] \ \text{env}
\]
\[
= \ [\ [\ \text{[\text{(f)}]}\ (\text{single_step} \ h)\ ]\ ]
\]

where

\( S_1, S_2 \) are Sum constructors
\( e_1\ldots e_n, g_1\ldots g_n, f \) are expressions

Pattern Matching Product Types

A product type is a structured type with only one constructor. For example num_pair defined below has the single constructor NPAIR.

\[\text{num\_pair} ::= \text{NPAIR} \ \text{num\_num}\]

Miranda implements ‘lazy product matching’ (Peyton Jones ‘87) and so no evaluation of the argument to a definition involving a product type is necessary before pattern matching takes place. Pattern matching can not fail here and so the \( \Lambda \) is again inserted for each field and the argument is expanded.

\[
\text{single_step} \quad \text{[[}\Lambda(P_1 \ e_1\ldots e_n) \ . \ f\ ]\ (P_2 \ g_1\ldots g_n)\ ] \ \text{env}
\]
\[
= \ [\ [\ \text{[\text{(f)}]}\ g_1\ldots g_n\ ]\]
\]

where

\( P_1, P_2 \) are product constructors
\( e_1\ldots e_n, g_1\ldots g_n, f \) are expressions

Pattern Matching Tuples

Tuples can be thought of as product types. For example a triple with type \((\alpha, \beta, \delta)\) could be represented as the product type:

\[\text{TRIPLE} \ \alpha \ \beta \ \delta\]
Chapter 7 - Evaluating Expressions

\[
\text{single_step} \ [(\lambda v \ldots \lambda p_n . e) \ a_1 \ldots a_n] \ \text{env} \\
= [(\lambda p_2 \ldots \lambda p_n . e[a_1/v]) \ a_2 \ldots a_n] \\
\text{single_step} \ [(\lambda k \ldots \lambda p_n . e) \ a_1 \ldots a_n] \ \text{env} \\
= [(\lambda p_2 \ldots \lambda p_n . e) \ a_2 \ldots a_n]
\]

where
\[v\] is a variable
\[k\] is a constant
\[p_1 \ldots p_n\] are variables or constants
\[a_1 \ldots a_n, e\] are expressions

Treating tuples in this way means that the definitions of \text{single_step} is very similar to the definition for product types, it is given below:

\[
\text{single_step} \ [(\Lambda (p_1 \ldots p_n) . f) \ (q_1 \ldots q_n)] \ \text{env} \\
= [(\Lambda p_1 \ldots \Lambda p_n . f) \ q_1 \ldots q_n]
\]

where
\[(p_1 \ldots p_n)\] is an n-tuple
\[(q_1 \ldots q_n)\] is an n-tuple
\[f\] is an expressions

\textit{Pattern Matched Lambda (\lambda)}

By the time an expression has been rewritten to the form \((\lambda a . e) \ b)\) all necessary pattern matching will have been performed and \(a\) will be either a variable or a constant. These two cases are handled by the following rewrite rules:

\textit{Pattern Failed Lambda (V)}

Once a pattern match has failed and \(\Lambda\) has been rewritten to become \(V\), then the entire expression containing the \(V\) can be rewritten to become \text{FAIL}. Note that \(V\) serves no purpose other than to explicitly indicate the place in which pattern matching has failed.

\[
\text{single_step} \ [(\lambda p_1 \ldots \lambda p_m . Vq . \Lambda p_0 \ldots \Lambda p_n . b) \ a_1 \ldots a_{n+1}] \ \text{env} = \text{FAIL}
\]

where
\[p_1 \ldots p_n\] are variables or constants
\[a_1 \ldots a_{n+1}, b\] are expressions

\textit{The ThinBar Operator: Representing Functions with a Number of Definitions}

So far pattern matching has only been discussed in terms of functions with single clauses. It is often the case that functions are written using a set of clauses. In order to represent functions of this form Prospero uses an operator called thinbar and written
Chapter 7 - Evaluating Expressions

"\|". Thinbar is so called to contrast it with the similar operator "\|\" (pronounced fatbar) which is used to similar effect in (Peyton Jones ‘87). A comparison of the two operators is presented in a later section of this chapter.

Thinbar is used to represent a definition of a function as the sum of a number of partial functions. For example, given the definition:

\[ f \ |
\begin{align*}
  1 &= \mathrm{rhs}1 \\
  2 &= \mathrm{rhs}2
\end{align*}
\]

thinbar would be used to represent the definition as follows:

\[ A1.\mathrm{rhs1} \ |
\begin{align*}
  (A2.\mathrm{rhs2} \ |
  \text{ERROR})
\end{align*}
\]

where \text{ERROR} is a value used to represent the failure of all patterns.

When a function is applied to a number of arguments, Thinbar ‘pulls in’ the arguments one by one, until all of the available arguments have been pulled in, or until the number of arguments required by the function have been pulled in.

Following the pulling in of the arguments, pattern matching of the clause which is the first operand of thinbar takes place. If the pattern matching succeeds then the application of thinbar is rewritten to become the first clause. If the pattern matching fails, then the thinbar application is rewritten to become the second operand of thinbar (i.e. the remaining clauses of the function).

This process is repeated until pattern matching succeeds for a clause, or until only the \text{ERROR} clause remains.
The rewrite rules for thinbar are as follows:

\[
\text{single_step} \, \left[ (\lambda p_1 \ldots \lambda p_n, b \ a_1 \ldots a_n) \mid e \ a_1 \ldots a_n \right] \ \text{env} \\
= \left[ (\lambda p_1 \ldots \lambda p_n, b \ a_1 \ldots a_n) \right] \\
\text{single_step} \, \left[ (\lambda p_1 \ldots \lambda p_n, \lambda p_{n+1} \ldots \lambda p_m, b \ a_1 \ldots a_n \mid e \ a_1 \ldots a_n) \ b_1 \ldots b_k \right] \ \text{env} \\
= \left[ (\lambda p_1 \ldots \lambda p_n, \lambda p_{n+1} \ldots \lambda p_m, b \ a_1 \ldots a_n \ b_1 \mid e \ a_1 \ldots a_n \ b_1) \ b_2 \ldots b_k \right] \\
\text{single_step} \, \left[ ((\lambda p_m \ldots \lambda p_n, \lambda p_{n+1} \ldots \lambda p_{n+k}, e) \ a_1 \ldots a_{n+k}) \mid d \right] \ \text{env} \\
= \left[ \text{single_step}((\lambda p_m \ldots \lambda p_n, \lambda p_{n+1} \ldots \lambda p_{n+k}, e) \ a_1 \ldots a_{n+k}) \ \text{env} \mid d \right] \\
\text{single_step} \, \left[ \text{ERROR} \ e_1 \ldots e_n \right] \ \text{env} = \left[ \text{ERROR} \right] \\
\text{single_step} \, \left[ \text{FAIL} \mid e \right] \ \text{env} = \left[ e \right]
\]

where
\begin{align*}
p_1 \ldots p_n & \text{ are patterns} \\
a_1 \ldots a_{n+1}, e_1 \ldots e_{n+1}, e, b & \text{ are expressions}
\end{align*}

7.1.9 Design of the Pattern Matching Mechanism

Most of the definition of `single_step` described above is not novel. The main difference between this evaluation scheme and the evaluation scheme normally adopted for implementations of a lazy functional language is in the area of pattern matching and the representation of functions with a number of clauses.

In this section I will discuss the equivalence of Prospero's evaluation scheme to the more standard scheme described in (Peyton Jones '87).

I also describe the alternative representations of pattern matching that were considered during the design of the mechanism described above. The goal of the design was to produce a mechanism which could be used to explain the success or failure of the pattern matching resulting from the application of any function to an argument. In order for this to be the case the mechanism needed to follow an intuitive left-to-right, top-to-bottom pattern matching style and to produce suitably fine grained reduction steps. The pattern matching also needed to be unoptimised so that no steps were skipped no matter how inefficient this made the mechanism.

Fatbar

To represent function definitions of the form
\[
f \ p_1,1 \ldots p_1,m = R_1 \\
\ldots \\
f \ p_n,1 \ldots p_n,m = R_n
\]
Peyton Jones (Peyton Jones '87 p61) introduces the \( \| \) operator. The translation rules (p68) are reproduced below.

\[
\text{TD} \left[ \begin{array}{l}
  f \ p_1,1 \ldots p_1,m = R_1 \\
  \vdots \\
  f \ p_n,1 \ldots p_n,m = R_n
\end{array} \right]
\]

\[
= f = (\lambda v_1 \ldots \lambda v_m. \ ( (\text{ATE} \| p_1,1 \ldots \text{ATE} \| p_1,m \). \text{TR} \| R_1 \}) \\
\quad v_1 \ldots v_m)
\]

\[
\quad \vdots \\
\quad ( (\text{ATE} \| p_n,1 \ldots \text{ATE} \| p_n,m \). \text{TR} \| R_n \}) \\
\quad v_1 \ldots v_m)
\]

\[
\| \text{ERROR})
\]

where
- \( f \) is a variable
- \( v_i \) is a variable not free in any \( R_j \)
- \( p_{i,j} \) is a pattern
- \( R \) is a right-hand side
- \( R_i \) is a right hand side
- \( a \| b = a \), if \( a \neq \text{FAIL} \)
- \( \text{FAIL} \| b = b \)
- \( \text{FAIL} \ E = E \)
- \( \text{TE}[\text{exp}] \) translates the expression \( \text{exp} \) from Miranda to a Lambda expression.
- \( \Lambda \) is a pattern matching \( \lambda \)

This translation scheme introduces a number of lambda abstractions \((\lambda v_1.\lambda v_m)\) into the definition in order to draw the arguments to the function into the operands of fatbar. Rather than introduce these lambda abstractions and the variables they bind into a definition, I elected to carry out the drawing in of arguments using the thinbar operator. The following section demonstrates the semantic equivalence of Peyton Jones' scheme and my own.

**Equivalence of Prospero and Peyton Jones' Mechanism**

All pattern matching in the Prospero evaluation scheme and that of Peyton Jones is carried out using pattern matching lambdas. The two schemes are semantically identical except in the area of failure to pattern match.

In Peyton Jones' scheme pattern match failure is represented using the symbol \( \text{FAIL} \). Prospero introduces an additional stage involving the rewriting of a pattern matching lambda to \( \text{V} \) before the \( \text{FAIL} \) symbol is introduced. \( \text{V} \) is always rewritten to \( \text{FAIL} \) and only serves to indicate exactly which part of the pattern matching has caused the
failure, as such the introduction and immediate rewriting of \( \triangledown \) does not alter the equivalence of the two schemes.

Definitions with a number of clauses are represented by Prospero’s thinbar operator and Peyton Jones’ fatbar. The only difference between these two operators is the way in which arguments are drawn in to the expressions which form their operands. Peyton Jones draws arguments into the operands of fatbar by introducing a number of lambda abstractions into each of the fatbars’ operands. The introduction of these lambdas can be seen in the extract from (Peyton Jones ‘87) given above.

Rather than introduce these extra lambda abstractions, Prospero passes the responsibility of drawing in the arguments to thinbar itself. By comparing the definition of single_step given above with the definitions of fatbar given in (Peyton Jones ‘87) it can be seen that these two schemes are equivalent.

Once pattern matching of a clause has succeeded or failed the thinbar and fatbar operators become the redex of an evaluation. Both of the operators behave in exactly the same way at this stage.

In summary, Peyton Jones’ scheme is equivalent to the Prospero scheme with the exceptions that an extra stage is introduced during pattern matching failure, and the responsibility for the drawing in of arguments prior to pattern matching is carried out using the thinbar operator, rather than by introducing extra lambda abstractions.

An Alternative Thinbar

An alternative definition of thinbar is one where arguments are pulled one by one into the partial functions which form the operands of thinbar, and pattern matching is carried out after each argument has been pulled in. Rather than producing a new set of definitions for this version of thinbar, I will demonstrate the way in which this new definition would work using the example given in Figure 7.1(b), which involves the evaluation of the definition given in Figure 7.1(a). Figure 7.1(c) shows the same evaluation using the definition of \( | \) given earlier in this chapter, and adopted in the present implementation of Prospero.
Chapter 7 - Evaluating Expressions

(a)  
```
f 1 2 = rhs1  
f x y = rhs2
```  

(b)  
```
f 2 3  
⇒ (\(λ.λ.λ.rhs1\) | (\(λ.λ.rhs2\) | ERROR)) 2 3  
⇒ ((\(λ.λ.rhs1\) 2) | ((\(λ.λ.rhs2\) | ERROR) 2)) 3  
⇒ ((\(λ.λ.rhs1\) 2) | ((\(λ.λ.rhs2\) | ERROR) 2)) 3  
⇒ (FAIL | ((\(λ.λ.rhs2\) | ERROR) 2)) 3  
⇒ (\(\lambda.\lambda.rhs2\) | ERROR) 2) 3  
⇒ (\(\lambda.\lambda.rhs2\) 2) | (ERROR 2) 3  
⇒ (\(\lambda.\lambda.rhs2\) 2) | (ERROR 2) 3  
⇒ (\(\lambda.\lambda.rhs2\) 2 3) | (ERROR 2 3))  
⇒ (\(\lambda.\lambda.rhs2\) 2 3) | (ERROR 2 3))  
⇒ (\(\lambda.\lambda.rhs2\) 2 3)  
...  
(c)  
```
f 2 3  
⇒ (\(λ.λ.λ.rhs1\) | (\(λ.λ.rhs2\) | ERROR)) 2 3  
⇒ ((\(λ.λ.rhs1\) 2) | ((\(λ.λ.rhs2\) | ERROR) 2)) 3  
⇒ ((\(λ.λ.rhs1\) 2 3) | ((\(λ.λ.rhs2\) | ERROR) 2 3))  
⇒ (FAIL | ((\(λ.λ.rhs2\) | ERROR) 2 3))  
⇒ (\(\lambda.\lambda.rhs2\) | ERROR) 2 3  
⇒ (\(\lambda.\lambda.rhs2\) 2) | (ERROR 2) 3  
⇒ (\(\lambda.\lambda.rhs2\) 2 3) | (ERROR 2 3))  
⇒ (\(\lambda.\lambda.rhs2\) 2 3) | (ERROR 2 3))  
⇒ (\(\lambda.\lambda.rhs2\) 2 3)  
...  
```

Figure 7.1 Examples of evaluations using an alternative thinbar operator

This example demonstrates that this alternative version of thinbar is more efficient than the version described above as the step involving pulling the second argument into the first definition is avoided by carrying out the pattern matching for the first argument early. However, the example given in Figure 7.2 shows that this model is not equivalent to the semantics of Miranda. Here the evaluation of \(ε\) to the single argument \(⊥\) (shown in Figure 7.2(b)) evaluates to \(⊥\), where as using the original definition of thinbar the evaluation will go no further than pulling in the first argument, giving the evaluation shown in Figure 7.2 (c).
Chapter 7 - Evaluating Expressions

(a)
\[ f \, 1 \, 2 = \text{rhs1} \]
\[ f \, x \, y = \text{rhs2} \]

(b)
\[ f \, \perp \]
\[ \Rightarrow (A_1.A_2.\text{rhs1} \mid A_x.A_y.\text{rhs2} \mid \text{ERROR}) \, \perp \]
\[ \Rightarrow ((A_1.A_2.\text{rhs1}) \, \perp \mid (A_x.A_y.\text{rhs2} \mid \text{ERROR}) \, \perp) \]
\[ \Rightarrow \perp \]

(c)
\[ f \, \perp \]
\[ \Rightarrow (A_1.A_2.\text{rhs1} \mid A_x.A_y.\text{rhs2} \mid \text{ERROR}) \, \perp \]
\[ \Rightarrow ((A_1.A_2.\text{rhs1}) \, \perp \mid (A_x.A_y.\text{rhs2} \mid \text{ERROR}) \, \perp) \]

Figure 7.2 Illustration of the change in operational behaviour of the alternative thinbar operator.

Case Statements

An alternative representation of pattern matching involves the use of case statements. If we take the following function definition:

\[
\text{leaf}_\text{inc} \, i \, (\text{Leaf} \, 1) \\
\quad = \text{Leaf} \, (1+i)
\]

\[
\text{leaf}_\text{inc} \, i \, (\text{Branch} \, l \, r) \\
\quad = \text{Branch} \, (\text{leaf}_\text{inc} \, i \, l) \, (\text{leaf}_\text{inc} \, i \, r)
\]

and translate this definition into a case statement as described in (Peyton Jones ‘87), then we get the following piece of enriched lambda calculus:

\[
\text{leaf}_\text{inc} \\
\quad = \lambda i.\lambda t \\
\hspace{1cm} \text{CASE} \, t \, \text{of} \\
\hspace{2cm} \text{Leaf} \, 1 \Rightarrow \text{Leaf} \, (l + i) \\
\hspace{2cm} \text{Branch} \, l \, r \Rightarrow \text{Branch} \, (\text{leaf}_\text{inc} \, i \, l) \, (\text{leaf}_\text{inc} \, i \, r)
\]

This representation does not meet the requirements of a pattern matching system because the pattern matching has strayed from the left-to-right, top-to-bottom intuition of pattern matching. Instead, the representation above uses a mixture of sequential and concurrent pattern matching where the \( i \)th pattern of each definition is matched in parallel and then the tree constructor patterns are matched sequentially.
Chapter 7 - Evaluating Expressions

The use of case statements as specified in (Peyton Jones ’87) also introduces a number of optimisations which result in a loss of information about the original patterns which would be unacceptable for our purposes.

7.2 Summary

In this chapter I have described a lazy evaluation scheme, which faithfully represents the Miranda programming system. The equivalence of this scheme and that described in (Peyton Jones ’87) is discussed.

This scheme, or model, produces all of the low-level information proposed in earlier chapters as necessary in order to reap the benefits of a rewriting approach. In particular the model represents the detailed processes of pattern matching and guard evaluation.

Basing the model on an enriched lambda calculus has allowed the representation of higher order functions and partial application of functions.

I have shown in earlier chapters of this thesis that, based upon the model described here, it is possible to produce information to users faced with a wide range of errors, and to produce information required by students of lazy functional programming. As such, one can conclude that the model described here forms a successful foundation for the production of meaningful views of an evaluation.
8 Prospero's Interface

The main themes of the research presented in this thesis are: to develop a model which could act as a foundation for views of an evaluation; to develop facilities with which to produce views of an evaluation; and to develop facilities with which to navigate through an evaluation history. A notable absentee from this list is the development of an interface. In fact, Prospero was implemented to be as independent of a ‘concrete’ interface as possible.

The interface used to generate examples in this thesis is an artefact of the need to have some kind of visible presentation of reduction stages. A very small proportion of the total effort involved in producing Prospero was expended on developing the interface. However, sufficient interest has been shown in the more novel aspects of this interface, that an outline description of these aspects seems warranted. The aspects described here include the graph-layout algorithm used in the rendering of reduction graphs, and the conjunction of functional and object-oriented programming to produce the current interface.

In order to be as interface independent as possible, Prospero incorporates an abstract model of an interface. This model greatly simplifies the development of concrete interfaces, and minimises the number of updates made to the display following a reduction step. A description of this ‘abstract interface’ is presented here.

This chapter will also place Prospero into the taxonomy of software visualisation systems described in (Price et al '92).

8.1 Interface Implementation Details

The current interface should be viewed as a prototype borne from the need to illustrate Prospero at work, rather than to demonstrate a novel presentation of the type of graphs produced by Prospero, and as such, lacks many of the facilities that one would expect to find in a modern interface. Indeed some of the facilities described
here are not implemented beyond their inclusion in the interface as menu options or buttons. The reader is referred to Appendix B for details of the implementation status of Prospero. The main reason for describing the implementation of the interface is that the issues involved in the combination of a modern graphical, window-based interface with a lazy functional programming language are understood but rarely tested. The only documented work in this area is that described in (Dwelly '89) and (Singh '91).

The interface was developed using Miranda and the BrouHaHa implementation of Smalltalk-80 (Goldberg & Robson '83).

Although the present interface to Prospero is written using Miranda and Smalltalk, Prospero was designed to be as interface independent as possible. To allow for this, most of the processing of user interaction and management of the content of the display is done in the part of Prospero written in Miranda, thus allowing the minimum amount of rewriting of code when experimenting with alternative interfaces to the system. In the following sections I will describe the section of the interface implemented in Miranda, followed by that implemented using Smalltalk-80 together with the way in which the two components were connected together.

8.1.1 Interaction using Lazily Evaluated Lists

It is common practice when writing interactive programmes with a lazy functional language to model the interaction using two infinitely long lists of characters. For example, a simple function which repeatedly prompts the user for a number, reads the numeral, and then prints that number of asterisks on the terminal, until the user enters zero, would have the type:

\[
\text{asterisks} :: [\text{char}] \to [\text{char}]
\]

The argument to the function can be treated as a list of characters entered by the user, and the result of the function is the list of characters displayed on the user’s terminal. As each character of user input is required by a program, then a character is taken from the head of the input list, and as output characters are generated, they are appended to the output list. The definition of the asterisks function is given in Figure 8.1.
The Miranda portion of Prospero can be viewed as a function which takes a list of characters as its argument and returns a list of characters as its result. (in order to simplify the following description, I will refer to this function as \textit{waf1}). These lists are used to connect a process running Miranda, to Smalltalk. This process is started up by the Smalltalk interface when execution of Prospero begins.

The list connecting Smalltalk to \textit{waf1} is used to notify \textit{waf1} that the user has made a request which requires a change in the expression being currently displayed. For example, a notification would be sent if the user were to request a single step, or the evaluation of a new expression, but would not be sent if the user requested a zoom or pan operation, as these operations change the display of the expression rather than the expression being displayed.

Following a change request sent from Smalltalk to \textit{waf1}, the list connecting \textit{waf1} to Smalltalk is used to notify the Smalltalk system of the changes to the expression being displayed. The Smalltalk system represents an expression as a graph with labelled nodes, so the changes are in the form of the creation, removal, labelling and connection of graph nodes.

\footnote{\textit{waf1} - (What, Another Functional Language) - name thanks to Steve Sommerville.}
8.1.2 The Virtual Display

One of the design goals of Prospero was to make it as independent of one particular user interface as possible, thus making it as simple as possible to change from one user interface to another. In order to achieve this, the majority of the code which controls the contents of the user interface is written in Miranda. This code will be referred to as the Virtual Display. The production of the Virtual Display means that in order to create a new user interface to Prospero, the interface implementation need only be capable of:

- displaying a representation of an evaluation stage;
- parsing updates to the display coming from the Miranda system;
- telling the Miranda system when an interaction has occurred which requires an update to the displayed information;
- carrying out any operations involving the displayed expression which do not require additional information about the expression (e.g. zoom, pan & scale).

By having a model of the user interface, waf1 can keep track of the state of the interface without having to interrogate the user interface system, thus simplifying the interface between waf1 and the interface implementation. The way the model has been designed also allows waf1 to minimise the amount of redrawing that takes place following a graph reduction step.

What is the Virtual Display?

The structure of the virtual display is shown in Figure 8.2. A virtual display has a number of windows; each window has a filter and a content. The filter is a Prospero filter as described in Chapter 5. The content is a model of a graph in the form of a number of sub-graphs; each sub-graph has a number of segment keys; each segment key is the unique identifier of a graphical entity on the screen.
Using this model \textit{wafl} keeps a representation of the current evaluation stage in terms of a number of sub-graphs. Each sub-graph directly maps on to the reduction graphs being used by Prospero’s graph-reducer to evaluate the current expression. By organising the information in this way \textit{wafl} can generate the minimum number of updates to the physical display by only generating updates for sub-graphs which have changed since the last display of the expression. For example, if a user requests that the expression is reduced by a single reduction step, and that reduction step results in the graph-reducer changing only one sub-graph, then \textit{wafl} need only generate update commands which reflect the change to that sub-graph.

The usefulness of the virtual display was demonstrated when it became necessary to switch from Prospero’s initial user interface (based on the VP system described below), to the current Smalltalk interface. As was intended, the switch involved no changes to the implementation of the virtual display, and the job of writing the Smalltalk interface was made easier by the simplicity of the interface between \textit{wafl} and Smalltalk.
8.1.3 The Smalltalk Interface

The visible, or concrete, part of the Prospero interface is written in Smalltalk-80. The standard approach to constructing interactive applications in Smalltalk-80 is to use the Model-View-Controller (MVC) framework (ParcPlace '92). Using this framework an application is divided into three components:

- the view is the part of the application visible to the user;

- the model is the underlying representation of the information presented to the user, and is independent of the way it is presented;

- the controller is used to process user input in such a way that interactions with the user, are reflected by changes in the model and/or the view.

Models can exist with, or without, associated views and controllers. In fact a model is 'unaware' of its view and responds to messages in exactly the same way whether it has zero, one or a number of views. Controllers and views on the other hand are not usually found outside an MVC relationship. Each view has a single model and controller associated with it. Each controller has a single model and view.

In the following sections I will describe the model and view used in Prospero. No description of the controller is warranted since a standard Smalltalk-80 controller is used to process mouse and keyboard input.

The View

The view used by Prospero is a composition of several scroll bars, buttons, menus and a main drawing on to which reduction graphs are rendered. The implementation of the scrolling behaviour, the buttons and the pop-up menus is carried out using standard techniques. The most complex part of the functionality of the view is the rendering of the reduction graph\(^1\).

The graph renderer uses a three-pass graph-layout algorithm. All three passes of the algorithm are depth-first. In the first pass, the layout of the graph is calculated. Each node calculates the space needed in order to display itself together with all of its

\(^1\) The graph-layout scheme described here was carried out independently of similar work which had gone before it (e.g. Tamassia, Di Battista & Batini '88). The author was unaware of this work at the time.
Chapter 8 - Prospero’s Interface

offspring. A node with no offspring ‘claims’ the space necessary to display its label in a rectangular box. Note that the space requested by each node is calculated with no regard for the actual size of the window in which the graph will finally be rendered.

The second pass, starts at the root of the graph and allocates the root node all of the actual window space available. The root node calculates its position within this space, and then divides the remaining space among its offspring. This division is based upon the space requested by each offspring during the first pass. Each sibling then carries out the same process with its siblings until the complete graph has been traversed. Should the available space be less than that requested by any node, the node will still position itself and its offspring in the space available, this may lead to overlapping nodes when attempting to render a graph which is too large for the available window. Note that shared graph nodes are not considered as a special case here: instead a shared node takes up the position resulting from the final time that it is required to calculate its position.

In the final pass, the graph is traversed, and each node is rendered at its associated position, at the same time edges are drawn between the nodes.

Note that the three separate passes, could be combined, and were only separated out to simplify the implementation of the interface.

*The Model*

While MVC is a framework suited to the construction of simple interactive applications, it is very common to find more complex applications using enriched versions of MVC. One example is when the application is not contained within the Smalltalk world, instead parts of the application, usually the part traditionally represented in the model, are held and manipulated by external applications. In this case an addition is made to the framework, I will refer to this addition as the *external model*. The extension to the traditional MVC framework is illustrated in Figure 8.3, which shows the references of the various components to each other, and the addition of an external model, referenced by the model.
In Prospero the *external application* is Prospero’s Miranda component. The *model* is a simple representation of a graph made up of a set of nodes. The *external model* has a character stream connecting it to applications running in the ‘outside world’.

As far as the view and controllers in such an application are concerned, there is no difference between this framework and the traditional MVC framework. Using a reference to the external model, the model can respond to messages from the *view*, and the *controller*, by forwarding these requests to the *external model*. The *external model* translates these requests into commands for the *external application* and forwards them using the character stream. The requests are processed by the *external application* (i.e. the Miranda component of Prospero), and the results are relayed back to the *external model* using the character stream. The *external model* then parses these
results, and passes them to the *model*, which modifies its state, before notifying the *view* that a change has occurred.

It is worth noting that in the implementation of Prospero, the framework described above involves a certain amount of duplicated information held in both the model, and the external application, i.e. the abstract interface described above, holds much the same information as the *model*. This is due to two separate design decisions.

First, as shown in the above section on the Abstract Interface, Prospero was designed to be interface independent. In order to achieve this, the Abstract Interface holds information which would normally only be the concern of the implementation of the interface.

Second, the duplication of data reduces the amount of data flowing between Miranda and Smalltalk, since both components of the system are effectively caching display information, and communicating changes to that information.

### 8.1.4 An Alternative Graph Display Technique

As an aside, it is worth briefly describing an alternative style of graph presentation implemented using the VP system described in (Billyard '92). VP takes a description of a directed graph and a PostScript definition of how each graph node is to be presented, and produces a window containing a representation of the graph.

The problems associated with the automatic layout of large graphs presented using the standard technique of using lines to represent arcs connecting nodes, are well known. As an attempt to overcome these problems, VP presents a node as an object with a number of parts. So a node P with 3 parts P1, P2 and P3, where P2 has parts P3 and P4, could be presented as shown in Figure 8.4.

![Diagram of an alternative graph display technique](image)

**Figure 8.4.** An example of the presentation of a graph in the VP graph system.
If we now say that any node in a graph can be represented as a VP node with a number of parts, each part representing one of its offspring, then the display shown in Figure 8.4 would be the representation of the graph shown in Figure 8.5.

Figure 8.5 The graph presented by VP in Figure 8.4

Displaying a graph using this technique means that the size of the representation of the graph is independent of the size of the graph. However, as a rule Prospero graph nodes have a small number of edges entering and exiting from them. This means they are much deeper than they are wide. This causes problems when displaying graphs in the style shown in Figure 8.4. A relatively shallow view of a graph is visible at any one time and it becomes very difficult to get a feel for the overall structure of the graph.

8.2 Classifying Prospero using a Taxonomy of Software Visualisation

In (Eisenstadt et al '90) "nine key dimensions that affect the design of visual programming and programming visualisation tools" are proposed. These ideas are expanded upon in (Price et al '92), where a taxonomy is described with which to provide a:

meaningful way of describing software visualization technology ... [providing] a clear and concise statement of the essential features, similarities, and differences between specific systems, [and developing] insights into the weaknesses of existing technologies and the needs for new technologies.
In this section I shall place Prospero into this taxonomy with the aim of highlighting its strengths and weaknesses, and to allow the reader to carry out a comparison between Prospero and other systems discussed by Price.

Rather than describing fully the sections of the taxonomy, I will simply outline them here, full details can be found in (Price et al '92). The taxonomy contains six sections each of which is further divided into a number of categories\(^1\).

---

A. Scope
What are the system's general characteristics?

1. System/Example
Is the work discussed a visualization system or an example of a visualization?

Prospero is a visualization system; it generates visualizations of Miranda programs.

2. Class of Program
What class of program is the system designed to visualize?

Prospero is designed to visualize any class of Miranda program.

3. Scalability
What restrictions are there on working with large programs or data sets?

This is a category which is emphasised in the taxonomy as a major failing of many visualization systems which are not of use for anything but toy problems. Of course there are two main reasons why a system may not be suitable for dealing with large problems. Either the interface to the system does not offer facilities which allow the user to cope with large amounts of data, or the implementation of a system can not cope with large problems because it requires too much processing time and too much memory.

I feel that Prospero’s interface - with its zoom and pan facilities, in combination with the filtering system which allows for the removal of large amounts of data - would support large problems. However, the present implementation of Prospero would have to be made far more efficient with respect to memory usage and execution speed.

---

\(^1\) Some categories which were clearly not applicable to Prospero have not been included in this chapter. In order to maintain consistency with the original taxonomy I have not re-numbered the categories.
before this became possible. The current implementation of Prospero is unsuitable for use with any evaluations larger than the examples presented in this thesis. This is not a problem inherent to Prospero, it is the result of a design whose main goals did not include efficiency, and an implementation written using an interpreted functional language.

4. Multiple Programs

*Can the system generate visualizations of multiple programs simultaneously?*

No.

5. Concurrency

*Does the system support the visualization of concurrent programs?*

No.¹

B. Content

*What gets visualized?*

7. Program/Algorithm Visualization

*Is the system designed to produce algorithm or program visualization? (i.e. is the system designed to educate the user about a general algorithm (algorithm visualization), or is it educating the user about a specific implementation of an algorithm (program visualization)?)*

Prospero produces a visualization of the implementation rather than the algorithm; although it could be used to highlight the pertinent points of a particular algorithm. If we expand the definition of this category by saying that a program visualization can be used as an algorithm visualization, but not vice-versa, then Prospero is certainly a program visualiser.

---

¹ A system to help with the programming of a parallel machine using a lazy functional language was the starting point for the research described in this thesis. This avenue was not pursued. It is clear however that this research area is an interesting and a vital one. Understanding the evaluation of a parallel algorithm written in a lazy functional language is notoriously difficult, especially when no explicit parallelising extensions are added to the language.
8. Code Visualization

*Can the system visualize the program code? (e.g. pretty printed source, call trees etc.).*

Parts of the code appear in the output of Prospero, so Prospero does visualise the program code.

9. Data Visualization

*If the system performs program visualization, can it visualize the program data?*

Yes.

10. Compile/Run time

*Is the data on which the visualization depends gathered at compile-time, at run-time, or both?*

Run time. The output of Prospero depends completely on the information that becomes available as an expression is evaluated.

11. Fidelity and Completeness

*Do the visual metaphors present the “true” and complete behaviour (Eisenstadt et al. 1990) of the underlying virtual machine?*

This point raises an interesting issue. While Prospero offers a semantically “true” and complete representation of its own underlying virtual machine, it is not clear whether it presents a “true” and complete representation of the Miranda virtual machine. While Prospero certainly presents a semantically “true” representation of Miranda, the virtual machine underlying Miranda is operationally different to that presented by Prospero, since Prospero presents an operational picture based on an enriched lambda calculus (Chapter 7), and Miranda is based on combinator reduction (Turner ’79). Thus, the presentation given by Prospero can be said to be semantically true, and operationally false.

C. Form

*What elements are used in the visualization?*

12. Medium

*What is the primary target medium for the visualization system? (e.g. paper, film, video, terminal, workstation)*

Prospero’s primary target medium is a workstation with a bit-mapped display. Although the development of a pretty printing interface such as that described above
would allow the system to be run on a simple terminal. The system could be used to
generate figures for presentations and publications. This is demonstrated by many of
the figures in this thesis.

13. Graphical Elements
What graphical elements are used in the visualization produced by the system?

The graphical elements used in the interface described here are lines, boxes and text.
(This is not to say that more complex graphical elements could not be used in a future
implementation.)

14. Colour
Does the system make use of colour in its visualizations?

No colour or shading is used in the interface. (Again, there is no reason why the
interface should not be expanded to incorporate these features.)

15. Animation
If the system gathers run-time data, is the resulting visualization animated or static?

There is no clear answer to this question. Prospero builds a history of an evaluation:
the subsequent presentation of this history could be said to be providing an animated
view of the evaluation (this is the line taken by Price when applying the taxonomy to
the Transparent Prolog Machine).

16. Multiple Views.
Does the system provide multiple synchronised views of different parts of the
software being visualized?

It was intended that Prospero would have a number of views of an evaluation, each
with a different filter associated with it. At the time of writing this is not the case (see
Appendix A).

17. Other Modalities
Does the system appeal to senses other than sight?

No.
Chapter 8 - Prospero’s Interface

D. Method

How is the visualization specified?

18. Specification Style

What style of visualization specification is used? (e.g. modification of source code, writing of additional visualization code, automatic visualization)

The output of Prospero is generated automatically, no additional code is required of the user.


If the visualization data is gathered at run time, is the visualization produced as a batch job from data recorded during a previous run, or is it produced live as the program executes?

Live.

20. Fixed/Customisable

Is the visualization which is generated completely fixed, or can the user customise it by some means?

The visualization is customisable.

22. Invasive

Does the program source code have to be modified in order to obtain a visualization?

No.

23. Customisation Language

If the visualization is customisable, how can the visualization be specified?

The visualization is customised using the filter system described in Chapter 5.

E. Interaction

How do you interact with and control the visualization?

25. Navigation

How well does the system support the navigation through a visualization of a large program or data set?

The navigation through the information in a stage of an evaluation could be said to be standard: the user may select-and-zoom, zoom in and out, and pan over a graph.
Navigation through the evaluation itself is achieved by stepping through the evaluation or using the search mechanism.

26. Elision

Does the system support techniques for eliding information or suppressing detail form the display?

Elision is achieved through the filtering system.

27. Temporal Control Mapping

If the visualization is based on data gathered at run time, what is the mapping between “program time” and “visualization time”?

There is no real mapping between the two types of time. This lack of mapping may cause users to hold misconceptions about the efficiency of their definitions. The mapping is made even less concrete by the effect that different filters can have on the definition of a step (see Chapter 5).

F. Effectiveness

How good is the visualization

28. Appropriateness and Clarity.

How well does the visualization communicate information about the software?

To answer this question would require experimental evaluation of Prospero (see Chapter 9)

29. Experimental Evaluation

Has the system been subjected to a good experimental evaluation?

No.

30. Production Use

Has the system been in production use for a significant period of time?

No.

8.2.1 Classification Summary

The taxonomy above is a large one. In this section I briefly summarise each section of the taxonomy with respect to Prospero.
Chapter 8 - Prospero’s Interface

Scope

Prospero can be used to visualize any Miranda program. The system cannot present more than one evaluation at once, and does not support the evaluation of expressions on a parallel machine.

Content

Visualizations are presented in terms of the user’s code being evaluated on an abstract machine carrying out graph reduction. Because Miranda supports higher-order values, there is no differentiation between functions and data in Prospero presentations. The visualizations are produced at run time.

Form

Prospero output is primarily aimed at the user of a workstation. The interface presented in this thesis does not take advantage of many of the graphical features of modern bit-mapped displays (see Chapter 9 for a discussion of future work in this area).

Method

To produce a visualization the user has to evaluate the expression in question using Prospero; no additional code is necessary. The presentation can be customised using filters which are at present written in Miranda.

Interaction

Prospero allows users to move through an evaluation using the search and step facilities described in Chapter 6.

Regarding temporal mapping, Prospero is an implementation of Miranda which is designed with the provision of information in mind, rather than efficient evaluation. This has an effect on the perceived efficiency of a user’s definitions.

Most modern day implementations of lazy functional languages are based on the translation of definitions into either combinators (Turner ’79) or super combinators (Hughes ’82). Prospero translates definitions into an enriched version of the lambda calculus (described in Chapter 7). No optimisation is carried out during the translation process; this lack of optimisation, combined with the possibility of a user watching an evaluation at the level of the lambda calculus, can lead to users building a false set of beliefs about the efficiency of their definitions.
Chapter 8 - Prospero's Interface

This problem is not a new one. Many modern implementations of imperative languages eliminate optimisation when compiling code which is to be viewed by a debugger. The facts are simple: optimisation of code removes information which could be useful for debugging purposes, the removal of this optimisation in order to provide more information, leads to users being told a story which is inaccurate as far as efficiency is concerned, but which is semantically accurate.

Effectiveness

No experimental evaluation of Prospero has been carried out, consequently no conclusions can be drawn about the effectiveness of Prospero. The reader is referred to Chapter 9 for a discussion of this area.

8.3 Summary

In this chapter a number of aspect of the current interface to Prospero have been explored. Since, the current interface has come out of the need to produce some kind of interface, rather than to explore an avenue of research, an in-depth description and evaluation of the interface has not been provided.

The chapter has outlined the implementation of the interface as two separate components: one written in Miranda and one in Smalltalk. The use of laziness to represent input and output character streams as lazy lists has been illustrated, together with the extension of Smalltalk’s MVC architecture to represent an application with an external implementation.

The current interface’s depth-first, multi-pass graph layout and rendering algorithm has also been outlined.

In order to place the current interface and underlying implementation in the context of other graphical debugging systems, the system has also been placed in the taxonomy for such systems, proposed in (Price et al '92).
The main aim of this thesis was to investigate the provision of information about the evaluation of expressions written in a lazy functional language, and to show that a system based upon a single underlying model could be used to provide a wide range of information to satisfy a wide range of user needs.

In the introduction to this thesis the impact of lazy evaluation on users and designers of systems such as Prospero was discussed. The work here has demonstrated that it is possible to design a system which offers a range of facilities to overcome these problems. Specifically: Prospero displays information to users without introducing any strictness into an evaluation; the design of a search system has been presented which allows control over the amount of strictness introduced; and an evaluation model has been produced which is faithful to that of the Miranda system, and which allows users to see the impact of pattern matching and guard evaluation on evaluation order.

A review of possible approaches to providing information about lazy functional language definitions has shown that while some approaches provide superior information, this information is specialised to satisfy a narrow range of the user's needs. More general approaches are available, and while failing to match the quality of the information provided by specialist approaches, they can be used to provide a wider range of views of an evaluation. In particular rewriting has been shown to be a good all-round approach to providing information.

A review of existing systems reinforced the findings of the review of approaches, showing the weaknesses of systems based on specialist approaches. The review of systems based on a rewriting approach, shows these systems to be lacking in a number of areas. The systems fail to provide the user with information about certain
aspects of their evaluation, for example none of the systems provide detailed information about pattern matching; the way in which information is presented is very basic, with no way for users to control the information that is presented, or to be shown a number of views of an evaluation; and the facilities available with which the user can navigate through an evaluation are also primitive, normally restricted to single-stepping.

By presenting Prospero, this thesis has demonstrated that by using a carefully designed underlying model, and by providing the correct facilities, it is possible to produce a system which provides quality information to users of a lazy functional language.

The work described in this thesis has shown that certain types of information about an evaluation can only be produced if the underlying model provides low-level rewriting information. Prospero has demonstrated the principle of providing low-level information and abstracting away from it in a variety of ways. In this way new views of an evaluation can be created as and when necessary, and low-level information, necessary to explain certain aspects of an evaluation, is always available. A complete description of this low-level model has been presented, and the equivalence of this representation to other accepted representations has been shown.

Illustrations throughout this thesis have shown that information about the rewriting of expressions can be presented to users at different levels of abstraction. To date, no ‘perfect’ level of abstraction has been discovered, although a number of systems are available which offer a single abstraction level. This thesis has shown that it is possible to avoid the trap of imposing a single abstraction level on the user, and to allow users to experiment with viewing the same evaluation in different ways. This has been achieved by providing a filtering system which allows new levels of abstraction to be created, either from scratch, or by composing existing filters in order to create new filtering effects.

The filter system has been shown to support the abstractions frequently used when writing definitions in a functional language; the elision of large data structures; the removal of evaluation detail of specific expressions; and the removal of low-level rewriting information.

While filters allow the user to abstract away from the detailed information available at every stage of an evaluation, they do not help the user to deal with the large number
of evaluation stages that may go to make up a complete evaluation history. The search system presented here shows that it is possible to provide users with advanced facilities with which to search for specific stages in an evaluation.

The description of the search system highlights the potential pitfalls of allowing searches to alter the order of an evaluation. Three possible search strategies, differentiated by the amount of strictness they introduce, have been presented. The three approaches offer the user a trade-off between introduced strictness, and the possible 'distance' between the point at which an expression is introduced into an evaluation, and the point at which a search involving this expression can be tested.

A number of other systems have appeared recently which have been influenced by this work described here.

The Miranda Calculator is described in Section 3.3.1. Many of the problems encountered during the implementation of Prospero were taken on board by the designers of the Miranda Calculator, and some of the ideas behind Prospero would have been incorporated had time allowed. For example, the provision of filtering facilities, a detailed representation of pattern matching, and more advanced navigation facilities would all have been provided.

The Hint system is described in Section 3.3.2. This system is, in certain respects, very close to Prospero, although the end use of the system, the production of profiling information, is different. Hint produces graphical presentations of evaluation histories which are based upon reduction graphs. Spacial filtering facilities provide the user with an equivalent to Prospero’s filter system. Hint’s temporal filters provide a way to search for specific stages in an evaluation, and as such provide a similar facility to Prospero’s searches.

9.1 Future Work

While the work described here has made a contribution to the field by exploring the potential of providing information about lazy evaluation using a rewriting approach, and by overcoming the problems inherent in this approach, there are a number of areas where there is clear potential for future work. The work can be divided into three main areas: field evaluation of the system and of the ideas put forward here; work required in order to transform the prototype system into a production one; and work which naturally follows on from that described in this thesis.
9.1.1 Experimental Evaluation

As it stands, Prospero is an unevaluated prototype system. In order to evaluate Prospero’s usefulness in the area of teaching and of removing errors from definitions, it will be necessary to carry out experimental evaluation of the system. A brief description of a possible evaluation is given below.

Evaluation of Prospero as a Teaching Tool

In order to evaluate Prospero as a teaching tool used by teachers to present information to students, and used by students to develop and observe their own functional language definitions, it will be necessary to carry out an experiment similar in style to the Calculator Project undertaken at Queen Mary and Westfield College and the Open University (Goldson '93a) (Fung & O’Shea '92).

The Calculator Project is a longitudinal study taking place over two years. In the first year the performance of students, undertaking a first year undergraduate introductory course in programming, is evaluated using the student’s results and through interviews with the students. The second year of the study involves the next year’s intake of students, who are taught similar material, but this time the teaching is supported by a variety of tools. These tools (which include the Miranda Calculator described in Chapter 3) aim to support the teaching of programming concepts, and to provide students with information about the evaluation of expressions written in Miranda.

The results of this experiment will be based upon the comparison of the performance of the two sets of students.

The experiment described above is a lengthy one, but represents one way in which to test the usefulness of a tool in a teaching environment. Of course the scale of such an experiment need not be as large as this one; it is possible to carry out a similar test with a smaller group of students taking a shorter course.

There are however some ethical problems with the approach outlined above, since the experiment involves the deprivation of one set of subjects, and inflicting untested tools upon another set. An alternative approach would be to carry out the evaluation proposed in the following section, but in a teaching context.
Chapter 9 - Conclusions & Future Work

**Evaluation of Prospero as an Error Finding Tool**

There are a number of styles of experiments which could be carried out in order to test the usefulness of Prospero as a tool with which users can better understand and so remove errors from their definitions.

Two groups of users could be presented with a number of sets of definitions each containing different types of error (e.g. non-terminating evaluations, application of a function to an argument outside of its domain). One group would be asked to correct the errors in the definitions equipped only with a functional language interpreter; the second group would be provided with Prospero. The results of this experiment would be used to show:

- the differences in performance between the two groups (i.e. how successful were they in finding the errors; how quickly did they find the errors; how successful were they in correcting the errors);

- the categories of errors for which Prospero was used;

- the facilities of Prospero which were used;

- additional facilities which could be provided by Prospero.

A second experiment could be carried out where the two groups were asked to write definitions to solve specific problems. Again a variety of tasks could be set (e.g. problems involving abstract data types, problems involving higher order functions). The results of this experiment would be similar to that described above:

- the differences in performance between the two groups;

- the categories of problems for which Prospero was used;

- the facilities of Prospero which were used;

- additional facilities which could be provided by Prospero.

**9.1.2 Filters**

There are a number of potential areas for future work on Prospero's filters. This work includes enhancements to the current filter system, and more interestingly, research
into the usefulness of different filters and the 'effects' which can be produced using filters.

Defining Filters

Most of the work which has been carried out to date has been on the design and implementation of a system which allows users to describe filters and use them to view evaluations.

This thesis is illustrated using a large number of examples. I have taken care throughout not to discuss the decisions made in choosing these illustrations, and any alternative illustrations which could have been used in their place. For example, my choice of representation of a balanced tree may not be one which some people consider useful.

Prospero is deliberately designed not to limit the user to a number of 'useful' views. Instead a wide range of views of an evaluation can be produced. Judging by the heated debate\(^1\) some of the illustrations used in this thesis have generated, there is clearly work to be done in exploring the types of view that can be produced.

To go further, it is possible to produce views of an evaluation which could be considered dangerously misleading, for example it is possible using temporal filters to make it appear that Prospero is carrying out call-by-value rather than call-by-reference. Whether these 'dangerous' views can be described in a way which would facilitate their eradication remains to be seen.

There is clearly a large amount of potential research work in this area. The different types of filters, and the uses to which these filters could be put has by necessity been left unexplored.

User-Definition of Filters

The nature of the work described here, has led to a situation where filters are defined using Miranda syntax. While this is acceptable when dealing with an experienced

---

\(^1\) This debate took place between the author and his supervisors (Bornat & Clarke) towards the end of the implementation of Prospero. The debate was part of a larger discussion on the use of Prospero as a teaching tool.
Miranda programmer, using the system to test various ideas on filtering in order to produce the results presented in this thesis, demanding that users define their own filters using the Miranda language is unsuitable for a number of reasons:

- by definition it is impossible for users who are learning to use Miranda to define their own filters in this way;

- even with the provision of pre-defined definitions to be used by an author to simplify the filter writing process, some knowledge of Prospero’s internal representation of expressions is at present still essential;

- designing filters using this process is complex and error prone.

These limitations shift filtering from being perceived as a powerful worthwhile tool, to being a difficult and inconvenient chore.

Perhaps a solution to this problem would be to allow users to use direct manipulation and programming-by-example techniques to define their filters. After all, it is most often the case that a need for filtering becomes apparent after an evaluation has been displayed. Ideally the user should be able to select an expression for which filtering is required, and then define a filter in terms of the value of the selected sub-expression. The user would need to provide information about: the properties of the expression which make it a candidate for filtering (e.g. the expression is the application of a tree constructor); the constant features of the new representation of the sub-expression (e.g. it should be a graph with a node marked tree as its root); and the variable parts of the expression (e.g. the arcs of the root node should lead directly to the leaves of the tree).

Of course the example given above is rather simple. An example of a more complex filter would be one which when applied to a tree structure containing numbers, presented the elements of the tree as the sum of all of its elements. Solving a problem such as this would take the system from the realm of program visualisation into that of visual programming. While this is an interesting proposition, it is well beyond the scope of the work described here, and probably even beyond the scope of this “future work” section.

Another alternative to the definition of filters which support abstraction introduced by Abstract Data Types would be to incorporate a filter description into the definition of the ADT. A similar approach to this already exists in Miranda, where the user must
define the way a value of an abstract data type will be printed by the Miranda system should an evaluation result in a value of that type.

9.1.3 Searches

The clearest area in which work has to be carried out on searches, is that the detailed design presented in this thesis should be implemented. Until the implementation work has been carried out there is no way of verifying the usefulness of the range of searches discussed in Chapter 6. It would be interesting to see whether the risk of introducing strictness into an evaluation is compensated by earlier success of searches. While it is possible to theorise about this, and to construct examples on paper, there is no replacement for testing out the different search strategies on real-world examples.

In a similar way to the definition of filters, the design of the search system is based upon the definition of searches using Miranda syntax. Again, this has the obvious drawbacks when dealing with novice users. A simplified method of defining the start and search conditions would have to be provided for the system to be a success.

9.1.4 Interface

A number of improvements to the interface of Prospero could be made. Some of these are described in this section.

More Adventurous Graphics

As it stands, Prospero is not a strong case for the use of a graphical presentation of evaluations. The interface could be extended to utilise layout techniques which are much more advanced than the basic hierarchical approach used throughout this thesis. Colour, shading and highlighting could be used to clarify and highlight properties of a presentation. Icons and shapes could be used to differentiate between different elements of a display. None of these facilities have been called upon in the work presented here. However, I feel that it is the availability of these facilities which is the strongest argument in favour of a graphical rather than textual approach to presenting evaluations.
Explanation of redex and reduction rules

The current implementation of Prospero provides no feedback to users about the choice of reduction rule applied at each reduction stage. An improvement to the interface would be to provide more information about which sub-expression was being rewritten and which rewrite rule was being applied.

Spatial Coherence

The simple graph layout algorithm used to display reduction stages does not guarantee to place sub-expressions which have a lifetime of more than one reduction stage, in the same place on the display during those reduction stages. A more advanced display algorithm which maintained the position of sub-expressions wherever appropriate would provide more coherence between the reduction stages, saving the user form ‘reading’ the whole display in order to parse the meaning of the displayed information.

Evaluation Stages in Context

A problem faced by users of Prospero’s search system would be akin to the ‘lost in hyperspace’ problem faced by users of hypertext systems (Nielsen ’90). The result of a search is to present a single stage of an evaluation with very little information about the context in which that evaluation stage occurs.

The only contextual information available is the information present in the evaluation history. This information could be used in various ways to provide an elided history of an evaluation stage to the user of the system.

An alternative approach to this problem is that taken by the designers of the Transparent Prolog Machine (Eisenstadt & Brayshaw ’87) where the user is never allowed to jump to an evaluation stage, instead all intermediate stages are displayed in a similar style to the ‘search’ facilities available on video recorders.

9.1.5 Improvements to the Implementation of the Interpreter

At present the implementation of Prospero is far too inefficient to be usable on all but the simplest of problems, such as those used throughout this thesis. There are two techniques which could be used to improve the scalability of the system.
Compilation of Hidden Evaluation

Prospero's filter system offers the user a powerful facility with which to remove detail about the evaluation of specific sub-expressions in an evaluation. If the evaluation of a sub-expression is never seen by the user, then there is no need to produce the large amounts of information normally produced by Prospero. It follows that more efficient techniques could be used to evaluate the sub-expression; the more efficient evaluation technique need only produce the correct result.

A simplification of this idea is used in the TPM system:

The user can instruct TPM to throw away information which would otherwise allow retroactive expansion of compressed nodes. This option allows the user to benefit from increased execution speed (since 'compressed' goals can simply be proved without additional tracer history storage overheads).

(Eisenstadt & Brayshaw '87)

Implementation Using a Functional Language Compiler

Prospero is a lazy functional language interpreter implemented using a lazy functional language interpreter. The speed at which Prospero evaluates expressions could be improved by at least an order of magnitude (Seward '92) by re-implementing Prospero using a modern day lazy functional language compiler. Note that, at the time of writing, I know of no such compiler for Miranda.

9.1.6 Scalability

In this section I will explicitly cover implementation factors which will directly influence the scalability of Prospero.

Filtering

Large scale problems will result in potentially large volumes of information. Part of the rationale behind the development of Prospero's filter system was to deal with exactly this problem. Filters are there to control the information presented to the user. I have shown in this thesis that the elision of large data structures, the removal of low-level evaluation detail, and the removal of evaluation detail of specific
expressions, are all attainable with filtering. This will directly improve the scalability of Prospero.

The preparation of a ‘library’ of pre-defined filters would also help with the scalability and usability of Prospero. Many useful filters can be created, which are independent of any specific set of definitions. (e.g. a filter which elides large lists, and filters which remove the evaluation detail of all functions in Miranda’s standard environment).

**Searches and Evaluation History**

There are no reasons why the search system described in this thesis will not scale up to large problems. Nor is there any reason that such a search system will fail to handle a large number of search conditions being active for the same evaluation.

One possible problem faced when dealing with larger evaluations, is that of very long evaluation histories. A simple fix to this is to set the maximum length of the evaluation history, and to remove stages from the beginning of the history, as more stages are added to the end. This was the solution adopted by the implementors of the Miranda calculator (Goldson ’93a). While this solution is a straightforward one, it is not completely satisfactory, as it restricts the information available to the user. An alternative future direction would be to investigate the possibility of compressing the history, for example it may be possible to record only the changes between evaluation stages, rather than all of the information about each stage.

**9.1.7 Future Work Summary**

In order to offer Prospero as a system which could be practically used to develop definitions written in Miranda, and to teach lazy functional programming, a number of improvements would have to be made to the system as it is described in this thesis. Alongside these improvements there is a need to carry out experimental evaluation of the system, either as it stands, or following the improvements described here.

The main improvements described are to the filter facility, the search facility, the system’s interface, and the system’s language interpreter. The filter and search system must both shift away from the need for users to use Miranda to interact with these facilities. The interface could be enhanced by providing more explanatory and contextual information, and by making fuller use of the potential of modern bit-mapped displays. The language interpreter must be made more efficient.
9.2 Concluding Remarks

Prospero has served as a framework with which to test a number of ideas on the production of information about lazy evaluations. As it stands it should be viewed as a foundation for further work on the efficient implementation of a system based on the ideas put forward here, and for the testing of these ideas in the field. This research has been fruitful in that it has produced an insight into the problems of providing information to the user of lazy functional languages, and has produced a set of novel solutions to these problems.

I conclude with a summary of the contributions of the thesis:

- the errors faced by users of a lazy functional language, and the needs of students of these languages have been reviewed;
- a comprehensive review of approaches to providing information about lazy evaluation of definitions, and systems based on these approaches has been presented;
- the usefulness of a system based on rewriting has been demonstrated;
- a case for the presentation of low-level information has been made, together with a description of a low-level model based on an enriched lambda calculus;
- a system based on this model has been implemented;
- facilities with which to build various types of abstract views of an evaluation have been described and implemented;
- a search system which allows the user to control the amount of potential strictness introduced into an evaluation has be described;
- a description of future directions of this work has been presented.
Hi,

As this the last year for the MSc HCI, we are holding resit exams at the end of this year instead of waiting until next year, as we normally do. I propose to organise resits as follows:

Wed 25th August: UE
Thurs 26 August: MMS
Fri 27 August: NLIS.

I hope this OK with everyone. For various reasons, the last week of August is more or less the only time available. But please let me know immediately if anyone has real problems with the proposed dates.

sylvia

Content-Type: text/x-vcard; charset=us-ascii;
   name="sylvia.vcf"
Content-Transfer-Encoding: 7bit
Content-Description: Card for Sylvia Wilbur
Content-Disposition: attachment;
   filename="sylvia.vcf"
APPENDIX A

The complete set of definitions for Figure 2.7

graph == ([vertex],[edge])
vertex == ([char],bool)
edge == ([char],[char],num)
tree == graph
is_old = False
is_new = True

search::graph->tree->tree

search g t
    = t , marked_new g = []
    = search g' t' , otherwise
where
    (g',t') = searchv (v,is_old) g t
    || marked_new returns name of first "new" vertex in graph
    || or empty list if there are non.
    marked_new ([],edges) = []
    marked_new (((v,is_old):vertices),edges) = v
    marked_new ((v:vs),edges) = marked_new (vs,edges)

searchv::vertex->graph->tree->(graph,tree)

searchv v g t
    = (g'',t')
where
    g' = mark_old vname g
    || ends of all edges starting at v
    v_list1 = [w | (v,w,n) <- all_edges; v = vname]
    sv_list1 = sort (mkset v_list1)
    fst (v,b) = v
    sv_list2 = sort all_vertices
intersect \( (x : xs) \) \( (y : ys) \) result
\[
= \text{intersect} \ xy \text{ result} \ (x) \ , \ (\text{fst} \ x) = y
= \text{intersect} \ (x:xs) \ y \text{ result} \ , \ (\text{fst} \ x) > y
= \text{intersect} \ xs \ (y:ys) \text{ result} \ , \ (\text{fst} \ x) < y
\]

intersecxt \( x \ y \) \( r = r \ , \ x = [] \lor y = []\)
[ends of all edges starting at \( v \) with their old/new flags]

vertex_list = intersect sv_list2 sv_list1 []

\((g'',t') = \text{searchv'} \ \text{vertex_list} \ v \ g' \ t\)
[marks vertex \( v \) old]

mark_old \( v \) \((\langle \langle v, b \rangle : vs), es \rangle) = \((\langle \langle v, is \_ old \rangle : vs), es \rangle)
mark_old \( v \) \((\langle \langle v', b \rangle : vs), es \rangle)
\[
= \((\langle \langle v', b \rangle : vs'), es'\rangle)
\]
where

\((vs', es') = \text{mark_old} \ v \ (vs, es)\)

(vname, new) \ = \ v
(all_vertices, all_edges) \ = \ g'

searchv'::[vertex]->vertex->graph->tree->(graph, tree)

searchv' \( (w:ws) \ v \ g \ (te, tv)\)
\[
= \text{searchv'} \ w \ v \ g' \ (te', (\langle (\text{vname}, \text{vname}, 0) : tv' \rangle)) \ , \ \text{new} \ \text{wname} \ g
= \text{searchv'} \ w \ v \ g \ (te, tv) \ , \ \text{otherwise}
\]
where

\((\text{vname}, \text{v}_{\_ old}) = v\)
\((\text{wname}, \text{w}_{\_ old}) = w\)
\((g', (te, tv')) = \text{searchv} \ (\text{wname}, \text{True}) \ g \ (te, tv)\)

searchv' [] \ v \ g \ t = (g, t)

new::[char]->graph->bool

new v \((\langle (v, b) : vs), es \rangle) = b\)
new v \((\langle v' : vs), es \rangle) = \text{new} \ v \ (vs, es)\)

test_graph = \((\langle \langle "b", \text{True},\rangle,\)
\(\langle "a", \text{True}, \rangle,\)
\(\langle "c", \text{True}, \rangle,\)
\(\langle "a", "b", 0, \rangle,\)
\(\langle "a", "c", 0, \rangle,\)
\(\langle "b", "c", 0, \rangle,\))\)

- 246 -
APPENDIX A

The complete set of definitions for Figure 2.7

graph == ([vertex],[edge])
vertex == ([char],bool)
edge == ([char],[char],num)
tree == graph
is_old = False
is_new = True

search::graph->tree->tree

search g t
    = t , marked_new g = []
    = search g' t' , otherwise
        where
            (g',t') = searchv (v,is_old) g t
            || marked_new returns name of first "new" vertex in graph
            || or empty list if there are non.
            marked_new ([],edges) = []
            marked_new (((v,is_old):vertices),edges) = v
            marked_new ((v:vs),edges) = marked_new (vs,edges)

searchv::vertex->graph->tree->(graph,tree)

searchv v g t
    = (g'',t'')
        where
            g' = mark_old vname g
            || ends of all edges starting at v
            v_list1 = [w | (v,w,n) <- all_edges;v = vname]
            sv_list1 = sort (mkset v_list1)
            fst (v,b) = v
            sv_list2 = sort all_vertices
Appendix A

intersect (x : xs) (y : ys) result
    = intersect xs ys (result ++ [x]) , (fst x) = y
    = intersect (x:xs) ys result , (fst x) > y
    = intersect xs (y:ys) result , (fst x) < y

intersect x y r = r , x = [] \lor y = []
[| ends of all edges starting at v with their old/new flags
vertex_list = intersect sv_list2 sv_list1 []
(g',t') = searchv' vertex_list v g' t
[| marks vertex v old
mark_old v (((v,b):vs),es) = (((v,is_old):vs),es)
mark_old v (((v',b):vs),es)
    = (((v',b):vs'),es')
    where
        (vs',es') = mark_old v (vs,es)

(vname,new) = v
(all_vertices,all_edges) = g'

searchv'::[vertex]->vertex->graph->tree->(graph,tree)
searchv' (w:ws) v g (te,ty)
    = searchv' ws v g' (te',((vname,wname,0):tv')) , new wname g
    = searchv' ws v g (te,ty) , otherwise
    where
        (vname,v_old) = v
        (wname,w_old) = w
        (g',(te',tv')) = searchv (wname,True) g (te,ty)

searchv' [] v g t = (g,t)

new::[char]->graph->bool
new v (((v,b):vs),es) = b
new v ((v':vs),es) = new v (vs,es)

test_graph = ((("b",True),
    ("a",True),
    ("c",True)),
    ("a","b",0),
    ("a","c",0),
    ("b","c",0)))
APPENDIX B

Current Implementation State

1. Language Implementation

Prospero offers a complete implementation of Miranda with the exception of the following facilities:

- ZF notation
- .. notation
- The system functions force, seq, system
- $+$ for input
- library mechanism
- abstract data types
- repeated variables in patterns

2. Filter System

At present the filter system offers only ‘simple’ filters. The ‘temporal’ filters described in chapter 5 are partially developed.

3. Navigation

At present the user is not offered the facility of carrying out a backwards step through the reduction history unless no filtering is being use, i.e. the user may only move backwards by a number of atomic reduction steps.

No part of search system described in Chapter 6 is implemented.
APPENDIX C

Introduction to Miranda

This appendix presents a brief introduction to the sub-set of the Miranda programming language used in this thesis. A list of the parts of the language which lay outside this sub-set can be found in Appendix B.

This introduction is aimed at the reader with some knowledge of functional programming, but with no knowledge of Miranda.

Writing Definitions

Miranda 'programs' are writing using sets of equations. The simplest type of equation available binds an identifier to a value. Examples of such equations are:

\[
\begin{align*}
  v &= 10 \\
  \pi &= 3.14 \\
  b &= \text{True}
\end{align*}
\]

The above set of equations forms a legal Miranda script. This illustrates the minimalist approach to Miranda programming: no type declarations are made here, and the equations are delimited using white-space.

The values on the right of the above equations are all examples of the pre-defined data types provided by Miranda. These data types include:

- `True` || booleans
- `1` || numbers
- `'c'` || characters
- `"text"` || strings
- `[1,2]` || lists
- `(1,True,"no")` || tuples

Tuples have a static size, may contain elements of mixed type, and are non-subscripted. Lists are dynamic data-structures which must contain elements of the same type. These lists are subscripted starting from 0. The above list could be represented using the list constructor operator `:` as follows

\[
1 : 2 : []
\]
Appendix C - Introduction to Miranda

**Defining Functions**

Functions in Miranda are first-class, i.e. they can be passed as arguments to other functions, and can be returned as the result of an evaluation.

Function definitions are created using equations. The function name and the names of its arguments are listed, followed by the definition of the function. Examples of such equations are given below:

\[
\text{sum } a \ b = a + b \\
\text{compile } \text{stream} = \text{code_generate } (\text{parse } (\text{lex } \text{stream}))
\]

Functions may be partially applied to their arguments, thus \text{sum } 1\ is a valid value in Miranda and represents a function which when applied to a number will evaluate to the result of incrementing the argument.

A function definition may involve more than one clause, these clauses are disambiguated using Miranda's pattern-matching mechanism. When applied to an argument the clauses of the function are matched against the argument in the order in which they are defined in the script. For example, if the function defined as:

\[
\text{fac } 1 = 1 \\
\text{fac } n = n \times \text{fac } (n-1)
\]

was applied to the argument 1, then the first clause of the definition would 'match' against the argument, otherwise the second 'catch all' clause would match and be used.

Another way of defining a function in terms of a number of clauses is by using guards. Guards allow a user to define a number of alternative clauses for a definition and to guard each of them using a condition. Each time a definition is evaluated, the guards are checked in order (top-to-bottom) until a guard is found which evaluates to True. If none of the guards evaluate to True then a run-time error occurs.

Guards are declared following the right-hand-side of a clause, they are separated from the clause using a comma. The keywords otherwise may be used to specify a final guard which always evaluates to True. An alternative definition of \text{fac} defined using guards would be:

\[
\text{fac } n = 1, n = 1 \\
\quad = n \times \text{fac } (n-1), \text{ otherwise}
\]
Appendix C - Introduction to Miranda

Evaluating Expressions in an Environment

Miranda is an interpreted system. The user can cause expressions to be evaluated by typing them at the interpreters command line. These expressions are evaluated in an environment of equations. As a default this environment contains a standard library of equations provided by the Miranda system. The contents of this library is described in the Miranda on-line documentation. The user may add equations to this environment, by writing a script containing a set of equations, and loading these equations into the system.

Additionally an environment local to a specific equation may be defined using the where keyword. For example:

\[
\text{val} = a + \text{inc } 3 \\
\text{where} \\
a = 10 \\
\text{inc } n = n + 1
\]

defines an equation with a local environment containing bindings for \(a\) and \(\text{inc}\).

Defining Types

Users may define their own types. Once again types are created using equations, with the type name and its value constructors equated using ::=. Value constructors always start with an uppercase letter, and may take zero or more arguments. Examples of user-defined types are:

\[
\text{num_tree ::= Leaf | Branch num_tree num num_tree} \\
\text{lexeme ::= Keyword [char] | Number num |} \\
\hspace{1cm} \text{Operator char | Variable [char]} \\
\text{fruit ::= Banana | Apple | Orange}
\]

Examples of values of these types are:

\[
\text{a_lexeme = Variable "a_var"} \\
\text{a_tree = Branch (Branch Leaf 3 Leaf) 4 Leaf} \\
\text{salad = [Banana, Apple, Banana, Orange]}
\]
Appendix C - Introduction to Miranda

Polymorphic types may also be defined by using the set of wild-cards (*, **, ***,...) to indicate polymorphism. For example a polymorphic version of the above tree would be defined as:

\[
\text{poly\_tree} ::= \text{Leaf} \mid \text{Branch} \text{poly\_tree} \times \text{poly\_tree}
\]
<table>
<thead>
<tr>
<th>Student ID</th>
<th>Name</th>
<th>Last Name</th>
<th>Country</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>970732031</td>
<td>Yeo</td>
<td>Keng Liang William</td>
<td>AA869</td>
<td>N</td>
</tr>
<tr>
<td>972469226</td>
<td>Quadros</td>
<td>Daryl Orvil</td>
<td>AJ115</td>
<td>N</td>
</tr>
<tr>
<td>971759732</td>
<td>Rehman</td>
<td>Fareed Mahmoud-Ur</td>
<td>BA514</td>
<td>N</td>
</tr>
<tr>
<td>960536053</td>
<td>Nasir</td>
<td>Kashif</td>
<td>BB186</td>
<td>N</td>
</tr>
<tr>
<td>960257515</td>
<td>Siddiqui</td>
<td>Navida</td>
<td>BB518</td>
<td>N</td>
</tr>
<tr>
<td>961147530</td>
<td>Alam</td>
<td>Shaha</td>
<td>BC637</td>
<td>N</td>
</tr>
<tr>
<td>961837044</td>
<td>White</td>
<td>Neil Peter</td>
<td>BD274</td>
<td>N</td>
</tr>
<tr>
<td>961851981</td>
<td>Omar-Shahza Saqibah</td>
<td></td>
<td>BD288</td>
<td>N</td>
</tr>
<tr>
<td>961886477</td>
<td>Ilperuma</td>
<td>Dinusha Tishan</td>
<td>BD318</td>
<td>N</td>
</tr>
<tr>
<td>962182198</td>
<td>Man</td>
<td>Wing Tak</td>
<td>BD582</td>
<td>N</td>
</tr>
<tr>
<td>962231418</td>
<td>Akhtar</td>
<td>Sadeeyah</td>
<td>BD658</td>
<td>N</td>
</tr>
<tr>
<td>962314490</td>
<td>Istaifi</td>
<td>Omar Sakandary</td>
<td>BD728</td>
<td>N</td>
</tr>
<tr>
<td>962364804</td>
<td>Olabode</td>
<td>Marianne Funmiola</td>
<td>BD766</td>
<td>N</td>
</tr>
<tr>
<td>962522512</td>
<td>Elmrabti</td>
<td>Jaber</td>
<td>BD857</td>
<td>N</td>
</tr>
<tr>
<td>961288305</td>
<td>Lee</td>
<td>Daehyung</td>
<td>BE153</td>
<td>N</td>
</tr>
<tr>
<td>962818172</td>
<td>Igbokwe</td>
<td>Adaora</td>
<td>BE200</td>
<td>N</td>
</tr>
<tr>
<td>963114066</td>
<td>Edeko</td>
<td>Oseyemen</td>
<td>BE451</td>
<td>N</td>
</tr>
<tr>
<td>963119821</td>
<td>Nwoye</td>
<td>Christopher Obiora</td>
<td>BE456</td>
<td>N</td>
</tr>
<tr>
<td>963520751</td>
<td>Lodge</td>
<td>Tom</td>
<td>BE642</td>
<td>N</td>
</tr>
<tr>
<td>963528724</td>
<td>Deasy</td>
<td>Daniel Gerard</td>
<td>BE657</td>
<td>N</td>
</tr>
<tr>
<td>962539923</td>
<td>Nathwani</td>
<td>Nirav</td>
<td>BF094</td>
<td>N</td>
</tr>
<tr>
<td>961248365</td>
<td>Patel</td>
<td>Raj</td>
<td>BF205</td>
<td>N</td>
</tr>
<tr>
<td>963607693</td>
<td>Patel</td>
<td>Imran</td>
<td>BH434</td>
<td>N</td>
</tr>
<tr>
<td>961724032</td>
<td>Haye</td>
<td>Trevor Richard</td>
<td>BH516</td>
<td>N</td>
</tr>
<tr>
<td>963032977</td>
<td>Ponnampallal Jeyakumar</td>
<td></td>
<td>BH596</td>
<td>N</td>
</tr>
<tr>
<td>953594489</td>
<td>Corkhill</td>
<td>Jamie Adam</td>
<td>BH873</td>
<td>N</td>
</tr>
<tr>
<td>964359486</td>
<td>Hook</td>
<td>Darren Lee</td>
<td>BJ469</td>
<td>N</td>
</tr>
<tr>
<td>952908247</td>
<td>Lovelock</td>
<td>Timothy Alexander</td>
<td>BJ607</td>
<td>N</td>
</tr>
<tr>
<td>952915898</td>
<td>Riaz</td>
<td>Farooq</td>
<td>BJ609</td>
<td>N</td>
</tr>
<tr>
<td>962521853</td>
<td>Luong</td>
<td>Chuong</td>
<td>BM075</td>
<td>N</td>
</tr>
<tr>
<td>962846359</td>
<td>Ejobe</td>
<td>Jiro</td>
<td>BM141</td>
<td>N</td>
</tr>
<tr>
<td>943418384</td>
<td>Arulrban</td>
<td>Krishnamoorthy</td>
<td>CD493</td>
<td>R</td>
</tr>
<tr>
<td>971931258</td>
<td>Youngjohns</td>
<td>Rosalind Anne</td>
<td>CD972</td>
<td>N</td>
</tr>
<tr>
<td>952729929</td>
<td>Iqbal</td>
<td>Atif</td>
<td>CE403</td>
<td>N</td>
</tr>
<tr>
<td>953247088</td>
<td>Khan</td>
<td>Ali Raza</td>
<td>CE468</td>
<td>N</td>
</tr>
</tbody>
</table>
March 2, 1999

The Research and Associate Student Office  
Queen Mary & Westfield College  
Mile End Road  
London E1 4NS  
UK  

TO WHOM IT MAY CONCERN  

I am writing this letter for supporting Mr. Bing Qian, my former student at Shanghai University of Engineering Science, in his applying for admission to your esteemed school as a doctoral student.

Mr. Qian came to my university and took my course of Marketing Management in the academic year from 1995 to 1996 when he was a senior student in his university. During his study, he showed me a strong willingness to learn and comprehend. He attempted seriously to sort out for himself a clearcut, functional proposition from seemingly tangled strings of theoretical ideas. Moreover, with an inherent curiosity, he frequently joined discussions with his classmates, raising stimulating and penetrating questions, displaying himself a keen observer and an incisive personality. Though his grade may not be the highest, his creativity and imagination, compared with most of his peers, impressed me deeply. After finishing his undergraduate thesis titled MARKETING OF INTEREST RATE OF CAPITAL IN CHINA he gave it to me to review. In the thesis he widely studied the experiences of western developed countries and presented his own viewpoint and proposals. In my opinion his proposals have important meaning in theory and reality for China. So he has showed his larger calibre in doing research work then. And Mr. Qian is one of the most talented and hardworking students I have encountered during my more than 30 years’ teaching career.

After graduating from Shanghai University of Engineering Science, he enrolled in the University of Liverpool in Britain for further study of his master’s degree. I believe he must have made more progress in his ability and intelligence. So, I have no doubt about his scholastic capacity for further academic growth, therefore I recommend him to your university without reserve. Should any additional information be needed, please let me know at your earliest convenience. I can be reached as listed above.

Sincerely yours

Xu, Bai Quan  
Professor  
Vice Dean of Management School  
Shanghai Jiaotong University
Presenting the Lazy Evaluation of Functions

Jonathan Paul Taylor

June 1996