University of London

Imperial College of Science, Technology and Medicine

Department of Computing

Automatic Methods for Program Transformation

by

CHIN Wei Ngan

This thesis is submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy (PhD) and the Diploma of the Imperial College (DIC)
Submitted - March 1990
Examined - May 1990

Abstract

The transformational approach to software development is recognised as an important formal route for software construction. One major benefit of this approach is the possibility of providing machine assistance to this development process. This thesis works towards this goal by systematizing four major classes of transformations into automated methods (or tactics).

The first class of transformations involves the *fusion* of *composed* expressions to eliminate unnecessary intermediate data structures and function calls. A producer-consumer model of functions is introduced to explain this transformation. With this model, we show how the *deforestation* algorithm of Wadler can be extended to *all* first-order programs. The extended transformation algorithm is presented, termination proof given and further improvements suggested. A similar consideration is also developed for compositions in set abstractions.

The second class of transformations involves the removal of certain *higher-order features* from *well-typed* programs. Three transformation algorithms are developed. Two of them are directly concerned with the removal of higher-order features. Termination proofs for these algorithms are given. A third algorithm preserves *full laziness* in a manner compatible with higher-order features removal. With these transformations, the extension of deforestation to higher-order programs is also developed.

The third class of transformations concerns the removal of redundant computation through *tupling*. A survey of past techniques is presented, followed by the development of a new analysis technique to discover *eureka* tuples to remove redundancy. This new technique makes novel use of *selection orderings* to search for eureka tuples. We give some methods to determine appropriate orderings and provide extensions to the basic analysis technique.

The last class of transformations involves the use of *constraints* (or *invariants*) to help improve recursive programs. Both the finite differencing tactic and a new base-case filter promotion tactic are part of this category of transformations. The necessary laws and semantic conditions for these two tactics are systematized.

Acknowledgements

The author would like to acknowledge the support received from the following parties:

National University of Singapore for generously providing the financial support and study leave necessary to pursue this work.

John Darlington for the supervision, insight and guidance necessary to bring my research to fruition.

Phil Wadler for inspiring much of the work done in Chapter 3 and 4.

Chris Hankin for clarifying my thoughts on abstract interpretation

David Lillie, Guido Jouret and David Sharp for acting as sounding boards and verifiers for some of my late ideas. D Lillie and G Jouret also provided invaluable feedback on the thesis.

Peter Harrison, Tony Field, David Sands, Hessam and Paul Kelly for carefully reading and commenting on various parts of this thesis

Helen, Keith, Nigel and Lyndon for bailing me out of the technical intrigues of programming HOPE+ and the transformation-based environment

The Rest of Functional Programming Section for contributing to a most stimulating place for research.

Bee Hoon for her understanding and encouragement.

Dedications

To Mum and Dad

Table of Contents

1. Introduction **1**

1.1. The Software Problem 1
1.2. A Suitable Language for Specification and Transformation 2
1.2.1. An Extended Functional Language 3
1.2.2. Types of Abstractions Available 4
1.2.2.1. Data Abstraction 4
1.2.2.2. Program Abstraction 5
1.2.2.3. Specification Abstraction 5
1.2.2.4. System Abstraction 7
1.3. A Suitable Program Transformation Methodology 7
1.3.1. Catalogue versus Generative Set Systems 8
1.3.2. Desirable Criteria for Transformation Systems 9
1.3.3. The Unfold/Fold Methodology 10
1.3.4. Correctness 12
1.3.5. Completeness 13
1.4. Contribution of Thesis 14

2. Automatic Methods **18**

2.1. A Pitfall to Avoid 18
2.2. A Comprehensive Transformation System 19
2.2.1. Transformation Processor and the Script Language 20
2.2.2. Program Repository 22
2.2.3. Lemma Library 22
2.2.4. Inference Sub-System 23
2.2.5. Methods or Tactics Bank 24
2.3. Common Structure of Tactics 25
2.4. An Algorithmic Tactic: Removing Useless Parameters 26
2.4.1. A Simple Functional Language 26
2.4.2. Phase 1: Analysis for Useless Parameters 27
2.4.3. Phase 2: Transformation 30
2.4.3.1. Transformation Algorithm 30
2.4.3.2. An Example 32
2.4.3.3. A Version of $\mathcal{U}$ which Carries the Definition Set 33
2.4.3.4. A Version of $\mathcal{U}$ which Returns Scripts 34
2.5. A Schematic Tactic: Conversion-to-Iteration 35
2.5.1. Constructing a Schematic Rule 36
2.5.2. Applying the Schematic Rule 38
2.5.3. The Stages for Rule Applications 41
2.5.4. Wider Applicability of our Approach 42
2.5.5. Benefits of Schematic Rules 43
2.6. Summary of Chapter 2 44

3. Fusion **45**

3.1. Introduction 45
3.1.1. Two Forms of Composition 45
3.1.1.1. Functional Composition 45
3.1.1.2. Set Abstraction 47
3.1.2. Consumer-Producer Model 49

3.2. Wadler's Deforestation 50
3.2.1. Pure Treeless Form 51
3.2.2. Deforestation Theorem 52
3.2.3. Transformation Algorithm 53
3.2.4. Making all Steps Explicit 54
3.2.5. Proof for Deforestation Theorem 55
3.2.6. Blazed Treeless Form 59
3.3. Our Extension to Deforestation 61
3.3.1. Parameter-Based Annotation Scheme 62
3.3.1.1. Variable-Only Criterion 62
3.3.1.2. Linearity Criterion 63
3.3.1.3. Annotation Scheme 64
3.3.2. Extended Treeless Form 65
3.3.3. Extended Deforestation Theorem 66
3.3.4. Transformation Algorithm 67
3.3.5. An Example 71
3.3.6. Proof for Extended Deforestation Theorem 72
3.3.7. A Minor Improvement to Extended Deforestation 82
3.4. Deforestation for First-Order Expressions 83
3.4.1. Universal Treelessness 84
3.4.2. Proof of Universal Deforestation Theorem 86
3.4.3. Converting Functions to Universal Treeless Form 87
3.4.3.1. Bottom-Up Conversion Strategy 88
3.4.3.2. Conversion for Non-Recursive Functions 88
3.4.3.3. Conversion for Recursive Functions 89
3.4.4. Example of Universal Treeless Conversion 90
3.5. Further Improvements to Deforestation 92
3.5.1. Eliminating Unnecessary Mutual Recursive Functions 93
3.5.2. Linearising Pattern-Matching Parameters 94
3.5.3. Laws to Improve Deforestation 95
3.6. Supercompilation - A Related Work 96
3.7. Fusion in Set Abstractions 97
3.7.1. Filter Promotion 98
3.7.1.1. Desirable Filter Form 98
3.7.1.2. Desirable Generator Form 99
3.7.1.3. Transformation Algorithm 100
3.7.1.4. An Example 102
3.7.1.5. Undesirable Expression Forms 104
3.7.1.5.1. Undesirable Generators 104
3.7.1.5.2. Undesirable Filters 105
3.7.1.6. An Extension (Patterns instead of Variables) 107
3.7.2. Fusions between Generators and Processors 109
3.7.2.1. Desirable Processor Form 109
3.7.2.2. Transformation Algorithm 110
3.7.2.3. An Example 111
3.8. Summary of Chapter 3 112

4. Higher-Order Removal 114

4.1. Introduction 114
4.1.1. A Simple Higher Order Language 115
4.1.2. Overview of General Application Elimination 116
4.1.3. Overview of Higher-Order Argument Elimination 117
4.1.4. Overview of Full Laziness Technique 119
4.1.5. Overview of Fully Lazy Higher-Order Argument Elimination 121
4.1.6. Outline of Stages for the Higher-Order Removal Tactic 122

4.2. Elimination of General Applications 124
4.2.1. Transformation Algorithm, $\mathcal{A}$ 124
4.2.2. An Example 127
4.2.3. Proof of Theorem 4.2 128
4.2.3.1. Proof for Lemma 4.3 (on applyless form) 128
4.2.3.2. Proof for Lemma 4.4 (on efficiency) 128
4.2.3.3. Proof for Lemma 4.5 (on termination) 128
4.3. Elimination of Higher-Order Arguments 130
4.3.1. Higher-Order Specialised Form 131
4.3.2. Transformation Algorithm, $\mathcal{R}$ 131
4.3.3. A Simple Example 135
4.3.4. An Optimisation for the Nested Rules 135
4.3.5. Further Examples 136
4.3.5.1. mapreduce 136
4.3.5.2. twice_map 137
4.3.5.3. fix-point operator and $\mathcal{R}$ 12 137
4.3.6. Proof of Theorem 4.7 139
4.4. Full Laziness Concern 145
4.4.1. Various Techniques for Full Laziness 146
4.4.1.1. $\mathcal{L}$(a) Extract out GT-MFEs from lambda abstractions. 146
4.4.1.2. $\mathcal{L}$(b) Convert grounded GT-MFEs to CFs 146
4.4.1.3. $\mathcal{L}$(c) Direct Unfolds for non-recursive function calls 147
4.4.1.4. $\mathcal{L}$(d) Indirect Unfolds by Re-Defining non-recursive functions. 147
4.4.1.5. Recursive Functions and the Space Leak Problem. 148
4.4.2. Full Laziness Transformation Algorithm 149
4.4.2.1. The $\mathcal{L}$ rules 149
4.4.2.2. The $\mathcal{E}$ rules 151
4.4.2.3. The $\mathcal{F}$ rules 153
4.4.3. Fully Lazy Elimination of Higher-Order Arguments 155
4.4.4. An Example 156
4.4.5. Possible Losses of Efficiency 157
4.4.5.1. FT-MFEs from g-type functions 158
4.4.5.2. FT-MFEs from recursive functions 158
4.4.5.3. Coping with the Loss of Full Laziness 159
4.4.6. Termination of the Full Laziness Transformation Rules 160
4.5. Deforestation for Higher-Order Expressions 161
4.5.1. Parameter Annotations for Higher-Order Deforestation 161
4.5.2. Higher-Order Treeless Form 162
4.5.3. Modifications to the Transformation Algorithm 163
4.5.4. An Example 165
4.5.5. Re-appearance of Higher-Order Features 165
4.5.6. Proof of Higher-Order Deforestation Theorem 167
4.6. Related Works 169
4.6.1. Restricted Higher-Order Facility 169
4.6.2. Translation to First-Order Programs using Explicit Closures 169
4.6.3. Partial Evaluation 170
4.7. Summary of Chapter 4 171

5. Tupling 173

5.1. Introduction 173
5.1.1. Memoisation and Tabulation 173
5.1.2. Tupling Transformation 174
5.2. Dependency Graph 176
5.2.1. Relevant Subsidiary Function Calls 177
5.2.2. Requirements of DG analysis 179
5.3. Past Analysis Techniques 179
5.3.1. Analysis using Algebraic Properties of Programs 179
5.3.2. Analysis using Dependency Graph 182
5.4. Pettorossi's Framework 183

5.5. Our Tupling Analysis Technique 185
5.5.1. An Example 186
5.5.2. Selection Orderings for Deterministic Analysis 187
5.5.3. AD-Based Selection Orderings 188
5.5.3.1. AD-Ordering 189
5.5.3.2. Descendant-Enhanced Ordering 191
5.5.3.3. A Characterisation for Recursion Structures of Function Calls 192
5.5.4. Function-Argument Selection Orderings 194
5.5.4.1. A Technique to Determine Function-Argument Orderings 195
5.5.4.2. Constructor-Based Ordering 197
5.5.4.3. Instantiate and Unfold Ordering 198
5.5.5. Examples 199
5.5.5.1. Example 1 - Common Generator 200
5.5.5.2. Example 2 - Periodic Commutative 200
5.5.5.3. Example 3 - Tower of Hanoi. 203
5.5.5.4. Example 4 - Mutual Recursions 204
5.6. An Extension to the Tupling Analysis Technique 205
5.6.1. Non-Linear Data Structure 205
5.6.2. Multiple Recursive Equations per Function 207
5.6.3. Simplification Rules 209
5.6.4. Logarithmic-Time Fibonacci 210
5.6.5. A Complex Linear-Recurrence Function 211
5.7. Limitations of Tupling Analysis 213
5.8. Summary of Chapter 5 214

6. Constraint-Based Transformations 215

6.1. Introduction 215
6.2. Finite Differencing 216
6.2.1. An Example 218
6.2.2. The Tactic 220
6.2.3. Phase 1: Analysis for Expensive Expressions 220
6.2.3.1. Identification of Expensive Expression 221
6.2.3.2. Maintenance Laws 222
6.2.3.3. Special Unchanged Laws 223
6.2.3.4. Generalisation to Multiple Maintenance Laws 223
6.2.4. Phase 2 : Transformation 224
6.2.5. Special Cases of Finite Differencing 226
6.2.6. Comments 227
6.3. Base-Case Filter Promotion 227
6.3.1. Continuous Filter Condition 228
6.3.2. Transformation 229
6.3.3. Incremental Filter Condition 231
6.3.4. Weaker Filter Condition 232
6.3.5. A More General Program Scheme 236
6.3.6. Comments 236
6.4. Keeping and Using the Invariants/Constraints 236
6.5. A Related Work - Global Search Algorithms 237
6.6. Summary of Chapter 6 239

7. Conclusion 240

7.1. Summary of Thesis 240
7.2. Underlying Computational Structures 241
7.3. Future Work 242

References 245

Appendix 1. Higher-Order Useless Parameter Eliminations 254

A1.1. Extension to Analysis Phase 254
A1.2. Extension to Transformation Phase 255

Appendix 2. Other Conversion-to-Iteration Schemes 257

A2.1. Two-Pass Iteration with Data Stack 257
A2.2. Scheme with Function Inversion 260
A2.2.1. Improving the 2-Pass Iteration to a 1-Pass Iteration 261
A2.2.2. Avoiding the Equality Test on the Recursion Argument 262
A2.3. Associative Dual Scheme 264
A2.4. Combining the Schematic Rules into a Single Tactic 265
A2.5. A Problem with the Lazy Semantics 266

Appendix 3. Derivation of Parallel Lemmas 269

A3.1 Three-Stage Synthesis 270
A3.1.1 Stage 1: Symbolic Evaluation to obtain Comparable Equations. 270
A3.1.2 Stage 2: Matching and Second-Order Generalisation. 271
A3.1.3 Stage 3: Synthesis of Unknown Functions. 272
A3.1.4 Embedding Synthesis within the Unfold/Fold Framework 273
A3.2 Classes of Programs which are Synthesizable 273
A3.2.1 Scheme 1- simple homomorphic form 274
A3.2.2 Scheme 2 - undefined for the nil or null case 274
A3.2.3 Scheme 3 - without the identity element 275
A3.2.4 Scheme 4 - with accumulating parameter 276
A3.2.5 Scheme 5 - requiring generalised programs 278
A3.2.6 Scheme 6 - deeper pattern-matching 279
A3.2.7 Scheme 7 - tail recursive functions 280
A3.2.8 Scheme 8 - more complex semantic conditions 281
A3.3 Scope and Limitations of our Synthesis Method 283

Appendix 4. Implementation of Tactics 285

A4.1. Overview of the Partial Evaluator and Tactics. 285
A4.1. Description of Implemented Tactics 287
A4.2. Symbolic Tree for Transformation 287
A4.4. Low-Level Access Functions 289

Appendix 5. A Termination Proof 292

A5.1. Pure Deforestation Termination Proof 292
A5.2. An Additional Proof Step for the Modified Rules 293

List of Figures

Figure 1.1 - Overview of Thesis 17
Figure 2.1 - Components of a Comprehensive Program Transformation System 19
Figure 2.2 - Translation Rules for Useless Parameter Analysis 29
Figure 2.3 - Transformation Rules for Eliminating Useless Parameters 31
Figure 2.4 - Modified Rules to Carry the Definition Set, ds 34
Figure 2.5 - Modified $\mathcal{U}$ Rules to Return Transformation Scripts 35
Figure 2.6 - Transforming a Generic Function to its Tail-Recursive Equivalent 37
Figure 2.7 - Function Equivalence Matching (FEM) between length and f1 40
Figure 3.1 - Fusion Transformation on sumdb 46
Figure 3.2 - Filter Promotion on the function all_path 48
Figure 3.3 - Deforesting a Simple Expression with $\mathcal{T}$ 52
Figure 3.4 - Wadler's Deforestation Rules 54
Figure 3.5 - Modified Rules to Make All Steps Explicit 55
Figure 3.6 - A Simple Example of Extended Deforestation 67
Figure 3.7 - $\mathcal{T}$ Rules for Extended Deforestation Algorithm 71
Figure 3.8 - Another Example of Extended Deforestation 72
Figure 3.9 - Proof of Lemma 3.14.2 (**gen_ren** produces smaller terms) 78
Figure 3.10 - Proof of Lemma 3.14.1($\mathcal{N}$ is bounded) 79
Figure 3.11 - Proof of Lemma 3.14.3 (eft{Ω} is preserved) 81
Figure 3.12 - Proof of Lemma 3.14.5 ($\mathcal{S}$ is bounded by $\mathcal{N}$ for all generalised terms) 82
Figure 3.13 - An Unsuccessful Application of $\mathcal{T}$ on a mix of treeless and non-treeless functions 83
Figure 3.14 - Two Additional Rules to Deal with the let construct 86
Figure 3.15 - Additional Proof of Lemma 3.14.1 to show that $\mathcal{N}$ is bounded by the new Universal Deforestation Rules 86
Figure 3.16 - Algorithm for Bottom-Up Conversion 88
Figure 3.17 - Rules to Deal with Non-Treeless Function Calls 90
Figure 3.18 - Additional Proof for Lemma 3.14.1 to show that it holds for Rules $\mathcal{T}$ 11 and $\mathcal{T}$ 12 90
Figure 3.19 - Converting Function rev_flatten 91
Figure 3.20 - Converting Function main 92
Figure 3.21 - $\mathcal{G}$ Rules for Filter Promotion 101
Figure 3.22 - An Example of Filter Promotion with the $\mathcal{G}$ Rules 103
Figure 3.23 - More General $\mathcal{G}$ rules which accept Complex Pattern Link 108
Figure 3.24 - Additional Rules for the Fusion of Generators and Processors 110
Figure 3.25 - An Example to Illustrate the Fusion of Generators and Producers 112
Figure 4.1 - Complete Transformation Stages for the Higher-Order Removal Tactic 123
Figure 4.2 - Rules for the Elimination of General Applications 126
Figure 4.3 - Successful Application of $\mathcal{A}$ to a Recursive Function 127
Figure 4.4 - Transformation Rules for Higher-Order Argument Eliminations 134
Figure 4.5 - A Simple Example for $\mathcal{R}$ 135
Figure 4.6 - Application of the Optimised $\mathcal{R}$ Rules to the map function 137
Figure 4.7 - Applying the $\mathcal{R}$ Rule to inf_int' 138
Figure 4.8 - A Looping Transformation Without the Inner Rules of $\mathcal{R}$ 12 139
Figure 4.9 - Proof that the Measure, S_T, is Bounded by All the Inner $\mathcal{R}$ rules 142
Figure 4.10 - Full Laziness Algorithm, $\mathcal{L}$, for Unfolding and Extraction 150
Figure 4.11 - $\mathcal{E}$ Rules for the Repeated Extraction of GT-MFEs 151
Figure 4.12 - $\mathcal{F}$ Rules to Find both Explicit and Implicit GT-MFEs 154
Figure 4.13 - Additional Sub-Rules for Fully Lazy Higher-Order Arguments Elimination 156
Figure 4.14 - Modified Rules for Higher-Order Deforestation 164
Figure 4.15 - New Rules for Higher-Order Deforestation 164
Figure 4.16 - Applying $\mathcal{T}$ to twice_map 165
Figure 4.17 - Deforesting a List of Functions 166
Figure 4.18 - Additional Proof of Lemma 3.14.1 to show that it holds for the New and Modified Rules of HO-Deforestation 168

Figure 5.1 - Compression of a Tree of calls to a DG. 177
Figure 5.2 - Symbolic DG for fib(n) 177
Figure 5.3 - DG for a Program with Common Generator Descent Condition 180
Figure 5.4 - DG for a Program with Commutative Descent Condition. 181
Figure 5.5 - Successive Frontiers (Cuts) During the Exploration of a DG 187
Figure 5.6 - Symbolic DG for factlist definition 189
Figure 5.7 - DG for factlist2 definition 190
Figure 5.8 - Descendant-Enhanced DG for factlist2 definition 191
Figure 5.9 - Different Recursion Structures of Function Calls 193
Figure 5.10 - DG for the Periodic Commutative Class of Programs as a Rectangular Array WHERE $e=d^m$ or $e=c^n$ 201
Figure 5.11 - A Continuous Tree of Cuts for dp(n) 206
Figure 5.12 - Tree of Cuts for a Multi-Recursive Cases Function 207
Figure 6.1 - Transformation Sequence for new_split 219
Figure 6.2 - Filter Promotion through Constraint for new_sub 230
Figure 6.3 - Filter Promotion using the Continuous Filter Condition 231
Figure 6.4 - Filter Promotion using the Incremental Filter Condition 232
Figure 6.5 - Filter Promotion using the Weaker Filter Continuous Condition 233
Figure 6.6 - Filter Promotion with two Weaker Filters 235
Figure A2.1 - Transforming to a Two-Pass Iteration with Data Stack 259
Figure A2.2 - Transforming to a Two-Pass Iteration without Data Stack 261
Figure A2.3 - Transforming to a Two-Pass Iteration without the Equality Test on Argument 264
Figure A2.4 - Transforming to Tail-Recursive Program with the Associative Dual Property 265
Figure A2.5 - An Integrated Conversion-to-Iteration Tactic 266
Figure A5.1 - Proof to Show that the Nesting Measure, **N**, is bounded 292
Figure A5.2 - Proof to show that the grammar form of fct is preserved 293
Figure A5.3 - Proof of Lemma A5.1 ($\mathcal{M}$ is well-founded) 295

List of Lemmas, Theorems, Definitions

Definition 3.1 - Pure Treeless Form 51
Theorem 3.2 - Deforestation 52
Lemma 3.3 - Adherence to (Pure) Treeless Expression Form 55
Lemma 3.4 - No Loss in Efficiency 56
Lemma 3.5 - Termination of Algorithm $\mathcal{T}$ 56
Definition 3.6 - Blazed Treeless Form 60
Definition 3.7 - Variable-Only Parameter Criterion 62
Definition 3.8 - Linearity Parameter Criterion 64
Definition 3.9 - A Parameter Annotation Scheme for Deforestation 64
Definition 3.10 - Extended Treeless Form 65
Theorem 3.11 - Extended Deforestation 66
Lemma 3.12 - Adherence to Extended Treeless Form 72
Lemma 3.13 - No Loss in Efficiency 72
Lemma 3.14 - Termination of Algorithm $\mathcal{T}$ 72
Lemma 3.14.1 - Nesting Measure, $\mathcal{N}$, is bounded by the $\mathcal{T}$ Rules 74
Lemma 3.14.2 - Each Nesting of Generalised and Extracted sub-terms is always less than the Source Expression 74
Lemma 3.14.3 - Each expression, during deforestation, always satisfies $\mathrm{eft}\{\Omega\}$ 75
Lemma 3.14.4 - Each Generalised or Extracted Subterm of $\mathrm{eft}\{\Omega\}$ always satisfies $\mathrm{eft}\{\Omega\}$ 76
Lemma 3.14.5 - Size, $\mathcal{S}$, of Generalised Expression is bounded by $\mathcal{N}$ 76
Definition 3.15 - Universal Treeless Form 85
Theorem 3.16 - Universal Deforestation 85
Definition 3.17 - Function Conversion 87
Theorem 3.18 - Universal Conversion 87
Lemma 3.19 - Conversion for Non-Recursive Functions 89
Lemma 3.20 - Conversion for Recursive Functions 89
Definition 3.21 - Desirable Filter Form 98
Definition 3.22 - Desirable Generator Form 99
Definition 3.23 - Desirable Processor Form 110
Definition 4.1 - Applyless Form 124
Theorem 4.2 - Elimination of General Applications 124
Lemma 4.3 - On Applyless Form 128
Lemma 4.4 - On Efficiency 128
Lemma 4.5 - On the Termination of $\mathcal{A}$ 128
Definition 4.6 - Higher-Order Specialised Form 131
Theorem 4.7 - Elimination of Higher-Order Arguments 132
Lemma 4.8 - On HO-Specialised Form 139
Lemma 4.9 - On Efficiency 139
Lemma 4.10 - On the Termination of $\mathcal{R}$ 139
Definition 4.11 - Annotation Scheme for Higher-Order Deforestation 161
Definition 4.12 - Higher-Order Treeless Form 162
Theorem 4.13 - Higher-Order Deforestation 163
Definition 5.1 - Dependency Graph 183
Definition 5.2 - Ancestor-Descendant Ordering 184
Definition 5.3 - Cut 184
Definition 5.4 - Progressive Sequence of Cuts 184
Definition 5.5 - Topmost Set 185
Definition 5.6 - Continuous Sequence of Cuts 185
Definition 5.7 - Dependency in a pair of Matching Tuples 193
Lemma A5.1 - Well-Founded Measure for $\mathcal{T}$ 2, 5 and 6a Rules 294

1. Introduction

1.1. The Software Problem

The past twenty years has seen another industrial revolution, this time of the information technology, which has been driven largely by the increase in sophistication and falling costs of computer hardware. Unfortunately, this rapid improvement in computer hardware technology has not been matched by a similar trend in software. Late schedules, over-run costs and suspect qualities are still features experienced in software projects. They are happening because software developers' productivity has not kept up with the demands of ever more complex tasks being entrusted to computer systems.

A fundamental cause of this problem is that there is an inherent *conflict* in the required characteristics of software. On the one hand, software must be specified in the *clearest* possible manner in order to aid in understandability, reasoning and modifiability. This objective is normally achieved by using various forms of high-level abstractions, e.g. abstract data types, weak algorithms[1] and modularized programs. On the other hand, software must be suitable for implementation by being *efficient* in its use of computer resources and be able to meet performance requirements.

These two characteristics, *clarity* and *efficiency*, cannot usually be satisfied simultaneously by a single piece of software. This is because high-level abstractions, like data abstraction, modularisation and weak algorithms, are constructs which make software easier to write and understand but are unfortunately inefficient to implement. Such constructs tend to be general and widely applicable. But for efficiency, we need different forms of programs which are specialised with respect to the way they are used. In particular, they have to exploit the specific contexts of programs for efficient implementation. Use of specialised contexts, as opposed to the generalised constructs, may improve efficiency but tend also to result in software codes that are more obscure, with higher interconnectivity between the various parts of the programs.

To harmonise both these conflicting requirements, many researchers have been advocating for a program transformation approach to software development [Darlington76,Burstall77, Green81,Bauer79,Balzer81]. In this approach, software is permitted to be specified using high-level constructs which are clear and easy to understand, without worrying about implementation concerns. After completing the software specification, it could be subjected to a sequence of *correctness-preserving* transformations that would gradually change the clear but inefficient specification into an equivalent program that is more efficient. Only correctness-preserving transformations should be used in order to guarantee that the transformed program will always preserve the intended meaning of its original specification.

The transformational approach represents an *earlier formalisation* of both the specification and design phases within the software life-cycle's *requirement*, *specification*, *validation*, *design*, *coding* and *testing* phases. Previously, the actual formalisation of software development occurs

[1] e.g. generate-and-test algorithms are weaker (and more general) than divide-and-conquer algorithms.

principally at the coding phase, with largely informal techniques used at the specification and design phases [Wirth71,Dijkstra76,Yourdon75]. *Earlier formalisation* in the software life-cycle can help to improve one fundamental problem of software, namely the difficulty of getting it correct with respect to users' requirement. This is so because:

(1) it is possible to *validate* system earlier through executable or verifiable specifications.

(2) high-level specifications are less likely to contain errors.

(3) correctness-preserving transformations (cf. design) will not introduce errors.

Such formalisations are very important because they open up opportunities for providing *machine assistance* in software development. These opportunities are the central theme for a proposal by Cordell Green, R Balzer et al [Green83], in the long-term, of providing Knowledge-Based Software Assistants (KBSA) which will put the machine at the centre of the software life-cycle activities. Possible areas where machine assistance could be provided include project management and supervision, acquisition and transformation of specifications, to automated facilities for testing, documentation and version control.

Our focus in this thesis is on the narrower but nevertheless important phase of transforming specifications to implementations. This phase has been viewed by many researchers as the key phase for large scale automation of the software construction process. However, the current state of art is presently some way from this goal of large scale automation.

This thesis contributes towards this goal by presenting a number of classes of transformations that can be automated. Each of these classes of transformations will be formalised into a transformation method, called *tactic*, which will be able to make appropriate use of analysis on program structures in order to suggest the corresponding transformations to improve the program. An overview of these tactics will initially be outlined in Section 1.4 before they are presented (in details) in Chapter 2 to 6. But first, we will address two preliminary concerns:

(1) a suitable language for specification and transformation (Section 1.2).

(2) a suitable methodology for program transformation (Section 1.3).

1.2. A Suitable Language for Specification and Transformation

The last decade has seen the development of a class of programming languages, called *declarative* languages, such as logic, functional, constraint and to some extent object-oriented languages[2], which are more suitable for writing software. Declarative languages represent another level of abstraction over imperative languages, like Pascal, Fortran, Modula. This is because they are more problem-oriented and possess better abstraction constructs for supporting the *specification of problems* rather than their *intricate procedural solutions*. In other

[2] Strictly speaking, object-oriented languages are not considered as declarative languages. However, their use of *local* rather than global states (plus corresponding *methods*) allow each object to be be viewed as a process (or function) which takes a list of requests and a state in order to return new state and replies to the requests (see (Perry89b]). In [Goguen87], a more direct formulation is used to combine the object-oriented paradigm with the functional paradigm. This involves the use of special modules to support objects, methods and classes. Both formulations allow objects to be integrated into the declarative paradigm.

words, declarative languages allow us to concentrate more on the 'what' rather than the 'how' of software and are increasingly recognised as a suitable way to specify software.

One very desirable mathematical property of declarative languages is *referential transparency*. This property makes declarative programs more amenable to mathematical reasoning and program transformation. The referential transparency property essentially states that each expression should denote the same value, wherever it may be used in a program. With this property, we can perform program manipulations by substituting one expression with another expression, which denotes the same value, and be sure of preserving the original meaning of the program. This is a basis by which safe manipulations of programs for transformations or proofs can be carried out.

The main reason why referential transparency is present in declarative languages, but not imperative languages, is that they do not have features like destructive assignment statements, which make expressions dependent on the computation history (or global memory state). Expressions that are sensitive to computation history are more difficult to manipulate safely.

1.2.1. An Extended Functional Language

One suitable declarative language for software specifications and transformational development is a functional language which has been extended to handle logic, constraint and object-oriented programming features. Such a language is currently under development at Imperial College, as a successor to HOPE+ [Perry89a,89b,Darlington89a][3]. The logic and constraint programming features permit very high-level specifications to be written, while the applicative/functional features allows efficient algorithms to be either written (by programmers) or derived (through transformations).

In a sense, this extended-functional language is a restricted form of the wide-spectrum language that have been used in a number program transformation systems e.g. CIP-L in CIP [CIP85,Bauer81], Anna/Ada in PROSPECTRA [KBruckner89], V in CHI [Phillips83]. A language is said to be wide-spectrum if it contains programming constructs from a range of programming styles. This spectrum of programming styles typically includes pre-algorithmic logic & set notations, applicative, imperative and even machine-oriented programming constructs. Wide-spectrum languages have been used in program transformation systems because they allow (localised) transformations to be applied, all the way, from high-level specifications to ultra-efficient implementations. Our extended language covers the *declarative-end* of this spectrum. It has been argued that this is quite sufficient because the gap between applicative and machine-oriented implementations can usually be carried out by automatic (optimising) compilation techniques.

Other functional languages, e.g Haskell [Haskell89], Miranda[4] [Turner85], Ponder [Fairbairn82], are also available. But as they do not possess logic and constraint-programming

---

[3] Hope was originally developed by Burstall, MacQueen and Sanella in Edinburgh [Burstall80]. It has since undergone major language extensions in Imperial College to include continuation-based I/O, absolute set abstractions and constraints.

[4] Miranda is a trademark of Research Software Ltd

features, we shall not use them in this thesis. These constructs are important because they are needed in certain classes of transformations (see Chapter 2 and 6, later).

1.2.2. Types of Abstractions Available

As mentioned earlier, our extended functional language can provide a number of high-level abstractions which are useful for specification purposes. Four kinds of abstractions that can be supported are:

- *data abstraction* to support common, re-usable abstract data types
- *program abstraction* to support common, re-usable programs
- *specification abstraction* to support high-level specifications using problem-independent algorithms or techniques
- *system abstraction* to support the programming of large systems with multiple components

We describe and illustrate the above abstractions next. In the process, we shall also be introducing the language constructs of HOPE informally.

1.2.2.1. Data Abstraction

There is an abstract way of defining data structures in HOPE through **data** declaration statements. These **data** statements can be used to define *enumerated* types, like week_days:

```
data week_days == mon ++ tue ++ wed ++ thur ++ fri ++ sat ++ sun;
```

More importantly, they can also be used to define complex *recursive data types*, such as list and tree, through *algebraic data type* statements, as follows:

```
data list(alpha)  == nil ++ alpha :: list(alpha);
data tree(alpha)  == leaf(alpha) ++ branch(tree(alpha),tree(alpha));
```

These recursive data types are defined using constructors, such as nil, ::, leaf and branch, which can be used to build large and complex data structures. They represent another level of abstraction over the dynamic data structures used in imperative languages, like Pascal of C. This is because implementation concerns, like allocation-deallocation of heap spaces and manipulations of pointers, will be automatically handled by the functional language compilers or interpreters.

Furthermore, the above type declarations are each parameterised with a type variable, alpha. This facility, known as *polymorphism*, is used to define generic data types - from which more specific types can be instantiated. Two examples of instantiated types are list of numbers and list of lists of numbers, as shown below.

```
type list_num     == list(num);
type list_list_num == list(list(num));
```

Polymorphism, algebraic data type notation and modules[5] can be used to define generic *abstract data types*, whose representations are hidden by a set of access functions. A classic example of this is the abstract data type for *stack*. This can be defined using the list structure as the concrete

[5] Module facility and separate compilation are supported by Hope+ (see [Sephton89]).

representation, together with a set of access functions, as declared below.

```
type stack(alpha)   == list(alpha)
dec empty           :  stack(alpha);
dec push            :  alpha # stack(alpha) -> stack(alpha);
dec top             :  stack(alpha) -> alpha;
dec pop             :  stack(alpha) -> stack(alpha);
```

These type declarations provide an interface for the abstract stack type, whose implementation is realised by a set of HOPE equations of the form, - - - LHS <= RHS, as shown below.

```
- - - empty          <= nil;
- - - push(a,stk)    <= a::stk;
- - - top(a::_)      <= a;
- - - pop(_::stk)    <= stk;
```

1.2.2.2. Program Abstraction

Apart from data abstraction, functional languages also support *program abstraction*. This abstraction can be used to support generic re-usable programs and is provided through polymorphism and the higher-order facility (or function parameterisation). An example of this is the following generic function for representing a class of binary divide-and-conquer algorithms.

```
dec d_c : alpha # (alpha -> (alpha # alpha)) # (beta # beta -> beta) # (alpha -> truval) # (alpha -> beta);
- - - d_c (x,divide,conquer,primitive,basic)
                    <= if primitive(x) then basic(x)
                       else let (a,b)==divide(x) in
                            conquer(d_c (a),d_c (b));
```

This abstract divide-and-conquer function can be re-used in a range of circumstances, with different instantiations applied to the function-type parameters, divide, conquer, primitive, basic.

More popular examples of *abstract programs* include those used for processing the recursive data structures. An example is the map function which produces an output list by applying its parameterised function, F, to each element of its given input list.

```
dec map : list(alpha) # (alpha -> beta) -> list(beta);
- - - map(nil,F)     <= nil;
- - - map(a::x,F)    <= F(a)::map(x,F);
```

Such abstract programs, using higher-order functions, are frequently used by functional programmers as highly re-usable software components. They provide the same kind of abstraction benefits to programs, as data abstractions provide for data structures.

1.2.2.3. Specification Abstraction

There is yet another form of abstraction, called *specification abstraction*, which uses high-level mathematical set and logic constructs to write software specifications. While program abstraction provides a facility for abstracting out commonly-used algorithms, specification abstraction stresses on weak algorithmic techniques that are problem-independent and more widely applicable.

As an example, consider how the sorting function can be specified in an abstract or declarative manner using the set abstraction, { p | p ∈ perm(x), ordered(p)}, as shown below.

```
dec sort , perm : list(num) -> set(list(num));
dec ordered : list(num) -> truval;
--- sort(x)          <= { p | p ∈ perm(x), ordered(p)};
--- perm(x)          <= if x=[] then { []}
                        else { a::p | a ∈ make_set(x), p ∈ perm(x - a)};
--- ordered([])      <= true;
--- ordered([a])     <= true;
--- ordered(a::b::x) <= a≤b ∧ ordered(b::x);
```

The above sorting specification states that a sorted output list, p, of input list, x, is a permutation of the original input (i.e p ∈ perm(x)) which is also ordered (i.e. ordered(p)). This specification is evaluated, as a generate-and-test algorithm, by generating all possible permutations using the perm function, before each of its possible solutions is tested by the ordered function.

Notice that we have managed to provide the above abstract specification through a set abstraction construct that is purely functional. This is so because each function call in the above set abstraction always takes in input values to produce outputs. Such a use of set abstraction is also known as the *relative set abstraction* or the *set comprehension* construct. It is now widely accepted as a standard feature of a number of modern functional languages, like HOPE+, Haskell and Miranda. However, in Miranda and Haskell, a variant of this construct, known as list comprehension, is used. This variant simply returns a list (rather than a set) which allows duplicate elements and has a positional order.

Apart from this purely functional usage, the set abstraction construct could also be used to support the more powerful predicate-logic language's *relations* (where any parameter field can act as either input or output). This use of set abstraction, known as *absolute set abstraction*, was first introduced in [Darlington86] as a means of integrating logic language feature into functional languages. It is achieved by allowing *functions* (in absolute set abstractions) to be evaluated backwards (i.e. given an output find its possible inputs) and thus behave as *relations*. An example of this is the following alternative specification for perm.

```
--perm([])      <= {[]};
--perm(a::x)    <= {p1<>[a]<>p2)) | p ∈ perm(x), p1<>p2 = p};
```

Here, we have an expression, p1<> p2 = p, making use of an append function, <>, which needs to be executed backwards in order to obtain various ways of splitting the p list into sublists, p1 and p2. Such backwards evaluation of functions is done through the narrowing mechanism, using both unification and rewriting (see [Pull90] for details). The ability to use functions in a multi-mode manner, like relations, provides for a more abstract and concise way of writing programs.

Recently, an even more general programming paradigm, called *constraint logic programming* scheme, was introduced by [Jaffar87]. This scheme is a proper generalisation of logic programming because Prolog II and III have been shown to be particular instances of this more general paradigm. In particular, the resolution mechanism of logic programming is just one of the problem-solving techniques which could be used in this more general paradigm. Other more

complex constraint-solving methods, e.g. linear equation solving, can also be used.

This more general paradigm has also been taken up in a proposed version of *constraint functional programming* [Darlington89a], through an even more general use of the set abstraction construct. Here, *constraints* are simply conditionals of the set abstraction construct, which could be evaluated with the help of appropriate problem-solving techniques. An example of this is the function, solve, for declaratively stating the solutions of two simultaneous linear equations, 3x+4y=5 and 7x+8y=8, as shown below

```
dec solve : void -> (num # num);
- - - solve() <= (x,y) st  3x+4y=5,7x+8y=8;
```

To evaluate the above function, we need a linear equations solver but this is again an implementation concern and not a concern of the programmer. Note the use of a special form of set abstraction, expr **st** $cond_1$,...,$cond_n$, which returns a singleton result, expr, satisfying the constraints, $cond_1$,...,$cond_n$.

1.2.2.4. System Abstraction

In [Shaw89], another form of abstraction, called *system abstraction*, is introduced for structuring and programming larger systems. For a while, software developers have been building systems using a number of popular abstract software architectures which are more suitable for structuring multi-component systems. Examples of these abstract system architectures include, the *layered hierarchical approach* for building operating and communications system, *pipes and filters* as is commonly used in UNIX, *data abstractions with inheritance* in object-oriented systems, *rule-based engines* for expert systems and *blackboard architecture* for multiple knowledge-base systems. These abstract system architectures are extremely suitable for structuring large software systems with multiple components.

Some of these system abstractions (e.g. layered approach, pipes and filters) can be directly supported by the hierarchical and compositional style of functional programs, while others (including object-oriented systems) can be embedded within the functional programming paradigm.

Closely related to this system abstraction capability, is the increasing realization that the expressiveness of functional languages can also be used to capture common program forms of different parallel computer architectures, e.g. SIMD, CSP, systolic, etc. This expressiveness can help to map (or transform) our programs directly to forms which are suitable for these parallel architectures and is the basis of a transformation-for-parallelism project currently under-way in Imperial College [Sharp90,Darlington89c].

All these different types of abstractions are supported in our extended functional language - making it extremely suitable as a specification language.

1.3. A Suitable Program Transformation Methodology

Almost all program transformation systems are based on source-to-source transformation rules which are *correctness-preserving* (i.e. preserve the meaning/value of the programs they

transform). These rules are normally classified as either *schematic* or *procedural* in form.

Schematic rules are typically of the form P1 → P2 **IF** C, where P1 and P2 are generic program schemes for representing classes of programs, and C is an applicability condition, consisting of a set of syntactic and/or semantic tests. An expression, e, can be transformed by a schematic rule if it matches the first program scheme, P1, and the corresponding applicability condition, C, is also satisfied[6]. When both the matching and applicability condition are satisfied, the expression, e, can be replaced by the second program scheme, P2, with the appropriate substitutions from the earlier matching. The simplest form of matching which can be used is a *first-order* matching. However, if function-type variables are present in the source program scheme, P1, then the second-order matching technique of [Huet78] should be applied. This form of schematic rules was used in an early Edinburgh transformation system [Darlington76] and in the Munich CIP[7] project [CIP85]. Such schematic rules are declarative in form and are easy to follow. However, not all rules can be expressed in this way.

Procedural rules, on the other hand, are normally specified in an algorithmic fashion as meta-functions of the type, expr # info -> expr. Each of these meta-functions takes an expression of the type, expr, together with some additional information of the type, info, to produce a transformed expression of the type, expr. The use of such procedural or algorithmic rules normally results in transformations which can be carried out more efficiently. Even more importantly, they are also more general because it should be theoretically possible to implement any computable transformations by procedural rules. However, their main weakness is that they are less declarative in form.

Both forms of rules (plus hybrid rules) have been used in various program transformation systems. In fact, there are even some proposals to exploit the declarative versus efficiency aspects of schematic and procedural rules. This is done by applying the transformational methodology to the development of the rules themselves. In particular, a recent ESPRIT project, called PROSPECTRA[8] [KBruckner89], allows transformation rules to be written as algebraic specifications between any two program schemes. These algebraic specifications could then be systematically transformed to more efficient algorithmic rules. To support the algorithmic rules, the PROSPECTRA system also uses a specially designed functional language, called TrafoLa [Heckmann88], which directly supports higher-order pattern matching (on expression trees) with back-tracking or non-determinism.

1.3.1. Catalogue versus Generative Set Systems

Though rules are normally classified as either procedural or schematic in form, transformation systems tend to be classified by the *nature* rather than the *form* of the rules. In particular, some researchers have classified transformation systems according to whether they are based on large sets of *complex* rules or smaller sets of more *elementary* rules. In [Partsch83], systems that are

---

[6] Syntactic tests can be evaluated directly for the transformation system. However, semantic tests are more difficult and normally need to be verified by either the user or a separate theorem-proving system.

[7] Computer-aided Intuition-guided Programming

[8] **PRO**gram Development by **SPE**Cification and **TRA**nsformation

based on large sets of complex rules are known as *catalogue* systems, while those that are based on small sets of elementary rules are known as *generative set* systems. These two approaches were also known as the *dictionary* and the *discovery* approaches in [Kieburtz81].

The good thing about the catalogue system is that each of its rules represents a fairly large step that could be applied to perform meaningful transformations. However, the main limitation is that the system's range of applicability is strictly dependent on the rules available. Furthermore, each new rule for the catalogue must, in general, be pre-proved by the users before it can be added to the catalogue.

In contrast, systems based on small sets of elementary rules are often nearly complete (when augmented with relevant domain-related laws) in their range of applicability, and do not require users to add pre-proven rules. However, the main problem of such systems is that each of the elementary rules result in only a small change. As a result, we often require a number of them, in appropriate sequence, to perform meaningful transformations. Organising these small rules to perform useful transformations is the key problem of generative set systems.

In the early days, there was a debate about which of the two approaches is better. However, it is now becoming clear that the two approaches address quite distinct concerns. In particular, the catalogue approach is more suitable for providing large automatable transformations, while generative set approach, with its generality, is useful for handling the widest possible range of transformations. More promising systems are therefore based on a hybrid approach which uses small rules for generality and relying on complex rules for meaningful transformations.

1.3.2. Desirable Criteria for Transformation Systems

Experimental program transformation systems have been around for some ten to twenty years now but are presently still some way from being accepted in industry. For these systems to be adopted as formal alternatives to software development, there are at least three important criteria which must be present. These are:

- *Safety* or *Correctness* criterion to ensure that all transformed programs are correct with respect to their specifications;
- *Generality* or *Completeness* criterion in order to let these systems cope with the widest possible range of transformations;
- *Automatability* to help construct transformations automatically whenever possible.

The first criterion is important because it is through this assurance of *correctness* that users can have confidence in their transformation systems. The second criterion of *generality* is necessary because software development is, by nature, an open-ended task where the programming solutions are nearly as varied as the problems themselves. Thus, in order not to be limited by the adopted methodology, we must choose a system which would allow the most flexibility. In this way, we could always fall back on user-prepared transformations whenever automated tools are not available. Lastly, the third criterion of *automatability* is vital for alleviating transformational programmers from much of the tedium of constructing transformations. This can help to improve productivity in software construction and is a powerful incentive for

encouraging the use of transformation systems.

Most current program transformation systems are presently unable to satisfy all the above three criteria simultaneously. Some of them can be automated but are not generally applicable, while others are general but harder to automate. This thesis intends to show that, though difficult, these objectives can be achieved simultaneously. The first two criteria of *safety* and *generality*, in essence call for a *generative set* transformation system. Such a system is based on a small set of elementary transformation rules which is in some sense correct and complete.

One of the earliest generative set approach for program transformation is the unfold/fold methodology that was introduced by Burstall and Darlington in [Burstall77]. We shall give an overview of this methodology in the next section and then show how the rules are in some sense correct and complete (under appropriate circumstances).

However, the reliance on elementary rules can make the third criterion of providing automated assistance non-trivial. This is because each elementary rule can only make a small change. As a result, appropriate combinations of the small rules are needed to perform useful transformations. This task of providing automated assistance within the unfold/fold methodology is the principal theme of this thesis. We shall introduce larger automatable transformations, known as transformation tactics, which are based on the unfold/fold rules. But first, we need to introduce the unfold/fold methodology.

1.3.3. The Unfold/Fold Methodology

The unfold/fold program transformation methodology is based on equality reasoning of functional programs that are expressed as equations. It consists of six elementary rules namely, *define*, *instantiate*, *unfold*, *abstraction*, *fold* and *laws*, as described below.

Define	Introduce a new equation (function definition) with a unique LHS.
Instantiate	Provide an appropriate instance of a given equation.
Unfold	Replace a function call by its corresponding function body, with the appropriate parameter substitutions.
Fold	Replace an expression that matches a function body by its corresponding function call.
Laws	Apply domain laws/lemmas of the language constructs and functions. e.g v(**if** a **then** b **else** c)=**if** a **then** v(b) **else** v(c)
Abstraction	Extract out one or more identical sub-expressions using the **where/let** construct.

The principal rules in the above list are *define*, *unfold*, *fold* and *laws*. The **where** abstraction step is actually a special *law* for the **let/where** language constructs. Also, the *instantiation* step can often be combined with *unfolding* by a process commonly called *narrowing*. This is made up of *instantiation* (via unification) followed by *unfolding*. We will sometimes refer to this *narrowing* step as just a general form of *unfolding*.

Let us consider how these rules could be used to improve a simple function, av_list, for computing the average of a list of numbers, as shown below.

```
dec av_list      :  list(num) -> real;
dec sum          :  list(num) -> num;
dec length       :  list(alpha) -> num;
- - - av_list(x)    <= sum(x)/length(x);
- - - sum([])       <= 0 ;
- - - sum(a::x)     <= a+sum(x) ;
- - - length([])    <= 0 ;
- - - length(a::x)  <= 1+length(x) ;
```

The present av_list function is inefficient because it involves the double traversal of the input list, x, by two recursive calls, sum(x) and length(x). To improve this function, we must group the two calls together into a new function definition. We perform this by initially *defining* a new function, say av_tup, as shown below.

```
define
- - - av_tup(x)     <= ( sum(x), length(x) );
```

This tuple function definition can be transformed to give two equations by *narrowing* on the function call, sum(x), as follows:

```
narrow on the sum call to give:
- - - av_tup([])    <= ( 0, length([]) );
- - - av_tup(a::x)  <= ( a+sum(x), length(a::x) );
```

This *narrowing* step corresponds to an *instantiate* followed by an *unfold* step on sum(x). After this, the two equations can be separately transformed. The first equation can be simplified by unfolding the length function call, to give:

```
unfold length call
- - - av_tup([])    <= ( 0, 0 );
```

The second equation can be transformed by three steps. These are an *unfold* on length call, followed by an *abstraction* step (using the **where** construct) and finally a *fold* step, as follows:

```
unfold length call
- - - av_tup(a::x)  <= ( a+sum(x), 1+length(x) );
abstract sum(x) and length(x)
- - - av_tup(a::x)  <= ( a+u,1+v ) where (u,v)==( sum(x), length(x) );
fold with av_tup(x)
- - - av_tup(a::x)  <= ( a+u,1+v ) where (u,v)==av_tup(x);
```

The above sequence of transformation produces the following function definition for av_tup:

```
dec av_tup       :  list(alpha) -> (num # num) ;
- - - av_tup([])    <= ( 0, 0 );
- - - av_tup(a::x)  <= ( a+u,1+v ) where (u,v)==av_tup(x);
```

This is a more efficient function because it traverses its input list only once in order to compute the two results. We can use it to improve the original function, av_list, by replacing the two recursive calls with an *abstract* step and a *fold* step, as follows:

```
- - - av_list(x)          <= sum(x)/length(x);
abstract sum(x) and length(x)
- - - av_list(x)          <= u/v where (u,v)==( sum(x),length(x) );
fold with the original definition of av_tup(x)
- - - av_list(x)          <= u/v where (u,v)==av_tup(x);
```

This sequence of steps is an instance of horizontal loop combination transformation. Other classes of transformations could also be performed using these simple but surprisingly powerful rules. We shall discuss the correctness and the completeness criteria of this methodology next.

1.3.4. CORRECTNESS

The six elementary rules of unfold/fold methodology essentially involves equational reasoning and cannot therefore result in inconsistent programs. However, the rules can only preserve *partial correctness*. This is because arbitrary use of *folding* steps can sometimes introduce non-termination (i.e. a *less defined* function). As an example, consider the following identity function, id.

```
dec id : alpha -> alpha;
- - - id(x)   <= x;
```

If the RHS of the above definition is folded against itself, we would obtain a function which is non-terminating (even though the original definition is not so), as shown below.

```
- - - id(x)   <= x;          ! fold id(x)
             <= id(x);
```

This loss of total correctness is clearly undesirable, but fortunately there are ways of preventing it from happening. We discuss three possible ways to ensure total correctness in a *strict* language.

The first technique is to employ termination proofs (immediately after transformation) to check that the transformed programs are terminating. Those programs that are terminating would be accepted as being valid because they could only have arisen out of totally correct transformation steps. This technique was used by Manna & Waldinger in a deductive system [Manna79] that had elementary rules similar to the unfold/fold rules. One problem of this technique is that termination proof is, in general, an undecidable problem. Nevertheless, there exist heuristics [Dershowitz79] that could be used.

A second more direct method for ensuring total correctness arised from a theoretical study in [Kott78] on the validity of unfold/fold rules. Kott considered each transformation sequence on a function definition to be made up of a series of unfoldings, simplification (laws) and folding steps. With a restricted form of simplification steps, he was able to show that total correctness can be guaranteed if at any one time there are always equal or more *unfold* steps than *fold* steps. If each sequence of transformations satisfies this condition, then erroneous introduction of non-

termination could never arise.

Finally, there is also a proposal in [Scherlis80] for a restricted use of fold steps which could guarantee total correctness. He showed that fold steps based on newly defined functions, which contain at least one function call which has been unfolded, will always be safe to carry out. A similar approach was also adopted in an unfold/fold system for logic programs in [Kanamori87]. This restricted use of folding appears to be the simplest method to adopt for total correctness but it does exclude certain categories of transformation. It could perhaps be used in conjunction with Kott's validity condition, to ensure a wide-range of safe transformations.

The above three techniques to guarantee total correctness are each given under the context of a *strict* language. In the case of non-strict language, additional precautions are needed to guarantee that non-strict behaviours are adhered to. [Runciman89] gave a consideration of how the *instantiation* step should be applied to comply with non-strict semantics. A brief discussion of this non-strict semantics correctness is given in Section A2.5 of Appendix 2 where we explore the strictness implications of conversion-to-iteration transformations.

1.3.5. COMPLETENESS

The unfold/fold methodology is *almost* complete in terms of the range of program improvement transformations which can be carried. This generality is certainly impressive.

However, there is an asymmetry in its rules which typically allows transformation to proceed from a *non-recursive* function to a *recursive* function but not vice-versa. This is the basis by which most program improvements are carried out but it may also be possible to have some examples which do not improve in this way.

An example of this is the identity function for lists,

```
dec id_list       :  list(alpha) -> list(alpha);
- - - id_list(x)  <= x;                                   (1.1)
```

which can be transformed to:

```
dec id_list       :  list(alpha) -> list(alpha);
- - - id_list(nil)  <= nil;                               (1.2)
- - - id_list(a::x) <= a::id_list(x);                     (1.3)
```

but not quite *vice-versa*, even though the first definition is more efficient.

This asymmetry could be overcome if a better complement of the unfold/fold rules is allowed. For example, if a step to *re-define* an existing function (suggested in [Burstall77] as a complement of the *define* step) is allowed, we could have proved that id_list(x)=x from equations (1.2) and (1.3) and then replace them with equation (1.1) by just a re-definition.

Alternatively, we could use the law, id_list(x)=x, to transform equation (1.3) to:

```
apply law id_list(x)=x to (1.3), giving:
- - - id_list(a::x)  <= a::x;                             (1.4)
```

After this, if a *generalise* step (as a complement to the *instantiation* step) is allowed, we could

generalise the two equations (1.2) and (1.4) to give the more efficient equation (1.1).

This example suggests two issues which need to be addressed to achieve better completeness.

- the transformation system should allow all steps which preserve correctness (including those which goes from loosely-specified to more-defined functions).
- it must be possible to derive or synthesize the necessary laws (using a reliable inference sub-system, perhaps the user!).

As shown in the last two sections, the unfold/fold methodology could satisfy both the correctness and completeness criteria. In the rest of this thesis, we shall see how the third criterion, automatability, could be satisfied.

1.4. Contribution of Thesis

The main contribution of this thesis is the systematisation of four major classes of transformations, namely *fusion*, *higher-order removal*, *tupling* and *constraint-based* transformations, which we shall show can be largely automated. These classes of transformations each correspond to a particular programming technique. The ability to automate these techniques implies successful encapsulation of the programming knowledge involved in implementing the transformations. Furthermore, we shall show that all these transformations can be formulated using the unfold/fold transformation methodology.

Chapter 2 presents a proposal for a comprehensive program transformation system (based on the unfold/fold methodology) which is capable of ensuring correctness, generality and machine assistance. We describe each of the system's key components that should be present. One of the most important components is the *methods/tactics bank*, for containing a collection of transformation tactics which are able to analyse user-programs and then suggest transformations to improve them. We shall show how these tactics can be organised as *meta-programs* (cf. LCF proof tactics [LCF79]) and then present two very simple tactics, one algorithmic and the other schematic, to perform simple but useful transformations. These two simple tactics are used to illustrate the possibility of automation within the unfold/fold methodology. More substantial transformation tactics are investigated in the next four chapters.

Chapter 3 considers an important class of program transformations, commonly known as **fusion**, which improves programs by merging (or fusing) function calls of composed expressions. Successful fusion can often improve programs by reducing the number of function calls used and eliminate the need for intermediate data structures. We introduce a *producer-consumer* model of function to explain this fusion transformation. In [Wadler88], a transformation algorithm, called *deforestation*, was presented. This algorithm can fuse any expressions which are composed from a *sub-class* of first-order functions with a certain syntactic property. This sub-class of functions is actually a description for good consumers and good producers for fusion purpose. Our contribution in this thesis, is a proposal for the extension of deforestation to cope with all (rather than just a sub-class of) first-order programs.

This extension helps to increase the number of instances where fusion could be applied. We make use of two apparatus for this extension. These two apparatus are a parameter annotation scheme and the **let** construct. They are used to identify *good* producers and consumers from *bad* producers and consumers. With these distinctions, fusion could be applied whenever a *good* producer meets a *good* consumer. We give the complete transformation algorithm to achieve this, together with a proof that our algorithm will always terminate. After that, we present an analogous technique that will enable independent function calls in *set abstractions* to be fused together. Set abstractions are usually used to specify generate-and-test programs. Successful fusions between generators and filters often correspond to a powerful program transformation technique, commonly known as filter promotion. This technique is capable of making drastic improvements by cutting off sub-spaces of infeasible solutions from the programs' search spaces.

Chapter 4 presents a new transformation tactic for removing certain **higher-order features** in functional programs. Higher-order facility provides additional expressive power to functional languages but it also carries additional overheads and complicates both analysis and transformation techniques. We show, in this chapter, how they can be removed by three fairly independent transformation algorithms which are applicable to well-typed functional programs. Two of the algorithms are used to eliminate a form of applications, called *general applications*, and a certain class of non-variable function-type arguments. A third transformation algorithm is concerned with preserving full laziness during the removal of function-type arguments. This algorithm differs from the conventional fully lazy lambda lifting technique, which requires the higher-order facility (rather than their removal) to preserve full laziness.

All three transformation algorithms work together for the efficient removal of most higher-order features. With this removal, the resulting expressions will be relatively simple and near first-order in form (with some residual higher-order features). We shall show how easy it is to extend the first-order deforestation technique of Chapter 3 to the resulting residual higher-order programs. This would provide an indirect route for deforestation to be applied to higher-order programs. Many other transformation or optimisation techniques could similarly rely on this higher-order removal for the extension of their techniques from first-order to higher-order programs.

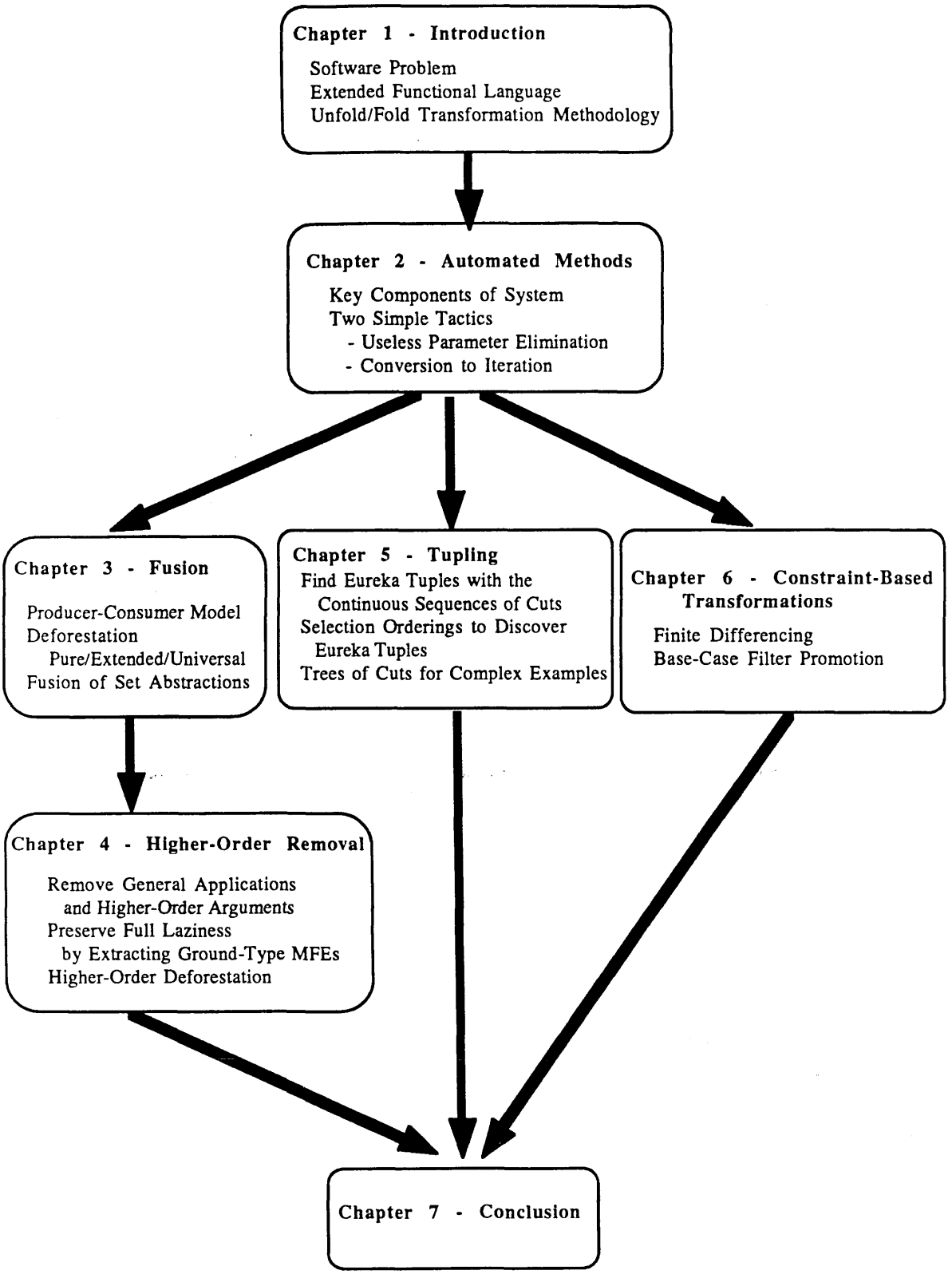
Chapter 5 deals with a slightly more complicated transformation tactic, called **tupling**. This tactic relies on the use of the *tuple structure* for grouping together separate function calls with common recursion variables. A major use of this tupling tactic is for the elimination of redundant calls in certain recursive functions. We present an analysis technique which works by systematically exploring the descendant calls of these recursive functions to discover eureka tuples (that would remove redundancy). A key feature of our technique is the use of an appropriate *selection ordering* to systematically search for eureka tuples. We show how this selection ordering could be heuristically determined either from the syntactic representation of function-argument pairs or from the constructor patterns of calls of functions with pattern

matching equations. This tupling analysis technique is one of the most general technique to-date among comparable work in the area. It is applicable to a very wide class of programs with redundant calls.

Chapter 6 investigates a class of transformations, called **constraint-based** transformations, which uses invariant or constraint relationships for program improvement. These constraint relationships must be maintained across recursive functions in order to allow optimisations to be made to their programs using the established invariants. Both finite differencing and a new (base-case) filter promotion technique are part of this category of transformations. We shall show that this class of transformation can be systematised by identifying the essential semantic conditions, which are needed to maintain the constraint relationships. These conditions must be successfully analysed and synthesized before our tactics could be applied to improve programs.

Chapter 7 summarises our achievements.

Figure 1.1 outlines the seven chapters of this thesis and provides an informal reading order for the various chapters.



Appendix 1 - Higher-Order Useless Parameter Eliminations
Appendix 2 - Other Conversion to Iteration Schemes
Appendix 3 - Derivation of Parallel Lemmas
Appendix 4 - Implementation of Tactics
Appendix 5 - A Termination Proof

Figure 1.1 - Overview of Thesis

2. Automatic Methods

2.1. A Pitfall to Avoid

There is a pitfall to avoid when attempting to provide automated methods for generative-set transformation systems. This pitfall is a lesson learnt from early theorem-proving systems which are similarly based on small sets of generative inference rules. Many of these systems originally relied on *uniform search strategies* that exhaustively search for proofs in the solution spaces of the specified theories. Such uniform strategies appear to be quite successful for small examples but immediately encounter the *combinatorial explosion* (or *computational intractability*) problem when larger examples are attempted. This phenomenon is caused by poor searching techniques in the large search space of the elementary inference rules.

Instead of uniform search strategies, we should attempt to provide appropriate *methods* or *tactics* that use various heuristics and analyses to help provide guided searches (for solutions). Such specialised methods or tactics could be discovered and codified into operational techniques, after careful identification of common patterns of problem-solving techniques in the particular problem domain. A number of well-known mathematical domains, such as *integration* or *differentiation*, already have collections of useful problem-solving methods that are widely applicable. These collections of methods, in effect, represent intelligent procedural knowledge of the respective problem domains.

Alan Bundy called this approach of avoiding the fallacy of uniform search strategies, the *meta-inference* or *meta-reasoning* technique [Bundy81]. Meta-inference techniques are concerned with formulating reasoning at the meta-level, on *methods* or *heuristics*, for more guided searches which would (in turn) induce object-level inferences. In contrast, object-level inference techniques are concerned with the formulation of general inference rules which directly apply to objects. This often results in blind (rather than guided) searches.

A number of successful problem-solving systems, covering different mathematical domains, have been classified in [Bundy83], as having employed (directly or indirectly) meta-inference techniques. Some examples of these systems include Boyer-Moore's inductive theorem proving system [Boyer79], the PRESS system for equation solving [Sterling82] and the Geometry Machine [Gelernter63]. The successful applications of meta-inference techniques in these systems are achieved through the exploitation of domain-specific problem-solving techniques.

Fortunately, the programming domain, as exemplified by program transformation, also has a similarly rich structure of techniques or methods which could be exploited to provide automated mechanisms. Examples of useful transformation techniques that have been identified, but not necessarily systematised yet, include *tupling*, *loop fusion*, *filter-promotion*, *convert-to-iteration*, *finite-differencing* and *dynamic programming*. Each of these techniques or tactics corresponds to a particular programming knowledge. Often, it either introduces good features or eliminate bad ones, with the overall objective of improving the efficiency of programs. All of these tactics have been observed to have fairly well-repeated sequences of rule applications.

In this chapter, we shall consider how this tactical or meta-inference approach could be provided for the unfold/fold methodology. We begin by proposing in Section 2.1 a comprehensive program transformation system, based on the unfold/fold methodology, which is able to satisfy the three criterion of *correctness*, *generality* and *automatability*. In particular, we suggest how tactics could be provided as meta-programs within this comprehensive system. After that, we claim in Section 2.2 that each transformation tactic can normally be structured into two distinct phases, namely an *analysis* before a *transformation* phase. To illustrate this common structure for tactics, we shall introduce in Sections 2.3 and 2.4 two very simple tactics, one *algorithmic* and the other *schematic* in style. The algorithmic tactic is used for eliminating useless parameters, while the schematic tactic is used to convert linear recursive functions to tail-recursive equivalent.

2.2. A Comprehensive Transformation System

In this section, we shall describe the essential components of a comprehensive transformation system, which is able to satisfy the three objectives of safety, generality and automatability. Such a system would contain a number of key components as outlined in Figure 2.1. We give a brief descriptions for each of these components before describing them further in the next few sub-sections.

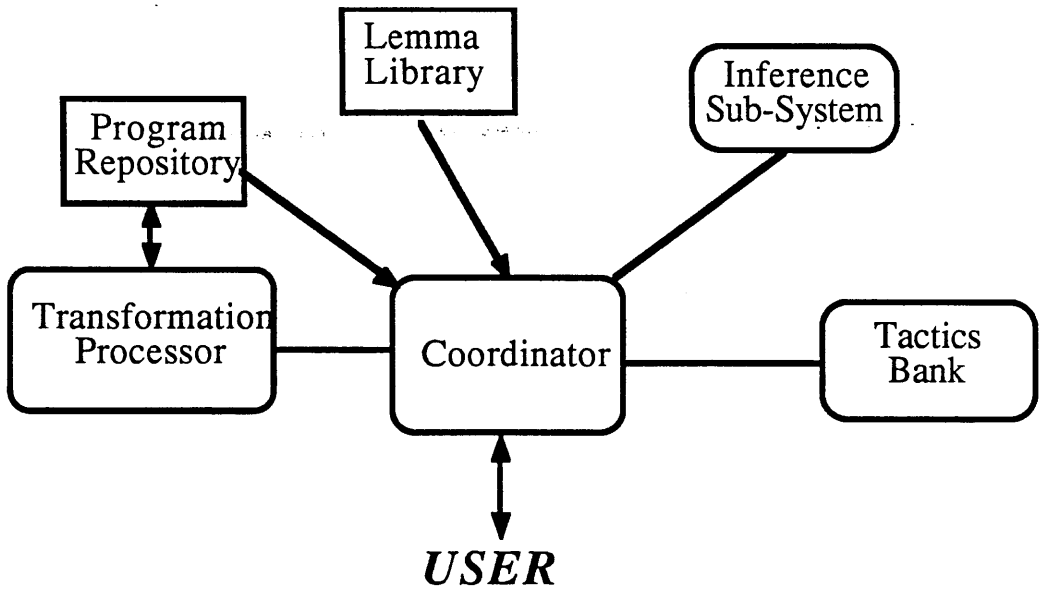


Figure 2.1 - Components of a Comprehensive Program Transformation System

At the centre of this system is a *coordinator* which accepts requests directly from the users and then directs these requests to the various system components. One of the these components is a *transformation processor* which will accept all valid instructions to transform user-programs that are stored in a *program repository*. This processor will only accept instructions based on a small set of correctness-preserving elementary rules. The use of these elementary rules will help to ensure both the generality of the system and the correctness of every transformation carried out by the processor.

Furthermore, to help provide automated mechanisms, we shall have a *methods/tactics bank*, consisting of both algorithmic and schematic tactics. These tactics would analyse user-programs

before suggesting (to the processor) the appropriate sequences of elementary steps required to improve the programs. Also, to make our transformation system more complete, we shall have an *inference sub-system* which is capable of either deriving or verifying laws about the user-programs. These laws (derived, verified or given) are frequently needed in transformations and we shall have a *lemma library* for storing them.

2.2.1. Transformation Processor and the Script Language

At the base of our system is a *transformation processor* which is capable of accepting and processing *transformation scripts* based on the unfold/fold rules. Each of these transformation scripts is a prescription for the appropriate sequence of transformation steps that is to be applied to a given user program. These steps/operations must conform to a special *script language.*

Such a processor and its associated script language have already been designed and implemented for the Flagship Transformation-Based Programming Environment. The script language is documented in [Pull88]. A sample list of the script operations allowed is shown below.

```
type pattern==string;
type hope_pattern==string;
data eqn_spec==all_eqns ++ eqspec(hope_pattern);
data eqn_defn==eqn_defn(hope_pattern);
data context==any ++ context_match(pattern);
data subterm==sbterm(hope_pattern # context) ++ locn(list(num));
data script ==
    unfold(subterm,eqn_spec).      ! unfold a subterm using the specified equation, eqn_spec.
++  fold(subterm,eqn_spec)         ! fold a subterm using the specified equation, eqn_spec.
++  replace(subterm,hope_pattern)  ! apply a lemma (second argument) to the subterm.
++  define(eqn_defn,script)        ! define a new equation, eqn_defn, and apply the specified
                                     transformation script to this new equation.
++  abstract(subterm)              ! apply the where abstraction to all subexpressions which are
                                     identical to specified subterm
++  substitute(subterm)            ! remove a let/where subterm by substituting the value of its local
                                     variable into the main expression of the let construct. This is a dual
                                     of the abstract operation.
++  compose (list(script))         ! apply a list of scripts in strict sequence.
++  branch (list(script))          ! apply a list of scripts in parallel by branching out into separate
                                     equations.
++  nil_script                     ! dummy script operator.
;
```

The argument, subterm, in some of the above operations refers to a sub-expression (within the current equation) which is to be transformed. This subterm could be specified using either be a simple *tree-address* location or a more sophisticated *context-matching* notation. As an example, consider the following program.

```
dec sumdb,sum      : list(num) -> num;
dec double         : list(num) -> list(num);
- - - sumdb(x)     <= sum(double(x));                                    (2.1)
- - - sum([])      <= 0;
- - - sum(a::x)    <= a+sum(x);
- - - double([])   <= [];
- - - double(a::x) <= 2*a::double(x);
```

To refer to the sub-expression double(x), in the RHS of equation for sumdb, we could use the tree-location, [2,2,1] (not [2,1], see footnote [1]). This tree location is simply a list of numbers to indicate by location, a particular sub-expression within the current equation. It is ideal for use in scripts that are automatically generated from tactics.

Alternatively, we could also use a context pattern, sbterm("double(x)",pattern) where pattern is used to provide a cross-reference to uniquely identify a sub-expression. In this case, pattern could either be:

any	if the sub-expression specified is already unique
context_match("- - -sumdb(x)<=_")	to provide an appropriate cross-reference using the parameter x.
context_match("- - -sumdb(x)<=sum(?)")	to narrow the scope of searching, where ? indicates the place to search for the sub-expression

Such subterms based on context patterns are easier for scripts which have to be manually prepared.

Let us consider a simple transformation script, called lc_script, below.

```
- - - lc_script <=
  branch [  compose [ unfold(locn [2,2,1],eqspec "double([])"), unfold(locn [2],eqspec "sum([])") ]
           ,compose [ unfold(locn [2,2,1],eqspec "double(a::x)" ), unfold(locn [2], eqspec "sum(a::x)" )
                    , fold (locn[2,2,2],eqspec "sumdb(x)" ) ]
         ];
```

This script is actually a prescription to apply a loop combination transformation to equation (2.1) of sumdb. It could be applied (by the transformation processor) through the following request,

```
apply_script(lc_script,"sumdb")
```

where apply_script takes a script and a function for the processor to apply the specified transformation.

We shall now describe how this script is interpreted by the transformation processor. Initially, the outermost script operation, branch, is used to split the transformation into two independent transformations which will generate separate equations.

The first branch, using the sub-script,

```
compose [ unfold(locn [2,2,1],eqspec "double([])"), unfold(locn [2],eqspec "sum([])") ]
```

would initially perform a narrowing (instantiate and unfold) on the inner function call, double(x), using the base-case double equation (specified by eqspec "double([])"), before a second unfold is applied to the outer call, sum([]). These two steps result in a base-case equation for sumdb, as

[1] Notice that the tree-address location is formulated for a higher-order language where apply nodes have to be taken into account. Thus, the ith argument of a function call, f(t1,..,tn), is given the relative address [2,i] rather than just [i] because this function call is internally represented as apply(f1,(t1,..,tn)).

shown below (with each script operation to be processed, shown on the right).

```
---sumdb(x)      <= sum(double(x));          ! unfold(locn [2,2,1],eqspec "double([])")
---sumdb([])     <= sum([]);                 ! unfold(locn [2],eqspec "sum([])")
---sumdb([])     <= [];
```

Similarly, the other sub-script,

```
compose [ unfold(locn [2,2,1],eqspec "double(a::x)" ), unfold(locn [2], eqspec "sum(a::x)" )
        , fold (locn[2,2,2],eqspec "sumdb(x)" ) ]
```

would initially perform a narrowing on the inner call double(x) with the recursive equation eqspec "double(a::x)", followed by an unfold of the outer sum call before a fold back on to the original definition of sumdb. This sequence of steps produces a recursive equation for sumdb, as shown below.

```
---sumdb(x)      <= sum(double(x));          ! unfold(locn [2,2,1],eqspec "double(a::x)" )
---sumdb(a::x)   <= sum(2*a::double(x));     ! unfold(locn [2], eqspec "sum(a::x)" )
---sumdb(a::x)   <= 2*a+sum(double(x));      ! fold (locn[2,2,2],eqspec "sumdb(x)" )
---sumdb(a::x)   <= 2*a+sumdb(x);
```

The result of applying the transformation script lc_script to sumdb is the following two transformed equations (for update into the program repository).

```
---sumdb([])     <= 0;
---sumdb(a::x)   <= 2*a + sumdb(x);
```

Such transformation scripts, based on the elementary rules, could be generated by either the users or automatically from the relevant tactics. They (the scripts) are the only means by which the system can be instructed to perform transformations on programs.

2.2.2. Program Repository

The program repository is used to store two types of information, namely:

- user functions
- definition set.

The user functions are the primary data present in the *program repository*. They could either be directly provided by users or indirectly obtained through program transformations. In addition to this, the program repository is allowed to keep a *definition set* which contains all the original equations that defined some of our functions before transformations. An example of an original equation is sumdb(x)=sum(double(x)) for the function sumdb in the previous sub-section. This equation was transformed to two new equations but we must continue to keep it in the program repository. It must be kept because if a similar expression occurs in another part of our program, we could apply a fold step to it and thus avoid the need for the introduction and transformation of another identical function. This allows the re-use of previous transformations.

2.2.3. Lemma Library

The *lemma library* is a place where useful lemmas or laws on functions are kept for use by

transformations. The presence of relevant lemmas is important because the completeness of the unfold/fold transformation systems is ultimately dependent on them. To start off with, this library must contain all useful laws for both *primitive functions* (e.g. +) and *language constructs* (e.g. **if** and **let**). Some examples are:

```
v(if a then b else c)   =  if a then v(b) else v(c)   ! distributive law of if
0+x                     =  x                          ! left identity law for primitive function +
```

In addition, lemmas of interest for *user-defined functions* must also be available. These can either be provided by user or be synthesized by the system. See Appendix 3 for the possibility of synthesizing some useful distributive lemmas.

2.2.4. Inference Sub-System

Transformation systems are normally incomplete without the presence of some general inference capability. In the unfold/fold methodology, this capability is *indirectly* provided by allowing *laws* to be used but little consideration has been given to the way these laws could be obtained.

For a transformation system to be as self-contained as possible, we need a general inference (both inductive and deductive) capability that is able to *verify* or *synthesize* laws (or lemmas) needed by transformations. To help provide this capability, we actually need a separate *inference sub-system.* However, such a sub-system need not have to use a separate logical framework. This is because most inference steps also involve equational reasoning and could therefore be embedded within the unfold/fold transformations.

For example, consider the right identity law, x<>[]=x, for a list append operator, <>, of the following definition.

```
- - -[] <> y        <= y;
- - -a::x <> y      <= a::(x<>y);
```

This law can be inductively proved by embedding the equational reasoning required in a sequence of transformations. This is achieved by initially defining a new function (representing the proof) for the above law, as follows:

```
define a proof function for the expression to be proved
- - -proof1(x)      <= x<>[] = x;
```

After that, we can narrow the append operator, <>, and follow it with a sequence of simplification steps, as shown below.

```
- - -proof1(x)      <= x<>[] = x;              ! unfold <>
- - -proof1([])     <= [] = [];                ! law (x=x) = true
                    <= true;
- - -proof1(a::x)   <= a::(x<>[])= a::x;       ! law (c(t1,..,tn)=c(t1',..,tn')) = (t1=t1' ∧ .. ∧ tn=tn')
                    <= a=a ∧ x<>[] = x;        ! law (x=x) = true
                    <= x<>[] = x;              ! fold proof1
                    <= proof1(x);
```

The final equations from the above transformations are:

```
- - -proof1([])     <= true;
- - -proof1(a::x)   <= proof1(x);
```

This transformation actually represented an inductive proof which split into two cases. The first case (represented by the base case equation) was simplified to true; while the second case used a fold step to form a recursive equation for proof1. This use of a fold step[2] in effect mimics the application of an induction hypothesis step, called *fertilisation* in [Boyer79].

The results for such inductive proofs are represented by the final form of the transformed (proof) functions. A proof is considered to be successful if only the values true and recursive calls (representing induction steps) are found in the final transformed program. This is indeed the case for the above proof, as represented by the transformed equations of proof1. With it, we have established the right-identity law for the append operator, <>.

Hilbert claimed that all mathematical activity can be formulated as proof in some formal system. Program transformation is one such activity and a number of researchers have attempted to formulate this as a theorem proving activity (e.g. [Manna80]). We could make the corollary claim that proofs (involving equational reasoning) can also be formulated within program transformation by the above embedding.

2.2.5. Methods or Tactics Bank

The methods or tactics bank is an extensible collection of *meta-programs* (cf proof tactics of [LCF79]) to provide either automatic or semi-automatic transformations. These meta-programs are allowed to analyse and manipulate user-programs but are not themselves allowed to make *direct* updates to the program repository. Instead, each meta-program should return a script, after analysing its given program. This script must then be passed to the transformation processor before the specified program is transformed and updated in the program repository.

Thus, tactics will typically be of the form:

```
dec tactics : program # add_info -> scripts;
```

where add_info signifies the additional information which may have to be supplied from the user.

Forcing the tactics to return scripts, which are then passed to the transformation processor, may involve duplicate work by both the tactic and the processor. However, this arrangement allows *ordinary* programmers (and not just system designers) to add tactics to the system *safely*. This is possible because tactics will always have to return scripts, containing correctness-preserving elementary rules, before they are accepted by the processors. This safe extension of the tactics bank (by both ordinary programmers and system designers) is a key difference between our system and other catalogue-based systems.

---

[2] We need to be careful about the fold step for proof (in the same way as that for transformation). It must not result in a loss of total correctness (i.e. the introduction of non-termination).

2.3. Common Structure of Tactics

The library of transformation tactics is, in effect, a collection of knowledge base for our program transformation domain. Possible transformation tactics range from those that are simple and completely automatable, to those that are complex and may require either user guidance or intelligent inference techniques to assist in the transformations.

Early optimisation techniques, frequently employed in imperative language compilers, such as *dead code elimination*, *constant folding* and *constant-code motion out of loops* can also be formulated at the functional language level. These techniques correspond to very simple automatable tactics that result in minor but useful optimisations.

Other more sophisticated transformation tactics are also possible. Some of them can even change the algorithmic structures present in user programs. For example, [Smith85] introduced a transformation tactic which was able to synthesize divide-and-conquer programs from generate-and-test programs. Similarly, [Bird87b] formulated theorems, using function-level notation, which state the conditions required to obtain *greedy* algorithms from certain kind of minimising/maximising search programs.

This thesis will examine a range of tactics, from simple tactics (like elimination of useless parameters and conversion-to-iteration) to more sophisticated tactics (like fusion, finite-differencing, filter-promotion and tupling). Our formulations of these tactics represent further systematisations and extensions over the present state-of-art for automated transformations. We show how each of these tactics can be formulated within the unfold/fold methodology.

From our experience in formulating these tactics, we observe that most tactics can be structured into two distinct phases, namely an *analysis* followed by a *transformation* phase. The analysis phase is used to determine if a given program satisfies the applicability conditions of its tactic. After the appropriate analysis, the transformation phase is normally achieved by applying a straightforward sequence of unfold/simplify/fold steps.

The form of analysis needed varies according to the particular transformation tactic. It may be a simple *syntactic test* to see if some program forms are satisfied or a *semantic test* to see if certain algebraic conditions are present, or even a combination of both. The analysis may also be on the function parameters to see if certain *static properties* are present. Or it may involve repeated *failure analysis* to obtain an appropriately *generalised* program form that is more suitably transformed (i.e. folded).

We shall see this common two-phase structure repeated for all our transformation tactics. To illustrate this, we shall consider two *very simple* tactics in the next two sections. One of them is an algorithmic tactic for eliminating useless parameters, while the other is a schematic tactic for performing conversion-to-iteration transformations. We shall show how both of them can be formulated as unfold/fold transformation tactics and also identify the analysis techniques needed to make these tactics automatable.

2.4. An Algorithmic Tactic: Removing Useless Parameters

It is frequently possible for functions to possess parameters that are *not* required in their computation. Such parameters are said to be *useless* and are undesirable because they cause both redundant computation and use up unnecessary storage spaces.

An example of this is the second parameter, w, of function traverse, as shown below:

```
dec traverse : list(num) # list(num) # list(num) -> list(num);
- - - traverse([],w,t)       <= t;
- - - traverse(a::x,w,t)     <= traverse(x,a::w,3*a::t);
```

The second parameter in this function is useless because it does not contribute to its function's result. This function may have been written by a careless programmer. But it is also possible for such a function to be an intermediate result of another program transformation tactic, e.g. *finite differencing* tactic of Chapter 6.

Useless parameters are one source of *dead codes*[3] for functional programs. Conventional compiler optimising techniques [Aho77] for detecting (with *data flow analysis*) and eliminating dead codes in imperative programs are therefore quite similar and could be applied here.

However, we would like to illustrate this elimination through the unfold/fold methodology, where correctness is formally assured. Our tactic for eliminating useless parameters will involve both an *analysis* and a *transformation* phase. The analysis phase will be used to identify which of the parameters are useless. This result can then be utilised by the transformation phase to apply a correct sequence of steps which would eliminate the identified parameters.

2.4.1. A Simple Functional Language

To simplify our presentation, we shall only consider this tactic for a simple first-order language. The complete treatment for higher-order programs can also be provided but this is more easily done after the removal of some of the higher-order features by another tactic that is presented in Chapter 4. For the interested readers, we have therefore provided the treatment for higher-order programs in Appendix 1.

Also, instead of the first-order HOPE language, we shall use an even simpler language which contains the minimum (essential) features that are typically present in a functional language. In particular, we shall use a **case** construct to replace the use of pattern-matching functions, **let** and conditional **if** constructs. The extension of our presentation to a fully featured (or sugared) language is straightforward and will not be shown.

Consider a simple first-order language where each user function is of the form:

$$f(v_1,..,v_n) \quad <= \quad tf;$$

and where $v_1,..,v_n$ are the parameter variables of the function and tf is a RHS term that can be represented by the grammar rules below.

---

[3] Another source of dead codes is those functions which are never referred to (directly or indirectly) by the main function of the program.

```
t    ::=   v                                          ! variable
         |  c(t1,..,tn)                               ! constructor term
         |  f(t1,..,tn)                               ! function call
         |  case t0 in p1 => t1 | .. | pr => tr end    ! case construct
p    ::=   v  | c(p1,..,pn)                           ! pattern
```

The first grammar rule, t, is used to specify either a variable, a constructor term, a function call or a **case** expression. The second grammar rule, p, is used to denote constructor patterns that are used in **case** constructs. With this simple first-order language, the earlier function, traverse, shall have to be translated to:

```
- - - traverse(x,w,t)   <= case x in [] => t
                          | a::x  => traverse(x,a::w,3*a::t) end;
```

Note that the **case** construct is not a standard feature of HOPE+. However, it could be provided because each **case** construct of the form:

case t_0 **in** $p_1 \Rightarrow t_1 \mid .. \mid p_r \Rightarrow t_r$ **end**

is simply equivalent to a lambda application of the form:

(**lambda** $p_1 \Rightarrow t_1 \mid .. \mid p_r \Rightarrow t_r$ **end**) t_0

2.4.2. Phase 1: Analysis for Useless Parameters

The analysis phase requires an appropriate technique to help identify useless parameters. For this task, we can turn to recent research on the *abstract interpretation* method (first introduced for functional programs in [Mycroft81], collection of papers in [AbsInterp87]) which is capable of providing a range of safe analysis techniques for various static properties of functional programs. The general method of abstract interpretation works by computing in the *abstract* domain using *abstract* functions (rather than the standard domain and standard functions) to discover the static properties of programs.

An illustrative example of the abstract interpretation method is the 'rule of sign' technique for finding the sign of the sum of two numbers. Using an *abstract sign domain* of + (positive), - (negative), ? (don't know), together with an abstract addition function, $add^{\#}$, represented by the semantic table below,

$add^{\#}$	+	-	?
+	+	?	?
-	?	-	?
?	?	?	?

we can quickly find a safe approximation for the sign of the sum of two numbers by interpreting in the abstract sign domain (rather than the standard numeric domain). For example, the sign of add(6,7) can be computed by interpreting $add^{\#}(+,+)$, under the sign domain, to give a value of + where add and $add^{\#}$ are respectively the standard and abstract functions for addition. Such interpretations in the abstract domains can normally be done more quickly[4] than interpretations in the standard domains but may involve a loss of information as we are only calculating safe approximations.

More useful static properties may also be analysed by abstract interpretation. Two analysis

[4] In fact, abstract interpretation can be performed in finite time (i.e. terminating) if a finite abstract domain is used. This is an essential condition for analysis of compile-time optimisation techniques.

techniques, which have attracted considerable interests, are:

(1) *strictness analysis* [Mycroft81,Burn87b] to determine the parameters which can be evaluated strictly in a lazy program, and

(2) *compile-time storage analysis* [JonesSB89] to determine places where constructor cells (storages) can be safely *overwritten* in a functional program.

Similarly, useless parameters may also be analysed by interpreting in an appropriate abstract domain. For this analysis, we introduce an abstract domain consisting of two values, *present* and *absent*, which are abbreviated as **P** and **A** respectively. The first value, **P**, is used to signify the possible presence of a parameter variable for computing the result of an expression, while the second value, **A**, is used to denote the definite absence of the parameter in the function's result.

With this abstract domain, each parameter of a function, say f, can be analysed *separately* by the abstract call, f#(..,**P**,..), where the parameter of interest is given the value **P**, while the rest of the parameters are each given the value **A**. If this parameter is possibly used in its function's body, then the result from the above call should be **P**; otherwise the value of **A** will be returned to signify that the parameter is not needed at all (i.e. useless). Notice that we have to analyse the parameters individually in order to find out if each parameter would (possibly) contribute towards the *result* of the function.

The outcome of this analysis is assured by the way the standard expressions are translated into abstract expressions by the syntax-directed translation rules, called **use**, in Figure 2.2. For example, each constructor term, $c(t_1,..,t_n)$, is translated to set_OR{**use**$[\![t_1]\!]$,..,**use**$[\![t_n]\!]$} because the *result* of a constructor term is dependent on the sum effect of its arguments. The semantic function, set_OR, and its auxiliary function, or, act as the least upper bound functions (summation of *result*) for the abstract domain where **A** is lower in the domain ordering than **P**. Also, the last translation rule (for the case construct) uses a let construct to propagate the abstract value of t_0 to each of the auxiliary variables, $v_1..v_{aux}$, of the patterns $p_1..p_r$. These values are used in the combined result of $t_1..t_r$, as represented by set_OR(v_0,**use**$[\![t_1]\!]$,..,**use**$[\![t_r]\!]$). The abstract expressions of t_0 and $t_1..t_r$ are combined together because the final result of the case expression is again dependent on all of them. These translation rules effectively provide an abstract expression to analogously compute the required flow analysis[5].

With this set of rules, each function of the form:

$$f(v_1,..,v_n) \quad <= \quad tf;$$

shall have an abstract function of the form:

$$f\#(v_1,..,v_n) \quad = \quad \textbf{use}\ [\![tf]\!]$$

[5] as used in traditional compiler analyses

```
dec use : expr -> expr;
use [[v ]]                                  <= v
use [[c(t1,..,tn) ]]                        <= set_OR {use[[t1 ]],..,use[[tn ]]}
use [[f(t1,..,tn) ]]                        <= f# (use [[t1 ]],..,use[[tn ]])
use [[case t0 in p1 => t1 | .. | pr => tr end ]]  <= let v0== use [[t0 ]] in
                                                    let v1..vaux==v0..v0 in
                                                        set_OR(v0, use [[t1 ]],..,use[[tr ]])
                                                    where v1..vaux==vars [ p1..pr]
                                                    Note that vars [ p1..pr] returns all variables in
                                                        the patterns, p1..pr.
WHERE
set_OR {}                                   <= A
set_OR {x}                                  <= x
set_OR {x} ∪ S                              <= or (x,set_OR(S))
or (A,x)                                    <= x
or (P,_)                                    <= P
```

Figure 2.2 - Translation Rules for Useless Parameter Analysis

Applying the syntax-directed translation rules for **use** to the function, traverse, would give us its abstract function through the following sequence of translations and simplifications[6].

```
traverse#(v,w,t)  =  use [[ case v in [] => t
                          | a::x => traverse(x,a::w,3*a::t) end]]
                  =  let v0==use [[ v ]] in
                     let (a,x)==(v0,v0) in
                        set_OR {v0,use [[ t ]] ,use [[ traverse(x,a::w,3*a::t) ]]}
                  =  let v0==v in
                     let (a,x)==(v0,v0) in
                        set_OR {v0,t ,traverse#(use [[ x]] ,use [[ a::w]] ,use [[ 3*a::t]] )}
                  =  let (a,x)==(v,v) in
                        set_OR {v,t ,traverse#(use [[ x]] ,use [[ a::w]] ,use [[ 3*a::t]] )}
                        :
                  =  set_OR {v, t, traverse#(v,or(v,w) ,or(v,t)) }
```

The final abstract function is recursive and can run into non-termination if directly computed. But as the domain of our abstract function is finite, we can use fix-point iteration to find the least fix-point value of the recursive function. A range of efficient fix-point iteration techniques can be found in [Martin90]. Using it, the semantic values for the three calls (of interest), which represent the analyses for the three parameters, are:

```
traverse#(P,A,A)  =  P
traverse#(A,P,A)  =  A
traverse#(A,A,P)  =  P
```

From these analyses, we can safely state that the first and the third parameters may be needed in our function's computation but not the second parameter because traverse#(**A**,**P**,**A**) returns a value of **A**.

---

[6] The simplification steps are not strictly needed for computing the abstract function. It is shown here to provide a clearer definition for the abstract function.

This method of analysing for the static properties of parameters is also known as *forward analysis* because we provide abstract inputs in order to compute the corresponding abstract output. Alternatively, there is another static analysis technique, known as *backward analysis*, which is also able to analyse for useless parameters. This alternative technique works by computing backwards, the possible *context* of inputs from the *context* of the required output. A range of backward analysis techniques is clearly presented in [Hughes88].

2.4.3. Phase 2: Transformation

After the identification of useless parameters, we can proceed with the second transformation phase to eliminate all the identified parameters. Initially, we shall informally describe, using traverse as our example, the necessary transformation steps needed in this phase.

Our transformation begins by defining a new function, say tr_good, to compute the same value as the original function, but *without* the useless parameter, as follows:

```
dec tr_good      : list(num) # list(num) -> list(num);
- - - tr_good(v,t)   <= traverse(v,w,t);                    (2.1)
  More ...
```

Dropping the useless parameter in the new function definition of (2.1) results in an *unbounded free variable*, w, where an unbounded free variable is a variable on the RHS which does not appear in the LHS of the equation. Because of this free variable, the definition of (2.1) is not a syntactically valid functional program. However, it is still meaningful because we are simply saying that the value of this variable is immaterial to the outcome of the function. More importantly, this can be regarded as a temporary condition which can be removed by subsequent transformations. In fact, an unfold step followed immediately by a fold step will be sufficient to remove the unbounded variable, as shown below.

```
- - - tr_good(v,t)   <= traverse(v,w,t);                    ! unfold traverse(v,w,t)
                    <= case v in [] => t
                         | a::x => traverse(x,a::w,3*a::t) end;   ! fold tr_good
                    <= case v in [] => t
                         | a::x => tr_good(x,3*a::t) end;
```

We are able to achieve this removal of the unbounded variable because the second parameter of traverse is indeed useless. With this removal, the new function, tr_good, can now be used to replace all occurrences of the old function, traverse, by *folding* with the original definition (2.1) of tr_good.

Notice that though this useless parameter analysis could be formally proven/shown to be correct (either by way of [Burn87a] or [Hudak88]), the unfold/fold rules will also ensure correctness in some sense. This is because if a wrong parameter is selected for elimination, the resulting transformed equation will still have an unbounded free variables on its RHS - giving the system sufficient reason to reject the transformation!

2.4.3.1. Transformation Algorithm

The above informal transformation can be formalised into a transformational algorithm, which

we will denote with the symbol, $\mathcal{U}$. This is possible because we need only three types of elementary rules, namely define, unfold and fold, to remove the useless parameters.

To simplify the presentation of our transformation algorithm, we shall assume the following:

1) Each identified useless parameter will be annotated with the symbol, **A**.
2) All the useless parameters, of each function, will be logically grouped together as the first m parameters. Each function call will therefore be represented as $f(t_1^{\mathbf{A}}..t_m^{\mathbf{A}},t_{m+1}..t_n)$ where $m \geq 0$ and $n \geq m$. Those functions without any useless parameters will simply have $m=0$.

These two assumptions simplify the presentation of our transformation algorithm but do not reduce generality. This is because we can either permute the parameters of each function to satisfy the above condition or modify the algorithm to handle arbitrary parameter orderings. Based on these assumptions, the transformation algorithm to eliminate useless parameters is made up of four $\mathcal{U}$ rules in Figure 2.3. These rules belong to a meta-function of the type:

$\mathcal{U}$: expr -> expr;

They will transform any expression, which may have useless parameters, to equivalent expression without useless parameters.

Given $f(v_1,..,v_n) = tf;$

(1) $\mathcal{U}[\![v]\!] \Leftarrow v$

(2) $\mathcal{U}[\![c(t_1,..,t_n)]\!] \Leftarrow c(\mathcal{U}[\![t_1]\!],..,\mathcal{U}[\![t_n]\!])$

(3) $\mathcal{U}[\![\text{case } t_0 \text{ in } p_1 \Rightarrow t_1 \mid .. \mid p_r \Rightarrow t_r \text{ end}]\!]$
$\Leftarrow \text{case } \mathcal{U}[\![t_0]\!] \text{ in } p_1 \Rightarrow \mathcal{U}[\![t_1]\!] \mid .. \mid p_r \Rightarrow \mathcal{U}[\![t_r]\!] \text{ end}$

(4) $\mathcal{U}[\![f(t_1^{\mathbf{A}}..t_m^{\mathbf{A}},t_{m+1}..t_n)]\!]$

a) If m=0 (no useless parameters)
$\Leftarrow f(\mathcal{U}[\![t_1]\!]..\mathcal{U}[\![t_n]\!])$

b) If m>0 and $f_old(v_{m+1},..,v_n)=f(v_1..v_m,v_{m+1}..v_n) \in def_set;$ **then** (fold)
$\Leftarrow f_old(\mathcal{U}[\![t_{m+1}]\!],..,\mathcal{U}[\![t_n]\!])$

c) otherwise (m>0) (define & unfold)
$\Leftarrow f_new(\mathcal{U}[\![t_{m+1}]\!],..,\mathcal{U}[\![t_n]\!])$

define f_new
and $def_set:=def_set \cup \{f_new(v_{m+1},..,v_n)=f(v_1..v_m,v_{m+1}..v_n)\}$
$f_new(v_{m+1},..,v_n)=f(v_1..v_m,v_{m+1}..v_n)$
unfold f ***call***
$f_new(v_{m+1},..,v_n)=\mathcal{U}[\![tf[v_1/v_1,..,v_n/v_n]]\!]$

Figure 2.3 - Transformation Rules for Eliminating Useless Parameters

A brief description for each of the four rules now follows. Rule 1 deals with variables. Rule 2 deals with constructor terms by recursively examining only their sub-terms. Rules 3 deals with **case** constructs, again by examining only the sub-terms. All these three rules simply apply recursive applications of $\mathcal{U}$ rules to the sub-terms (where available) without affecting the outermost constructs. The main rule for eliminating the useless parameters occurs in Rule 4. This rule deals with function calls of the form $f(t_1^{\mathbf{A}}..t_m^{\mathbf{A}},t_{m+1}..t_n)$ where there are three courses of possible action.

If none of the parameters in the function is useless, i.e. m=0, then the call is left alone with recursive transformation applied to just the arguments (see $\mathcal{U}$ 4a). However, if there are some useless arguments (i.e. m>0), together with the fact that there is a previously defined function (i.e f_old) with a RHS that matches the call, then a fold is applied to remove the useless arguments (see $\mathcal{U}$ 4b). Otherwise, we will replace the function call with a newly defined function (i.e f_new), without the useless parameters, and proceed to transform the body of the new definition, after the original call has been unfolded (see $\mathcal{U}$ 4c).

Notice also that both the steps, $\mathcal{U}$ 4b and $\mathcal{U}$ 4c, use a definition set in the form of a *global variable*, def_set, for all newly defined functions. This definition set is extended by step $\mathcal{U}$ 4c for each newly defined function. In addition, it is also used by $\mathcal{U}$ 4b to check if folding can be applied for re-occurring function calls. Presently, we are using the global variable to simplify the presentation of our rules (i.e. avoid an additional rule parameter). However, this non-functional global variable can also be removed, as suggested later in Section 2.4.3.3.

2.4.3.2. An Example

The meta-function, $\mathcal{U}$, is used to transform expressions. To use it on programs, we will have to apply the rules to each RHS of functions, as follows:

$$--f(v_1,..,v_n) \quad <= \quad \mathcal{U}[\![tf]\!]$$

Let us illustrate this application to the earlier example, traverse. Applying it to the RHS of the function results in the following sequence of steps, where the rules taken at each step is shown on the right.

```
--- traverse(v,w^A,t) <= U[[ case v in [] => t
                              | a::x => traverse(x,a::w^A,3*a::t) end ]]   !U 3
                    <= case U[[ v ]] in [] => U[[ t ]]
                         | a::x => U[[ traverse(x,a::w^A,3*a::t) ]] end   !U 1,1
                    <= case v in [] => t
                         | a::x => U[[ traverse(x,a::w^A,3*a::t) ]] end   !U 4c, define tr_good
                    <= case v in [] => t
                         | a::x => tr_good(U[[ x ]],U[[ 3*a::t ]]) end    !U 1,2
                    <= case v in [] => t
                         | a::x => tr_good( x,U[[ 3*a ]]::U[[ t ]]) end   !U 1,..
                    <= case v in [] => t
                       | a::x => tr_good(x,3*a:: t ) end
```

At one of the above steps, a new function, called tr_good, is introduced by $\mathcal{U}$ 4c. This new function will also be transformed by the $\mathcal{U}$ rules to a function without the useless parameter, as shown below.

```
Define
- - - tr_good(v,t)      <=  traverse(v,w,t)
Unfold traverse
- - - tr_good(v,t)      <=  U[[case v in [] => t
                              | a::x => tr_good(x,3*a:: t ) end ]]        !U 3
                        <=  case U[[ v ]] in [] => U[[ t ]]
                              | a::x => U[[tr_good( x,3*a:: t ) ]] end    !U 1,1,4a
                        <=  case v in [] => t
                              | a::x => tr_good(U[[x]],U[[3*a:: t]]) end  !U 1,2,..
                        <=  case v in [] => t
                              | a::x => tr_good(x, 3*a:: t) end
```

The result of this transformation is the following more efficient function definition for traverse.

```
- - -traverse(v,w^A,t)  <=  case v in [] => t
                              | a::x => tr_good(x,3*a:: t ) end ;
- - -tr_good(v,t)       <=  case v in [] => t
                              | a::x => tr_good(x, 3*a:: t) end ;
```

2.4.3.3. A Version of $\mathcal{U}$ which Carries the Definition Set

As noted earlier, the $\mathcal{U}$ rules are presently specified in a non-functional way because they make use of a *global* variable, def_set, for storing newly defined functions. This non-functional apparatus can be removed by changing our rules from the type:

```
U : expr -> expr ;
```

to the type:

```
U : expr # def_set -> expr # def_set ;
```

where def_set is passed as both a parameter and a result. The necessary changes to the rules are shown in Figure 2.4. These changes are quite straightforward and involve the passing of definition set, ds, from one $\mathcal{U}$ call to another in order to accumulate each new function definition introduced. To help us do this, we also have an auxiliary meta-function, $\mathcal{U}$_list, which operates on a list of expressions (rather than a single expression) by passing the definition set which results from one expression to the next expression along the list.

```
dec U : expr # def_set -> expr # def_set;
dec U_list : list(expr) # def_set -> expr # def_set;
(1) U [[ v ]] ds                    <=  (v, ds)
(2) U [[ c(t1,...,tn) ]] ds         <=  let (t'1..t'n, ds')== U_list [[ t1..tn ]] ds in
                                        (c(t'1,...,t'n), ds')
(3) U [[ case t0 in p1 => t1 | .. | pr => tr end ]] ds
                                    <=  let (t'0t'1..t'n, ds')== U_list [[ t0 t1..tr ]] ds in
                                        (case t'0 in p1 => t'1 | .. | pr => t'r end, ds')
(4) U [[ f(t1^A..tm^A,tm+1..tn) ]] ds
    a) if m=0 (no useless parameters)
                                    <=  let (t'1..t'n, ds')== U_list [[ t1..tn ]] ds in
                                        (f(t'1,...,t'n),ds')
    b) if m>0 and f_old(vm+1,..,vn)=f(v1..vm,vm+1..vn) ∈ ds ; then (fold)
                                    <=  let (t'm+1..t'n, ds')== U_list [[ tm+1..tn ]] def in
                                        (f_old(t'm+1..t'n),ds')
    c) otherwise (m>0) (define & unfold)
                                    <=  let ds'== {f_new(vm+1,..,vn)=f(v1..vm,vm+1..vn)} ∪ ds
in
                                        let (t'm+1..t'n,ds'')== U_list [[ tm+1..tn ]] ds' in
                                        let (tf', ds''')== U [[ tf[v1/v1,...,vn/vn] ]] ds'' in
                                            (f_new(t'm+1,..,t'n),ds''')
                                        new function to repository
                                            f_new(vm+1,..,vn)=tf'
    U_list [[ [] ]] ds              <=  ([],ds)
    U_list [[ t::rest ]] ds         <=  let (t',ds')==U [[t]] ds in
                                        let (rest',ds'')==U_list [[rest]] ds' in
                                            (t'::rest',ds'')
```

Figure 2.4 - Modified Rules to Carry the Definition Set, ds

2.4.3.4. A Version of $\mathcal{U}$ which Returns Scripts

Furthermore, as our transformation tactic is supposed to return a script rather than the transformed expression, we should actually change the meta-function to the type:

```
U : expr  # location # def_set -> script # def_set ;
```

where location indicates the position of the present expression, expr, within its equation.

This new version of meta-function, $\mathcal{U}$, is shown in Figure 2.5. Again, there is no major problem involved in the change. Instead of returning a transformed expression, we simply return a corresponding script that records the desired transformation. The only additional work needed is the correct maintenance of the location parameter. This parameter is used to indicate the correct tree-address for the current sub-expression.

```
dec U : expr # location # def_set -> scripts # def_set
dec U_list : list(expr) # location # def_set # num -> scripts # def_set
(1)  U[[v]]loc ds                    <=   (nil_script,ds)
(2)  U[[c(t1,...,tn)]]loc ds         <=   U_list[[t1..tn]] loc<>[2] ds 1
(3)  U[[case t0 in p1 => t1 | .. | pr => tr end]]loc ds
                                     <=   U_list[[t0t1..tr]] loc ds 0
(4)  U[[f(t1^A..tm^A,tm+1..tn)]] loc ds
     a) if m=0 (no useless parameters)
                                     <=   U_list[[t1..tn]] loc<>[2], 1
     b) if m>0 and f_old(vm+1,...,vn)=f(v1..vm,vm+1..vn) ∈ ds; then (fold)
                                     <=   let (sc,ds')==U_list[[tm+1..tn]] loc<>[2],ds, n+1 in
                                            (compose[fold(locn loc,eqspec "f_old(vm+1,...,vn)"),sc],ds')
     c) otherwise (m>0) (define & unfold)
                                     <=   let ds'== {f_new(vm+1,...,vn)=f(v1..vm,vm+1..vn)} ∪ ds in
                                          let (sc_arg,ds")== U_list[[tm+1..tn]] loc<>[2] ds' 1 in
                                          let (sc_tf, ds'")== U[[tf[v1/v1,...,vn/vn]]][2] ds" in
                                          (compose[define(" f_new(vm+1,...,vn)=f(v1..vm,vm+1..vn)",
                                               compose[unfold(locn [2],eqspec "f(v1,...,vn)"),sc_tf]
                                               ,sc_arg] ,ds'")
     U_list[[[]]]loc ds no           <=   (nil_script,ds)
     U_list[[t::rest]] loc ds no     <=   let (st,ds')==U[[t]]loc<>[no] ds in
                                          let (srest,ds")==U_list[[rest]]loc ds' no+1 in
                                            (compose[st,srest],ds")
```

Figure 2.5 - Modified U Rules to Return Transformation Scripts

In the presentation of the other algorithmic tactics in Chapter 3 and 4, we shall leave out the versions of meta-functions which return scripts or carry the definition set. These corresponding versions can be easily formulated, as shown here for the U rule. We shall ignore them to simplify the presentation of our tactics.

2.5. A Schematic Tactic: Conversion-to-Iteration

The next tactic to be presented is the *conversion-to-iteration* tactic for transforming linear recursive functions to tail-recursive equivalent. Tail-recursive functions are desirable because they correspond directly to the imperative while loops.

For example, the higher-order (generic) tail-recursive function, f_it, shown below,

```
- - -f_it(x,w,B,H,N)   <=   if B(x) then w                  ! a tail recursive function
                           else f_it(N(x),H(x,w),B,H,N);
```

is actually equivalent to the following function from an *imperative* language, e.g. Pascal, where while loops and assignment statements are allowed.

```
f_it(x,w,B,H,N)      =    while not(B(x)) do              ! an imperative function
                           begin
                            w:=H(x,w); x:=N(x)
                           end
                          return w;                        ! result of function call is in w
```

Successful transformations to tail-recursive form could thus improve recursive functional programs by eliminating the need for the storage-expensive recursion mechanism.

The general techniques for conversion-to-iteration transformations are now pretty well-understood. One of the earliest presentations for some of these techniques were made in [Cooper66] and [Darlington76]. More recent and thorough treatments can also be found in [Bauer82] as *transformation schemes* and in [Harrison86] as *theorems* between linear-recursive functions and their tail-recursive equivalent. These techniques have largely been presented as transformations between pairs of equivalent program schemes.

The most general of these transformation schemes could convert *any* linear-recursive function to its tail-recursive equivalent by employing a two-pass iteration. The first-pass is used to find the base-case value of the last recursive call and to collect a data stack of argument values. These are then used in the second iterative pass to compute the function call, bottom-up, from the base-case value. However, this most general scheme offers little improvement because of the need for a two-pass iteration and a data stack.

More efficient conversion-to-iteration transformations normally require additional syntactic and semantic conditions to be present. These extra conditions can be specified in alternative transformation schemes. A comprehensive list of these schemes is documented in [Bauer82]. They can be combined into a conversion-to-iteration tactic for transforming any applicable linear recursive function to its tail-recursive counterpart.

Program transformation techniques, like conversion-to-iteration, which can be structured into a limited number of general transformation schemes, are suitable for formulation as schematic transformations. This category of transformations has traditionally been specified in catalogue systems, like CIP, where the schematic rules are provided and pre-proved by users/designers.

What we would like to show in this thesis is that schematic rules can also be present and applied in generative-set systems. The main difference is that these schematic rules would be *constructively* built from the elementary rules, rather than directly provided by the users. The advantage in doing this is that separate proofs for these rules will be unnecessary because they will be correct by construction.

Specifically, we would like to show that schematic rules can be provided under the unfold/fold methodology. In CIP, a wider form of scheme expressions (where free functional variables are allowed) was provided to specify schematic rules. We shall show that this is unnecessary because program schemes are just generic programs which can be formulated using *higher-order* functions. In addition, the applicability conditions of schematic rules can be specified (in our higher-order programs) as attached *constraints*.

In the rest of this section, we shall *construct* a simple schematic rule using the unfold/fold rules and then examine the techniques required to *apply* it within our transformation methodology.

2.5.1. Constructing a Schematic Rule

One sub-class of linear recursive functions which always have an efficient tail recursive equivalent, is commonly referred to as having a special *right-commutative* outer functional and *constant-at-termination* property. This sub-class of functions can be represented by the

following higher-order function:

```
- - -f1(x,B,H,N,c)        <= if B(x) then c
                            else H(f1(N(x),B,H,N,c));
```

with four fixed parameters, B,H,N,c, and one recursion parameter, x. The base case result of this function, c, is a constant (hence the term constant-at-termination) and its outer function operator, H, is right-commutative because it does not use the recursion variable, x, as one of its inputs.

We select this sub-class of functions to illustrate because its transformation has never been done before under the unfold/fold framework. This is because it requires the constraint specification mechanism which was not available in (un-extended) functional programs.

Specifically for this transformation, we will need to define a new function, call it f1_it, in terms of the old function, f1, as follows:

$$\text{- - -f1_it}(x,w,B,H,N) \;\Leftarrow\; H^n(f1(x,B,H,N,c)) \text{ st } w=H^n c \; ; \qquad (2.1)$$

This new function has an additional parameter, w. It also has a RHS definition that is rather unusual as it applies the outer functional parameter, H, to the function call, f1(x,B,H,N,c), n times where n is a logical variable that is specified indirectly through the constraint $w=H^n c$. The n-times application of the parameter, H, can be defined as follows.

$$H^0(x) = x$$
$$H^{n+1}(x) = H^n(H(x))$$

Also, the constraint used, $w=H^n c$, is solely to assist in program transformation and is not suitable for direct execution. With its help, we are able to transform this new function into an efficient tail-recursive equivalent.

```
- - -f1_it(x,w,B,H,N)  <=1  Hⁿ(f1(x,B,H,N,c)) st w=Hⁿc;                ! unfold f1
                      <=2  Hⁿ(if B(x) then c
                           else H(f1(N(x),B,H,N,c)) st w=Hⁿc;          ! distribute if construct
                      <=3  if B(x) then Hⁿ(c)
                           else HⁿH(f1(N(x),B,H,N,c)) st w=Hⁿc;        ! substitute w=Hⁿc
                      <=4  if B(x) then w
                           else HⁿH(f1(N(x),B,H,N,c)) st w=Hⁿc;        ! law HⁿH=Hⁿ⁺¹
                      <=5  if B(x) then w
                           else Hⁿ⁺¹(f1(N(x),B,H,N,c)) st w=Hⁿc;       ! law a=b → H(a)=H(b)
                      <=6  if B(x) then w
                           else Hⁿ⁺¹(f1(N(x),B,H,N,c) st Hw=Hⁿ⁺¹c; ! fold f1_it
                      <=7  if B(x) then w
                           else f1_it(N(x),H(w),B,H,N)
```

Figure 2.6 - Transforming a Generic Function to its Tail-Recursive Equivalent

The detailed unfold/fold sequence of transformation steps to achieve this is shown in Figure 2.6. Notice how the constraint is used to simplify the expression, $H^n(c)$, to w, in step $\Leftarrow_4$. Also, the fold rule in step $\Leftarrow_7$, requires $w=H^n(c)$ to be converted to $H(w)=H^{n+1}(c)$ before the

main recursive expression can be successfully folded.

The final transformed equation for f1_it, shown below, is tail recursive and is free of the previous constraint because the logical variable, n, is no longer required in the main expression.

```
- - -f1_it(x,w,B,H,N)   <= If B(x) then w
                          else f1_it(N(x),H(w),B,H,N);
```

To complete our schematic rule, there is one last thing to be done. Notice that the relationship between the old and the new functions, in the original definition (2.1) of f1_it, is not totally clear. This can be made explicit by instantiating the logical variable n to 0, followed by simplification steps, as shown below.

Instantiate with n=0

f1_it(x,w,B,H,N) = H^0(f1(x,B,H,N,c)) **st** w=H^0c

Apply Law $H^0x=x$

f1_it(x,w,B,H,N) = f1(x,B,H,N,c) **st** w=c

Substitute w=c **from Constraint**

f1_it(x,c,B,H,N) = f1(x,B,H,N,c)

The final equation from this simplification, f1_it(x,c,B,H,N)= f1(x,B,H,N,c), directly relates the tail recursive function with the original linear recursive function, and is a representation of our schematic rule between higher-order functions. In the unfold/fold methodology, this schematic rule can be regarded as a lemma between two generic functions. With it, any function that is an instance of f1 can be safely transformed into its tail-recursive form f1_it. We can represent this schematic transformation rule with the equality relationship, as shown below.

Schematic Rule 1 - for f1_it

f1(x,B,H,N,c) = f1_it(x,c,B,H,N)

In this particular case, no constraints are needed for the equality to hold. Other schematic rules may be more complex (see Appendix 2).

2.5.2. Applying the Schematic Rule

Having shown the way schematic rules are constructed, we are now ready to examine how they can be used to transform expressions within the unfold/fold system. Each of these schematic rules is, in essence, an equality relationship (possibly with some constraints) between two generic expressions involving higher-order functions. This rule can be applied to instances of expressions, which match the first generic expression (and satisfy the constraints), by replacing them with corresponding instances of the second generic expression.

However, the primary form of matching used in the unfold/fold rules is *first-order* matching. This is unfortunately not appropriate for generic expressions of schematic rules which may have function-type variables to instantiate. For generic expressions, it is better to use the *second-order matching* technique of [Huet78] to help us determine if an expression (or a function) is an instance of another given generic expression (or function).

More specifically, we have formulated a matching technique, called Function Equivalence Match (FEM), that is based on a mix of *equational reasoning* and *second-order matching* for

testing to see if a function is an instance of another higher-order function. This technique begins by equating a specific function to the generic one and then apply equational reasoning to see if the two functions match, and if so, obtain the instantiations for complete function equivalence. Like inductive proofs (of Section 2.2.5), we shall see that this technique can also be embedded within the unfold/fold framework.

As an example, consider the following linear recursive function for finding the length of a list.

```
- - -length([])      <= 0;
- - -length(a::x)    <= 1+length(x);
```

This function is actually an instance of the higher-order function, f1. To test if this is indeed the case, we must initially define an equivalence function, call it match1, to equate the length function to an f1 call, as shown below.

```
- - -match1(x,B,H,N,c)    <= length(x) = f1(x,B,H,N,c);
```

In this equivalence definition, each of the *unchanged (or fixed)* parameters, i.e. B,H,N,c, is kept independent while the *changing (or recursion)* parameters, i.e. x, of the two calls are made equal. Which parameters are fixed and which are changing (in a recursive function) can be detected by simple parameter analysis.

The detailed transformation to show that the two function calls match is given in Figure 2.7. The fold step in $<=_{12}$ corresponds to an induction step during the matching. Also, second-order matching is applied in $<=_{14}$ to instantiate the function-type parameter variable, H, as H(z)=1+z.

The end-result of this transformation-based matching is shown below.

```
- - -match1([],B,H,N,c)    <=   B([])=true ∧ 0 = c;
- - -match1(a::x,B,H,N,c)<=   B(a::x)=false ∧ N(a::x)=x ∧ 1+z = H(z)
                               ∧ match1(x,B,H,N,c);
```

It essentially contains the following substitutions for the four fixed parameters,

```
B    =   lambda [] => true | a::x => false end
N    =   lambda a::x => x end
H    =   lambda z => 1+z end
c    =   0
```

in order for function length to be made equal to an f1 function call.

There are three consistency checks which can be used to determine if a matching task is successful. They are:

1) no part of the conjunctive expression must ever be false
2) each equality relationship (obtained from the matching) which involves a fixed functional parameter, e.g. B,N,H, must be a valid equation, e.g. B([])=true but not B([])=x where x is not a fixed parameter[7].
3) descendant recursive call, e.g. match1(x,B,H,N,c), are allowed. This is because it represents a match for the recursive function.

[7] Fixed parameters, being a constant, can appear on the RHS of an equality without having to be present on the LHS.

Define a new matching task

```
- - -match1(x,B,H,N,c) <=1   length(x) = f1(x,B,H,N,c)
```

Unfold length([])

```
- - -match1([],B,H,N,c) <=2   0 = f1 ([],B,H,N,c)                                  ! unfold f1
                       <=3   0=if B([]) then c else H(f1(N([]),B,H,N,c))           ! law C(e) = (e=z) ∧ C(z)
                       <=4   B([])=z ∧ 0=if z then c else H(f1(N([]),B,H,N,c))     ! instantiate z=true
                       <=5   B([])=true ∧ 0=if true then c else H(f1(N([]),B,H,N,c))
                                                                                   ! law if true then a else b = a
                       <=6   B([])=true ∧ 0=c
```

Unfold length(a::x)

```
- - -match1(a::x,B,H,N,c)<=7   1+length(x) = f1(a::x,B,H,N,c)                     ! unfold f1
                       <=8   1+length(x) = if B(a::x) then c else H(f1(N(a::x),B,H,N,c))
                                                                                   ! law C(e) = (e=z) ∧ C(z)
                       <=9   B(a::x)=z  ∧ 1+length(x) = if z then c else H(f1(N(a::x),B,H,N,c))
                                                                                   ! instantiate z=false
                       <=10  B(a::x)=false  ∧ 1+length(x) = H(f1(N(a::x),B,H,N,c))
                                                                                   ! law C(e) = (e=z) ∧ C(z)
                       <=11  B(a::x)=false ∧ 1+y =H(f1(N(a::x),B,H,N,c))
                              ∧ length(x)=y                                        ! fold against match1
                       <=12  B(a::x)=false ∧ 1+y =H(f1(N(a::x),B,H,N,c))
                              ∧ y=f1(x,B,H,N,c) ∧ match1(x,B,H,N,c)                ! subs y=f1(x,B,H,N,c)
                       <=13  B(a::x)=false ∧ 1+f1(x,B,H,N,c) = H(f1(N(a::x),B,H,N,c)
                              ∧ match1(x,B,H,N,c)                                  ! second-order match
                       <=14  B(a::x)=false ∧ N(a::x)=x ∧ 1+z = H(z)
                              ∧ match1(x,B,H,N,c)
```

Figure 2.7 - Function Equivalence Matching (FEM) between length and f1

This transformation-based matching has therefore established the following equality.

```
length(x)   =   f1(lambda [] => true | a::r => false end,
                    lambda a::r => r end, lambda x=> 1+x end, 0, x)
```

It could be used as an alternative definition for length which could then be transformed (using Schematic Rule 1) to:

```
length(x)   =   f1_it(lambda [] => true | a::r => false end,
                    lambda a::r => r end, lambda x=> 1+x end, 0,x)
```

We now have a function, length, which is tail-recursive because it calls the tail-recursive function, f1_it. However, there are some higher-order arguments present in this call which can be removed. This removal can be automatically applied by another tactic, called higher-order removal tactic, which is described in Chapter 4. Its application will give us following efficient first-order program.

```
- - -length(x)        <= length_it(x,0);
- - -length_it([],w)  <= w;
- - -length_it(a::x,w) <= length_it(x,w+1);
```

The final program is now tail-recursive and could be directly compiled into an efficient while

loop by a functional language compiler.

The FEM technique has not been implemented yet. At first glance, this looks like an arbitrary hard technique because we have formulated it as an equational reasoning between two function calls. Indeed, such equational reasoning is, in general, an undecidable problem because we could also use it to test if two functions are equivalent under this framework! However, the primary task of FEM is to test if a function is an instance of another higher-order function. Because of this, we could (by suitable restrictions) view the FEM technique as a simple extension of the second-order matching algorithm of Huet & Lang [Huet78]. The simple extension involves the ability to match a pattern-matching function with a conditional function and the independent treatment of fixed parameters (see Section 2.5.4 on a discussion of this extension). With appropriate restrictions, the author believe that it should be possible to build an FEM technique that is no less automatable than the second-order matching algorithm. This work is, however, left to a later date.

2.5.3. The Stages for Rule Applications

The above application of schematic rules can be summarised as consisting of three stages. These are:

1) Function Equivalence Matching to check if a function is an instance of another higher-order function.
2) Transformation of expression by (a) replacing each matching function call with its equivalent higher-order function call and then (b) apply the appropriate schematic rule between higher-order functions.
3) Tidy-up by removing all unnecessary higher-order arguments, where possible.

The first two stages can be viewed as the *analysis* and *transformation* phases of the conversion-to-iteration tactic. These two phases are sufficient for obtaining tail-recursive program, albeit a higher-order one. As for the third tidying-up stage, we will regard it as the application of a separate transformation tactic (which normally follows our conversion-to-iteration tactic).

To illustrate the generality of this schematic rule, let us consider a linear recursive version of the fibonacci function.

```
- - -fib_tup(0)        <= (1,1);
- - -fib_tup(n+1)      <= H(fib_tup(n));
- - -H(u,v)            <= (u+v,u);
```

This function can be similarly shown by FEM to be an instance of the higher-order function, f1. Application of a subsequent transformation step to f1_it, followed by the removal of higher-order arguments would similarly convert fib_tup to the following tail-recursive equivalent.

```
- - -fib_tup(n)        <= fib_it(n,(1,1));
- - -fib_it(0,(u,v))   <= (u,v);
- - -fib_it(n+1,(u,v)) <= fib_it(n+1,H(u,v));
```

2.5.4. Wider Applicability of our Approach

Our application of schematic rules, involving detailed steps for matching and an additional stage for higher-order removal, may look unnecessarily long but it is more flexible. The earlier example already illustrated this flexibility by successfully matching a *conditional* function, f1, with a *pattern-matching* function, length. In addition to this, the transformation-based matching is not obstructed (or confused) by the presence of parameter variables - a perceived *weakness* often cited by proponents of function-level languages, like FP, [Harrison86,Kieburtz81] against object-level functional languages, like HOPE.

To see why, we shall illustrate an example where the presence of extra parameters does not confuse our technique. This example, add_len, is a slightly modified version of the function length, with an extra fixed parameter, b, that is added to its result for each item of its input list.

```
- - -add_len([],b)     <= 0;
- - -add_len(a::x,b)   <= b+add_len(x,b);
```

Direct second-order matching, as used by catalogue systems like CIP, between this function and the program scheme for f1 will fail. This is because the extra parameter, b, will upset the matching.

However, the more flexible Function Equivalence Matching is able to find a match between add_len and f1. This is done by initially providing the following equivalence definition,

```
- - -match2(x,b,B,H,N,c)  <= add_len(x,b) = f1 (x,B,H,N,c);
```

where each of the fixed parameters (including b of add_len) is kept independent. After that, a sequence of corresponding transformation and matching steps could be applied to try to prove that add_len is an instance of f1. These steps would result in the following.

```
- - -match2([],b,B,H,N,c)  <= B([])=true ∧ 0 = c;
- - -match2(a::x,b,B,H,N,c)<= B(a::x)=false ∧ N(a::x)=x ∧ b+z = H(z)
                              ∧ match2(x,b,B,H,N,c);
```

The above substitutions for the fixed parameters could then be used to provide an alternative definition for add_len, in terms of f1, before it is transformed to a tail-recursive f1_it call, as shown below.

```
- - -add_len(x,b)    <= f1(lambda [] => true | a::r => false end,
                          lambda a::r => r end,lambda x=> b+x end,0,x);
                    <= f1_it(lambda [] => true | a::r => false end,
                          lambda a::r => r end,lambda x=> b+x end,0,x);
```

A final tidying-up stage could similarly be applied to remove the higher-order arguments. This gives the following efficient tail-recursive program.

```
- - -add_len(x,b)       <= add_len_it(x,b,0);
- - -add_len_it([],b,w)  <= w;
- - -add_len_it(a::x,b,w) <= add_len_it(x,b,w+b);
```

Using this approach, our matching technique is less sensitive to the presence of additional parameter variables. We simply have to equate changing parameters together and allow the fixed parameters to be independent for FEM.

2.5.5. Benefits of Schematic Rules

So far, we have presented a conversion-to-iteration scheme which applies to programs that have the right-commutative and constant-at-termination property. Apart from this particular scheme, there are also a number of other conversion-to-iteration schemes (with different syntactic and semantic conditions) for linear recursive functions. These other schemes would be needed in a more complete transformation tactic. For this purpose, we have presented three more conversion-to-iteration schemes in Appendix 2. In particular, we show how these schemes can also be constructed directly using the unfold/fold rules.

One of the primary benefits of organising transformations into schematic rules is that they represent large, generally-applicable rules which are easier to apply. The clever combinations of elementary steps (in Figure 2.6) need only be applied once to discover the schematic rules. Once discovered, the result of each of these rules could be applied to any program instance which matches the source program scheme and satisfies its constraints. The matching and some syntactic constraints can be checked by the transformation system. In addition, we may also need the more general inference capability to verify the semantic constraints. These checks are often much easier than trying to re-discover the clever combinations of steps for each program.

Another potential benefit of the schematic rules is that they could be used as the basis of a more sophisticated *learning component* within our program transformation system. In [Chin89], we outlined this possibility by introducing a learning capability, based on the *explanation-based generalisation* technique [DeJong86, Mitchell86]. Explanation-based generalisation technique is a recent learning technique which could learn from *single* examples with the help of domain knowledge that is able to explain or reason about the examples. This contrasts with *similarity-based generalisation* technique [Mitchell82,Michalski83] which could only learn from two or more close examples by noting their similarities and differences.

In the case of program transformation, we can learn from a single transformation instance (containing source program, target program and script) by generalising it to a transformation scheme (source scheme, target scheme, and constraints). This is achieved by applying generalisations to the source and target program to obtain their generic (higher-order) counterparts in such a way that all critical transformation steps (serving as explanations) are preserved in the new scheme (particularly the steps leading to the self-recursive fold). Schematics rules are suitable for this form of learning because they capture generalised versions of program classes quite naturally.

2.6. Summary of Chapter 2

In this chapter, we outlined the key components of a comprehensive transformation system, based on the unfold/fold methodology, which satisfies the *generality*, *safety* and *automatability* criteria. An important component, from the point of view of automated assistance, is the methods/tactics bank which contains *meta-programs* as our transformation tactics. These meta-programs are required to return scripts (for the transformation processor) rather than make direct updates to program repository itself. Such an arrangement helps to ensure safety.

We also noted that tactics are usually structured into an analysis and a transformation phase, with the analysis phase playing a key role for automating transformations. To illustrate this structure, we presented two very simple tactics, namely a useless parameter elimination tactic and a conversion-to-iteration tactic. The algorithmic useless parameter elimination tactic uses an abstract interpretation technique to help identify the useless parameters before its transformation algorithm is applied to remove these redundant parameters. Correspondingly, the schematic conversion-to-iteration tactic uses a special matching technique (based on equational reasoning and second-order matching) to check if a program is an instance of another higher-order one. This matching effectively represents the necessary analysis (before transformation) to indicate if a given program can be transformed to its tail-recursive equivalent.

In the next four chapters, we shall present four classes of more substantial tactics. Each of these tactics will involve some form of analysis before transformations. Also, each of them will be based on the elementary unfold/fold rules.

3. Fusion

3.1. Introduction

Structural and modular approaches to software development, as first advocated in [Dijkstra72, Parnas72], involve a style of program development where sound techniques (like abstraction, modularisation, hierarchical development, information hiding) are used to structure large systems into meaningful components. This approach has been shown to produce systems that are generally more complete and robust. In terms of actual implementation, the software is often structured into hierarchies of modules, where complex programs are built or *composed* from simpler ones. Such modularisation helps to simplify the task of implementing large systems. It also aids in software reusability.

Though good for software development, modular or compositional styles of programming is, in general, less efficient. This is because compositions of separate programs, while offering better clarity and modularity, usually involve *more* computation time and space than if they were integrated or fused together. The process of integrating separate programs, which are in some form of composition, into a single definition can often be achieved by a program transformation technique that is commonly referred to as **fusion**. The availability of this technique is extremely useful coupled with the modular style software development. It implies that we can continue to use this style without incurring the runtime penalties of modularisation.

We examine two common forms of program composition in functional programs and look at how the fusion transformation can help to improve their efficiency.

3.1.1. Two Forms of Composition

Two possible ways in which programs can be composed together in functional languages are

- functional composition, where function calls are connected by nesting (or juxtaposition).
- set abstractions, where function calls are connected by intermediate variables.

3.1.1.1. Functional Composition

Functional composition is one of the basic means by which programs are put together in applicative languages. Essentially, this is used to route the output of one function call to the input of another. A simple example of this is the composition, F(G(x)), in the definition below

```
---P(x)          <= F(G(x)) ;
```

where the inner function call, G(x), is supposed to produce an *intermediate output* for the outer function call, F.

Such a composition of functions can be inefficient. In particular, if the intermediate output *produced* by the inner function call is a *large (recursive) data structure*, our program may incur considerable amount of storage and computation expense in order to construct this data before it is used or *consumed* by the outer function call.

An example to illustrate this phenomenon is the following definition of sumdb.

```
dec sumdb,sum: list(num) -> num;
dec double: list(num) -> list(num);
dec sqr: num -> num;
- - -sumdb(x)        <= sum(double(x));
- - -sum([])        <= 0;
- - -sum(a::x)      <= a+sum(x);
- - -double([])     <= [];
- - -double(a::x)   <= 2*a::double(x);
```

This function, sumdb, has a composition of two simpler functions, sum and double. A call to it would cause the inner call, double(x), to produce an *intermediate* list by doubling every item in its input list, x, before the intermediate list is totalled up by the outer function, sum. This creation of an intermediate list is expensive (for both strict and non-strict languages).

A more efficient definition can be obtained by fusing the two composed function calls together and in the process, eliminates the need for the intermediate list. This can be done by a straightforward sequence of transformations, involving only the unfold and fold steps, as shown in Figure 3.1. Our transformation initially unfolds (narrows) the inner call, double(x). This produces two symbolic intermediate terms at steps $<=_2$ and $<=_4$, which can be partially consumed by unfolding the outer function call. One of these two steps, results in a base case value of 0 (at $<=_3$), while the other results in an expression which can be folded against the original definition of sumdb (at $<=_6$). The end-result is a more efficient fused definition, shown below, which no longer needs the intermediate list.

```
- - -sumdb([])      <= 0 ;
- - -sumdb(a::x)    <= 2*a+sumdb(x) ;
```

- - -sumdb(x)	$<=_1$	sum(double(x))	! narrow double(x)
- - -sumdb([])	$<=_2$	sum([])	! unfold sum([])
	$<=_3$	0	
- - -sumdb(a::x)	$<=_4$	sum(2*a::double(x))	! unfold sum(a::x)
	$<=_5$	2*a+sum(double(x))	! fold back to sumdb
	$<=_6$	2*a+sumdb(x)	

Figure 3.1 - Fusion Transformation on sumdb

This kind of transformation is also known as *(vertical) loop combination*[1]. The merging of the two recursive calls into a single self-recursive definition, has the effect of *halving* the number of function calls required. It can thus be viewed as combining the two recursive loops into a single equivalent loop. Martin Feather makes heavy use of this particular transformation technique (also called *combining* tactic) to semi-automatically transform (using his ZAP transformation system) a small multi-pass compiler into a more efficient two-pass compiler [Feather79,82].

[1] There is also another form of loop combination, called horizontal loop combination, which involves the tupling of independent recursive calls instead of the fusion of nested calls. An example of this was given in Section 1.3.3. This horizontal loop combination can be considered as a special case of the Tupling Tactic in Chapter 5.

3.1.1.2. Set Abstraction

Set abstractions provide another way in which program units can be composed together. In this case, intermediate variables are used to link separate program units together. This is frequently used to concisely specify *generate and test* programs. A schematic example of this is the following function,

$$---prog(x) \quad <= \quad \{p | p \in gen(x), filter_1(p), ..., filter_n(p)\} ;$$

where each output represented by intermediate variable, p, is taken from a generator, gen(x). This intermediate output must satisfy every test, $filter_1(p),...,filter_n(p)$, before it is accepted as one of the results.

The use of (relative) set abstraction and the associated programming style provides higher-level abstractions to functional programs. However, this style of generate and test programming is often very inefficient. This is because loosely composed generators and filters frequently cause many infeasible intermediate solutions to be generated. Such inefficiency can normally be removed by using a fusion technique to merge the generator and filters together. This has the potential benefit of obtaining a superior generator, which could automatically cut off some *sub-spaces* of infeasible solutions. These cut-offs are achieved by parts of the filter being *promoted* into the new generator during the transformation - hence the name **filter promotion** [Darlington78] for this kind of transformations.

As an example, consider the following generate-and-test program for finding all complete paths in a set of nodes for a given graph.

```
dec all_path,perm: set(nodes) -> list(nodes);
dec path: list(nodes) -> truval;
dec path': nodes # list(nodes) -> truval;
dec arc: nodes # nodes -> truval;
---all_path(s)      <= {p | p ∈ perm(s),path(p)} ;
---perm({})         <= {[]} ;
---perm(s)          <= {a::p | a ∈ s,p ∈ perm(s-a)} ;
---path([])         <= true ;
---path(a::x)       <= path'(x,a) ;
---path'([],a)      <= true ;
---path'(b::x,a)    <= arc(a,b) ∧ path'(x,b) ;
```

The main function of this program, all_path, has a generator, p ∈ perm(s), that produces all possible permutations, p, from a set of nodes, s. Each of these solutions is then tested by path (which make use of a graph-specific function arc) to check if it is a valid path within the given graph.

This classic generate-and-test program is grossly inefficient. To improve its efficiency, we can apply the fusion transformation, as shown in Figure 3.2. This transformation results in the following fused program, with an auxiliary function, all_path', introduced at step $<=_{10}$.

```
- - -all_path({})      <= {[]} ;
- - -all_path(s)      <= {a::p | a ∈ s,p ∈ all_path'(s-a,a)} ;
- - -all_path'({},a)  <= {[]} ;
- - -all_path'(s,a)   <= {b::p | b ∈ s,arc(a,b),p ∈ all_path'(s-b,b)} ;
```

The above program is a more efficient because part of the filter, denoted by arc(a,b), has been promoted into the recursive generator of all_path' during the transformation. As a result, subspaces of infeasible solutions are now avoided by the better generator. Before this transformation, the complexity of our program was O(n!), where n is the number of nodes. With filter promotion, the complexity of the transformed program has been reduced to between O(n) and O(n!), depending on the connectivity of the input graph.

```
- - -all_path(s)      <=1   {p | p ∈ perm(s),path(p)}                  ! unfold perm(s)
- - -all_path({})     <=2   {p | p ∈ {[]},path(p)}                     ! substitute p
                      <=3   {[] | path([])}                            ! unfold path([])
                      <=4   {[] | true}                                ! remove true
                      <=5   {[]}
- - -all_path(s)      <=6   {p | p ∈ {a::p | a ∈ s,p ∈ perm(s-a)},path(p)}  ! substitute p
                      <=7   {a::p | a ∈ s,p ∈ perm(s-a),path(a::p)}     ! unfold path
                      <=8   {a::p | a ∈ s,p ∈ perm(s-a),path'(p,a)}     ! define all_path'
                      <=9   {a::p | a ∈ s,p ∈ all_path'(s-a,a)}
define
- - -all_path'(s,a)   <=10  {p | p ∈ perm(s),path'(p,a)}                ! unfold perm(s)
- - -all_path'({})    <=11  {p | p ∈ {[]},path'(p,a)}                  ! simplify
                      <=12  {[]}
- - -all_path'(s,a)   <=13  {p | p ∈ {b::p | b ∈ s,p ∈ perm(s-b)},path'(p,a)}  ! substitute p
                      <=14  {b::p | b ∈ s,p ∈ perm(s-b),path'(b::p,a)}  ! unfold path'
                      <=15  {b::p | b ∈ s,p ∈ perm(s-b),arc(a,b),path'(p,b)}  ! fold all_path'
                      <=16  {b::p | b ∈ s,arc(a,b),p ∈ all_path'(s-b,b)}
```

Figure 3.2 - Filter Promotion on the function all_path

Apart from generators and filters, there is another form of composition in the set abstraction construct that may also benefit from the fusion technique. This other form of composition is between generators and *processors* which give results to the set abstraction. An example of this is illustrated in the following program,

```
- - -all_cost(s)       <= {cost_path(p) | p ∈ all_path(s)} ;
- - -cost_path([])     <= 0 ;
- - -cost_path(a::x)   <= cost_path'(x,a) ;
- - -cost_path'([],a)  <= 0 ;
- - -cost_path'(b::x,a) <= cost(a,b)+cost_path'(x,b) ;
```

where the generator is the previous function, all_path, and the processor function is cost_path for computing the cost of the entire path. Using the previously fused definition of all_path, this pair of generator and processor can be similarly transformed to give us the following fused program.

```
- - -all_cost({})      <= {0} ;
- - -all_cost(s)      <= {c | a ∈ s,c ∈ all_cost'(s-a,a)} ;
- - -all_cost'({},a)  <= {0};
- - -all_cost'(s,a)   <= {cost(a,b)+c | b ∈ s,arc(a,b),c ∈ all_cost'(s-b,b)} ;
```

The intermediate list has been completely eliminated and we have also avoided some redundant computations which are experienced by the cost function.

3.1.2. Consumer-Producer Model

Both functional composition and set abstraction share some similarity; notably, the manner in which intermediate outputs are *produced* by one function call and then *consumed* by another. As a result, the transformation techniques to fuse these composed function calls are quite similar. They typically require the *define*, *unfold* and *fold* steps, making the mechanisation of the technique seems promising.

To help us explain this fusion technique, we would like to introduce a **producer-consumer model** of functions. Under this model, each function call is viewed as both a *consumer* of inputs (through its parameters), and at the same time a *producer* of output (through its results).

The objective of the fusion transformation is to try to *fuse*, where possible, compositions of consumers and producers together so that the resulting program could operate as an integrated (strongly synchronised) and more efficient unit. We have seen fusion applied to a composition between sum (as a consumer) and double (as a producer) in order to give an integrated function, sumdb. Similarly, the composition between perm (as a producer) and path (as a consumer) was also fused to give an integrated program containing two functions, all_path and all_path'.

However, not all compositions between function calls can be successfully fused together. Some of these function calls may not be in the appropriate form for fusion. Let us call a producer that is in an appropriate form for fusion, a *good producer*, and a consumer that is in an appropriate form for fusion, a *good consumer*. Correspondingly, we shall also call consumers and producers, that are not in the appropriate forms for fusion, *bad producers* and *bad consumers*, respectively. With this characterisation, a composition can be successfully fused if both the producer and consumer are of the appropriate form (i.e. *good* producer and *good* consumer). But what are these appropriate forms?

Recently, Phil Wadler invented an important transformation technique called **deforestation** [Wadler88], which is capable of eliminating some intermediate data terms in functional compositions. He formulated this as an automatic transformation algorithm that can successfully transform any expressions which are composed solely from a *sub-class* of first-order functions. This sub-class of functions has a syntactic property, called **pure treelessness**. This property is actually a description for functions which are both *good producers* and *good consumers* in the above producer-consumer model. We shall present the notation used and main results of his work in Section 3.2.

Our contribution in this chapter is a gradual extension of this area of transformation to include all first order functions, rather than just a subset, for possible deforestation. We shall introduce

two apparatus for this extension. Firstly, in Section 3.3, we introduce a novel parameter annotation scheme which can be used to classify parameters of functions into two categories, namely those that are suitable for deforestation and those that are not. More specifically, they actually constitute a way of differentiating *good consumers* from *bad consumers*. With this annotation scheme, the class of expressions which can be deforested is considerably extended. We give the target expression form, transformation algorithm and prove that our algorithm terminates. Secondly, in Section 3.4, we widen the class of expressions which can be deforested to all first-order expressions. We achieve this by making use of the **let** construct to help make certain situations *appear* deforested. This is only needed whenever we encounter *bad producer* functions. With this apparatus, we are able to provide deforestation for all first-order programs.

We complete this area of work, in Section 3.5, by suggesting further improvements that are possible for deforestation. We also make a comparison between Turchin's supercompilation and our extended deforestation in Section 3.6. Finally, in Section 3.7, we apply analogous results to the way compositions in set abstractions can be fused together.

3.2. Wadler's Deforestation

Wadler's deforestation work is based on a simple first-order functional language, where each term or expression, t, can be described by the following one-line BNF grammar:

$$t ::= v \mid c\ (t_1,..,t_n) \mid g\ (t_0,t_1,..,t_n) \mid f\ (t_1,..,t_n)$$

Here, v is a variable, c is a constructor and f and g are two kinds of user-defined functions. These two kinds of functions are used to distinguish functions with pattern-matching parameters[2] from those without pattern-matching parameters, and are defined as follows:

$$\begin{aligned} &--f(v_1,..,v_n) && <= && tf \\ \\ &--g(p_1,v_1,..,v_n) && <= && tg_1 \\ &\quad\vdots && && \quad\vdots \\ &--g(p_r,v_1,..,v_n) && <= && tg_r \end{aligned}$$

where each $p_i=c_i(v'_1,..,v'_j)$ is a pattern and $tf,tg_1,..,tg_r$ are first-order terms.

Notice further that there are two simple restrictions placed on the pattern-matching parameters of g-type functions. These are:

- only one pattern-matching parameter (the first one) is allowed per g-type function
- each pattern, $p_i=c_i(v_1',..,v_j')$, is allowed to be only a single constructor deep. In [Augustsson85], this is called a *simple pattern.*

[2] The original deforestation work of [Wadler88] is based on a first-order language which uses **case** constructs in the place of g-type functions. This old formulation tended to produce large programs by maximally unfolding all pattern-matching functions. In this chapter, we shall use the more recent formulation by Ferguson and Wadler (in [Ferguson88]) which uses pattern-matching functions to replace **case** constructs.

Despite these two restrictions, there is no loss of generality because methods exist [Wadler87a,Augustsson85] to transform functions, involving arbitrary number of pattern matching parameters with nested patterns, to equivalent functions with the above restrictions. These methods were given for expressions involving the **case** constructs but could easily be adapted for the above language.

Far from reducing generality, the two restrictions are extremely useful. They help to simplify the forms of functions permitted and as a result make both the specification of the transformation algorithm and the proof of its termination much simpler.

3.2.1. Pure Treeless Form

Using this first-order language, Wadler and Ferguson identified a sub-class of expressions which forms the basis of deforestation. This sub-class of expressions has a special syntactic characteristic, called **pure treelessness**. An expression is said to be pure treeless[3] if it has *no intermediate terms* and is *linear*. This characteristic can be defined formally, as follows:

Definition 3.1 - Pure Treeless Form

A term is said to be *pure treeless* if it is *linear* and satisfies the following grammar:

$tt ::= v \mid c\ (tt_1,..,tt_n) \mid f\ (v_1,..,v_n) \mid g\ (v_0,v_1,..,v_n)$

WHERE f and g are *pure treeless* functions and each free variable, v, occurs only once (i.e. is *linear*) in the expression.

Correspondingly, a function is said to be *pure treeless* if its definition (RHS of its defining equations) is made up of *pure treeless* terms.

The most distinctive feature of the above pure treeless grammar is that each function call, $f(v_1,..,v_n)$ or $g(v_0,v_1,..,v_n)$, must take only variables as its arguments. This condition helps to ensure that *no* function call nesting exists and hence there are *no* intermediate terms in the expression, as required. The second criterion of requiring terms to be *linear* is also important because *non-linear* parameter arguments can cause losses in efficiency when large expressions (which are expensive to compute) are duplicated by unfolding. This is clearly undesirable and one way of preventing this is to strictly enforce the linearity criterion for all pure treeless expressions.

Three examples of pure treeless functions are:

```
dec append: list(alpha) # list(alpha) -> list(alpha);
dec rev_tree: list(alpha) # list(alpha) -> list(alpha);
dec twos: list(num);
- - -rev_tree(leaf(a))        <= leaf(a);
- - -rev_tree(node(x,y))      <= node(rev_tree(y),rev_tree(x));
- - -append([],y)             <= y;
- - -append(a::x,y)           <= a::append(x,y);
- - -twos()                   <= 2::twos();
```

[3] has been called pure treeless form to distinguish it from other restricted or extended forms of treelessness. We shall see the other forms of treelessness later.

Notice that each RHS term of these three functions is linear and satisfies the pure treeless grammar. The first two functions are pattern-matching g-type functions, while the third function is an f-type function without any parameters.

3.2.2. Deforestation Theorem

The pure treeless form is actually a way of characterising functions which are both good producers and good consumers. With this treeless form, Wadler was able to formulate the main result of deforestation as the following theorem.

Theorem 3.2 - Deforestation

> Every linear term, containing only occurrences of (pure) treeless functions, can be effectively transformed, by the Deforestation Algorithm, to an equivalent (pure) treeless term, without loss of efficiency.

Under this theorem, the sub-class of expressions which can be transformed are those making use of only *pure treeless* functions. An expression which belongs to this sub-class is append(append(x,y),z). This expression satisfies the requirements of the Deforestation Theorem and can thus be transformed to an equivalent treeless expression by the Deforestation Algorithm. The rules of this algorithm, denoted by ***T***, are given and described in the next section. We briefly illustrate, in Figure 3.3, how this transformation algorithm works by applying it to the simple expression, append(append(x,y),z). The end result of this transformation is a treeless expression, app(x,y,z), which uses a newly defined function, app, that is also transformed to treeless form. The resulting program is more efficient because it no longer produces an intermediate list.

```
T [[ append(append(x,y),z) ]] can be transformed by deforestation to:
                    ⇒   T [[ append(append(x,y),z)]]        ! T 7, define app
                    ⇒   app(x,y,z)
define
- - -app(x,y,z)     <=  append(append(x,y),z)               ! unfold append(x,y)

- - -app([],y,z)     <=  T [[ append(y,z) ]]                 ! T 4, treeless call
                    <=  append(y,z)
- - -app(a::x,y,z)  <=  T [[ append(a::append(x,y),z) ]]     ! T 5, unfold append
                    <=  T [[ a::append(append(x,y),z) ]]    ! T 2, skip constructor
                    <=  T [[ a ]] ::T [[ append(append(x,y),z) ]]  ! T 1, skip variable
                    <=  a::T [[ append(append(x,y),z) ]]    ! fold against app
                    <=  a::app(x,y,z)
```

Figure 3.3 - Deforesting a Simple Expression with *T*

3.2.3. Transformation Algorithm

The deforestation algorithm for the Wadler's theorem is made up of seven[4] rules in Figure 3.4. These rules rely principally on the *syntax* of expressions to direct transformation steps, that are made up of the define, unfold and fold operations. We briefly describe each of these rules next.

Rules $\mathcal{T}$ 1 and 2 are used to handle variables and outer constructor terms. These rules leave the outer expressions intact and transform the inner sub-expressions, where applicable. Rules $\mathcal{T}$ 3 and 4 are used to deal with f- and g-type function calls which are already treeless (with variables as their arguments). Such calls are just left unchanged.

The main deforestation rules are $\mathcal{T}$ 5, 6 and 7, which may each have one or more intermediate terms to eliminate. These three rules use a context notation, ...t..., to describe a sub-term, t, inside a *nesting* of pattern-matching g-type function calls, where:

$$...t... ::= t \mid g\ (...t...,t_1,...,t_n)$$

More specifically, the context notation is used to refer to an *innermost* function call, last_call, in a *nesting* of g-calls, e.g. $g_1(g_2(...g_N(\text{last_call},t_1,..,t_n)...,t_1',..,t_n'),t_1'',..,t_n'')$, where there are zero or more ($N \geq 0$) outer g-calls, $g_1,..,g_N$. For example, the innermost call for the expression, append(append(x,y),append(a,b)), is append(x,y) and not append(a,b) because the first call lies in the pattern-matching parameter of the outer append function call but not the second call.

Rule $\mathcal{T}$ 6 handles each innermost f-type function call, while rule $\mathcal{T}$ 7 handles each innermost g-type function call with a variable in its pattern-matching argument. Each of these calls is unfolded/narrowed to produce an intermediate result, possibly with an outer constructor. Those intermediate results with outer constructors are consumed (or propagated) by rule $\mathcal{T}$ 5, which unfolds each g-type function call with a matching constructor term as its first argument. The overall effect of these three rules is to unfold/narrow some *innermost* function calls and then propagate out any intermediate constructor terms produced. This corresponds to the typical transformation sequence for the fusion technique.

The given rules involve principally the *unfold* steps (in $\mathcal{T}$ 5,6 and 7) and a *define* step (in $\mathcal{T}$ 7). For some expressions which may re-occur during transformation, we also need the *fold* steps in this otherwise infinite transformation. These fold steps are considered to be *implicit* steps needed in Wadler's deforestation rules. They can be implemented by checking each new expression to see if it is a re-occurrence (modulo renaming[5]) of the previous expressions encountered by the $\mathcal{T}$ 6 and $\mathcal{T}$ 7 rules. Rule $\mathcal{T}$ 5 can be left out of folding because there could never be an infinite sequence of just $\mathcal{T}$ 5 steps.

---

[4] Actually, the original deforestation algorithm of Wadler consists of only 5 rules. We have added two more rules, $\mathcal{T}$ 3 and 4, to explicitly detect function calls that are already treeless and hence do not require further transformations. These are just minor optimisations.

[5] An expression which differs from another expression by just the names of variables used. In other words, an exact match.

GIVEN $f(v_1,...,v_n) = tf$
and $g(p_1,v_1,...,v_n) = tg_1$
:
$g(p_r,v_1,...,v_n) = tg_r$ where $p_i=c_i(v_1',...,v_j')$

(1) $T[\![v]\!] \Leftarrow v$

(2) $T[\![c(t_1,...,t_n)]\!] \Leftarrow c(T[\![t_1]\!],...,T[\![t_n]\!])$

(3) $T[\![f(v_1,...,v_n)]\!] \Leftarrow f(v_1,...,v_n)$

(4) $T[\![g(v_0,v_1,...,v_n)]\!] \Leftarrow g(v_0,v_1,...,v_n)$

(5) $T[\![...g\ (c_i(t_1',...,t_j'),t_1,...,t_n)...]\!] \Leftarrow T[\![...tg_i[t_1'/v_1',...,t_j'/v_j',t_1/v_1,...,t_n/v_n]...]\!]$

(6) $T[\![...f(t_1,...,t_n)...]\!] \Leftarrow T[\![...tf[t_1/v_1,...,t_n/v_n]...]\!]$

(7) $T[\![...g(v_0,t_1,...,t_n)...]\!] \Leftarrow g_new(v_0,w_1,...,w_j)$
where $w_1,...,w_j = vars[...g(v_0,t_1,...,t_n)...] - v_0$
define $g_new(v_0,w_1,...,w_j) = ...g(v_0,t_1,...,t_n)...$
unfold on the innermost call, g, ***to give:***
$g_new(p_1,w_1,...,w_j) = T[\![...tg_1\ [t_1/v_1,...,t_n/v_n]...]\!]$
:
$g_new(p_r,w_1,...,w_j) = T[\![...tg_r\ [t_1/v_1,...,t_n/v_n]...]\!]$
Note: vars[ex] will return all free variables in expression, ex.

Figure 3.4 - Wadler's Deforestation Rules

Furthermore, to enable folding, an *implicit* define step is also needed to introduce new function definitions for expressions encountered by ***T*** 6. This define step could either:

1) be made explicit (as in ***T*** 7) for each new expression encountered (by ***T*** 6) that has not been previously defined, or
2) be inserted retrospectively by re-transformation whenever an expression re-occurs.

The first option of using explicit define steps may cause too many (unnecessary) intermediate functions to be defined, while the second option (using re-transformation) is both inefficient and difficult to implement. The proposal made in [Ferguson88], uses a *labelled tree* to give a trace of steps taken and corresponds essentially to the first option where unnecessary[6] intermediate functions may be introduced. We suggest a better alternative in the next section.

3.2.4. Making all Steps Explicit

The implicit define and folding steps of Wadler's deforestation rules can be made explicit by making simple modifications to rules ***T*** 6 and 7, as shown in Figure 3.5. Firstly, we add a new parameter, ds, to carry all new function definitions introduced so far by the ***T*** rule. This set of new definitions is used to check for foldable expressions at steps ***T*** 6b and 7a.

Secondly, *explicit* define steps are used by both ***T*** 6c and ***T*** 7b. Each of these steps introduces a new function definition for each new expression encountered. However, there is an optimisation possible for expressions of the form,...$f(t_1,...,t_n)$...., which are encountered by ***T*** 6.

[6] It was suggested by Wadler, in personal communication, that unnecessary intermediate functions may be removed by an additional post-processing stage.

If the innermost f-type call is from a non-recursive function, then a re-occurrence of this expression cannot occur and we can directly unfold such innermost calls, as specified in rule *T* 6a.

For example, the expression append(twos(),y) may re-occur during transformation because the f-type function call, twos(), is recursive. This expression must therefore be handled by either *T* 6b or *T* 6c. In contrast, another expression, append(id(x),y) where id(x)=x, cannot re-occur during its transformation because the innermost f-type function call, id(x), is from a non-recursive function. It can therefore be handled in a more efficient way by *T* 6a. This optimisation will be shown (in Section 3.2.5) not to affect the termination property of the deforestation algorithm.

(6) $T[\![...f(t_1,..,t_n)...]\!]\,ds$

a) if f is non-recursive then (unfold)

$\Leftarrow \quad T[\![...tf[t_1/v_1,..,t_n/v_n]...]\!]\,ds$

b) if $f_old(w_1,..,w_j) = ...f(t_1,..,t_n)... \in ds$ **then** (fold)

$\Leftarrow \quad f_old(w_1,..,w_j)$

where $w_1,..,w_j == vars[...f(t_1,..,t_n)...]$

c) otherwise (define & unfold) $\Leftarrow \quad f_new(w_1,..,w_j)$

where $w_1,..,w_j == vars[...f(t_1,..,t_n)...]$

where $new_ds == ds \cup \{f_new(w_1,..,w_j) = ...f(t_1,..,t_n)...\}$

define $f_new(w_1,..,w_j) = ...f(t_1,..,t_n)...$

unfold on the innermost call, f, to give:

$f_new(w_1,..,w_j) = T[\![...tf[t_1/v_1,..,t_n/v_n]...]\!]\,new_ds$

(7) $T[\![...g(v_0,t_1,..,t_n)...]\!]\,ds$

a) if $g_old(v_0,w_1,..,w_j)=...g(v_0,t_1,..,t_n)... \in ds$ **then** (fold)

$\Leftarrow \quad g_old(v_0,w_1,..,w_j)$

where $w_1,..,w_j == vars[...g(v_0,t_1,..,t_n)...] - v_0$

b) otherwise (define & unfold) $\Leftarrow \quad g_new(v_0,w_1,..,w_j)$

where $w_1,..,w_j == vars[...g(v_0,t_1,..,t_n)...] - v_0$

where $new_ds == ds \cup \{ g_new(v_0,w_1,..,w_j)=...g(v_0,t_1,..,t_n)...\}$

define $g_new(v_0,w_1,..,w_j)=...g(v_0,t_1,..,t_n)...$

unfold on the innermost call, g, ***to give:***

$g_new(p_1,w_1,..,w_j)=T[\![...tg_1\ [t_1/v_1,..,t_n/v_n]...]\!]\,new_ds$

:

$g_new(p_r,w_1,..,w_j)=T[\![...tg_r\ [t_1/v_1,..,t_n/v_n]...]\!]\,new_ds$

Figure 3.5 - Modified Rules to Make All Steps Explicit

3.2.5. Proof for Deforestation Theorem

The deforestation theorem can be proved by showing the following three lemmas, which together, demonstrate the validity of the theorem.

Lemma 3.3 - Adherence to (Pure) Treeless Expression Form

Each expression, which satisfies the condition of containing only treeless functions, can always be transformed by the Deforestation Algorithm to an equivalent pure treeless expression.

Lemma 3.4 - No Loss in Efficiency

Each expression, which results from the Deforestation Algorithm, will be no less efficient than the original un-transformed expression, where efficiency is measured by the number of function calls that will be invoked.

Lemma 3.5 - Termination of Algorithm *T*

Under the pre-condition of using expressions containing only (restricted) treeless functions, the Deforestation Algorithm (***T*** rules) will *always* terminate.

Proof for Lemma 3.3 (on treeless form)

The first lemma can be proved by structural induction on the seven ***T*** rules of the Deforestation Algorithm. It can be trivially shown that every expression and new function defined in the RHS of each of the ***T*** rules is indeed treeless. Also, the equivalence between the original and the transformed expression automatically follows from the correctness of the unfold/fold rules.

Proof for Lemma 3.4 (on efficiency)

The second lemma can again be proved by structural induction. The only transformation steps used by the rules are the define, unfold and fold steps.

The *unfold* step always results in a more efficient expression by virtue of requiring one *less* function call, provided no large expressions are duplicated. This is the case because all the expressions used are linear. Hence, this step will always produce a gain in efficiency. The *define* step introduces a new intermediate function, which we would regard as a loss of efficiency. However, each define step is always accompanied by an unfold step. Thus, the inefficiency introduced by define is always cancelled out by an unfold.

Lastly, each fold step against a previously defined function can, in general, cause either a gain or a loss in efficiency - depending on the sequence of transformation steps which have been applied between the define step and its correspondingly fold step. If the sequence of transformation steps prior to a given fold step (but after the define step) results in a gain in efficiency, then this fold step will allow the efficiency gain to be *amplified* in the newly introduced recursive call. Correspondingly, if the prior sequence of steps results in a loss of efficiency, then this fold step will *amplify* the loss. As all the other rules are efficiency-preserving, the fold steps of the ***T*** rules will never cause a loss of efficiency. Hence, the efficiency-preserving characteristic of the ***T*** rules.

Proof for Lemma 3.5 (on termination)

The termination proof for the transformation algorithm is the most difficult of the three lemmas. To simplify the termination proof, Ferguson and Wadler [Ferguson88] actually used a slightly more restricted form of treelessness where each term is permitted to have only a single outer constructor. This form, known as **restricted treeless** form, is used in place of pure treeless form and can be defined by the following grammar rules:

$$rt ::= rt' \mid c\ (rt'_1,..,rt'_n)$$
$$rt' ::= v \mid f\ (v_1,..,v_n) \mid g\ (v_0,v_1,..,v_n)$$

where f and g are *restricted treeless* functions and each term is linear

Even though this is a more restrictive form, there is no loss of generality involved because each *pure treeless* expression can always be converted to an equivalent *restricted treeless* expression by introducing suitably defined intermediate functions. With the restricted treeless form, the termination proof in [Ferguson88] was given in two parts. We briefly outline the proof in this section in order to give the reader a flavour of how termination proofs could be given for transformation algorithms.

Firstly, some *measure*, related to the size of expressions occurring during transformation, must be shown to be *bounded* by each of the $\boldsymbol{T}$ rules. The measure used, called *function nesting*, was as follows:

$$\begin{aligned}
\mathbf{N}[\![v]\!] &= 0 \\
\mathbf{N}[\![c(t_1,..,t_n)]\!] &= \max\{\mathbf{N}[\![t_1]\!],..,\mathbf{N}[\![t_n]\!]\} \\
\mathbf{N}[\![f(t_1,..,t_n)]\!] &= 1+\max\{\mathbf{N}[\![t_1]\!],..,\mathbf{N}[\![t_n]\!]\} \\
\mathbf{N}[\![g(t_0,..,t_n)]\!] &= 1+\max\{\mathbf{N}[\![t_0]\!],..,\mathbf{N}[\![t_n]\!]\}
\end{aligned}$$

It was shown to be bounded by proving that:

For each transformation step, $\boldsymbol{T}[\![t]\!] \Leftarrow \ldots \ldots \boldsymbol{T}[\![t_i]\!] \ldots\ldots, \forall i.\ \mathbf{N}[\![t]\!] \geq \mathbf{N}[\![t_i]\!]$

This is possible because all the unfoldings in $\boldsymbol{T}$ involve only restricted treeless function calls. These unfoldings will never increase the function nesting of their expressions because each treeless function's body has a measure of at most one, which at worst, replaces the nesting of the function call just unfolded.

Notice that the function nesting measure does not take into account constructor terms. Showing that this nesting measure is bounded is <u>not</u>, on its own, sufficient for termination proof. What is needed is a proof to show that the *size* of expressions (represented by the depth of expressions tree) encountered by the transformation rules is bounded, with respect to the initial expression. By using the restricted treeless form, Ferguson and Wadler provided this proof by showing that each term encountered by their $\boldsymbol{T}$ rules can only have a *single* constructor. This was proved by showing that each new term encountered at the RHS of $\boldsymbol{T}$ satisfied the grammar form of fct, whenever the previous term (on the LHS of $\boldsymbol{T}$) satisfied fct where fct was a grammar (for expressions with a single constructor each) of the following definition:

$$\begin{aligned}
&\text{fct} && ::= \text{ft} \mid \ldots c(t_1,..,t_n) \ldots \\
&\ldots t \ldots && ::= t \mid g\,(\ldots t \ldots, ft_1,..,ft_n) \\
&\text{ft} && ::= v \mid f\,(ft_1,..,ft_n) \mid g\,(ft_0,..,ft_n) \quad \text{WHERE f and g are restricted treeless functions}
\end{aligned}$$

The initial term is assumed to satisfy the grammar of ft without loss of generality (because each constructor can always be replaced by a corresponding intermediate treeless function).

The presence of an upper bound on the size of expressions which could be encountered by Wadler's $\boldsymbol{T}$ rules (for any given initial expression) is sufficient for termination proof. This is because:

(1) With an upper bound on the size of expressions handled by the ***T*** rules, the number of different expressions (based on an initial *finite* set of function and constructor symbols) which could be encountered during transformation will be corresponding *finite*. This implies that there will only be a finite number of steps involving ***T*** 5,6,7 (of Figure 3.4), as any re-occurrences of expressions (for these steps) are assumed to be terminated by the *implicit* fold steps.

(2) The other recursive rule, ***T*** 2, could never form infinite sequences of rule applications on its own. This is because it operates on successively smaller sub-expressions.

Because of these two reasons, the four recursive rules, ***T*** 2,5,6,7, could never form an infinite sequence of applications for any given initial expression. Hence, the termination of algorithm, ***T***.

Wadler's rules relied on the presence of *implicit* fold steps to handle expressions which re-occur during transformation. This was already noted in Section 3.2.3 to require either a complex retrospective mechanism for introducing the necessary define steps or a less than optimal solution which introduces an intermediate function for each new expression encountered. To improve this situation, we introduced *explicit* define&unfold rules for ***T***6c and ***T***7b in Section 3.2.4, together with the corresponding explicit fold rules, ***T***6b and ***T***7a. We also introduced direct unfold rules, ***T***5 and ***T***6a, which we claimed do not require the fold steps. We shall outline the additional termination proof which allows this optimisation.

With the (slightly) modified rules, the number of explicit define&unfold steps will continue to be *finite*. This is because there is still an upper bound on the size of expressions which could be encountered. However, the direct unfold rules of ***T***5 and ***T***6a could no longer rely on the presence of fold steps to terminate any re-occurrences of their expressions. This is still fine provided we can prove that, during the transformation, there could only be *finite sequences* of steps involving just the (other recursive) rules ***T*** 2, ***T*** 5 and ***T*** 6a.

If we are able to prove this, then the total number of recursive ***T*** steps will *always* be finite for any given expression. This is because each addition of a *finite sequence* of steps for ***T*** 2,5,6a, among the *finite number* of the define&unfold steps (***T*** 6c,T7b) will *never* extend the total number of steps into an infinite one. Hence, the termination of the modified ***T*** rules. The other rules, namely ***T*** 1,3,4,6b,7a, need not be considered because they are the terminating cases.

To show that there could never be an infinite sequence of steps, involving just rules ***T*** 2,5,6a, we must show the presence of some well-founded decreasing measure for these rules. A well-founded decreasing measure is one which decreases to its smallest value in a finite number of steps. Informally, this is possible because:

(1) Rule *T* 5 propagates constructor terms outwards. This means that the position of the constructor within our expression (after each application of this rule) can be used to show a well-founded measure which decreases. This measure would guarantee that there could never be an infinite sequence of just *T* 5 steps.

(2) Rule *T* 6a is used to unfold non-recursive f-type function calls. To show a well-founded decreasing measure, we could assign a *function hierarchy number*, H [[p]], to each function p (for both f-type and g-type functions), such that each of its auxiliary function, say p_aux, has a hierarchy number, H [[p_aux]] that is *less than* H [[p]]. Correspondingly, each mutual recursive function, p_rec of p, should have a hierarchy number, H [[p_rec]], that is *equal to* H [[p]].

With this function hierarchy numbers, we can show that the expression which results from each unfolding of a non-recursive function call will have a decreasing multi-set (bag) measure that is made up of the hierarchy numbers of all its functions. This (well-founded) decreasing measure will show that there could never be an infinite sequence of the just *T* 6a steps.

(3) Lastly, the *T* 2 rule deals with successively smaller sub-expressions. We could therefore use the size of expression as the basis for a (well-founded) decreasing measure for this rule.

Of course, the above three (different) measures are presently applicable to each of the rules individually. What is needed is a *single* well-founded measure which can be shown to be decreasing for all the three rules simultaneously. If this can be proved, then there could only be finite sequences of steps involving just *T* 2, *T* 5 and *T* 6a. There is indeed such a measure, taken by combining the above three separate measures into a single complex measure. This new measure and the proof that it is decreasing for the three rules is given in the Appendix 5. With this well-founded measure, the total number of recursive *T* rules will always be finite (under the conditions of the Deforestation Theorem). Hence, the termination of the modified *T* rules.

This style of termination proof will be repeated for a number of transformation algorithms in Chapter 3 and 4. We shall always begin by showing that the explicit define&unfold steps (with fold rules) will only occur a finite number of times. This is possible as long as there is a bound on the size of expressions used by these define&unfold rules. After that, we shall show that there is a well-founded decreasing measure for the *other* recursive rules. This will help to confirm that the total number of recursive steps will be finite because finite sequences of the other recursive steps (from the well-founded measure) could never increase the overall number of steps (within the finitely many define&unfold steps) by an infinite number.

3.2.6. Blazed Treeless Form

One important extension to the above work is to widen the definition of treeless form so that deforestation can be safely applied to an even a larger class of expressions. Wadler has proposed differentiating terms, depending on their type information, to either:-

1) *recursive* type terms, like lists and trees, which can be gainfully deforested;

   or

2) simple *atomic* type terms, like true-false and num[7], which are better left alone because of negligible gains possible from recovering unit storage.

This differentiation is indicated by *annotating* each term (or expression) with a $^{\oplus}$ if it is of *recursive* type, or with a $^{\circ}$ if it is of *atomic* type. Wadler called this term-based annotation process **blazing** and provided the following definition[8] for blazed treeless terms.

Definition 3.6 - Blazed Treeless Form

A term is considered to be *blazed treeless* if it satisfies the following grammar:

$bt ::= bv \mid c\ (bt_1,..,bt_n)^{\oplus} \mid f\ (bv_1,..,bv_n)^{\oplus} \mid g\ (bv_0,..,bv_n)^{\oplus}$

$bv ::= v^{\oplus} \mid v^{\circ} \mid c\ (bv_1,..,bv_n)^{\circ} \mid f\ (bv_1,..,bv_n)^{\circ} \mid g\ (bv_0,..,bv_n)^{\circ}$

WHERE f and g are blazed treeless functions; and $v^{\oplus}$ must be linear; v° can be non-linear

Correspondingly, a function is considered to be *blazed treeless* if its definition is made up of *blazed treeless* terms.

The grammar of this form is a considerable extension of pure treeless form. Nested calls are now allowed for terms which are annotated as $^{\circ}$. In addition, non-linearity is also allowed for variables which are annotated as $^{\circ}$. As a result, blazed treeless form can now include many more functions. An example of a function which is blazed-treeless but not pure treeless is:

```
- - - up_to([],n)      <= [];
- - - up_to(a::x,n)    <= n::up_to(x,n+1);
```

This function is not pure treeless because of the nested call up_to(x,n+1) and the non-linearity of n.

A similar Deforestation Theorem was given in [Wadler88] for blazed treelessness, together with a modified transformation algorithm. The main modification to this algorithm involves the extraction of all subterms which are annotated as $^{\circ}$ before introducing new function definitions. This generalisation enables fold operations for recursive expressions and also ensures that large expressions are not duplicated by the unfolding of non-linear parameters.

As an example, the expression double(up_to(x,1)) can be deforested by initially extracting out all subterms annotated as $^{\circ}$, as follows.

```
let v==1° in double(up_to(x⊕,v°)⊕)⊕
```

[7] The basic num type is presently assumed to be a directly implemented type. This is why it is considered as an *atomic* type. Alternatively, we could also define the number type recursively in *peano*-style. In this case, it would be considered as a *tree-like* type. Of course, there is nothing to stop us from having both versions of num in our programs (referring to them by different names).

[8] The original blazed treeless definition were formulated for expressions with **case** construct but we have adapted it here for pattern-matching functions.

This is then deforested to:

```
let v==1⁼ in d_to(x⁺,v⁼)⁺
WHERE
- - -d_to([],n)        <= [];
- - -d_to(a::x,n)      <= let v==n+1⁼ in 2*n⁼::d_to(x⁺,v⁼)⁺;
```

Using our consumer-producer model, blazed treelessness can be viewed as a way of marking all atomic-type terms as *bad* producers and *bad* consumers. This is because the fusion of expressions involving them do not result in any significant gains in efficiency. Annotating them differently will allow these terms (or compositions) to be disregarded during deforestation.

3.3. Our Extension to Deforestation

Despite its wider applicability, the type-based annotation of Wadler does not go far enough. This is because there are still many functions outside blazed treeless form which can also be safely deforested. The main feature of these functions is that *some* but *not all* of their *recursive-type* parameters are suitable places for deforestation.

An example is the following iterative list reversal function.

```
dec rev_it : list(alpha) # list(alpha) -> list(alpha);
- - -rev_it([],w)       <= w;
- - -rev_it(a::x,w)     <= rev_it(x,a::w);
```

This function has two linear, list-type parameters. It is not considered (blazed) treeless because it has a nested constructor term, a::w, as the second argument of the recursive call, rev_it(x,a::w). In contrast, the first parameter does not have this problem and appears to satisfy the treeless requirement of not having an intermediate term because there is a *variable*, x, as the first argument of its recursive call. Thus, it should not be surprising to learn that an expression like rev_it(double(x),[]), with an intermediate term in the first parameter of rev_it, can still be safely and successfully deforested.

This consideration suggests that a change to the annotation scheme should be made. Instead of annotating individual *terms*, our new scheme will directly annotate each function's *parameter* with either a $^+$ or a $^=$ to indicate if the parameter is suitable for deforestation or not.

Firstly, as before, atomic-type parameters are not suitable for deforestation. This is because the gains arising from eliminating the atomic-type arguments' unit storage are negligible. Hence, they should continue to be annotated $^=$, as in blazing. But what about recursive-type parameters? There are two kinds of recursive-type parameters, as we have just seen in rev_it. There are some parameters which are *suitable* for the safe elimination of intermediate terms and these should be annotated as $^+$. At the same time, there are also parameters which are *unsuitable* for the elimination of intermediate terms and these should be annotated as $^=$. This additional consideration (on annotating all parameters) gives us a more complete way of distinguishing *good* consumers from *bad* consumers.

As it turns out, there are some specific criteria to help us distinguish between these parameters. We present these parameter criteria next.

3.3.1. Parameter-Based Annotation Scheme

Apart from type information, there are two other important criteria which need to be considered for determining if an argument is suitable for deforestation or not. These are:

- Variable-Only Criterion
- Linearity Criterion

3.3.1.1. Variable-Only Criterion

A close examination of the way deforestation's transformation works, suggests that there is a very useful parameter criterion which could guarantee successful folding in recursive expressions that are being transformed. This criterion, which will be known as the **variable-only** parameter criterion, was implicitly used in pure treelessness where all function calls are obliged to have *only* variables as their arguments. However, instead of requiring all parameters of function calls to be variable-only, we can relax our consideration by permitting parameters to be either variable-only or not.

For example, the earlier rev_it function has two parameters, one of which is variable-only while the other is not. Another example is the function swap below. The first three parameters of this function are variable-only while the fourth parameter is not, as dictated by the recursive call, swap(x,c,b,w<>[a]), of the second equation.

```
dec swap : list(alpha) # beta # beta # list(alpha) -> (beta,list(alpha));
- - -swap([],b,c,w)    <= (b,w);
- - -swap(a::x,b,c,w)  <= swap(x,c,b,w<>[a]);
```

This criterion is primarily relevant to parameters of *recursive* functions where a parameter of a function is considered to be variable-only if the corresponding parameter argument in each *recursive call* of our function definition is made up of only variables. Non-recursive functions need not worry about this criterion because their function calls can always be unfolded without requiring folding. This syntactic criterion can be formally defined as follows:

Definition 3.7 - Variable-Only Parameter Criterion

Given a set of *mutually recursive* functions, $p_1..p_k$, where $k \geq 1$. The jth-parameter, v_j, of the ith-function with definition:

$$p_i(v_1,...,v_j,...,v_n) <= tf_i$$

is considered to be *variable-only* if all its recursive calls, $p_i(t_1,...,t_j,...,t_n)$, in each of the definition of mutual recursive functions, $p_1..p_k$, have the jth-argument, t_j, as a *variable*.

Also, all parameters of *non-recursive* functions are trivially considered to be *variable-only*, since there are no recursive calls.

This simple syntactic criterion is sufficient to ensure successful folding of recursive expressions during deforestation. Two informal reasons (though not the only reasons) for this are:

- If this criterion holds for the pattern-matching parameter, then this parameter is likely to be used as a consuming parameter for deforestation purposes (see first parameter of rev_it).
- If this criterion holds for the non pattern-matching parameters, then it is likely that only some form of permutation in the arguments will take place during transformation and this will help ensure folding (see second and third parameters of function swap).

In contrast, those parameters which are *not* variable-only tend to accumulate results in their arguments. This may cause infinite number of function calls (containing different arguments) to occur during transformation, thus preventing folding from taking place (unless we apply appropriate generalisations). For example, try transforming the expression, rev_it(double(x),[]), without ever generalising the second argument, []. This would simply result in the following infinite number of expressions (containing the rev_it call) for new function definitions (during its transformation).

rev_it(double(x_1),[a])
rev_it(double(x_2),[a,a_1])
:
rev_it(double(x_n),[a,a_1,...,a_{n-1}])
:

3.3.1.2. Linearity Criterion

Apart from the variable-only criterion, there is another very important parameter criterion, called **linearity**, which is needed to ensure no loss of efficiency in the transformed programs. This criterion is important because functions with non-linear parameters run the risk of losing efficiency when expensive sub-expressions are duplicated by unfolding. An example of this is the function, twotimes, with a non-linear parameter, x, as shown below.

```
- - -twotimes(x)      <= x+x;
```

Calls of such functions must be considered carefully. If a large expression appears in the non-linear argument, e.g. twotimes(y*y), then the call should *not* be unfolded, as its unfolded expression, y*y+y*y, will be less efficient because of the duplicated arguments. However, if a *variable* or a *constant* is present in the non-linear parameter position, then the call can still be unfolded without loss of efficiency, e.g.

twotimes(a)	can be efficiently unfolded to	a+a,	and
twotimes(4)	can be efficiently unfolded to	4+4	

But how should the linearity criterion be determined? The simplest method of identifying non-linear parameters is to check if multiple occurrences of parameter variables are present in each equation. Those parameters which occur more than once are simply considered to be non-linear. This method can be performed quite efficiently and is used by Wadler.

However, it is not accurate enough. This is because some parameter variables, though occurring more than once in their function's definition, are only evaluated once (even in the

call-by-textual substitution parameter mechanism). A simple example of this is the parameter, c, of function p below.

```
---p (b,c)          <= if (b,2*c,c);
---if (true,x,y)    <= x;
---if (false,x,y)   <= y;
```

This parameter, c, occurred twice in the body of p. However, each occurrence of c appears in disjoint paths of the conditional function, if. This means that c is *unshared* and will only ever be evaluated once. As a result, function calls like p(b,twotimes(c)), can still be unfolded without any loss of efficiency, assuming that we have a lazy rather than strict evaluation mechanism. Hence, a more accurate linearity criterion is one which augments syntactic counting with a modified form of **sharing analysis** [Hudak85] for multiply occurring parameters. This criterion can be defined as follows:

Definition 3.8 - Linearity Parameter Criterion

A parameter of a function definition is said to be linear if

- it occurs at most once in each equation's RHS, or
- it occurs more than once but is *unshared* (according to sharing analysis).

Otherwise it is considered to be non-linear.

The linearity criterion is an important consideration for unfolding. In [Sestoft88], this consideration is also known as *duplication risk analysis.*

3.3.1.3. Annotation Scheme

The above two parameter criteria, together with the type criterion of blazed treelessness, can be used in the following parameter annotation scheme for determining the suitability of a parameter term for deforestation.

Definition 3.9 - A Parameter Annotation Scheme for Deforestation

A parameter of a function definition will be annotated as $^{\diamond}$, and therefore marked as a suitable candidate for deforestation, if and only if, it satisfies all the three criteria below:

- is a *recursive* type
- is *linear*
- is *variable-only*

If it does not satisfy all three criteria, then the parameter will be annotated as $^{\circ}$.

As an example, the earlier function rev_it will have the following annotations.

```
---rev_it([]◇,w°)      <= w;
---rev_it(a::x◇,w°)    <= rev_it(x,a::w) ;
```

Both parameters of this function are *recursive-type* and *linear* but only the first parameter is *variable-only*, hence the annotations are as shown.

3.3.2. Extended Treeless Form

With the new annotation scheme, the parameters of each function can be logically separated into two groups of either $^{\oplus}$ or $^{\ominus}$ annotation. Henceforth, we shall represent this logical separation by using the following three forms of function calls for both the f- and g-type functions:-

$$f(t_1^{\oplus}..t_m^{\oplus},t_{m+1}^{\ominus}..t_n^{\ominus})$$
$$g(t_0^{\oplus},t_1^{\oplus}..t_m^{\oplus},t_{m+1}^{\ominus}..t_n^{\ominus})$$
$$g(t_0^{\ominus},t_1^{\oplus}..t_m^{\oplus},t_{m+1}^{\ominus}..t_n^{\ominus})$$

For the purposes of exposition, we shall require all $^{\oplus}$ parameters to occur before $^{\ominus}$ parameters, apart from the pattern-matching parameters of g-type calls which may either be $^{\oplus}$ or $^{\ominus}$. It should be noted that this is merely for convenience in describing the transformation (later). It does not represent a programming restriction as we can cater for it by either automatically permuting the parameters of the functions during transformation or simply modify our transformation rules to accept arbitrary orderings. With this assumption, we are able to define a new kind of treelessness, called the **extended treeless** form, as follows.

Definition 3.10 - Extended Treeless Form

A term is said to be *extended treeless* if it satisfies the grammar below.

$$et ::= v \mid c(et_1,..,et_n) \mid f(v_1^{\oplus}..v_m^{\oplus},et_{m+1}^{\ominus}..et_n^{\ominus}) \mid g(v_0^{\oplus},v_1^{\oplus}..v_m^{\oplus},et_{m+1}^{\ominus}..et_n^{\ominus}) \mid g(et_0^{\ominus},v_1^{\oplus}..v_m^{\oplus},et_{m+1}^{\ominus}..et_n^{\ominus})$$

Note that all arguments in parameter positions annotated as $^{\oplus}$ must be variables, as in $v_1..v_m$; while those arguments in parameter positions annotated $^{\ominus}$ can be terms, as in $et_{m+1}..et_n$. Also, g and f are *extended treeless* functions.

Similarly, a function is said to be *extended treeless* if and only if its definition is made up of *extended treeless* terms.

This extended treeless, or e-treeless, form is a generalisation of blazed treelessness because:

1) recursive-type arguments, annotated as $^{\ominus}$, are allowed to contain nested terms, and
2) linearity is no longer a pre-requisite on any variable.

It is a significant generalisation of treelessness. As an example, consider the following set of functions for finding an infinite list of prime numbers by the sieving technique. With blazed treelessness, only functions, if, fl', filter and inf_int are considered to be (blazed) treeless but not functions, prime and sieve. However, with the extended form of treelessness, *all* the six functions are (extended) treeless.

```
dec prime : list(num);
dec sieve,inf_int : list(num) -> list(num);
dec filter : list(num) # num -> list(num);
- - -prime()          <= sieve(inf_int(1)) ;
- - -sieve ( []ᵉ)    <= [] ;
- - -sieve ( a::xᵉ)  <= a::sieve(filter(x,a)) ;
- - -inf_int ( nᵉ)   <= n::inf_int(n+1) ;
- - -filter ( []ᵉ, aᵉ)   <= [] ;
- - -filter (b::xᵉ,aᵉ)   <= fl'(x,b,a) ;
- - -fl' ( xᵉ, bᵉ, aᵉ )  <= if (b mod a=0,filter(x,a),b::filter(x,a)) ;
- - -if (trueᵉ,xᵉ,yᵉ)    <= x ;
- - -if (falseᵉ,xᵉ,yᵉ)   <= y ;
```

Of the above functions, sieve appears to be the most interesting. This function is not considered to be blazed-treeless because its definition contains a recursive call, sieve(filter(x,a)), with an intermediate list-type term. However, by annotating sieve's parameter to ᵉ, we have managed to include this function under the new extended treeless form. As a result, we could now deforest expressions which use these functions. We shall see an example of this later.

Also, unlike the restricted treeless form, we allow arbitrary nesting of constructor terms to be used. This helps to avoid unnecessary intermediate functions which will otherwise cause the programs to be larger and less efficient. For example, consider the following e-treeless function:

```
dec thrice_ones:list(alpha) -> list(num)
- - -thrice_ones([])    <= [] ;
- - -thrice_ones(a::x)  <= 1::1::1::thrice_ones(x) ;
```

The previous restricted form of treelessness would have required this function to be translated into the following alternative program, with two intermediate functions to break the nesting of constructors.

```
- - -thrice_ones([])    <= [] ;
- - -thrice_ones(a::x)  <= 1::thrice_ones'(x) ;
- - -thrice_ones'(x)    <= 1::thrice_ones''(x) ;
- - -thrice_ones''(x)   <= 1::thrice_ones(x) ;
```

This restriction was imposed to allow the proof of termination in [Ferguson88]. We shall show that this is unnecessary by providing a different termination proof in Section 3.3.6 .

3.3.3. Extended Deforestation Theorem

A theorem for deforesting expressions, which use e-treeless functions, can be stated as follows:

Theorem 3.11 - Extended Deforestation

> Every term, even non-linear ones, containing constructors and occurrences of functions which are *extended treeless*, can be effectively transformed, by the Extended Deforestation Algorithm, to an equivalent *extended treeless* term, without any loss of efficiency.

The Extended Deforestation Algorithm referred to in the above theorem will safely and successfully eliminate all intermediate data terms that appear in parameter positions annotated as $^{\oplus}$. Each such elimination will normally produce a gain in efficiency. We shall present the transformation algorithm in Section 3.3.4 and the proof for the above theorem in Section 3.3.6, but first, let us look at a simple expression, rev_it(double(x),[]). This expression cannot be deforested using Wadler's transformation algorithm because the function rev_it is not considered to be pure or blazed treeless. However, it is an e-treeless function. We can therefore use our Extended Deforestation Algorithm, given in Figure 3.7, to transform this expression into an e-treeless equivalent. Such a transformation is illustrated in Figure 3.6, where a new function, rev_db, is defined. This transformation resulted in the expression, rev_db(x,[]), together with the following e-treeless function:

```
- - -rev_db([],w)      <= w ;
- - -rev_db(a::x,w)    <= rev_db(x,2*a::w) ;
```

Notice that the rules had (correctly) generalised the second argument of rev_it, in the expression rev_it(double(x),[]), before the function rev_db was defined. The reason for this is that the second parameter of rev_it does *not* have the variable-only criterion.

$\mathcal{T}$ [[rev_it(double(x)$^{\oplus}$,[]$^{=}$)]] *can be transformed to:*

	$\Rightarrow$	$\mathcal{T}$ [[rev_it(double(x),[])]]	! $\mathcal{T}$ 9b, define rev_db
	$\Rightarrow$	rev_db(x, $\mathcal{T}$ [[[]]])	! $\mathcal{T}$ 3
	$\Rightarrow$	rev_db(x, [])	

define

- - -rev_db(x,w)	<=	rev_it(double(x),w)	! narrow double(x)
- - -rev_db([],w)	<=	$\mathcal{T}$ [[rev_it([],w)]]	! $\mathcal{T}$ 6a, unfold rev_it
	<=	$\mathcal{T}$ [[w]]	! $\mathcal{T}$ 1
	<=	w	
- - -rev_db(a::x,w)	<=	$\mathcal{T}$ [[rev_it(2*a::double(x),w)]]	! $\mathcal{T}$ 6a, unfold rev_it
	<=	$\mathcal{T}$ [[rev_it(double(x),2*a::w)]]	! $\mathcal{T}$ 9a, Fold with rev_db
	<=	rev_db(x,$\mathcal{T}$ [[2*a::w]])	! $\mathcal{T}$ 2,
	<=	rev_db(x,$\mathcal{T}$ [[2*a]]::$\mathcal{T}$ [[w]])	! $\mathcal{T}$ 1,3,1,2
	<=	rev_db(x,2*a::w)	

Figure 3.6 - A Simple Example of Extended Deforestation

3.3.4. Transformation Algorithm

The Extended Deforestation Algorithm is made up of the nine transformation rules in Figure 3.7. To shorten these rules, we use the abbreviation, $t_{1..n}$, to represent a list of arguments, $t_1,...,t_n$; and $\mathcal{T}$ [[t]]$_{1..n}$ to represent $\mathcal{T}$ [[t_1]],...,$\mathcal{T}$ [[t_n]]. Also, each expression used for this transformation can be any combination of constructors and extended-treeless functions, as shown below:

$$eft ::= v \mid c(eft_1,...,eft_n) \mid f(eft_{1..m}^{+},eft_{m+1..n}^{-}) \mid g(eft_0^{+},eft_{1..m}^{+},eft_{m+1..n}^{-}) \mid g(eft_0^{-},eft_{1..m}^{+},eft_{m+1..n}^{-})$$

The first five rules (Rules $\mathcal{T}$ 1 to $\mathcal{T}$ 5) deal with simple situations where the outermost expressions (function calls or constructors) are already in extended treeless form. These outermost expressions are left alone, with the transformation rules applied only to the remaining subexpressions.

The main elimination rules (for intermediate terms) are $\mathcal{T}$ 6 to $\mathcal{T}$ 9. Each of these rules deals with a function call, say last_call, which lies in the leftmost innermost position of pattern-matching parameter positions, annotated as $^{+}$, of zero or more outer g-type calls. This leftmost innermost call could only be one of the following forms:

$$f(t_1..t_m^{+},t_{m+1}..t_n^{-})$$
$$g(v_0^{+},t_1..t_m^{+},t_{m+1}..t_n^{-})$$
$$g(c\ t'_1..t'_j^{+},t_1..t_m^{+},t_{m+1}..t_n^{-})$$
$$g(t_0^{-},t_1..t_m^{+},t_{m+1}..t_n^{-})$$

If it is not one of the above calls, then it must be of the form $g(t_0^{+},t_1..t_m^{+},t_{m+1}..t_n^{-})$, where there is another inner function call in the pattern-matching argument, t_0, that is annotated as $^{+}$.

Such an innermost function call could be concisely represented by the expression, ...last_call..., which uses a context notation (represented by outer ellipses) that is defined by the following grammar.

$$...t... ::= t \mid g(...t...^{+},t_{1..m}^{+},t_{m+1..n}^{-})$$

This expression is actually a shortened notation for

$$g_1(g_2(...g_N(last_call^{+},t_1..t_m^{+},t_{m+1}..t_n^{-})...^{+},t'_1..t'_{m'}^{+},t'_{m'+1}..t'_{n'}^{-})^{+},t''_1..t''_{m''}^{+},t''_{m''+1}..t''_{n''}^{-})$$

where $N \geq 0$. With the help of this notation, the (leftmost) innermost function call for the expression, append(sieve(inf_int(2^{-})$^{-}$)$^{+}$,y^{+}), is sieve(inf_int(2^{-})$^{-}$) and *not* inf_int(2^{-}) because the former lies in a $^{+}$ pattern-matching parameter position but not the latter.

In each of these rules, we can either perform a *define & unfold*, *unfold* or *fold* step. In the case of the *define & unfold* steps ($\mathcal{T}$ 6b,7c,8b,9b), a *generalisation* and *renaming* process using the auxiliary meta-function, **gen_ren**, is always performed on the expression first. This is done before a new function definition is introduced, whose innermost call is unfolded, before further transformation.

The generalisation and renaming meta-function, **gen_ren**, simultaneously *extracts* out all subterms that are annotated[9] as $^{-}$ and *renames* all the variables that are annotated as $^{+}$. This ensures that the resulting expression is linear and and has only variables for arguments annotated as $^{-}$. Thus, given a term, eft, this meta-function would return a tuple of five items, $(eft^{\sim},te_{1..s},ve_{1..s},vo_{1..k},vn_{1..k})$, where $eft^{\sim}$ is the generalised and linearised expression, $te_{1..s}$ are

[9] by virtue of its occurrence in the annotated parameters.

the extracted subterms annotated as $^{\square}$ and $ve_{1..s}$ are the corresponding substituted variables. Also, $vo_{1..k}$ are the old variables with $^{\oplus}$ annotations and $vn_{1..k}$ are the new renamed variables. As an example, **gen_ren** $[\![rev_it(double(x^{\oplus})^{\oplus},[]^{\square})]\!]$ would return:

$$(rev_it(double(a^{\oplus})^{\oplus},b^{\square}),\ [[]],\ [b],\ [x],\ [a])$$

where the generalised and renamed expression is $rev_it(double(a^{\oplus})^{\oplus},b^{\square})$, extracted sub-term in [], replacement variable is b, old name of variable annotated as $^{\oplus}$ is x and new variable name is a.

We represent by $eft^{\sim}$ the generalisation and renaming of a term eft. Similarly, the generalisation and renaming of context expression ...eft... will be represented by the notation ...$eft^{\sim}$...$^{\sim}$. These generalised expressions will conform to the following grammar:

$$\begin{aligned} eft^{\sim} &::= v \mid c(eft^{\sim}_1,..,eft^{\sim}_n) \mid f(eft^{\sim}_{1..m}{}^{\oplus},v_{m+1..n}{}^{\square}) \mid g(eft^{\sim}_0{}^{\oplus},eft^{\sim}_{1..m}{}^{\oplus},v_{m+1..n}{}^{\square}) \\ &\quad \mid g(v_0{}^{\square},eft^{\sim}_{1..m}{}^{\oplus},v_{m+1..n}{}^{\square}) \\ ...t...^{\sim} &::= t \mid g(...t...^{\sim}{}^{\oplus},eft^{\sim}_{1..m}{}^{\oplus},v_{m+1..n}{}^{\square}) \end{aligned}$$

The generalisation operation is important because it can help to ensure the termination of the transformation algorithm through successful folding. We shall formally see this requirement later, in the termination proof of Section 3.3.6.

The fold steps ($\mathcal{T}$ 7b,8a,9a) must work in tandem with the corresponding define steps for recursive expressions. Each of these fold steps checks previously defined function definitions in def_set[10] to see if there is an earlier function definition whose RHS matches exactly with the generalised and renamed expression. Whenever such a definition exists, we apply the fold step in preference to the define step.

Also, we have a *direct* unfold step in rule $\mathcal{T}$ 6a and $\mathcal{T}$ 7a. This is applied in rule $\mathcal{T}$ 6a to deal with the consumption of nested constructor terms within outer g-type function calls. We apply direct unfolding whenever the arguments annotated as $^{\square}$ in the innermost matching g-call are all *variables*. This can be done, without worrying about folding, because there can never be an infinite sequence of steps requiring just the rule $\mathcal{T}$ 6a. The only time the define step of $\mathcal{T}$ 6b is used is when one or more of the arguments annotated as $^{\square}$ are non-variables.

As for Rule $\mathcal{T}$ 7a, there are two pre-conditions which must be satisfied before we are allowed to perform direct unfolding. These pre-conditions are:

- the expression is non-recursive, otherwise a define step is needed for subsequent folds, and
- the $^{\square}$ arguments of the innermost function call (an f-type call) are variables, otherwise loss of efficiency may result from duplicating its non-linear arguments.

[10] To simplify the $\mathcal{T}$ rules, def_set is not passed as a rule parameter. Instead, it is presently assumed to be a global data structure. See Section 2.4.3.3 for the necessary changes to enable this definition set to be passed around as a rule parameter.

If both these pre-conditions are not satisfied, then we simply perform an *indirect* unfold through the define step of $\mathcal{T}$7c.

(1) $\mathcal{T}[\![v]\!] \Leftarrow v$

(2) $\mathcal{T}[\![c(t_1,...,t_n)]\!] \Leftarrow c(\mathcal{T}[\![t_1]\!],...,\mathcal{T}[\![t_n]\!])$

(3) $\mathcal{T}[\![f(v_{1..m}^{+},t_{m+1..n}^{-})]\!] \Leftarrow f(v_{1..m},\mathcal{T}[\![t]\!]_{m+1..n})$

(4) $\mathcal{T}[\![g(v_0^{+},v_{1..m}^{+},t_{m+1..n}^{-})]\!] \Leftarrow g(v_0,v_{1..m},\mathcal{T}[\![t]\!]_{m+1..n})$

(5) $\mathcal{T}[\![g(t_0^{-},v_{1..m}^{+},t_{m+1..n}^{-})]\!] \Leftarrow g(\mathcal{T}[\![t_0]\!],v_{1..m},\mathcal{T}[\![t]\!]_{m+1..n})$

(6) $\mathcal{T}[\![...g(c_i(t'^{+}_{1..j}),t_{1..m}^{+},t_{m+1..n}^{-})...]\!]$

a) if $t_{m+1..n}$ are all variables (*unfold*)

$\Leftarrow \mathcal{T}[\![...tg_i[t'/v'_{1..j},t/v_{1..n}]...]\!]$

b) otherwise (*define & unfold*) $\Leftarrow$ f_new($vo_{1..k},\mathcal{T}[\![te]\!]_{1..s}$)

where$(...g(c_i(t'^{\sim}_{1..j}),t^{\sim}_{1..m},ve_{m+1..n})...^{\sim},te_{1..s},ve_{1..s},$
$vo_{1..k},vn_{1..k}) ==$ **gen_ren**$[\![...g(c_i(t'_{1..j}),t_{1..m},t_{m+1..n})...]\!]$

define

f_new($vn_{1..k},ve_{1..s}$)=$...g(c_i(t'^{\sim}_{1..j}),t^{\sim}_{1..m},ve_{m+1..n})...^{\sim}$

unfold on call f ***to give:***

f_new($vn_{1..k},ve_{1..s}$)= $\mathcal{T}[\![...tg_i[t'^{\sim}/v'_{1..j},t^{\sim}/v_{1..n},ve/v_{m+1..n}]...^{\sim}]\!]$

(7) $\mathcal{T}[\![...f(t_{1..m}^{+},t_{m+1..n}^{-})...]\!]$

a) if f is non-recursive and $t_{m+1..n}$ are all variables (*unfold*)

$\Leftarrow \mathcal{T}[\![...tf[t/v_{1..n}]...]\!]$

b) if f_old($vn_{1..k},ve_{1..s}$)=$...f(t^{\sim}_{1..m},ve_{m+1..n})...^{\sim} \in$ def_set; **then** (*fold*)

$\Leftarrow$ f_old($vo_{1..k},\mathcal{T}[\![te]\!]_{1..s}$)

where $(...f(t^{\sim}_{1..m},ve_{m+1..n})...^{\sim},te_{1..s},ve_{1..s},$
$vo_{1..k},vn_{1..k}) ==$ **gen_ren**$[\![...f(t_{1..m},t_{m+1..n})...]\!]$

c) otherwise (*define & unfold*) $\Leftarrow$ f_new($vo_{1..k},\mathcal{T}[\![te]\!]_{1..s}$)

where$(...f(t^{\sim}_{1..m},ve_{m+1..n})...^{\sim},te_{1..s},ve_{1..s},$
$vo_{1..k},vn_{1..k}) ==$ **gen_ren**$[\![...f(t_{1..m},t_{m+1..n})...]\!]$

define and add to def_set

f_new($vn_{1..k},ve_{1..s}$)=$...f(t^{\sim}_{1..m},ve_{m+1..n})...^{\sim}$

unfold on call f ***to give:***

f_new($vn_{1..k},ve_{1..s}$)= $\mathcal{T}[\![...tf[t^{\sim}/v_{1..m},ve/v_{m+1..n}]...^{\sim}]\!]$

(8) $\mathcal{T}[\![...g(v_0^{+},t_{1..m}^{+},t_{m+1..n}^{-})...]\!]$

a) if g_old($vn_0,vn_{1..k},ve_{1..s}$)=$...g(vn_0,t^{\sim}_{1..m},ve_{m+1..n})...^{\sim} \in$ def_set; **then** (*fold*)

$\Leftarrow$ g_old($v_0,vo_{1..k},\mathcal{T}[\![te]\!]_{1..s}$)

where $(...g(vn_0,t^{\sim}_{1..m},ve_{m+1..n})...^{\sim},te_{1..s},ve_{1..s},$
$v_0vo_{1..k},vn_0vn_{1..k}) ==$ **gen_ren**$[\![...g(v_0,t_{1..m},t_{m+1..n})...]\!]$

b) otherwise (*define & unfold*) $\Leftarrow$ g_new($v_0,vo_{1..k},\mathcal{T}[\![te]\!]_{1..s}$)

where$(...g(vn_0,t^{\sim}_{1..m},ve_{m+1..n})...^{\sim},te_{1..s},ve_{1..s},$
$v_0vo_{1..k},vn_0vn_{1..k}) ==$ **gen_ren**$[\![...g(v_0,t_{1..m},t_{m+1..n})...]\!]$

define and add to def_set

g_new($vn_0,vn_{1..k},ve_{1..s}$)=$...g(vn_0,t^{\sim}_{1..m},ve_{m+1..n})...^{\sim}$

narrow on call g ***to give:***

g_new($p_1,vn_{1..k},ve_{1..s}$)= $\mathcal{T}[\![...tg_1[t^{\sim}/v_{1..m},ve/v_{m+1..n}]...^{\sim}]\!]$

:

g_new($p_r,vn_{1..k},ve_{1..s}$)= $\mathcal{T}[\![...tg_r[t^{\sim}/v_{1..m},ve/v_{m+1..n}]...^{\sim}]\!]$

(9) $\mathcal{T}$ [[...g(t_0°,$t_{1..m}^{+}$,$t_{m+1..n}^{\circ}$)...]]

a) if g_old(ve_0,$vn_{1..k}$,$ve_{1..s}$)=...g(vn_0,$t\tilde{}_{1..m}$,$ve_{m+1..n}$)...$\tilde{}$ ∈ def_set; **then** (*fold*)

⇐ g_old($\mathcal{T}$ [[t_0]],$vo_{1..k}$,$\mathcal{T}$ [[te]]$_{1..s}$)

where (...g(ve_0,$t\tilde{}_{1..m}$,$ve_{m+1..n}$)...$\tilde{}$,$t_0te_{1..s}$,$ve_0ve_{1..s}$,
$vo_{1..k}$,$vn_{1..k}$) == **gen_ren**[[...g(t_0,$t_{1..m}$,$t_{m+1..n}$)...]]

b) **otherwise** (*define & unfold*) ⇐ g_new($\mathcal{T}$ [[t_0]],$vo_{1..k}$,$\mathcal{T}$ [[te]]$_{1..s}$)

where(...g(ve_0,$t\tilde{}_{1..m}$,$ve_{m+1..n}$)...$\tilde{}$,$t_0te_{1..s}$,$ve_0ve_{1..s}$,
$vo_{1..k}$,$vn_{1..k}$) == **gen_ren**[[...g(t_0,$t_{1..m}$,$t_{m+1..n}$)...]]

define and add to def_set

g_new(ve_0,$vn_{1..k}$,$ve_{1..s}$)=...g(ve_0,$t\tilde{}_{1..m}$,$ve_{m+1..n}$)...$\tilde{}$

narrow on call g ***to give:***

g_new(p_1,$vn_{1..k}$,$ve_{1..s}$)= $\mathcal{T}$ [[...tg_1[$t\tilde{}$/$v_{1..m}$,ve/$v_{m+1..n}$]...$\tilde{}$]]

:

g_new(p_r,$vn_{1..k}$,$ve_{1..s}$)= $\mathcal{T}$ [[...tg_r[$t\tilde{}$/$v_{1..m}$,ve/$v_{m+1..n}$]...$\tilde{}$]]

WHERE

gen_ren[[v]] ⇐ (v$\tilde{}$,[],[],[v],[v$\tilde{}$]) ! where v$\tilde{}$ is a new variable

gen_ren[[c(t_1,...,t_n)]] ⇐ (c($t\tilde{}_1$,...,$t\tilde{}_n$),te_lt,ve_lt,vo_lt,vn_lt)

where ($t\tilde{}_1$..$t\tilde{}_n$,te_lt,ve_lt,vo_lt,vn_lt)==**gen_list**[[t_1..t_n]]

gen_ren[[f($t_{1..m}^{+}$,$t_{m+1..n}^{\circ}$)]]

⇐ (f($t\tilde{}_{1..m}$, $ve_{m+1..n}$),te_lt<>t_{m+1}..t_n,ve_lt<>$ve_{m+1..n}$,vo_lt,vn_lt)

where ($t\tilde{}_{1..m}$,te_lt,ve_lt,vo_lt,vn_lt)==**gen_list**[[$t_0t_{1..m}$]]

gen_ren[[g(t_0^{+},$t_{1..m}^{+}$,$t_{m+1..n}^{\circ}$)]]

⇐

(g($t\tilde{}_0$,$t\tilde{}_{1..m}$,$ve_{m+1..n}$),te_lt<>t_{m+1}..t_n,ve_lt<>$ve_{m+1..n}$,vo_lt,vn_lt)

where ($t\tilde{}_0$,$t\tilde{}_{1..m}$,te_lt,ve_lt,vo_lt,vn_lt)==**gen_list**[[$t_0t_{1..m}$]]

gen_ren[[g(t_0°,$t_{1..m}^{+}$,$t_{m+1..n}^{\circ}$)]]

⇐ (g(ve_0,$t\tilde{}_{1..m}$,$ve_{m+1..n}$),te_lt<>t_0t_{m+1}..$t_n$ve_lt<>$ve_0ve_{m+1..n}$,
vo_lt,vn_lt)

where ($t\tilde{}_{1..m}$,te_lt,ve_lt,vo_lt,vn_lt)==**gen_list**[[$t_{1..m}$]]

gen_list[[t_1..t_n]] ⇐ ($t\tilde{}_1$..$t\tilde{}_n$,te_lt_1..te_lt_n,ve_lt_1..ve_lt_n,vo_lt_1..vo_lt_n,vn_lt_1..vn_lt_n)

where ∀ i ∈ 1..n ($t\tilde{}_i$,te_lt_i,ve_lt_i,vo_lt_ivn_lt_i,)==**gen_ren**[[t_i]]

Figure 3.7 - $\mathcal{T}$ Rules for Extended Deforestation Algorithm

3.3.5. An Example

We illustrate the increased capability of extended deforestation with a slightly more complex expression, second(prime()). This expression finds a list of alternate prime numbers. It uses the e-treeless function, prime (from Section 3.3.2), and a treeless function, second, as shown below.

```
dec second,second' : list(alpha) -> list(alpha);
- - -second([])      <= []
- - -second(a::x)    <= second'(x)
- - -second'([])     <= []
- - -second'(a::x)   <= a::second(x)
```

In its original form, the expression presently constructs an intermediate list. This intermediate list can be eliminated by Extended Deforestation as shown by the transformation sequence in Figure 3.8, where two new function definitions, new_sieve and new_sieve', are introduced. The new definitions are correctly generalised using rule $\mathcal{T}$9b, by extracting out all sub-terms

(arguments) annotated as $^{\circ}$. This generalisation enables the functions introduced to be successfully folded during transformation.

```
T [[second(prime()°)]]
                    ⇒   T [[second(prime()) ]]                  ! T 7a, unfold prime()
                    ⇒   T [[second(sieve(inf_int(1))) ]]        ! T 9b, define new_sieve
                    ⇒   new_sieve(T [[ inf_int(1) ]])           ! T 3
                    ⇒   new_sieve(inf_int(T [[ 1 ]]))           ! T 2
                    ⇒   new_sieve(inf_int(1))
define
---new_sieve( x° )  <=  second(sieve(x° ))                       ! narrow sieve(x)
---new_sieve( []° )  <=  T [[second([]) ]]                       ! T 6a, unfold second
                    <=  T [[ [] ]]                              ! T 2
                    <=  []
---new_sieve( a::x°) <=  T [[ second(a::sieve(filter(x,a))) ]]   ! T 6a, unfold second
                    <=  T [[ second'(sieve(filter(x,a))) ]]     ! T 9b, define new_sieve'
                    <=  new_sieve'(T [[ filter(x,a) ]])         ! T 3
                    <=  new_sieve'(filter(x,a))
define
---new_sieve'( x°)   <=  second'(sieve(x°))                      ! narrow sieve(x)
---new_sieve'( []°)  <=  T [[second'([]) ]]                      ! T 6a, unfold second'
                    <=  T[ [] ]]                                ! T 2
                    <=  []
---new_sieve'(a::x°) <=  T [[ second'(a::sieve(filter(x,a))) ]]  ! T 6a, unfold second'
                    <=  T [[ a::second(sieve(filter(x,a))) ]]   ! T 2
                    <=  T [[ a ]]::T [[ second(sieve(filter(x,a))) ]]  ! T 1
                    <=  a::T [[ second(sieve(filter(x,a))) ]]   ! T 9a, Fold with new_sieve
                    <=  a::new_sieve(T [[ filter(x,a) ]])       ! T 3
                    <=  a::new_sieve(filter(x,a))
```

Figure 3.8 - Another Example of Extended Deforestation

3.3.6. Proof for Extended Deforestation Theorem

The proof for the Extended Deforestation Theorem will essentially follow that of the basic Deforestation Theorem in Section 3.2.6. Again, three lemmas have to be proved in order to support the validity of our theorem. These lemmas are:

Lemma 3.12 - Adherence to Extended Treeless Form

Each expression, which satisfies the theorem's condition of containing only constructors and e-treeless functions, can always be transformed by Extended Deforestation Algorithm to an equivalent e-treeless expression.

Lemma 3.13 - No Loss in Efficiency

Each expression transformed by the Extended Deforestation Algorithm will not result in a loss of efficiency, where efficiency is measured by the number of function calls invoked.

Lemma 3.14 - Termination of Algorithm T

Under the condition of the theorem, the Extended Deforestation Algorithm, using T rules, will *always* terminate.

Proof for Lemma 3.12 (on e-treeless form)

Lemma 3.12 can be trivially shown to be correct by using structural induction on every expression and newly defined function that results from the transformation rules. Also, the equivalence between original and transformed expressions holds by the correctness of the unfold/fold rules.

Proof for Lemma 3.13 (on efficiency)

No loss of efficiency results from the transformation because every define step in the rules is always accompanied by an unfold step. This ensures that there will be *no* net increases in the number of intermediate function calls being introduced. Furthermore, all non-linear arguments annotated as $^-$ will always be variables before unfolding. Hence, these rules will never result in a transformed expression which is less efficient than the original expression.

Proof for Lemma 3.14 (on termination)

The termination proof for the Extended Deforestation Algorithm must again show that the size of expressions occurring during transformation is bounded (w.r.t. the original expression). In fact, we shall show that the RHS of each new function definition introduced (rather than each expression encountered) will have a size that is bounded w.r.t the given initial expression. If this can be established, then the number of different new functions which could be introduced (based on a finite set of function and constructor symbols) will also be finite. This means that each transformation of a given expression will have a finite number of define&unfold steps, with any re-occurrences handled by the corresponding fold steps. The other recursive rules can be separately shown to have some decreasing measure (or ordering) to confirm that there will only be finite sequences of their steps. The net effect of these separate proofs is to show that the total number of the recursive $\mathcal{T}$ steps will be finite (and therefore terminating) for any given application of the algorithm, under the conditions of the Extended Deforestation Theorem.

Thus, for the termination proof, we shall initially show that there is a measure (not the size of expression) that is bounded by all expressions occurring during transformation. After that, we shall show that the size of expressions used for new function definitions is, in turn, bounded by this measure. This would confirm that there is a finite number of define&unfold step for any given expression. Lastly, we shall outline the presence of a well-founded decreasing measure for the rest of the recursive $\mathcal{T}$ rules.

The measure, we shall show will be bounded, is the function nesting on arguments which are annotated as $^+$. This nesting measure, $\mathcal{N}$, can be defined as follows:

$$\mathcal{N}[\![v]\!] = 0$$

$$\mathcal{N}[\![c(t_1,..,t_n)]\!] = \max\{\mathcal{N}[\![t_1]\!],..,\mathcal{N}[\![t_n]\!]\}$$

$$\mathcal{N}[\![f\ (t_{1..m}^{+},t_{m+1..n}^{-})]\!] = \max\{1+\max\{\mathcal{N}[\![t]\!]_{1..m}\},\mathcal{N}[\![t]\!]_{m+1..n}\}$$

$$\mathcal{N}[\![g\ (t_0^{+},t_{1..m}^{+},t_{m+1..n}^{-})]\!] = \max\{1+\max\{\mathcal{N}[\![t_0]\!],\mathcal{N}[\![t]\!]_{1..m}\},\mathcal{N}[\![t]\!]_{m+1..n}\}$$

$$\mathcal{N}[\![g\ (t_0^{-},t_{1..m}^{+},t_{m+1..n}^{-})]\!] = \max\{1+\max\{\mathcal{N}[\![t]\!]_{1..m}\},\mathcal{N}[\![t_0]\!],\mathcal{N}[\![t]\!]_{m+1..n}\}$$

where $\mathcal{N}[\![t]\!]_{1..n}$ is a short-hand notation for $\mathcal{N}[\![t_1]\!],..,\mathcal{N}[\![t_n]\!]$ and max{}=0.

Notice that this measure is *not* directly related to the size of expressions because it ignores the constructors and sub-terms in parameter positions annotated as $^{\circ}$. Notice further that each e-treeless expression will have an $\mathcal{N}$ measure which is ≤ 1.

The lemma, stating that this nesting measure is bounded by each transformation step of the $\mathcal{T}$ rules, is given below.

Lemma 3.14.1 - Nesting Measure, $\mathcal{N}$, is bounded by the $\mathcal{T}$ Rules

The nesting measure, $\mathcal{N}$, on expressions being transformed is bounded by all nine Extended Deforestation $\mathcal{T}$ rules.

For each transformation step, $\mathcal{T}[\![t]\!] \Leftarrow .....\mathcal{T}[\![t_i]\!].....,\forall i.\ \mathcal{N}[\![t]\!] \geq \mathcal{N}[\![t_i]\!]$

The proof for the lemma is presented in Figure 3.10. The crux of the proof is that all unfoldings of e-treeless function calls, with only variables in the arguments annotated as $^{\circ}$, will never increase the nesting measure, $\mathcal{N}$. Furthermore, since the $\mathcal{T}$ rules make use of the meta-function, gen_ren, to generalise and extract out sub-terms from the expression, we also need a secondary result in our proof which states that:

Lemma 3.14.2 - Each Nesting of Generalised and Extracted sub-terms is always less than the Source Expression

The *generalised expression* and each *extracted sub-term* from gen_ren will always has a nesting measure which is less than or equal to the source expression. This condition can be formally stated as follows:

$\mathcal{N}[\![t]\!] \geq \max\{\mathcal{N}[\![t^{\sim}]\!], \mathcal{N}[\![te]\!]_{1..s}\}$ **where** $(t^{\sim}, te_{1..s}, _, _, _) = =$ **gen_ren**$[\![t]\!]$

The proof for this secondary result can be shown by structural induction on each equation of gen_ren. This is detailed in Figure 3.9.

Having shown that the measure, $\mathcal{N}$, is bounded by the rules, we shall now proceed to show that the size of expressions for new function definitions is, in turn, bounded as well. More specifically, we shall show that the size of *generalised* expressions from gen_ren (used by each new function definition) is bounded by the measure, $\mathcal{N}$.

We provide the rest of our proof in two parts. The first part shows that all the expressions (occurring during transformation) conform to a special grammar form. After this, we show that the size of generalised expressions from this special grammar form is bounded by the nesting measure, $\mathcal{N}$.

Part One - Expressions conform to a Special Grammar Form

We observe that in any *finite* program, there is always some maximum finite nesting on its constructor terms. Let us assume a very large number, say Ω, to denote the maximum possible *contiguous* (unbroken by function call) constructor nestings for our program. We further assume that this limit is so large that it will never be reached in practice by any program. Using it, we can introduce an alternative grammar, eft, which implicitly states that such a limit exist, as follows:

$$
\begin{array}{lll}
\text{eft} & ::= & \text{eft}\{\Omega\} \\
\text{eft}\{x\} & ::= & \text{eft}\{x\text{-}1\} \mid c(\text{eft}\{x\text{-}1\}_1,..,\text{eft}\{x\text{-}1\}_n) \\
\text{eft}\{0\} & ::= & v \mid f(\text{eft}_{1..m}{}^{\diamond},\text{eft}_{m+1..n}{}^{\circ}) \mid g(\text{eft}_0{}^{\diamond},\text{eft}_{1..m}{}^{\diamond},\text{eft}_{m+1..n}{}^{\circ}) \\
 & & \quad \mid g(\text{eft}_0{}^{\circ},\text{eft}_{1..m}{}^{\diamond},\text{eft}_{m+1..n}{}^{\circ})
\end{array}
$$

These rules represent a class of grammars with different limits, Ω, that are progressively more general (cover larger classes of expressions) with bigger Ω. As this limit can be arbitrary large, we can always assume that there is some version of this class of grammars which will accept any expression that the original grammar, eft, accepts.

Correspondingly, the *e-treeless* form can also be specified in this manner with the grammar:

$$
\begin{array}{lll}
\text{et} & ::= & \text{et}\{\Omega\} \\
\text{et}\{x\} & ::= & \text{et}\{x\text{-}1\} \mid c(\text{et}\{x\text{-}1\}_1,..,\text{et}\{x\text{-}1\}_n) \\
\text{et}\{0\} & ::= & f_{id}(v) \mid f(v_{1..m}{}^{\diamond},\text{etv}_{m+1..n}{}^{\circ}) \mid g(v_0{}^{\diamond},v_{1..m}{}^{\diamond},\text{etv}_{m+1..n}) \\
 & & \quad \mid g(\text{etv}_0{}^{\circ},v_{1..m}{}^{\diamond},\text{etv}_{m+1..n}{}^{\circ}) \\
\text{etv} & ::= & v \mid \text{et}\{\Omega\}
\end{array}
$$

Again, we permit an arbitrary large nesting, Ω, of constructors to appear in the outer expression and in each argument, that is annotated as $^{\circ}$. However, this time we also use a special identity function, $f_{id}(v)=v$, in place of each variable, v, found directly within a constructor term (not variables within a function call). The use of this special identity function does not involve a loss of *generality* or *efficiency* because they can be easily *added* or *removed* from the e-treeless expression. We can visualise this identity function as existing to separate out *variables* from *constructors*, through intermediate functions. With it, we can show that any substitutions of arguments, satisfying the grammar of $\text{eft}\{\Omega\}$, into the variables of a term of the above e-treeless form, $\text{et}\{\Omega\}$, will result in a term which satisfies the original grammar form, $\text{eft}\{\Omega\}$. This condition will help to show that our transformation preserves the form of $\text{eft}\{\Omega\}$.

Using these alternative grammar forms, we can show that all expressions occurring under extended deforestation will always satisfy the grammar of $\text{eft}\{\Omega\}$. Such a result can be stated as follows:

Lemma 3.14.3 - Each expression, during deforestation, always satisfies $\text{eft}\{\Omega\}$

> Given an expression of the form, $\text{eft}\{\Omega\}$, for transformation by the $\mathcal{T}$ rules, the expression encountered during the recursive application of rules will continue to satisfy the same grammar form, $\text{eft}\{\Omega\}$.

The proof for Lemma 3.14.3 is given in Figure 3.11. In the proof, we also make use of the following secondary lemma which states that the generalised term and the extracted sub-terms from **gen_ren** will always satisfy the grammar form of $\text{eft}\{\Omega\}$.

Lemma 3.14.4 - Each Generalised or Extracted Subterm of $eft\{\Omega\}$ always satisfies $eft\{\Omega\}$

Given a term, ...t..., together with the extracted subterms, $te_{1..s}$, and the generalised term, ...t~...~, as shown below.

$(...t^{\sim}...^{\sim}, te_{1..s}, _, _) == \mathbf{gen_ren}[\![...t...]\!]$

If the term, ...t..., satisfies $eft\{\Omega\}$, then the extracted subterms and generalised term will continue to satisfy the grammar of $eft\{\Omega\}$. This can be formally stated as follows:

$...t... \in eft\{\Omega\} \Rightarrow (\forall\, i \in 1..s \;\; te_i \in eft\{\Omega\}, t^{\sim} \in eft\{\Omega\}, \forall\, x \in eft\{\Omega\} \Rightarrow ...x...^{\sim} \in eft\{\Omega\})$

The proof for this secondary result is trivial because the corresponding grammar the generalised expressions, $eft^{\sim}\{\Omega\}$:

$$
\begin{array}{lll}
eft^{\sim} & := & eft^{\sim}\{\Omega\} \\
eft^{\sim}\{x\} & := & eft^{\sim}\{x\text{-}1\} \mid c(eft^{\sim}\{x\text{-}1\}_1,..,eft^{\sim}\{x\text{-}1\}_n) \\
eft^{\sim}\{0\} & ::= & v \mid f(eft^{\sim}{}_{1..m}{}^{\diamond}, ve_{m+1..n}{}^{\circ}) \mid g(eft^{\sim}{}_0{}^{\diamond}, eft^{\sim}{}_{1..m}{}^{\diamond}, ve_{m+1..n}{}^{\circ}) \\
 & & \quad \mid g(ve_0{}^{\circ}, eft^{\sim}{}_{1..m}{}^{\diamond}, ve_{m+1..n}{}^{\circ})
\end{array}
$$

is a subset of the grammar of $eft\{\Omega\}$ where all arguments annotated as $^{\circ}$ are generalised to variables. Thus, each extracted subterm from the generalised expressions will also satisfy the grammar form of $eft\{\Omega\}$.

Part Two - Size of Generalised Terms is bounded by $\mathcal{N}$

The size of generalised term, satisfying the grammar form $eft^{\sim}\{\Omega\}$, must now be shown to be bounded by function nesting, $\mathcal{N}$. We introduce a new measure, $\mathcal{S}$, on the *actual* nesting of all constructors and functions to denote the actual size of expressions as follows.

$$
\begin{array}{lll}
\mathcal{S}\,[\![v]\!] & = & 0 \\
\mathcal{S}\,[\![c(t_1,..,t_n)]\!] & = & 1+\max\{\mathcal{S}\,[\![t_1]\!],..,\mathcal{S}\,[\![t_n]\!]\} \\
\mathcal{S}\,[\![f\,(t_{1..m}{}^{\diamond}, t_{m+1..n}{}^{\circ})]\!] & = & 1+\max\{\mathcal{S}\,[\![t_1]\!],..,\mathcal{S}\,[\![t_n]\!]\} \\
\mathcal{S}\,[\![g\,(t_0{}^{\diamond}, t_{1..m}{}^{\diamond}, t_{m+1..n}{}^{\circ})]\!] & = & 1+\max\{\mathcal{S}\,[\![t_0]\!],..,\mathcal{S}\,[\![t_n]\!]\} \\
\mathcal{S}\,[\![g\,(t_0{}^{\circ}, t_{1..m}{}^{\diamond}, t_{m+1..n}{}^{\circ})]\!] & = & 1+\max\{\mathcal{S}\,[\![t_0]\!],..,\mathcal{S}\,[\![t_n]\!]\}
\end{array}
$$

This size measure, $\mathcal{S}$, can be shown to be bounded by the function nesting, $\mathcal{N}$, as stated in the following lemma.

Lemma 3.14.5 - Size, $\mathcal{S}$, of Generalised Expression is bounded by $\mathcal{N}$

The nesting measure, $\mathcal{N}$, directly bounds the size, $\mathcal{S}$, of generalised expression, denoted by $eft^{\sim}$.

In fact, we can prove that the following bounded relation, $\mathcal{S}\,[\![eft^{\sim}\{\Omega\}]\!] \leq \Omega+1+(\Omega+1)^{*}\mathcal{N}\,[\![eft^{\sim}\{\Omega\}]\!]$, holds for each generalised terms, $eft^{\sim}\{\Omega\}$. This proof is given in Figure 3.12 by showing the relation holds for each case of the grammar rule, $eft^{\sim}\{x\}$.

The presence of a bound on the *size* of each new function definition, together with a finite (initial) set of constructor and function symbols implies that the number of new definitions introduced is bounded. This means that there is a *finite* number of different (modulo renaming) new function definitions which could be introduced. As a result, the number of define&unfold

steps ($\mathcal{T}$7c,8b,9b) will be *finite* for the transformation of each expression, that satisfies the Extended Deforestation Theorem conditions. The other (recursive) $\mathcal{T}$ rules, can be divided into two groups, namely:

1) those which operate on successively smaller sub-expressions, $\mathcal{T}$ 2,3,4,5,7b,8a,9b, and
2) the *direct unfold* rules ($\mathcal{T}$ 6a,7a) and the *define&unfold* rule ($\mathcal{T}$ 6b), without the corresponding fold rules.

The rules of the first group have a decreasing measure (size of expression) and cannot therefore have infinite sequences between themselves. The rules of the second group are a little problematic because they do not decrease the size of expressions. However, the unfold steps of rules $\mathcal{T}$ 6a,6b propagate constructor terms outwards. There is therefore a decreasing measure on the number of deep constructors that are present in the expression (after each application of these rules). Correspondingly, the direct unfold step of $\mathcal{T}$7a replaces a non-recursive function by its set of auxiliary functions. We could again use the function hierarchy numbers (see Section 3.2.5) to show that the overall expression has a decreasing measure whenever a non-recursive function call is unfolded. This would give a well-founded measure for rule $\mathcal{T}$7a.

These two well-founded decreasing measures can be combined with the size of expressions to give a more complex measure that is decreasing for these two groups of recursive $\mathcal{T}$ rules. A detailed proof for such a well-founded measure is given in [Chin90]. This proof is very similar in style to that given in Appendix 5 for the Basic Deforestation Algorithm.

With this well-founded measure, each sequence of steps involving these rules will always be finite. As these finite sequences of steps could only occur among a finite number of the define&unfold rules, the overall sequence of (recursive) steps could never be infinite. Hence, the Extended Deforestation Algorithm will terminate under the conditions of its theorem.

Proof of Lemma 3.14.2

Prove: For each equation of meta-function, **gen_ren**

$$\mathcal{N}[\![t]\!] \geq \max\{\mathcal{N}[\![t\tilde{}]\!], \mathcal{N}[\![te_lt]\!]\} \textbf{where}(t\tilde{}, te_lt, _, _, _) == \textbf{gen_ren}[\![t]\!]$$

Case (1) t=v :

From definition of $\mathcal{N}$: $\mathcal{N}[\![v]\!] = 0 = \max\{\mathcal{N}[\![v\tilde{}]\!]\}$

Hence: $\mathcal{N}[\![v]\!] \geq \max\{\mathcal{N}[\![v\tilde{}]\!]\}$ **where**$(v\tilde{}, [], _, _, _) == \textbf{gen_ren}[\![v]\!]$

Case (2) $t=c(t_1,..,t_n)$:

From definition of $\mathcal{N}$: $\mathcal{N}[\![c(t_1,..,t_n)]\!] = \max\{\mathcal{N}[\![t_1]\!],..,\mathcal{N}[\![t_n]\!]\}$

Given $\forall i \in 1..n$ $(t\tilde{}_i, te_lt_i, _, _, _) == \textbf{gen_ren}[\![t_i]\!]$,

by induction, we have $\forall i \in 1..n$ $\max\{\mathcal{N}[\![t_i]\!]\} \geq \max\{\mathcal{N}[\![t\tilde{}_i]\!]\mathcal{N}[\![te_lt_i]\!]\}$

thus: $\mathcal{N}[\![c(t_1,..,t_n)]\!] \geq \max\{\mathcal{N}[\![t\tilde{}_1]\!],..,\mathcal{N}[\![t\tilde{}_n]\!]\mathcal{N}[\![te_lt]\!]_{1..n}\}$

From definition of $\mathcal{N}$: $\geq \max\{\mathcal{N}[\![c(t\tilde{}_1,..,t\tilde{}_n)]\!]\mathcal{N}[\![te_lt]\!]_{1..n}\}$

Hence: $\mathcal{N}[\![c(t_1,..,t_n)]\!] \geq \max\{\mathcal{N}[\![c(t\tilde{}_1,..,t\tilde{}_n)]\!]\mathcal{N}[\![te_lt]\!]_{1..n}\}$

where $(c(t\tilde{}_1,..,t\tilde{}_n), te_lt_{1..n}, _, _, _) == \textbf{gen_ren}[\![c(t_1,..,t_n)]\!]$

Case (3) $t=f(t_{1..m}^{\oplus}, t_{m+1..n}^{\ominus})$:

From definition of $\mathcal{N}$:

$\mathcal{N}[\![f(t_{1..m}, t_{m+1..n})]\!] = \max\{1+\max\{\mathcal{N}[\![t]\!]_{1..m}\}, \mathcal{N}[\![t]\!]_{m+1..n}\}$

Given $\forall i \in 1..n$ $(t\tilde{}_i, te_lt_i, _, _, _) == \textbf{gen_ren}[\![t_i]\!]$,

by induction, we have $\forall i \in 1..m$ $\max\{\mathcal{N}[\![t_i]\!]\} \geq \max\{\mathcal{N}[\![t\tilde{}_i]\!]\mathcal{N}[\![te_lt]\!]_i\}$

Thus: $\mathcal{N}[\![f(t_{1..m}, t_{m+1..n})]\!] \geq \max\{1+\max\{\mathcal{N}[\![t\tilde{}]\!]_{1..m}, \mathcal{N}[\![te_lt]\!]_{1..m}\}, \mathcal{N}[\![t]\!]_{m+1..n}\}$

$\geq \max\{1+\max\{\mathcal{N}[\![t\tilde{}]\!]_{1..m}\}, 1+\max\{\mathcal{N}[\![te_lt]\!]_{1..m}\}, \mathcal{N}[\![t]\!]_{m+1..n}\}$

$\geq \max\{1+\max\{\mathcal{N}[\![t\tilde{}]\!]_{1..m}\}, \mathcal{N}[\![te_lt]\!]_{1..m}, \mathcal{N}[\![t]\!]_{m+1..n}\}$

From definition of $\mathcal{N}$: $\geq \max\{\mathcal{N}[\![f(t\tilde{}_{1..m}, ve_{m+1..n})]\!], \mathcal{N}[\![te_lt]\!]_{1..m}, \mathcal{N}[\![t]\!]_{m+1..n}\}$

Hence: $\mathcal{N}[\![f(t_{1..m}, t_{m+1..n})]\!] \geq \max\{\mathcal{N}[\![f(t\tilde{}_{1..m}, ve_{m+1..n})]\!], \mathcal{N}[\![te_lt]\!]_{1..m}, \mathcal{N}[\![t]\!]_{m+1..n}\}$

where $(f(t\tilde{}_{1..m}, ve_{m+1..n}), t_{m+1..n}, te_lt_{1..n}, _, _, _) == \textbf{gen_ren}[\![f(t_{1..m}, t_{m+1..n})]\!]$

Case (4) $t=g(t_0^{\oplus}, t_{1..m}^{\oplus}, t_{m+1..n}^{\ominus})$: Similar to Case (3)

Case (5) $t=g(t_0^{\ominus}, t_{1..m}^{\oplus}, t_{m+1..n}^{\ominus})$: Similar to Case (3)

Figure 3.9 - Proof of Lemma 3.14.2 (gen_ren produces smaller terms)

Proof of Lemma 3.14.1

Assume Lemma 3.14.2: $\mathcal{N}[\![t]\!] \geq \max\{\mathcal{N}[\![t\tilde{}]\!], \mathcal{N}[\![te]\!]_{1..s}\}$ **where** $(t\tilde{}, te_{1..s}, _, _, _) == \textbf{gen_ren}[\![t]\!]$

Prove: For each transformation step, $\mathcal{T}[\![t]\!] \Leftarrow .....\mathcal{T}[\![t_i]\!].....,\forall i.\, \mathcal{N}[\![t]\!] \geq \mathcal{N}[\![t_i]\!]$

Case for Rule $\mathcal{T}$ 1 - v : nothing to prove

Case for Rule $\mathcal{T}$ 2- $c(t_1,..,t_n)$:

From the definition: $\mathcal{N}[\![c(t_1,..,t_n)]\!] = \max\{\mathcal{N}[\![t_1]\!],..,\mathcal{N}[\![t_n]\!]\}$

- thus $\forall i \in 1..n$ $\mathcal{N}[\![t_i]\!] \leq \mathcal{N}[\![c(t_1,..,t_n)]\!]$

Case for Rule $\mathcal{T}$ 3 - $f(v_{1..m}, t_{m+1..n})$:

From the definition: $\mathcal{N}[\![f(v_{1..m}, t_{m+1..n})]\!] = \max\{\mathcal{N}[\![t]\!]_{m+1..n}\}$

- thus $\forall i \in m+1..n$ $\mathcal{N}[\![t_i]\!] \leq \mathcal{N}[\![f(v_{1..m}, t_{m+1..n})]\!]$

Case for Rule $\mathcal{T}$ 4 - $g(v_0^{\oplus}, v_{1..m}, t_{m+1..n})$: Similar to Case for Rule $\mathcal{T}$ 3

Case for Rule $\mathcal{T}$ 5 - $g(t_0^{\ominus}, v_{1..m}, t_{m+1..n})$: Similar to Case for Rule $\mathcal{T}$ 3

Case for Rule $\mathcal{T}$ 6 - $...g(c_i(t'_{1..j})^{\oplus},t_{1..m}{}^{\oplus},t_{m+1..n}{}^{\ominus})...$, where $g(c_i(v_1'..v_j'),v_1..v_n)=tg_i$:

a) $t_{m+1..n}$ are variables:

From Definition: $\mathcal{N}[\![g\ (c_i(t'_{1..j}),t_{1..m},t_{m+1..n})\ ]\!]$

$= \max\{1+\max\{\mathcal{N}[\![t']\!]_{1..j},\mathcal{N}[\![t]\!]_{1..m}\},\mathcal{N}[\![t]\!]_{m+1..n}\}$

Since $t_{m+1..n}$ are variables $= 1+\max\{\mathcal{N}[\![t']\!]_{1..j},\mathcal{N}[\![t\ ]\!]_{1..m}\}$

Since g is e-treeless, $\mathcal{N}[\![tg_i\ ]\!]\leq 1$ $\geq \mathcal{N}[\![tg_i\ ]\!]+\max\{\mathcal{N}[\![t']\!]_{1..j},\mathcal{N}[\![t]\!]_{1..m}\}$

$\geq \mathcal{N}[\![tg_i\ [t'/v'_{1..j},t/v_{1..m},t/v_{m+1..n}]\ ]\!]$

Adding a context to each side:

- $\mathcal{N}[\![...tg_i\ [t'/v'_{1..j},t/v_{1..n}]...\ ]\!]\leq \mathcal{N}[\![...g(c_i(t'_{1..j}),t_{1..m},t_{m+1..n})...\ ]\!]$

b) otherwise:

From Lemma 3.14.2 on **gen_ren**:

$\mathcal{N}[\![...g\ (c_i(t'_{1..j}),t_{1..m},t_{m+1..n})...\ ]\!]$

$\geq \max\{\mathcal{N}[\![...g(c_i(t'^{\sim}_{1..j}),t^{\sim}{}_{1..m},ve_{m+1..n})...^{\sim}\ ]\!],\mathcal{N}[\![te\ ]\!]_{1..s}\}$

- Thus : $\forall\ i\in 1..s$: $\mathcal{N}[\![te_i]\!]\leq\mathcal{N}[\![...g\ (c_i(t'_{1..j}),t_{1..m},t_{m+1..n})...\ ]\!]$

Also, From Definition: $\mathcal{N}\ [\![g(c_i(t'^{\sim}_{1..j}),t^{\sim}{}_{1..m},ve_{m+1..n})\ ]\!]$

$= \max\{1+\max\{\mathcal{N}[\![t'^{\sim}]\!]_{1..j},\mathcal{N}[\![t^{\sim}]\!]_{1..m}\},\mathcal{N}[\![ve]\!]_{m+1..n}\}$

$= 1+\max\{\mathcal{N}[\![t'^{\sim}]\!]_{1..j},\mathcal{N}[\![t^{\sim}]\!]_{1..m}\}$

Since g is e-treeless, $\mathcal{N}[\![tg_i\ ]\!]\leq 1$ $\geq \mathcal{N}[\![tg_i\ ]\!]+\max\{\mathcal{N}[\![t'^{\sim}]\!]_{1..j},\mathcal{N}[\![t^{\sim}\ ]\!]_{1..m}\}$

$\geq \mathcal{N}[\![tg_i[t'^{\sim}\ /v'_{1..j},t^{\sim}\ /v_{1..m},ve/v_{m+1..n}]\ ]\!]$

- Thus: $\mathcal{N}[\![...tg_i[v'\ /v'_{1..j},t^{\sim}\ /v_{1..m},ve/v_{m+1..n}]...^{\sim}]\!]\leq \mathcal{N}\ [\![...g\ (c_i(t'_{1..j}),t_{1..m},t_{m+1..n})...\ ]\!]$

Case for Rule $\mathcal{T}$ 7 $...f\ (t_{1..m}{}^{\oplus},t_{m+1..n}{}^{\ominus})...$, where $f(v_1,..,v_n)=tf$:

From Lemma 3.14.2 on **gen_ren**:

$\mathcal{N}[\![...f(t_{1..m},t_{m+1..n})...\ ]\!]\ \geq\ \max\{\mathcal{N}[\![...f(t^{\sim}{}_{1..m},ve_{m+1..n})...^{\sim}\ ]\!]\mathcal{N}[\![te\ ]\!]_{1..s}\}$

a) $t_{m+1..n}$ are variables: Similar to Case for Rule $\mathcal{T}$ 6a

b) apply fold against old definition:

- From Lemma 3.14.2: $\forall\ i\in 1..s$: $\mathcal{N}[\![te_i]\!]\leq\mathcal{N}[\![...f(t_{1..m},t_{m+1..n})...\ ]\!]$

c) otherwise: Similar to Case for Rule $\mathcal{T}$ 6b

Case for Rule $\mathcal{T}$ 8 - $...g(v_0{}^{\oplus},t_{1..m}{}^{\oplus},t_{m+1..n}{}^{\ominus})...$:

From Lemma 3.14.2 on **gen_ren**:

$\mathcal{N}[\![...g(v_0,t_{1..m},t_{m+1..n})...\ ]\!]\geq\ \max\{\mathcal{N}[\![...g(vn_0,t^{\sim}{}_{1..m},ve_{m+1..n})...^{\sim}\ ]\!],\mathcal{N}[\![te\ ]\!]_{1..s}\}$:

a) apply fold against old definition:

- From Lemma 3.14.2: $\forall\ i\in 1..s$: $\mathcal{N}[\![te_i]\!]\leq\mathcal{N}[\![...g(v_0,t_{1..m},t_{m+1..n})...\ ]\!]$

b) otherwise:

- From Lemma 3.14.2: $\forall\ i\in 1..s$: $\mathcal{N}[\![te_i]\!]\leq\mathcal{N}[\![...g(v_0,t_{1..m},t_{m+1..n})...\ ]\!]$

Also, From Definition: $\mathcal{N}[\![g(vn_0,t^{\sim}{}_{1..m},ve_{m+1..n})\ ]\!]$

$= \max\{1+\max\{\mathcal{N}[\![vn_0]\!],\mathcal{N}[\![t^{\sim}]\!]_{1..m}\},\mathcal{N}[\![ve]\!]_{m+1..n}\}$

$= 1+\max\{\mathcal{N}[\![t^{\sim}]\!]_{1..m}\}$

For each equation $g(p_i,v_1,..,v_n)=tg_i$:

Since g is e-treeless, $\mathcal{N}[\![tg_i\ ]\!]\leq 1$ $\geq \mathcal{N}[\![tg_i\ ]\!]+\max\{\mathcal{N}[\![t^{\sim}\ ]\!]_{1..m}\}$

$\geq \mathcal{N}[\![tg_i[v'\ /v'_{1..j},t^{\sim}\ /v_{1..m},ve/v_{m+1..n}]\ ]\!]$

- Thus: $\mathcal{N}[\![...\ tg_i[v'\ /v'_{1..j},t^{\sim}\ /v_{1..m},ve/v_{m+1..n}]...^{\sim}]\!]\leq \mathcal{N}[\![...g(v_0,t_{1..m},t_{m+1..n})...\ ]\!]$

Case for Rule $\mathcal{T}$ 9 - $...g(t_0{}^{\ominus},t_{1..m}{}^{\oplus},t_{m+1..n}{}^{\ominus})...$: Similar to Case for Rule $\mathcal{T}$ 8

Figure 3.10 - Proof of Lemma 3.14.1($\mathcal{N}$ is bounded)

Proof of Lemma 3.14.3

Assume Lemma 3.14.4: $...t... \in eft\{\Omega\} \Rightarrow (\forall\, i \in 1..s\ \ te \in eft\{\Omega\},\ t^{\sim} \in eft\{\Omega\}$
$,\ \forall\, x \in eft\{\Omega\} \Rightarrow ...x...^{\sim} \in eft\{\Omega\})$ **where** $(...t^{\sim}...^{\sim}, te_{1..s}, _, _) ==$ **gen_ren**$[\![...t...]\!]$

Prove: For each step, $\mathcal{T}\ [\![\ t\ ]\!] \Leftarrow .....\mathcal{T}\ [\![\ t_i\ ]\!]$ ....., $\forall\, i.\ t \in eft\{\Omega\} \Rightarrow t_i \in eft\{\Omega\}$

Case for Rule $\mathcal{T}$ 1 - v : nothing to prove

Case for Rule $\mathcal{T}$ 2 - $c(t_1,..,t_n)$:

$c(t_1,..,t_n) \in eft\{\Omega\} \Rightarrow c(t_1,..,t_n) \in c(eft\{\Omega\text{-}1\}_1,..,eft\{\Omega\text{-}1\}_n)$

- thus $\forall\, i \in 1..n \Rightarrow t_i \in eft\{\Omega\text{-}1\}$
  $\Rightarrow t_i \in eft\{\Omega\}$

Case for Rule $\mathcal{T}$ 3 - $f(v_{1..m}^{\oplus}, t_{m+1..n}^{\ominus})$:

$f(v_{1..m}, t_{m+1..n}) \in eft\{\Omega\} \Rightarrow f(v_{1..m}, t_{m+1..n}) \in f(eft\{\Omega\}_{1..m}, eft\{\Omega\}_{m+1..n})$

- thus $\forall\, i \in m+1..n \Rightarrow t_i \in eft\{\Omega\}$

Case for Rule $\mathcal{T}$ 4 - $g(v_0^{\oplus}, v_{1..m}^{\oplus}, t_{m+1..n}^{\ominus})$: similar to case for $\mathcal{T}$ 3

Case for Rule $\mathcal{T}$ 5 - $g(t_0^{\ominus}, v_{1..m}^{\oplus}, t_{m+1..n}^{\ominus})$: similar to case for $\mathcal{T}$ 3

Case for Rule $\mathcal{T}$ 6 - $...g(c_i(t'_{1..j})^{\oplus}, t_{1..m}^{\oplus}, t_{m+1..n}^{\ominus})...$, where $g(c_i(v_1'..v_j'), v_1..v_n) = tg_i$:

a) $t_{m+1..n}$ are variables:

$...g\ (c_i(t'_{1..j}), t_{1..m}, t_{m+1..n})... \in eft\{\Omega\} \Rightarrow g\ (c_i(t'_{1..j}), t_{1..m}, t_{m+1..n})$
$\in g\ (c_i(eft\{\Omega\text{-}1\}_{1..j}), eft\{\Omega\}_{1..m}, v_{m+1..n})$
$\Rightarrow \forall\, i \in 1..j\ t'_i \in eft\{\Omega\text{-}1\}, \forall\, i \in 1..m\ t_i \in eft\{\Omega\}$

Since g is e-treeless, $tg_i \in et\{\Omega\}$, $\Rightarrow tg_i\,[t'/v'_{1..j}, t/v_{1..m}, t/v_{m+1..n}] \in eft\{\Omega\}$

- Adding the context: $\Rightarrow ...tg_i\,[t'/v'_{1..j}, t/v_{1..m}, t/v_{m+1..n}]... \in eft\{\Omega\}$

b) otherwise:

From Lemma 3.14.4 on **gen_ren** $\Rightarrow ...g\ (c_i(t^{\sim}{}'_{1..j}), t^{\sim}{}_{1..m}, ve_{m+1..n})\ ...^{\sim} \in eft\{\Omega\}$,
$\forall\, i \in 1..s\ \ te_i \in eft\{\Omega\}$

- Thus: each extracted term $\Rightarrow te_i \in eft\{\Omega\}$

Also, $\Rightarrow \forall\, i \in 1..j\ t^{\sim}{}'_i \in eft\{\Omega\text{-}1\}, \forall\, i \in 1..m\ t^{\sim}{}_i \in eft\{\Omega\}$

Since g is e-treeless, $tg_i \in et\{\Omega\}$, $\Rightarrow tg_i\,[t^{\sim}{}'/v'_{1..j}, t^{\sim}/v_{1..m}, ve/v_{m+1..n}] \in eft\{\Omega\}$

- Adding the generalised context: $\Rightarrow ...tg_i\,[t^{\sim}{}'/v'_{1..j}, t^{\sim}/v_{1..m}, ve/v_{m+1..n}]...^{\sim} \in eft\{\Omega\}$

Case for Rule $\mathcal{T}$ 7 $...f\ (t_{1..m}^{\oplus}, t_{m+1..n}^{\ominus})...$:

a) $t_{m+1..n}$ are variables: Similar to Case for $\mathcal{T}$ 6a

b) apply fold against old expression:

From Lemma 3.14.4 on **gen_ren** $\Rightarrow ...f\ (t_{1..m}^{\oplus}, t_{m+1..n}^{\ominus})\ ...^{\sim} \in eft\{\Omega\}$,
$\forall\, i \in 1..s\ \ te_i \in eft\{\Omega\}$

- Thus: each extracted term, $\Rightarrow te_i \in eft\{\Omega\}$

c) otherwise : Similar to the generalise, define, unfold step of $\mathcal{T}$ 6b

Case for Rule $\mathcal{T}$ 8 ...$g(v_0^{\diamond},t_{1..m}^{\diamond},t_{m+1..n}^{\circ})$....,

a) apply fold against old expression: Similar to Case for $\mathcal{T}$ 7b

b) otherwise:

From Lemma 3.14.4 on **gen_ren** $\Rightarrow$ ...g $(v_0,t\tilde{}_{1..m},ve_{m+1..n})$ ...$\tilde{}$ $\in$ eft{Ω} ,
$\forall$ i $\in$ 1..s $te_i \in$ eft{Ω}

- Thus: each extracted term $\Rightarrow$ $te_i \in$ eft{Ω}

Also, $\Rightarrow$ $\forall$ i $\in$ 1..m $t\tilde{}_i \in$ eft{Ω}

For each equation, $g(p_i,v_1,..,v_n) = tg_i$

Since g is e-treeless, $tg_i \in$ et{Ω}, $\Rightarrow$ $tg_i\,[v'/v'_{1..j},t\tilde{}/v_{1..m},ve/v_{m+1..n}] \in$ eft{Ω}

- Adding the generalised context: $\Rightarrow$ ...$tg_i\,[v'/v'_{1..j},t\tilde{}/v_{1..m},ve/v_{m+1..n}]$...$\tilde{}$ $\in$ eft{Ω}

Case for Rule $\mathcal{T}$ 9 - ...$g(t_0^{\circ},t_{1..m}^{\diamond},t_{m+1..n}^{\circ})$...: Similar to Cases for $\mathcal{T}$ 8

Figure 3.11 - Proof of Lemma 3.14.3 (eft{Ω} is preserved)

Proof of Lemma 3.14.5

To Prove: For each grammar case of eft$\tilde{}${x}, $\mathcal{S}[\![\text{eft}\tilde{}\{x\}]\!] \leq (x+1)+(\Omega+1)*\mathcal{N}[\![\text{eft}\tilde{}\{x\}]\!]$

Case (1), when $x \neq 0$, eft$\tilde{}${x} $\rightarrow$ eft$\tilde{}${x-1}:

To Prove: $\mathcal{S}[\![\text{eft}\tilde{}\{x-1\}]\!] \leq (x+1)+(\Omega+1)*\mathcal{N}[\![\text{eft}\tilde{}\{x-1\}]\!]$

By Induction: $\mathcal{S}[\![\text{eft}\tilde{}\{x-1\}]\!] \leq ((x-1)+1)+(\Omega+1)*\mathcal{N}[\![\text{eft}\tilde{}\{x-1\}]\!]$

$\Rightarrow$ $\leq (x)+(\Omega+1)*\mathcal{N}[\![\text{eft}\tilde{}\{x-1\}]\!]$

$\Rightarrow$ $\leq (x+1)+(\Omega+1)*\mathcal{N}[\![\text{eft}\tilde{}\{x-1\}]\!]$

Hence the above proof.

Case (2), when $x \neq 0$, eft{x} $\rightarrow$ c(eft$\tilde{}${x-1}$_1$,..,eft$\tilde{}${x-1}$_n$):

To Prove: $\mathcal{S}[\![c(\text{eft}\tilde{}\{x-1\}_1,..,\text{eft}\tilde{}\{x-1\}_n)]\!] \leq x+1+(\Omega+1)*\mathcal{N}[\![c(\text{eft}\tilde{}\{x-1\}_1,..,\text{eft}\tilde{}\{x-1\}_n)]\!]$

From Definition: $\mathcal{S}[\![c(\text{eft}\tilde{}\{x-1\}_1,..,\text{eft}\tilde{}\{x-1\}_n)]\!]$

$= 1+\max\{\mathcal{S}[\![\text{eft}\tilde{}\{x-1\}_1]\!],..,\mathcal{S}[\![\text{eft}\tilde{}\{x-1\}_n]\!]\}$

From Induction $\forall$ i $\in$ 1..n, $\mathcal{S}[\![\text{eft}\{x-1\}_i]\!] \leq (x-1)+1+(\Omega+1)*\mathcal{N}[\![\text{eft}\{x-1\}_i]\!]$:

$\leq 1+\max\{(x-1)+1+(\Omega+1)*\mathcal{N}[\![\text{eft}\{x-1\}_1]\!],..,(x-1)+1+(\Omega+1)*\mathcal{N}[\![\text{eft}\{x-1\}_n]\!]\}$

$\Rightarrow$ $\leq 1+(x-1)+1+(\Omega+1)*\max\{\mathcal{N}[\![\text{eft}\{x-1\}_1]\!],..,\mathcal{N}[\![\text{eft}\{x-1\}_n]\!]\}$

$\Rightarrow$ $\leq x+1+(\Omega+1)*\max\{\mathcal{N}[\![\text{eft}\{x-1\}_1]\!],..,\mathcal{N}[\![\text{eft}\{x-1\}_n]\!]\}$

From the Definition of $\mathcal{N}$: $\leq x+1+(\Omega+1)*\mathcal{N}[\![c(\text{eft}\tilde{}\{x-1\}_1,..,\text{eft}\tilde{}\{x-1\}_n)]\!]$

Hence the above proof.

Case (3), when $x=0$, eft{x} $\rightarrow$ v:

To Prove: $\mathcal{S}[\![v]\!] \leq 1+(\Omega+1)*\mathcal{N}[\![v]\!]$

By the definitions of + and *:

$0 \leq 1+(\Omega+1)*0$

Hence, by the definitions of $\mathcal{S}$ and $\mathcal{N}$ we have:

$\mathcal{S}[\![v]\!] \leq 1+(\Omega+1)*\mathcal{N}[\![v]\!]$

Case (4), when x=0, $eft\{x\} \rightarrow f(eft^{\sim}\{\Omega\}_{1..m}^{\diamond}, v_{m+1..n}^{\circ})$:

To Prove: $\mathcal{S}[\![f(eft^{\sim}\{\Omega\}_{1..m}, v_{m+1..n})]\!] \leq (0+1)+(\Omega+1)*\mathcal{N}[\![f(eft^{\sim}\{\Omega\}_{1..m}, v_{m+1..n})]\!]$

From Definition: $\mathcal{S}[\![f(eft^{\sim}\{\Omega\}_{1..m}, v_{m+1..n})]\!]$

$= 1+\max\{\mathcal{S}[\![eft^{\sim}\{\Omega\}]\!]_{1..m}\}$

From Induction $\forall i \in 1..m$, $\mathcal{S}[\![eft\{\Omega\}_i]\!] \leq \Omega+1+(\Omega+1)*\mathcal{N}[\![eft\{\Omega\}_i]\!]$:

$\leq 1+\max\{\Omega+1+(\Omega+1)*\mathcal{N}[\![eft\{\Omega\}_1]\!], ..., \Omega+1+(\Omega+1)*\mathcal{N}[\![eft\{\Omega\}_m]\!]\}$

$\leq 1+\max\{(\Omega+1)*(1+\mathcal{N}[\![eft\{\Omega\}_1]\!]), ..., (\Omega+1)*(1+\mathcal{N}[\![eft\{\Omega\}_m]\!])\}$

$\leq 1+(\Omega+1)*(1+\max\{\mathcal{N}[\![eft\{\Omega\}]\!]_{1..m}\})$

From the Definition of $\mathcal{N}$: $\leq 1+(\Omega+1)*\mathcal{N}[\![f(eft^{\sim}\{\Omega\}_{1..m}, v_{m+1..n})]\!]$

Hence the above proof.

Case (5), when x=0, $eft\{x\} \rightarrow g(eft^{\sim}\{\Omega\}_0^{\diamond}, eft^{\sim}\{\Omega\}_{1..m}^{\diamond}, v_{m+1..n}^{\circ})$: Similar to Case (4)

Case (6), when x=0, $eft\{x\} \rightarrow g(v_0^{\circ}, eft^{\sim}\{\Omega\}_{1..m}^{\diamond}, v_{m+1..n}^{\circ})$: Similar to Case (4)

Figure 3.12 - Proof of Lemma 3.14.5 ($\mathcal{S}$ is bounded by $\mathcal{N}$ for all generalised terms)

3.3.7. A Minor Improvement to Extended Deforestation

There is a minor improvement possible to rule $\mathcal{T}3$ of Extended Deforestation. Previously, this rule left each outer f-type function call, which is already e-treeless, unchanged. This is not the most desirable thing to do because *non-recursive* f-type function calls, which do not cause any loss of efficiency, can always be improved by direct unfolding. Thus, we can change $\mathcal{T}3$ to the following:

(3) $\mathcal{T}[\![f(v_{1..m}^{\diamond}, t_{m+1..n}^{\circ})]\!]$

a) if f is non-recursive and $t_{m+1..n}$ are all variables/constants

$\Leftarrow \quad \mathcal{T}[\![tf[v/v_{1..m}, t/v_{m+1..n}]...]\!]$

b) otherwise $\Leftarrow \quad f(v_1..v_m, \mathcal{T}[\![t]\!]_{m+1..n})$

The new direct unfold step ($\mathcal{T}3a$) need not introduce any new function definition because the f-type call, being non-recursive, cannot re-occur by unfolding. In addition, Lemmas 3.14.1 to 3.14.5 of the termination proof continue to hold with this new step.

Notice that this improvement can only be applied to $\mathcal{T}$ 3 and not to analogous rules, $\mathcal{T}$ 4 and 5, of g-type function calls. This is because g-type function calls cannot be directly unfolded because we would like to adhere to the language restriction of one pattern-matching parameter per g-type function.

3.4. Deforestation for First-Order Expressions

Thus far, both the basic and extended Deforestation Algorithms are only applicable to expressions composed *solely* from treeless functions. But what about those expressions which use a *mix* of treeless and non-treeless functions? Such expressions may still contain intermediate terms that are worth eliminating and we should strive to extend our technique to work for them as well!

Let us look at an example of a non-treeless function, called rev_flatten, as shown below.

```
dec rev_flatten : list(list(alpha)) -> list(alpha);
- - -rev_flatten([])     <= [] ;
- - -rev_flatten(a::x)   <= append(rev_flatten(x)+,a+) ;
```

This function operates on a list of lists and is used to simultaneously flatten and reverse the outer list. The main reason why the above function is *non-treeless* is that its second equation contains a recursive term in the first parameter (annotated as $^+$) of the append function call. As a result, the Extended Deforestation Algorithm cannot guarantee the safe transformation of expressions which use this non-treeless function. This includes the RHS expression of the function itself.

For example, if we attempt to transform the RHS of rev_flatten definition, using $\mathcal{T}$, we will simply enter into non-termination as shown in Figure 3.13. Here, we get progressively larger intermediate function definitions, rev_flatten_1,rev_flatten_2,..., rev_flatten_N,.., being introduced as new functions!

---rev_flatten([])	<=	$\mathcal{T}$ [[[]]]	! $\mathcal{T}$ 2
	<=	[]	
---rev_flatten(a::x)	<=	$\mathcal{T}$ [[append(rev_flatten(x),a)]]	! $\mathcal{T}$ 9b, define rev_flatten_1
	<=	rev_flatten_1(x,a)	
define			
---rev_flatten_1(x,a_1)	<=	append(rev_flatten(x),a_1)	! narrow rev_flatten
---rev_flatten_1([],a_1)	<=	$\mathcal{T}$ [[append([],a_1)]]	! $\mathcal{T}$ 6a
	<=	$\mathcal{T}$ [[a_1]]	! $\mathcal{T}$ 1
	<=	a_1	
---rev_flatten_1(a_2::x,a_1)	<=	$\mathcal{T}$ [[append(append(rev_flatten(x),a_2),a_1)]]	! $\mathcal{T}$ 9b, define rev_flatten_2
	<=	rev_flatten_2(x,a_2,a_1)	
define			
---rev_flatten_2(x,a_2,a_1)	<=	append(append(rev_flatten(x),a_2),a_1)	! narrow rev_flatten
:		:	
define			
---rev_flatten_N(x,a_N..a_1)	<=	append(...append(rev_flatten(x),a_N)...,a_1)	! narrow rev_flatten
:		:	

Figure 3.13 - An Unsuccessful Application of $\mathcal{T}$ on a mix of treeless and non-treeless functions

The main reason for this non-termination is that the rev_flatten function is itself non-treeless and should not therefore be selected for unfolding. Unfolding such non-treeless function calls will

simply result in successively larger expressions, and violate the bounded nesting measure property. Under our consumer-producer model, this kind of recursive functions are actually the *bad producers*!

There is simply no way in which rev_flatten can be converted to treeless form by the purely unfold/fold means of the deforestation algorithm. Only special transformation techniques which make use of semantic laws, like associativity, *may* possibly be able to convert such functions to e-treeless form. However, this use of laws is not generally automatable.

All is not lost, as there is one very neat method (or trick!) which could be used to make any non-treeless expression *appear* treeless. This method involves the use of the **let** construct to extract out the offending subterm, so that the expression becomes *pseudo-treeless*. The earlier equation for rev_flatten can be made pseudo-treeless by this trick, as follows:

```
- - -rev_flatten(a::x)  <=  let v==rev_flatten(x) In append(v,a) ;
```

Here, a local variable, v, is introduced by the **let** construct to abstract out the offending subterm, rev_flatten(x), in append(v,a). The overall effect of this is that both subexpressions, rev_flatten(x) and append(v,a), of the **let** construct are now treeless. As a result, the entire equation *appears* treeless.

One *caveat* in this use of **let** constructs is that it is too general and can actually convert any expression into pseudo-treeless form. This is a cause for concern because we ought to avoid pseudo-treelessness if an expression can be deforested into *genuine* treeless form. As an example, let us look at the sumdb function.

```
- - -sumdb(x)          <=  sum(double(x)) ;
```

This definition contains a nesting of two treeless functions, sum and double. It is not yet treeless but can be transformed into genuine treeless form by the deforestation algorithm. As such, it would be unnecessary to use the pseudo-treeless **let** construct, in the following way:

```
- - -sumdb(x)          <=  let v==double(x) In sum(v) ;
```

The trouble with the pseudo-treeless form is that the abstracted intermediate term has *not* been eliminated. It is merely separated out to make the overall expression appears treeless. Because of this, pseudo-treeless **let** constructs should only be used when deforestation rules cannot cope. In the next few sub-sections, we shall present a safe generalisation of Extended Deforestation which will only introduce **let** constructs when really needed. This safe generalisation will allow deforestation to be applicable to any first-order expressions.

3.4.1. Universal Treelessness

We shall call a new treeless form which uses the pseudo-treeless **let** construct, the **universal treeless** form, and formally define it as follows.

Definition 3.15 - Universal Treeless Form

A term is said to be *universal treeless* if it satisfies the grammar below:

$$ut ::= ut' \mid \textbf{let } v==ut'_1 \textbf{ in } ut'_2$$

$$ut' ::= v \mid c(ut'_1,...,ut'_n) \mid f(v_1^{+}..v_m^{+},ut'_{m+1}^{-}..ut'_n^{-})$$
$$\mid g(v_0^{+},v_1^{+}..v_m^{+},ut'_{m+1}^{-}..ut'_n^{-}) \mid g(ut'_0^{-},v_1^{+}..v_m^{+},ut'_{m+1}^{-}..ut'_n^{-})$$

where $v_1..v_m$ are variables annotated $^{+}$, and $ut'_{m+1}..ut'_n$ are argument terms annotated $^{-}$; and g and f are *universal-treeless* functions.

Correspondingly, a function is said to *universal treeless*, if its definition is made up of *universal treeless* terms.

This new universal treeless, or u-treeless, form is a simple generalisation of e-treeless form with the addition of a **let** construct. To simplify matters, we have restricted this **let** construct to be present only as an outermost construct. This does not involve a loss of generality because inner and nested **let** constructs can always be removed by replacing them with intermediate functions. Also, we shall restrict the use of this **let** construct to be present only in the f-type functions, again, without loss of generality.

With this universal treeless form, we propose a corresponding deforestation theorem as follows:

Theorem 3.16 - Universal Deforestation

Any first-order term, containing only constructors and *universal treeless* functions, can be effectively transformed, by Universal Deforestation Algorithm, to an equivalent *universal treeless term*, without any loss of efficiency.

The transformation algorithm, referred by this new theorem, is a simple generalisation of the Extended Deforestation Algorithm. Apart from the original nine $\mathcal{T}$ rules (in Figure 3.7), we need two more rules (to deal with the **let** constructs and f-type functions containing the **let** constructs). These two rules are presented in Figure 3.14, as rules $\mathcal{T}$ 10 and $\mathcal{T}$ 7d.

Rule $\mathcal{T}$ 10 preserves each outermost **let** construct by transforming just its sub-expressions, while $\mathcal{T}$ 7d is an alternative step of $\mathcal{T}$ 7. This alternative step handles each expression which contains an innermost f-type function call with a **let** construct. We can view each function, with a **let** construct, as a function which is pseudo-treeless. If it appears in the expression as an innermost function call, then it can be unfolded by $\mathcal{T}$ 7d; but we also need the following distributive law:

...; ,

$$\ldots \textbf{let } v==t_1 \textbf{ in } t_2\ldots \Rightarrow \textbf{let } v==t_1 \textbf{ in } \ldots t_2\ldots .$$

to allow the **let** construct to be made outermost. This law is used implicitly in $\mathcal{T}$ 7d and is always allowed because **let** constructs are *not* nested under our language restrictions. It is useful because the presence of outermost **let** construct makes it easier for our rules to identify all f-type functions which are pseudo-treeless (or bad producers).

(7) $\mathcal{T}[\![...f(t_{1..m}^{\oplus}, t_{m+1..n}^{-})...]\!]$

d) if $f(v_1,..,v_n)$=let v==tl_1 in tl_2 **then**

$\Leftarrow$ f_new($vo_{1..k}$, $\mathcal{T}[\![te]\!]_{1..s}$)

where(...$f(t^{\sim}_{1..m}, ve_{m+1..n})...^{\sim}$, $te_{1..s}$, $ve_{1..s}$, $vo_{1..k}$, $vn_{1..k}$) == **gen_ren**[[...$f(t_{1..m}, t_{m+1..n})$...]]

define

f_new($vn_{1..k}$, $ve_{1..s}$)=...$f(t^{\sim}_{1..m}, ve_{m+1..n})...^{\sim}$

unfold on call f ***to give:***

f_new($vn_{1..k}$, $ve_{1..s}$)=let v==$\mathcal{T}[\![tl_1\ [t^{\sim}/v_{1..n}, ve/v_{m+1..n}]]\!]$ in $\mathcal{T}[\![...tl_2\ [t^{\sim}/v_{1..n}, ve/v_{m+1..n}]...^{\sim}]\!]$

(10) $\mathcal{T}[\![$let v==t_1 in $t_2]\!]$ $\Leftarrow$ let v==$\mathcal{T}[\![t_1]\!]$ in $\mathcal{T}[\![t_2]\!]$

Figure 3.14 - Two Additional Rules to Deal with the let construct

3.4.2. Proof of Universal Deforestation Theorem

The proof for Universal Deforestation Theorem follows by extending from the proof of the Extended Deforestation Theorem in Section 3.3.6. Like the original nine rules of Extended Deforestation, each of the two new rules for Universal Deforestation could only produce terms which are u-treeless (by structural induction). In addition, they do not cause a loss in efficiency because each intermediate function defined is always compensated by an unfold step.

Additional Proof Cases for Lemma 3.14.1

Prove: The nesting measure, $\mathcal{N}$, on expressions being transformed is also bounded by rule $\mathcal{T}$ 7d and 10.

Assume: $\mathcal{N}[\![t]\!] \geq \max\{\mathcal{N}[\![t^{\sim}]\!], \mathcal{N}[\![te]\!]_{1..s}\}$ **where** $(t^{\sim}, te_{1..s}, _, _, _)$==**gen_ren**[[t]]

Case for Rule $\mathcal{T}$ 7 ...f $(t_{1\cdot\cdot m}^{\oplus}, t_{m+1..n}^{-})...$:

d) given $f(v_1,..,v_n)$=let v==tl_1 in tl_2

From Lemma on **gen_ren**, we have:

$\mathcal{N}[\![...f(t_{1\cdot\cdot m}, t_{m+1..n})...]\!] \geq \max\{\mathcal{N}[\![...f(t^{\sim}_{1\cdot\cdot m}, ve_{m+1..n})...^{\sim}]\!], \mathcal{N}[\![te]\!]_{1..s}\}$

- Thus : $\forall i \in 1..s$: $\mathcal{N}[\![te_i]\!] \leq \mathcal{N}[\![...f(t_{1\cdot\cdot m}, t_{m+1..n})...]\!]$
- Also, From Definition: $\mathcal{N}[\![f(t^{\sim}_{1\cdot\cdot m}, ve_{m+1..n})]\!]$

$= \max\{1+\max\{\mathcal{N}[\![t^{\sim}]\!]_{1..m}\}, \mathcal{N}[\![ve]\!]_{m+1..n}\}$

$\geq 1+\max\{\mathcal{N}[\![t^{\sim}]\!]_{1..m}\}$

Since f is u-treeless, $\mathcal{N}[\![$let v==tl_1 in $tl_2]\!] \leq 1$

$\geq \mathcal{N}[\![$let v==tl_1 in $tl_2]\!] + \max\{\mathcal{N}[\![t^{\sim}]\!]_{1..m}\}$

$\geq \max\{\mathcal{N}[\![tl_1]\!], \mathcal{N}[\![tl_2]\!]\} + \max\{\mathcal{N}[\![t^{\sim}]\!]_{1..m}\}$

$\geq \max\{\mathcal{N}[\![tl_1]\!] + \max\{\mathcal{N}[\![t]\!]_{1..m}\}, \mathcal{N}[\![tl_2]\!] + \max\{\mathcal{N}[\![t^{\sim}]\!]_{1..m}\}\}$

$\geq \max\{\mathcal{N}[\![tl_1[t^{\sim}/v_{1..n}, ve/v_{m+1..n}]]\!], \mathcal{N}[\![tl_2[t^{\sim}/v_{1..n}, ve/v_{m+1..n}]]\!]\}$

Thus: $\mathcal{N}[\![...tl_2[t^{\sim}/v_{1..n}, ve/v_{m+1..n}]...^{\sim}]\!] \leq \mathcal{N}[\![...f(t_{1..m}, t_{m+1..n})...]\!]$

and $\mathcal{N}[\![tl_1[t^{\sim}/v_{1..n}, ve/v_{m+1..n}]]\!] \leq \mathcal{N}[\![...f(t_{1..m}, t_{m+1..n})...]\!]$

Case for Rule $\mathcal{T}$ 10 let v==t_1 in t_2 :

From Definition: $\mathcal{N}[\![$let v==tl_1 in $tl_2]\!] = \max\{\mathcal{N}[\![tl_1]\!], \mathcal{N}[\![tl_2]\!]\}$

Therefore, $\mathcal{N}[\![tl_1]\!] \leq \mathcal{N}[\![$let v==$tl_1$ in $tl_2]\!]$, $\mathcal{N}[\![tl_2]\!] \leq \mathcal{N}[\![$let v==$tl_1$ in $tl_2]\!]$

Figure 3.15 - Additional Proof of Lemma 3.14.1 to show that $\mathcal{N}$ is bounded by the new Universal Deforestation Rules

Most importantly, the two new rules do not cause the termination property to be lost. This is because they continue to bound the function call nesting measure $\mathcal{N}$, given in Section 3.3.6, when we include the following additional definition.

$$\mathcal{N}[\![\text{let } v==t_1 \text{ in } t_2]\!] \quad = \quad \max\{\mathcal{N}[\![t_1]\!], \mathcal{N}[\![t_2]\!]\}$$

The proof that this nesting measure is bounded by the two new rules is given in Figure 3.15. Furthermore, the restriction on single outer **let** construct helps to ensure that there will never be more than one such construct in our expression. As a result, the actual size of expressions occurring during transformation will continue to be bounded by $\mathcal{N}$. Thus, the proof that $\mathcal{T}$ terminates goes through, as before.

3.4.3. Converting Functions to Universal Treeless Form

The Universal Deforestation Theorem, as presented, is directly formulated for dealing with individual *expressions*. But what is really needed is a re-formulation of deforestation to transform individual *functions* to u-treeless form. We refer to this process of transforming individual function to treeless form as **conversion**, and define it as follows:

Definition 3.17 - Function Conversion

A function, k, which can either be an f- or g-type definition,

$$\begin{array}{lll} \text{- - -}k(v_1,..,v_n) & <= & t \\ \text{or - - -}k(p_1,v_1,..,v_n) & <= & t_1 \\ \vdots & & \\ \text{- - -}k(p_r,v_1,..,v_n) & <= & t_r \quad \text{where } p_i=c_i\ (v_1',..,v_j') \end{array}$$

is considered to be successfully *converted* to a universal treeless function if and only if the application of the deforestation algorithm, $\mathcal{T}$, to each RHS term of the function's definition (as shown below) will terminate and result in universal treeless expression.

$$\begin{array}{lll} \text{- - -}k(v_1,..,v_n) & <= & \mathcal{T}[\![t]\!] \\ \text{or - - -}k(p_1,v_1,..,v_n) & <= & \mathcal{T}[\![t_1]\!] \\ \vdots & & \\ \text{- - -}k(p_r,v_1,..,v_n) & <= & \mathcal{T}[\![t_r]\!] \end{array}$$

With the help of the **let** constructs, we claim that it is always possible to convert *any* first-order function safely to an equivalent u-treeless function. This result can be stated as follows:

Theorem 3.18 - Universal Conversion

All first order functions can be safely *converted* to universal treeless form.

The proof for this Conversion Theorem can be given by showing the following three main results, which together support the validity of the theorem.

- applying conversion, in a *bottom-up manner*, will help ensure that auxiliary functions are always converted (to u-treeless form) before the main function
- *non-recursive* functions, with auxiliary functions which are u-treeless, can always be successfully converted.
- *recursive* functions, with auxiliary functions which are u-treeless, can also be successfully converted.

These three results will be given in the next few sub-sections.

3.4.3.1. Bottom-Up Conversion Strategy

The set of functions of any given programs is normally specified in a defining *hierarchy* with each function making possible use of both its *auxiliary* as well as its *mutually* recursive functions. The auxiliary functions are those which are lower in the defining hierarchy, while the mutually recursive functions are considered to be at the same level. As each expression to be deforested must make use of functions which are u-treeless, we will normally have to apply our conversion to *auxiliary* functions (which are lower in the hierarchy) before the main function and its mutual recursive set. This can be done through a **bottom-up conversion** strategy.

```
bottom_up(f,env) <=
    If visited(f,env) then env
    else
        let mut_set == mutual_recursive(f,env) ∪ {f} in
        let aux_set == ∪ { auxiliary_fn(m) | m ∈ mut_set} in
            all_convert(mut_set,all_bottom_up(aux_set,env));
all_bottom_up(aux_set,env) <=
        if aux_set={} then env
        else let (a, rest)==singleton_split(aux_set) in
            all_bottom_up(rest,bottom_up(a,env));
all_convert(mut_set,env) <=
        If mut_set={} then env
        else let (m, rest)==singleton_split(mut_set) in
            all_convert(rest,convert_and_mark(m,env));
WHERE
    env - data containing function definitions, hierarchy and visited markings
    singleton_split - given a set, return an arbitrary element together with the set with
        that element removed.
    visited - test if a function has been marked as visited
    mutual_recursive - returns a set of mutual recursive functions
    auxiliary_fn - returns a set of auxiliary functions
    convert_and_mark - apply deforestation algorithm to function's definition and
        mark the function as visited
```

Figure 3.16 - Algorithm for Bottom-Up Conversion

The bottom-up conversion strategy would have to be recursively applied because the auxiliary functions may themselves be making use of other auxiliary functions. This recursive application can be specified as an algorithm in Figure 3.16. The given algorithm simultaneously considers each function and its set of mutually recursive functions (if any), in such a way that all their auxiliary functions are dealt with, before the main function and its recursive set. Each function which has been converted is marked as *visited* so that none of the function is ever transformed more than once. If this is not done, some functions may be processed more than once because we normally have a graph rather than tree structure for the defining hierarchy.

With this strategy, we can inductively assume when considering a function for conversion, that its auxiliary functions will have been converted to universal treeless form. This assumption is used in the next two lemmas on function conversions.

3.4.3.2. Conversion for Non-Recursive Functions

Each function of a given program is either *recursive* or *non-recursive*. We propose two lemmas, in this and the next section, to show that both kinds of functions can be converted to universal treeless form.

Assuming the use of the bottom-up conversion strategy, the conversion for each non-recursive function can be stated as follows:

Lemma 3.19 - Conversion for Non-Recursive Functions

> Every non-recursive function, using only auxiliary functions which are u-treeless, can always be safely converted into an equivalent u-treeless form, without any loss of efficiency.

Proof

Let k be a non-recursive function with auxiliary functions, $ka_1..ka_m$ ($m \geq 0$), which are all u-treeless. Each RHS term of this non-recursive function, k, is simply an expression which contains *only* u-treeless functions (i.e. $ka_1..ka_m$). Such terms satisfy the requirement of Universal Deforestation Theorem and can therefore be deforested successfully by its transformation algorithm. Hence, the validity of the above lemma on the conversion of non-recursive functions.

3.4.3.3. Conversion for Recursive Functions

Similarly, the conversion for recursive functions can be stated as follows:

Lemma 3.20 - Conversion for Recursive Functions

> Every set of mutually recursive functions, using only auxiliary functions which are u-treeless, can always be safely converted to an equivalent set of u-treeless functions.

Proof

Let $k_1..k_s$ ($s \geq 1$) be a set of mutually recursive functions whose auxiliary functions, $ka_1..ka_m$ ($m \geq 0$), are all universal treeless. Each RHS term of these mutually recursive functions may now contain a *mix* of both treeless functions (i.e auxiliaries $ka_1..ka_m$) and *possibly* non-treeless functions (i.e recursive $k_1..k_s$).

As seen earlier with rev_flatten example, such a mix of expressions can cause non-termination if the deforestation rules are not applied carefully. However, it is still possible to perform safe transformation on such *mixed* expressions if we can ensure that all the current recursive function calls, $k_1..k_s$, (possibly non-treeless) are never unfolded. This assurance is provided by two additional rules, in Figure 3.17, for handling these possibly non-treeless function calls, denoted with the symbol k. (Note that the k functions can be either f-type or g-type functions.)

Rule 11 merely skips over each outer non-treeless k call, in order to recursively transform only its arguments. Rule 12 is used to handle a context expression, $...k(t_1,...,t_n)...$, of nested g-calls with an innermost k call. This innermost k call should *not* be unfolded. Moreover, as this is one of the situation which may be non-treeless, we must use a **let** construct to abstract out the k-call so as to make the expression appears treeless. We also use an intermediate f-type function, f_new, to contain this **let** construct in order to ensure that the construct is outermost.

These two additional rules are sufficient to convert any recursive function to universal treeless form. A close examination of the expressions resulting from these two rules shows that they are indeed u-treeless. In particular, notice that the parameters of each recursive function, k, are

currently *un-annotated.* This parameter annotation would only be assessed after deforestation has converted the set of recursive functions to u-treeless form. At the very worst, all the parameters could be annotated as $^{\circ}$, and the expression would trivially satisfy the u-treeless form.

(11) $\mathcal{T}[\![k(t_1,..,t_n)]\!] \Leftarrow k(\mathcal{T}[\![t_1]\!],..,\mathcal{T}[\![t_n]\!])$

(12) $\mathcal{T}[\![...k(t_1,..,t_n)...]\!] \Leftarrow$ f_new$(vo_{1..k},\mathcal{T}[\![te]\!]_{1..s})$

where $(...k(t\tilde{}_1,..,t\tilde{}_n)...\tilde{},te_{1..s},ve_{1..s},$
$vo_{1..k},vn_{1..k}) ==$ **gen_ren**$[\![...k(t_1,..,t_n)...]\!]$

define f_new$(vn_{1..k},ve_{1..s})$= let v==$\mathcal{T}[\![k(t\tilde{}_1,..,t\tilde{}_n)]\!]$ in $\mathcal{T}[\![...v...\tilde{}]\!]$

WHERE gen_ren rule for k calls is:

gen_ren$[\![k(t_1,..,t_n)]\!] \Leftarrow (k(t\tilde{}_1,..,t\tilde{}_n),te_lt,ve_lt,vo_lt,vn_lt)$

where $(t\tilde{}_1..t\tilde{}_n,te_lt,ve_lt,vo_lt,vn_lt)$==**gen_list**$[\![t_1..t_n]\!]$

Figure 3.17 - Rules to Deal with Non-Treeless Function Calls

Furthermore, the new deforestation rules can also be shown to terminate. This is because none of the possibly non-treeless function calls are ever unfolded. As a result, we still have a bounded measure, as the proof in Figure 3.18 shows. Thus, the Universal Deforestation Algorithm, in conjunction with these two additional rules (for handling the current recursive functions) can be used to safely convert any recursive function to its u-treeless form, as claimed in the Recursive Function Conversion Lemma.

Additional Proof for Lemma 3.14.1

Prove: The nesting measure, $\mathcal{N}$, on expressions being transformed is bounded by rule $\mathcal{T}$ 11 and 12.

Assume Lemma 3.14.2: $\mathcal{N}[\![t]\!] \geq \max\{\mathcal{N}[\![t\tilde{}]\!],\mathcal{N}[\![te]\!]_{1..s}\}$

where $(t\tilde{},te_{1..s},_,_,_)$==**gen_ren**$[\![t]\!]$

Define: $\mathcal{N}[\![k(t_1,..,t_n)]\!] = 1+\max\{\mathcal{N}[\![t_1]\!],..,\mathcal{N}[\![t_n]\!]\}$

Case for Rule $\mathcal{T}$ 11 - $k(t_1,..,t_n)$:

From the definition: $\mathcal{N}[\![k(t_1,..,t_n)]\!] = 1+\max\{\mathcal{N}[\![t_1]\!],..,\mathcal{N}[\![t_n]\!]\}$

- thus $\forall i \in 1..n \quad \mathcal{N}[\![t_i]\!] \leq \mathcal{N}[\![k(t_1,..,t_n)]\!]$

Case for Rule $\mathcal{T}$ 12 - $...k(t_1,..,t_n)...$:

From Lemma on **gen_ren** for k (with a result similar to constructor terms):

$\mathcal{N}[\![...k(t_1,..,t_n)...]\!] \geq \max\{\mathcal{N}[\![...k(t\tilde{}_1,..,t\tilde{}_n)...\tilde{}]\!],\mathcal{N}[\![te]\!]_{1..s}\}$:

- thus : $\forall i \in 1..s$: $\mathcal{N}[\![te_i]\!] \leq \mathcal{N}[\![...k(t_1,..,t_n)...]\!]$

Also from lemma $\mathcal{N}[\![...k(t_1..t_n)...]\!] \geq \mathcal{N}[\![...k(t\tilde{}_1..t\tilde{}_n)...\tilde{}]\!]$, we get:

- $\mathcal{N}[\![...k(t_1,..,t_n)...]\!] \geq \mathcal{N}[\![...v...\tilde{}]\!]$ (another generalisation)
- $\mathcal{N}[\![...k(t_1,..,t_n)...]\!] \geq \mathcal{N}[\![k(t\tilde{}_1,..,t\tilde{}_n)]\!]$ (an extracted sub-term)

Figure 3.18 - Additional Proof for Lemma 3.14.1 to show that it holds for Rules $\mathcal{T}$ 11 and $\mathcal{T}$ 12

3.4.4. Example of Universal Treeless Conversion

With the Universal Conversion Theorem, any first order function can now be transformed into its u-treeless equivalent. We illustrate this by showing how the bottom-up conversion strategy would apply to the following set of four functions.

```
dec main:list(list(num)) -> list(num);
dec list_double:list(list(num)) -> list(list(num));
- - -main(x)           <= rev_flatten(list_double(x));
- - -list_double([])   <= [];
- - -list_double(a::x) <= double(a)::list_double(x);
- - -double([])        <= [];
- - -double(a::x)      <= 2*a::double(x);
- - -rev_flatten([])   <= [];
- - -rev_flatten(a::x) <= append(rev_flatten(x),a);
```

The above function, main, takes in a list of lists of numbers and uses the function list_double to double every item in its input, before the intermediate output is flattened (and reversed) by rev_flatten. One sequence in which the bottom-up conversion could be applied is: 1) double 2) list_double 3) rev_flatten 4) main. This sequence ensures that all auxiliary functions are converted before their parent functions.

The first two functions, double and list_double, are already in u-treeless form. Applying the deforestation rules will make no changes to them. The third function, rev_flatten, could however be deforested as shown in Figure 3.19. This conversion results in two mutually recursive functions, rev_flatten and rev_flat', with a new **let** construct introduced in rev_flat'.

```
- - -rev_flatten([])     <= T[[ [] ]]                                  !T 2
                        <= []
- - -rev_flatten(a::x)   <= T[[ append(rev_flatten(x),a) ]]            !T 12, define rev_flat'
                        <= rev_flat'(x,a)
define
- - -rev_flat'(x,a)      <= let v==T[[ rev_flatten(x) ]] in T[[ append(v,a) ]]   !T 11
                        <= let v==rev_flatten(T[[ x ]]) ]] in T[[ append(v,a) ]] !T 1
                        <= let v==rev_flatten(x) in T[[ append(v,a) ]]  !T 4
                        <= let v==rev_flatten(x) in append(v,a)
```

Figure 3.19 - Converting Function rev_flatten

There is actually no gain in efficiency[11] arising from the conversion of function rev_flatten. This conversion is however needed to enable the function main to be improved. In particular, the deforestation of main, as shown in Figure 3.20, manages to successfully eliminate an intermediate term. This last conversion results in the following final deforested program.

```
- - -main([])          <= [];
- - -main(a::x)        <= main'(x,a);
- - -main'(x,a)        <= let v==main(x) in main_aux(v,a);
- - -main_aux([],a)    <= double(a);
- - -main_aux(b::v,a)  <= b::main_aux(v,a);
```

[11] In fact, there is a loss in efficiency from the introduction of an intermediate function with a let construct, without being compensated by a corresponding unfold. This always occurs for T12 and is a penalty of pseudo-treelessness. Nevertheless, this loss of efficiency can be undone by a post-processing stage, after deforestation, which unfolds each function with a let construct.

```
---main(x)          <= T [[ rev_flatten(list_double(x)) ]]          ! T 8b, narrow list_double

---main([])         <= T [[ rev_flatten([]) ]]                      ! T 6a, unfold rev_flatten
                    <= T [[ [] ]]                                   ! T 2
                    <= []
---main(a::x)       <= T [[ rev_flatten(double(a)::list_double(x)) ]] ! T 6a,unfold rev_flatten
                    <= T [[ rev_flat'(list_double(x),double(a) ) ]] ! T 7d, define main'
                    <= main'(x,a)
define
---main'(x,a)       <= rev_flat'(list_double(x),double(a) )
                    <= let v==T [[ rev_flatten(list_double(x )) ]]
                         in T [[ append(v,double(a)) ]]             ! T 8a,fold with main
                    <= let v==main(x) in T [[ append(v,double(a)) ]] ! T 8b, define main_aux'
                    <= let v==main(x) in main_aux(v,a)
define
---main_aux(v,a)    <= append(v,double(a))                          ! narrow append
---main_aux([],a)   <= T [[ double(a) ]]                            ! T 4
                    <= double(a)
---main_aux(b::v,a) <= T [[ b::append(v,double(a)) ]]               ! T 2
                    <= T [[ b]] ::T [[ append(v,double(a)) ]]       ! T 1
                    <= b::T [[ append(v,double(a)) ]]               ! T 8a,fold with main_aux
                    <= b::main_aux(v,a)
```

Figure 3.20 - Converting Function main

3.5. Further Improvements to Deforestation

The deforestation transformation, though complete for first-order programs, may at times be less than optimal. Some of these less than desirable situations are:

1) It is sometimes possible for recursive programs to have *some* unnecessary mutual recursive function calls (which can be unfolded away). When this happens, we describe the program as having a *poor* mutual recursion structure. It is undesirable because deforestation will *propagate* the poor structure to newly transformed programs - leading to bigger programs which could have been improved.

2) Certain pattern-matching parameters may be *variable-only* but *non-linear*. This is sometimes caused by non-linearity in one or more of its auxiliary variables, resulting in the pattern-matching parameter being conservatively marked as unsuitable for deforestation. Such parameters should be *linearised*, if possible, to improve deforestability.

3) Some pseudo-treeless expressions, which cannot be deforested by the unfold/fold steps, can often be made into genuine treeless form by using special laws. These laws, if available, would boost the performance of the deforestation transformation.

We examine these issues and suggest some improvements in the next three sub-sections.

3.5.1. Eliminating Unnecessary Mutual Recursive Functions

One source of unnecessary intermediate functions may be found in some mutually recursive programs. This could happen with functions that are *badly written.* For example, we could have re-specified the rev_it function to use an intermediate mutual-recursive function, intm, as follows.

```
- - -rev_it([]+,w+)     <= w ;
- - -rev_it(a::x+,w+)   <= intm(x+,a::w-) ;
- - -intm(x+,w-)        <= rev_it(x+,w+) ;
```

The intermediate function introduced serves no useful purpose. It may have made the second parameter of rev_it *appears* variable-only but this is merely transferring the accumulating condition to the second parameter of intm. Far from being useful, such intermediate functions are undesirable because they result in bigger and inefficient programs.

Another possible problem of intermediate mutual recursive function is that it may actually introduce a form of pseudo-treelessness. An example of this is the following.

```
- - -prog([],z)        <= z ;
- - -prog(a::x,y)      <= prog_mut(x,double(y)) ;      ! a kind of pseudo-treelessness
- - -prog_mut(x,z)     <= prog(x,triple(z)) ;
```

This program has a mutual recursive call, prog_mut(x,double(y)) in the definition for prog, which can be regarded as existing to abstract out the sub-term, double(y). Our deforestation rules cannot presently unfold this prog' call because this call will be considered to be a *possibly* non-treeless function call by rule $\mathcal{T}$ 11. In contrast, if this function call can be unfolded, it would have helped to reveal an expression, triple(double(y)), for genuine deforestation, as shown below.

```
- - -prog([],y)        <= y ;
- - -prog(a::x,y)      <= prog(x,triple(double(y))) ;
```

These two examples illustrate the problem caused by certain unnecessary mutual recursion. We have a problem because our rules will propagate (but not create[12]) these poor recursion structures to newly transformed program. To illustrate this propagation, consider the following function which uses the above rev_it function.

```
- - -rev_db(x,w)       <= rev_it(double(x),w) ;
```

This function can be transformed by deforestation to:

```
- - -rev_db([],w)      <= w ;
- - -rev_db(a::x,w)    <= intm_db(x,2*a::w) ;
- - -intm_db(x,w)      <= rev_db(x,w) ;
```

which has an identical mutual recursion structure to the above rev_it function. We are presently unable to prevent this propagation with our rules (unless expensive retrospective re-transformation with implicit define steps of Section 3.2.3 is resorted to).

---

[12] the $\mathcal{T}$ 7 rule presently introduces a *new* intermediate function definition for every *recursive* expression with an innermost f-type call. This can propagate mutual recursion structures but not create new ones.

All is not lost because we can introduce a *pre-processing* stage (before deforestation) to avoid such poor mutual recursions in the first place. This pre-processing stage would unfold each intermediate mutual recursive function call which satisfies the following conditions:-

1) is an f-type function, and
2) is strictly mutual recursive with no *self-recursive*[13] call in its definition, and
3) must not cause loss of efficiency from its non-linear arguments.

3.5.2. Linearising Pattern-Matching Parameters

The linearity criterion for each *pattern-matching* parameter is determined by the total status of *all* the variables of its patterns. This is occasionally an undesirable situation because certain pattern-matching parameters may be variable-only but still marked as unsuitable for deforestation, whenever one or more of its auxiliary variables are found to be non-linear. An example of this phenomenon is the pattern-matching parameter of function, square, for squaring every element in an input list, as shown below.

```
---square([]ᵉ)      <= [] ;
---square(a::xᵉ)    <= a*a::square(x) ;
```

The parameter of this function is actually variable-only but is non-linear because of its auxiliary variable, a. As a result, the parameter has to be marked as ᵉ. This means that we could not eliminate intermediate terms in expressions like square(double(x)).

Fortunately, we can improve this situation by applying techniques that could make such pattern-matching parameters linear. A simple technique is to introduce an intermediate function to abstract out each sub-expression which has multiple occurrences of the auxiliary variables. Applying this technique to square would give us:

```
---square(a::xᵉ)    <= sq(a)::square(x) ;
---sq(aᵉ)           <= a*a ;
```

This linearising technique could help to improve deforestability. However, it may sometimes be possible for the introduced function call, which is non-recursive (e.g. sq), to be unfolded during deforestation; thus un-doing our earlier attempt at making the parameter linear.

A more permanent and simpler solution is to use the **let** construct to abstract out each non-linear auxiliary variable, as shown below.

```
---square(a::xᵉ)    <= let v==a in v*v::square(x) ;
```

This alternative use of **let** construct (which is not to be eliminated) will help improve deforestability and make the linearised parameter more permanent.

---

[13] If the mutual recursive function, calls *itself* and another mutual recursive call, we do not select it for unfolding because the process will be non-terminating otherwise.

It is important to note that this linearising technique should only be applied under certain circumstances. In particular, we will only consider its use if the pattern-matching parameter in question satisfies the following two conditions.

1) is variable-only, and
2) with the non-linearity caused by auxiliary variables, and not by the main descent variable (e.g. a non-linear auxiliary variable, a, satisfies the conditions for linearisation allowed but not a non-linear descent variable, x, of the above example).

If both these conditions are not satisfied, then we are not likely to obtain any improvements by linearising the pattern-matching parameter.

3.5.3. Laws to Improve Deforestation

The deforestation algorithms presented relied on just the *define*, *unfold* and *fold* transformation steps for eliminating intermediate terms. However, certain expressions may also require *laws* to perform genuine deforestation on their intermediate terms. An example of this is the function, size_tree, for computing the size of a binary tree in terms of function length and flatten_tree, as shown below.

```
- - -size_tree(t)              <= length(flatten_tree(t));
- - -flatten_tree(leaf(a))    <= [a];
- - -flatten_tree(node(x,y))  <= append(flatten_tree(x),flatten_tree(y));
```

If we apply the original universal deforestation rules to these functions, we will only obtain a partially deforested program, which uses the pseudo-treeless **let** construct, as shown below.

```
- - -size_tree(leaf(a))       <= 1+0;
- - -size_tree(node(x,y))     <= size_tree'(x,y);
- - -size_tree'(x,y)          <= let v==flatten(x) In app(v,y);
- - -app([],y)                <= size_tree(y);
- - -app(a::x,y)              <= 1+app(x,y);
- - -flatten(leaf(a))         <= [a];
- - -flatten_tree(node(x,y))  <= flat'(x,y);
- - -flat'(x,y)               <= let v==flatten_tree(x) In app_flat(v,y);
- - -app_flat([],y)           <= flatten_tree(y);
- - -app_flat(a::x,y)         <= a::app_flat(x,y);
```

In contrast, if the distributive law of length, namely length(append(x,y)=length(x)+length(y), is permitted, then we are able to obtain the following fully deforested function, size_tree, which is also much smaller.

```
- - -size_tree(leaf(a))       <= 1+0;
- - -size_tree(node(x,y))     <= size_tree(x)+size_tree(y);
```

It would obviously be desirable to obtain the latter program with genuine deforestation where possible[14]. A preliminary investigation suggests that two major changes may have to be made

[14] assuming that the appropriate laws are made available from either the user or a synthesis sub-system (see Appendix 3).

to Universal Deforestation if these laws are to be incorporated into our transformation algorithm. These two major changes are:

- We may have to withdraw the use of **let** constructs, and substitute it with explicit (function) annotations for the bad producers.
- We may have to allow the unfolding of bad producer functions if laws exist to distribute the terms that are causing the non-treeless characteristics.

These two changes are fairly major. A thorough re-consideration of Universal Deforestation is therefore needed to ensure that the laws could be safely incorporated. We leave this as an area for future research.

3.6. Supercompilation - A Related Work

In [Turchin86], an automatic program transformation technique, called *supercompilation*, for applicative REFAL[15] programs was presented. This technique works by symbolically evaluating a given program in order to observe the possible history of computational states, called *configurations*, which could result from the program. Based on this computational history, Turchin's supercompiler would try to construct/compile a better program.

The symbolic evaluation used, called *driving*, is based on the outside-in evaluation of the lazy semantics. This outside-in driving will essentially *locate* one or more *innermost* function calls, which are currently preventing the *outermost* function call from being unfolded. Each of these *located* function calls is unfolded to produce another symbolic expression (as the next configuration). This process is repeatedly applied in order to build a history or graph of configurations.

The graph of configurations, during driving, is potentially *infinite*. In order to obtain a *finite* set of configurations, supercompilation needs to perform two additional tasks, namely *folding* and *generalisation*. Folding is applied for a configuration that matches a previous configuration, while generalisation is needed to obtain a more general configuration, for a configuration that is *close* but does not match a previous configuration. This more general configuration is then selected for re-supercompilation. It is needed because driving and folding alone may still produce an infinite number of configurations. Thus, the generalisation algorithm, presented in [Turchin88], is a key part of supercompilation. It is needed to ensure termination of the supercompilation process.

The sub-class of transformations which could be performed by Turchin's supercompiler, appears to be similar to what we could achieve with the Universal Deforestation Algorithm. It essentially covers the fusions of functional compositions.

---

[15] the REcursive Function Algorithmic Language

The principal difference between Turchin's supercompilation technique and our Universal Deforestation is that the former uses an *on-the-fly* generalisation technique for finding more general configurations whenever any two configurations are close enough. This approach to generalisation was classified as an **on-line** approach in [JonesN88a] because it performs generalisations at transformation-time. In contrast, our Universal Deforestation algorithm uses a parameter annotation scheme (before transformation) to support an **off-line** generalisation technique.

This off-line (pre-processing) technique is used to determine in advance the various places (i.e. parameter positions) where generalisations must be applied. It is a simpler and more efficient technique than Turchin's on-line technique which requires both a complex algorithm to determine when generalised configurations have to be introduced, and *re-supercompilation* of the generalised configurations whenever they are introduced. In addition, our off-line technique allows the Universal Deforestation algorithm to be extended to higher-order programs (see Chapter 4 later). Turchin's super-compiler is currently available for only first-order REFAL programs.

3.7. Fusion in Set Abstractions

In set abstractions, *intermediate variables* are often used to route the output of one function call to the input of another. A simplified form of program to illustrate this phenomenon is shown below.

$$prog(v_1,..,v_n) \Leftarrow \{ process(w,v_1,..,v_n) \mid w \in generate(v_1,..,v_n),\ filter(w,v_1,..,v_n)\};$$

Here, we have an intermediate variable, w, which is used to link together three functions, generate, filter and process. This is a *generate-test-process* program. The generator function call, $generate(v_1,..,v_n)$, produces a set of *intermediate outputs*. Each of these intermediate outputs, denoted by w, must satisfy the test, $filter(w,v_1,..,v_n)$, before it is converted into an appropriate result by $process(w,v_1,..,v_n)$. Often, the intermediate outputs are recursive data structures. As a result, the opportunities for fusion in set abstractions are similar to those we encounter for fusion in functional compositions (deforestation).

Two kinds of fusions can be identified in the above program. The first kind involves the fusion between a generator, $w \in generate(v_1,..,v_n)$, and its filter, $filter(w,v_1,..,v_n)$. This kind of fusion is frequently known as *filter promotion*, where the filter and generator are fused together in order to produce a better overall generator. We have already seen an example of this in the introductory section of this chapter.

The second kind of fusion is between the fused generator and the outer processor function call, $process(w,v_1,..,v_n)$. This fusion is closer in form to deforestation because the intermediate term, represented by w, can be *truly* eliminated. Compare this with the filter promotion technique, which merely merges the generator and filter together, without actually eliminating its intermediate output term.

In the next two sections, we shall show how the algorithmic transformation technique of deforestation can be formulated analogously for these two forms of fusion.

3.7.1. Filter Promotion

Unlike deforestation where each user-defined function may act as either a *producer* and/or a *consumer* of intermediate outputs, the functions used in set abstractions have clearly distinct roles. In particular, all generators will act as *producers*, while all filters will act as *consumers* of intermediate outputs produced by the generators.

Furthermore, the types of these functions are also different. Assuming the presence of an intermediate output of (recursive) type, Rtype, the typical function types for generators and filters are as follows:

dec generate : $(\alpha_1..\alpha_n)$ -> set(Rtype);

dec filter : (Rtype # $\beta_1..\beta_n$) -> truval;

Each generator must return a set of results, while each filter is a test function which returns a boolean value.

These two classes of functions (with distinct roles) could either be f-type or g-type functions. However, as they play different roles in fusion, we shall further differentiate these functions with capitalised suffix subscripts, $_G$ and $_F$, to distinguish between generator and filter functions respectively.

3.7.1.1. Desirable Filter Form

In the case of filter functions, we are allowed to have the following f-type and g-type functions.

$f_F(v_1,..,v_n)$ <= btf

$g_F(p_1,v_1,..,v_n)$ <= btg_1

:

$g_F(p_n,v_1,..,v_n)$ <= btg_n WHERE $p_i=c_i(v'_1,..,v'_j)$, and the suffix, $_F$, denotes a filter function

The RHS terms of these functions shall be boolean terms. For the purpose of fusion, we would like to differentiate the filter functions into those which are suitable for fusion and those which are not. Those filter functions which are suitable must be good consuming functions. The main criterion which can be used to determine if we have a good consuming function is whether its RHS boolean terms satisfy a certain desirable form, as defined below.

Definition 3.21 - Desirable Filter Form

A filter function is regarded to be of *desirable* form for filter promotion, if each of its RHS terms satisfies the following grammar.

bt::=bt' | $bt_1 \wedge bt_2$ | $bt_1 \vee bt_2$

bt'::=v | true | false | $g_F(v_0{}^+,t_1,..,t_n)$ | $f_F(t_1,..,t_n)$ |$k_F(t_1,..,t_n)$

where $t_1,..,t_n$ are general terms, k_F (either f-type or g-type) is a filter function that does not satisfy the *desirable* filter form and g_F & f_F are desirable filter functions.

Notice that each desirable g-type filter function call, $g_F(v_0{}^+,t_1,..,t_n)$, must only have a variable in its pattern-matching argument. This requirement ensures that the desirable g-type filter functions

are good consumer functions - a condition essential for successful fusion. We shall see later how this desirable expression form can be used in a transformation algorithm for merging filters and generators together. At the moment, there is one assumption which could be taken to help simplify our transformation algorithm.

Our algorithm could be made much simpler if we are able to assume that only g-type functions are considered as good filter functions. In other words, we would like to regard f-type filter functions as being *unsuitable* for fusion. This is a reasonable assumption to make because only g-type functions are the real consumers. The f-type filter functions are either bad filter functions or are intermediate functions which could disappear (before fusion is applied) by direct unfolding. We shall therefore make this assumption in our transformation algorithm.

3.7.1.2. Desirable Generator Form

Similarly, generators are allowed to be either f-type or g-type functions, as shown below.

$$
\begin{array}{lcl}
\text{- - -}f_G(v_1,..,v_n) & <= & stf_1 \\
\text{- - -}g_G(p_1,v_1,..,v_n) & <= & stg_1 \\
 & : & \\
\text{- - -}g_G(p_n,v_1,..,v_n) & <= & stg_n \text{ WHERE } p_i=c_i(v'_1,..,v'_j), \text{ and suffix } _G \text{ denotes a generator function}
\end{array}
$$

But this time, we do not require desirable generators to be just g-type functions. This is because both f-type and g-type functions could be *good producers*. The main factor which determines if a generator is a *good producer* is whether its RHS terms satisfy the following form.

Definition 3.22 - Desirable Generator Form

A function is considered to be a desirable generator, if each of its RHS terms satisfies the following form.

$$
\begin{array}{lll}
st & ::= st' \mid st_1 \cup st_2 \mid st_1 \cap st_2 & \text{! outermost set expressions} \\
st' & ::= st'' \mid \{\} \mid \{ p \mid bt''_1,..,bt''_r \} & \text{! set results} \\
st'' & ::= v \mid f_G(t_1,..,t_n) \mid g_G(t_0,t_1,..,t_n) & \text{! set variable and function calls} \\
bt'' & ::= w \in st'' \mid bt' & \text{! generator or filter} \\
p & ::= v \mid c(p_1,..,p_n) & \text{! variable or constructor term}
\end{array}
$$

where w is an intermediate variable

The above expression form represents a particular canonical form for set expressions. It is also a desirable form for producing intermediate terms that could be consumed by other functions. This is because:

- The set union and intersection operators, placed on the outermost positions, help to normalise the expression. This normalisation enables distributive laws to be applied whenever these operators are present (or unfolded) inside another set abstraction.
- We have restricted the set results to either intermediate results, like $\{ p \mid bt''_1,..,bt''_r \}$ where p is a constructor term, or make direct use of other good generator functions, like $f_G(t_1,..,t_n)$ or $g_G(t_0,t_1,..,t_n)$.

  This restriction ensures that either suitable intermediate outputs (for the consuming filters) or good producers (for further fusion) are provided.

With both these desirable expression forms (for generators and filters), we can now present a transformation algorithm for the filter promotion technique.

3.7.1.3. Transformation Algorithm

The transformation algorithm for fusing filters and generators together (in set abstractions) is more complex than the deforestation algorithm for functional compositions. Apart from the define, unfold and fold steps, we shall also need a number of *lemmas* to help transform set abstractions into a form which is similar to the canonical form of desirable generators.

To simplify our presentation, we shall fuse only *a pair* of one desirable generator and one desirable filter at a time. Selecting one pair of generator and filter at a time corresponds to a form of incremental transformation which have to be applied repeatedly. This must be performed until no more pairs of desirable generators and filters are found.

To help us achieve this, we shall use the following notation for an *arbitrary* selection of a pair of desirable generator and filter from the set abstraction, under transformation, as follows:

$\{... \mid w \in \text{generate}(t_1,..,t_m),\ \text{filter}(w,t_{m+1},..,t_n),...\}$

The selected pair is taken arbitrarily from the set abstraction and need not necessarily be the first two sub-terms (conditions) of the set abstraction. More generally, we shall use the notation, {...|t,...}, to denote the selection of a sub-term, t, arbitrarily from the conditions, t,..., within the set abstraction.

Such a selected pair of sub-expressions would be used to define a new function that is to be fused, as shown below.

$\text{---new_generate}(v_1,..,v_n) <= \{w \mid w \in \text{generate}(v_1,..,v_m),\ \text{filter}(w,v_{m+1},..,v_n)\};$

Except for the intermediate variable, w, we shall generalise and linearise (as shown above) all the other arguments, $t_1..t_m,t_{m+1}..t_n$, of the two function calls, $\text{generate}(t_1,..,t_m)$ and $\text{filter}(w,t_{m+1},..,t_n)$. This total generalisation will help to maximise the chances of successful transformation through appropriate folding. With it, we can perform the following sequence of transformation steps in order to achieve filter promotion.

1) Narrow on the generator call to produce an intermediate output.
2) Apply relevant lemmas to distribute out the unions or intersections of intermediate output, in order to ensure canonical form for our expression.
3) Unfold the filter function call, if possible, and perform further normalisation where applicable.
4) Apply folding, if applicable.
5) If there are more pairs of generators and filters, either define a new function and repeat the same transformation sequence or fold it with a previously defined function (if it matches).
6) If there are no more pairs of generators and filters, then terminate.

(1) $\mathcal{G}\,[\![\{\}]\!] \Leftarrow \{\}$

(2) $\mathcal{G}\,[\![v]\!] \Leftarrow v$

(3) $\mathcal{G}\,[\![\{p\}]\!] \Leftarrow \{p\}$

(4) $\mathcal{G}\,[\![A \cup B]\!] \Leftarrow \mathcal{G}\,[\![A]\!] \cup \mathcal{G}\,[\![B]\!]$

(5) $\mathcal{G}\,[\![A \cap B]\!] \Leftarrow \mathcal{G}\,[\![A]\!] \cap \mathcal{G}\,[\![B]\!]$

(6) $\mathcal{G}\,[\![\{... \mid x \vee y,...\}]\!] \Leftarrow \mathcal{G}\,[\![\{... \mid x,...\}]\!] \cup \mathcal{G}\,[\![\{... \mid y,...\}]\!]$

(7) $\mathcal{G}\,[\![\{... \mid x \wedge y,...\}]\!] \Leftarrow \mathcal{G}\,[\![\{... \mid x,y,...\}]\!]$

(8) $\mathcal{G}\,[\![\{... \mid w \in A \cup B,...\}]\!] \Leftarrow \mathcal{G}\,[\![\{... \mid w \in A,...\}]\!] \cup \mathcal{G}\,[\![\{... \mid w \in B,...\}]\!]$

(9) $\mathcal{G}\,[\![\{... \mid w \in A \cap B,...\}]\!] \Leftarrow \mathcal{G}\,[\![\{... \mid w \in A,...\}]\!] \cap \mathcal{G}\,[\![\{... \mid w \in B,...\}]\!]$

(10) $\mathcal{G}\,[\![\{... \mid \text{true},...\}]\!] \Leftarrow \mathcal{G}\,[\![\{... \mid ...\}]\!]$

(11) $\mathcal{G}\,[\![\{... \mid \text{false},...\}]\!] \Leftarrow \{\}$

(12) $\mathcal{G}\,[\![\{w \mid w \in t\}]\!] \Leftarrow t$

(13) $\mathcal{G}\,[\![\{... \mid w \in \{\},...\}]\!] \Leftarrow \{\}$

(14) $\mathcal{G}\,[\![\{... \mid w \in \{p \mid b_1,..,b_m\},...\}]\!] \Leftarrow \mathcal{G}\,[\![\{...[p/w] \mid ... [p/w], b_1,...,b_m\}]\!]$

(15) $\mathcal{G}\,[\![\{... \mid g_F(c_i(p_1,..,p_j)^+,t_1,..,t_n),...\}]\!]$

a) If $t_1..t_n$ are variables/constants, then (unfold)

$\Leftarrow \mathcal{G}\,[\![\{...\mid bgt_i[p/v'_{1..j},t/v_{1..n}],...\}]\!]$

b) otherwise (define & unfold) $\Leftarrow f_{G_new}(t_1..t_n,v_{n+1}..v_z)$

define

$f_{G_new}(v_1..v_n,v_{n+1}..v_z)=\{... \mid g_F(c_i(p_1..p_j)^+,v_1..v_n),...\}$

unfold the generator

$f_{G_new}(v_1..v_n,v_{n+1}..v_z)=\mathcal{G}\,[\![\{...\mid bgt_i[p/v'_{1..j},v/v_{1..n}],...\}]\!]$

where $v_{n+1}..v_z$==vars [{... |...}]

(16) $\mathcal{G}\,[\![\{... \mid w \in f_G(t_1,..,t_m),g_F(w^+,t_{m+1},..,t_n),...\}]\!]$

a) If $f_{G_old}(v_1,..,v_n)=\{ w \mid w \in f_G(v_1,..,v_m),\ g_F(w,v_{m+1},..,v_n)\} \in$ def_set, **then** (fold)

$\Leftarrow \mathcal{G}\,[\![\{...\mid w \in f_{G_old}(t_1..t_m,t_{m+1}..t_n),...\}]\!]$

b) otherwise (define & unfold) $\Leftarrow \mathcal{G}\,[\![\{...\mid w \in f_{G_new}(t_1..t_m,t_{m+1}..t_n),...\}]\!]$

define and add to def_set

$f_{G_new}(v_1,..,v_n)=\{ w \mid w \in f_G(v_1,..,v_m),\ g_F(w,v_{m+1},..,v_n)\}$

unfold the generator

$f_{G_new}(v_1,..,v_n)=\mathcal{G}\,[\![\{ w \mid w \in stf,\ g_F(w,v_{m+1},..,v_n)\}]\!]$

(17) $\mathcal{G}\,[\![\{... \mid w \in g_G(t_0,t_1,..,t_m),g_F(w^+,t_{m+1},..,t_n),...\}]\!]$

a) If $g_{G_old}(v_0,v_1,..,v_n)=\{ w \mid w \in g_G(v_0,v_1,..,v_m),\ g_F(w,v_{m+1},..,v_n)\} \in$ def_set, **then** (fold)

$\Leftarrow \mathcal{G}\,[\![\{...\mid w \in g_{G_old}(t_0,t_1..t_m,t_{m+1}..t_n),...\}]\!]$

b) otherwise (define & unfold) $\Leftarrow \mathcal{G}\,[\![\{...\mid w \in g_{G_new}(t_0,t_1..t_m,t_{m+1}..t_n),...\}]\!]$

define and add to def_set

$g_{G_new}(v_0,v_1,..,v_n)=\{ w \mid w \in g_G(v_0,v_1,..,v_m),\ g_F(w,v_{m+1},..,v_n)\}$

narrow on the generator

$g_{G_new}(p_1,v_1,..,v_n)=\mathcal{G}\,[\![\{ w \mid w \in stg_1,\ g_F(w,v_{m+1},..,v_n)\}]\!]$

:

$g_{G_new}(p_r,v_1,..,v_n)=\mathcal{G}\,[\![\{ w \mid w \in stg_r,\ g_F(w,v_{m+1},..,v_n)\}]\!]$

(18) $\mathcal{G}\,[\![\{...|...\}]\!] \Leftarrow \{...|...\}$ If none of the above rules applies anymore

Figure 3.21 - $\mathcal{G}$ Rules for Filter Promotion

Such an iterative sequence of transformation steps can be performed by sixteen rules in Figure 3.21. These rules are ordered in such a way that, should any two rules be applicable at

the same time, the lower numbered rules are selected in preference.

Rules 1 to 5 are used to handle outer constructs which are already in the desirable expression form. These are skipped over to transform just their sub-expressions (where applicable). Rules 6 to 9 apply distributive laws of both boolean and set operators (in relation to the set abstraction construct) in order to normalise the expression under transformation. Rules 10 to 13 help to identify opportunities for simplification in the set abstractions. This can happen whenever part of the expression is reduced to either true or false ($w \in \{\}$ is also equal to false) or to a set by itself ($\{w \mid w \in t\}$ is equivalent to t).

Also, Rule 14 is used to collapse an inner set abstraction (an intermediate result from unfolding a generator) by substituting the variable, w, with the constructor term, p (from the inner set abstraction). The rest of the conditions, $b_1..b_m$ (from the inner set abstraction), are also appended to the main set abstraction. Rule 15 identifies a filter function call with an intermediate term that can be consumed by unfolding. Either a direct or indirect unfold takes place here.

The main rules for discovering pairs of desirable generators (f- or g-type) and filters (g-type) are Rules 16 and 17. Rule 16 deals with an f-type generator, while Rule 17 deals with a g-type generator. In both cases, we can apply either a *define & unfold* step or a *fold* step. In the case of the define & unfold step, all the arguments (except the intermediate variable) of the two calls are generalised before we introduce a new function definition. After that, the generator function call is unfolded before the original expression and the newly defined function are subjected to further transformations. Also, the fold step works in tandem with the define & unfold step for folding pairs of generators and filters that re-occur.

Lastly, Rule 18 is applied whenever none of the earlier rules are applicable. This could happen when *all* pairs of desirable generators and filters have been selected for fusion.

3.7.1.4. An Example

We shall now illustrate the application of transformation algorithm, $\mathcal{G}$, through a simple example.

Consider the function, all_length, for finding the length of all sub-sequences (of an input list) which are valid paths of a given graph. This function can be defined using a set abstraction with a generator, $w \in$ sub(x), and a filter, path(w), as shown below. The two functions, sub(x) and path(w), are of the desirable forms for generators and filters respectively.

```
- - -all_length(x)      <=   {length(w) | w ∈ sub(x), path(w)};
- - -sub([])           <=   {[]};
- - -sub(a::x)         <=   sub(x) ∪ {a::w | w ∈ sub(x)};
- - -path([])          <=   true;
- - -path(a::x)        <=   path'(x,a);
- - -path'([],a)       <=   true;
- - -path'(b::x,a)     <=   arc(a,b) ∧ path'(x,a);
```

We also have a processor function call, length(w), but this function call will be handled by a later

transformation technique in Section 3.7.2.

Our present concern is the application of transformation algorithm, $\mathcal{G}$, to the RHS of function all_length. This results in a sequence of transformation steps, as shown in Figure 3.22, causing the original generator and filter to be successfully fused together.

- - -all_length(x)	<=	$\mathcal{G}$ [[{ length(w) | w ∈ sub(x), path(w)}]]	! $\mathcal{G}$ 17b,define sub_path
	<=	$\mathcal{G}$ [[{ length(w) | w ∈ sub_path(x)}]]	! $\mathcal{G}$ 18,nothing left
	<=	{ length(w) | w ∈ sub_path(x)}	
define			
- - -sub_path(x)	<=	{ w | w ∈ sub(x), path(w)}	! narrow sub
- - -sub_path([])	<=	$\mathcal{G}$ [[{ w | w ∈ { []}, path(w)}]]	! $\mathcal{G}$ 14,subs w
	<=	$\mathcal{G}$ [[{ [] | path([])}]]	! $\mathcal{G}$ 15,unfold path([])
	<=	$\mathcal{G}$ [[{ [] | true}]]	! $\mathcal{G}$ 6,simplify
	<=	$\mathcal{G}$ [[{ []}]]	! $\mathcal{G}$ 3,terminate
	<=	{ []}	
- - -sub_path(a::x)	<=	$\mathcal{G}$ [[{ w | w ∈ sub(x) ∪ {a::w | w ∈ sub(x)}, path(w)}]]	! $\mathcal{G}$ 10,distribute ∪
	<=	$\mathcal{G}$ [[{ w | w ∈ sub(x), path(w)}]]	! $\mathcal{G}$ 17a,fold
		∪ $\mathcal{G}$ [[{ w | w ∈ {a::w | w ∈ sub(x)}, path(w)}]]	! $\mathcal{G}$ 14,subs w
	<=	$\mathcal{G}$ [[{ w | w ∈ sub_path(x)}]]	! $\mathcal{G}$ 12,simplify
		∪ $\mathcal{G}$ [[{ a::w | w ∈ sub(x), path(a::w)}]]	! $\mathcal{G}$ 15,unfold path
	<=	sub_path(x) ∪ $\mathcal{G}$ [[{ a::w | w ∈ sub(x), path'(w,a)}]]	! $\mathcal{G}$ 17b,define sub_path'
	<=	sub_path(x) ∪ $\mathcal{G}$ [[{ a::w | w ∈ sub_path'(x,a)}]]	! $\mathcal{G}$ 18,no more
	<=	sub_path(x) ∪ { a::w | w ∈ sub_path'(x,a)}	
define			
- - -sub_path'(x,a)	<=	{ w | w ∈ sub(x), path'(w,a)}	! narrow sub
- - -sub_path'([],a)	<=	$\mathcal{G}$ [[{ w | w ∈ { []}, path'(w,a)}]]	! $\mathcal{G}$ 14,subs w
	<=	$\mathcal{G}$ [[{ [] | path'([],a)}]]	! $\mathcal{G}$ 15a,unfold path'
	<=	$\mathcal{G}$ [[{ [] | true}]]	! $\mathcal{G}$ 10,simplify
	<=	$\mathcal{G}$ [[{ []}]]	! $\mathcal{G}$ 3,terminate
	<=	{ []}	
- - -sub_path'(b::x,a)	<=	$\mathcal{G}$ [[{ w | w ∈ sub(x)	
		∪ {b::w | w ∈ sub(x)}, path'(w,a)}]]	! $\mathcal{G}$ 8,distribute ∪
	<=	$\mathcal{G}$ [[{ w | w ∈ sub(x), path'(w,a)}]]	! $\mathcal{G}$ 17a,fold
		∪ $\mathcal{G}$ [[{ w | w ∈ {b::w | w ∈ sub(x)}, path'(w,a)}]]	! $\mathcal{G}$ 14,subs w
	<=	$\mathcal{G}$ [[{ w | w ∈ sub_path'(x,a)}]]	! $\mathcal{G}$ 12,simplify
		∪ $\mathcal{G}$ [[{ b::w | w ∈ sub(x), path'(b::w,a)}]]	! $\mathcal{G}$ 15,unfold path'
	<=	sub_path'(x,a)	
		∪ $\mathcal{G}$ [[{ b::w | w ∈ sub(x), arc(a,b),path'(w,b)}]]	! $\mathcal{G}$ 17a,fold
	<=	sub_path'(x,a)	
		∪ $\mathcal{G}$ [[{ b::w | w ∈ sub_path'(x,b), arc(a,b)}]]	! $\mathcal{G}$ 18,no more
	<=	sub_path'(x,a) ∪ { b::w | w ∈ sub_path'(x,b), arc(a,b)}	

Figure 3.22 - An Example of Filter Promotion with the $\mathcal{G}$ Rules

The end result is a new pair of efficient generators, sub_path and sub_path'.

```
- - -all_length(x)       <=   {length(w) | w ∈ sub_path(x)};
- - -sub_path([])        <=   {[]};
- - -sub_path(a::x)      <=   sub_path(x) ∪ { a::w | w ∈ sub_path'(x,a)};
- - -sub_path'([],a)     <=   {[]};
- - -sub_path'(b::x,a)   <=   sub_path'(x,a) ∪ { b::w | w ∈ sub_path'(x,b), arc(a,b)};
```

This application of fusion has successfully promoted part of the filter, arc(a,b), into the new generator function, sub_path'. Such a promotion improves the above program by helping to cut off sub-spaces of infeasible solutions from its search space.

3.7.1.5. Undesirable Expression Forms

Our transformation algorithm for filter promotion requires both the generators and filters to be in their respective desirable forms before they can be selected for fusion. However, it may (at times) be possible for functions to be defined in a form which is presently *unsuitable*, but could be subsequently converted or transformed to the desirable form for fusion. In such circumstances, it would be advantageous to apply these conversions before the filter promotion transformation. Doing this would help to improve the chances of successful fusion.

In the next two sub-sections, we shall investigate two possible unsuitable forms and suggest appropriate techniques which could transform them into desirable forms.

3.7.1.5.1. Undesirable Generators

For ease of specification, some generator functions may be written in absolute set abstractions, rather than relative set abstractions, where functions are required to be executed backwards using the unification mechanism. Two examples are the functions front and sub_list, as shown below, for obtaining all initial sub-lists and arbitrary sub-lists, respectively, from their input lists.

```
- - -front(x)       <=   { w | w<>_=x };
- - -sub_list(x)    <=   { w | _<>w<>_=x };
```

The above definitions are more concise in form but they are also unsuitable for the earlier transformation algorithm. In their present forms, our technique will simply mark these two functions as undesirable generators.

However, we can transform these two functions to the desirable generator form. This is achieved by removing, where possible, all the unification (or backwards execution) features. For this very purpose, we can turn to a transformation tactic, called the *removal-of-unification* tactic, which was recently formulated by Helen Pull. A systematic description of this tactic appears in [Pull90], and we shall briefly illustrate the essence of this tactic with a simple example.

Pull's removal-of-unification tactic operates in two phases. The first phase is made up of a series of transformations consisting of narrowing, unfolding, simplification and folding steps, which would work together to transform expressions with the unification features to equivalent expressions without them. This transformation phase often results in a set of equations which

may have overlapping LHS. The second phase is meant to resolve the possible overlaps of LHS. This is done by applying appropriate instantiations and union of set equations.

To illustrate these two phases, let us consider this transformation for the function, front, given earlier. The first transformation phase would apply the following sequence of steps in order to remove the unification features.

```
- - -front(x)          <=   { w | w<>_=x };          ! narrow <>
- - -front(x)          <=   { [] | _=x };            ! substitute _=x
                       <=   { [] };
- - -front(x)          <=   { a::w | a::w<>_=x };    ! unify a::w<>_ with x
- - -front(a::x')      <=   { a::w | a=a,w<>_=x' };  ! fold back with front
                       <=   { a::w | w ∈ front(x') };
```

This sequence essentially consists of a narrowing step (on <>), followed by simplification steps (e.g. a=a -> true) before a fold operation. It results in two equations with an *overlapping* LHS, as shown below.

```
- - -front(x)          <=   { [] };
- - -front(a::x')      <=   { a::w | w ∈ front(x') };
```

Narrowing of a function call introduces or-parallelism whenever logical variables (as opposed to bound variables on the LHS of equations) of absolute set abstractions are instantiated. This or-parallelism semantics implies that a set result from the various branches of narrowing must now be returned. Equations could therefore be allowed to be overlapping because we must take the set union of the results from each of the equations during execution. However, this overlap of equations (and the need to take a union of the results) is not very satisfactory from the standpoint of functional programs.

Because of this, a second phase of symbolic manipulation was devised by Pull to get rid of overlapping equations. In particular, notice that the LHS of the first equation is more general than the LHS of the second equation. To resolve this overlap, an *instantiation* of the first equation using list's two cases [] and a::x' is required, giving:

```
- - -front([])         <=   { [] };
- - -front(a::x')      <=   { [] };
- - -front(a::x')      <=   { a::w | w ∈ front(x') };
```

This instantiation results in three equations. Two of them (the second and third equations) have a *common* LHS. These two equations can be combined by uniting the two RHS with a set union, as shown in the second equation below.

```
- - -front([])         <=   { [] };
- - -front(a::x')      <=   { [] } ∪ { a::w | w ∈ front(x') };
```

Once this is done, the final equations will no longer be overlapping. Furthermore, they also satisfy the desirable generator form and are without the unification features.

3.7.1.5.2. Undesirable Filters

Like generators, filters may also be in the wrong form for fusion. Consider the following generate-and-test program to find all sub-sequences (from the input list) whose lengths are less

than 8.

```
- - -short_sub(x)      <=   { w | w ∈ sub(x), length(w)<8};
- - -sub([])          <=   { []} ;
- - -sub(a::x)        <=   sub(x) ∪ { a::w | w ∈ sub(x)};
- - -length([])       <=   0;
- - -length(a::x)     <=   1+length(x);
```

The filter of this program is actually an expression, length(w)<8. The outermost function of this filter expression is a primitive operator, <, that is certainly not the desired pattern-matching function needed for fusion. However, there is another function call length(w) within this filter expression, that is a proper function call for consuming the intermediate output, w, of the generator.

If this function, length, is tail-recursive or has an associated function which is tail-recursive, we will be able to convert the whole expression into a desirable filter function. As it turns out, there exists an equivalent tail recursive version of this function that is obtainable through the conversion-to-iteration transformation tactic. This tail-recursive function, length_it, is defined to be related to the original length function, as follows.

```
- - -length(x)          <=   length_it(x,0);
- - -length_it([],m)    <=   m;
- - -length_it(a::x,m)  <=   length(x,m+1);
```

Using this tail-recursive function, we can define a new filter function, as shown below,

```
- - -fil_fn(x,m)       <=   length_it(x,m)<8;
```

which can be directly transformed (by an unfold followed by a fold step) to:

```
- - -fil_fn([],m)      <=   m<8;
- - -fil_fn(a::x,m)    <=   fil_fn(x,m+1);
```

Such a filter function is of the desirable form that we need. The steps used to obtain it can be automated because the conversion-to-iteration tactic and the transformation to a tail-recursive filter function can be mechanised. With the new filter function, we can replace the filter expression, length(w)<8, in the earlier generate-and-test function, as shown below.

```
- - -short_sub(x)      <=   { w | w ∈ sub(x), fil_fn(w,0)};
```

With this replacement, we can apply the $\mathcal{G}$ rules to the function, short_sub, to obtain the following fused program.

```
- - -short_sub(x)      <=   n_sub(x,0);
- - -n_sub([],m)       <=   { [] | m<8};
- - -n_sub(a::x,m)     <=   n_sub(x,m) ∪ {a::p | p ∈ n_sub(x,m+1)};
```

The main improvement to this fused definition is a merged generator which does not need to re-compute the length of each generated sub-sequence. This is possible because the length of the partially evaluated sub-sequence is kept in an accumulating parameter, m, of function, n_sub.

However, the promoted filter, m<8, has ended up in the base-case equation rather than the recursive equation of the generator function, n_sub. As a result, there is actually no cut-off in the search-space of the transformed program. This example shows that not all fusions between

generators and filters necessarily produce generators which cut off sub-spaces of infeasible solutions.

Nevertheless, those filters that have ended up in the base-case equations could sometimes be promoted into the recursive portion of generators by another transformation technique. This other technique uses invariants (rather than fusion) for its filter promotion and is presented in Chapter 6. Our present fusion technique is still useful because it can help to prepare generate-and-test programs (by transforming them to recursive functions with base-case filters) for this other form of filter promotion.

3.7.1.6. An Extension (Patterns Instead of Variables)

In this section, we will consider a simple extension to widen the class of expressions which could be handled by the transformation algorithm, $\mathcal{G}$.

Often, a constructor pattern rather than a simple intermediate variable is used to link separate program units together in the set abstractions. An example of this is the following program, taken from the paragraph problem in [Bird86], for generating all valid permutations of paragraphs (lists of lists of words) from a given text of words (a list of words).

```
dec paragraph,lines: list(word) -> set(list(list(word)));
- - -paragraph(x)      <=   {w| w ∈ lines(x),nonil(w)};
- - -lines([])         <=   {[]::w | w ∈ lines([])} ∪ {[]};
- - -lines(a::x)       <=   {[]::w | w ∈ lines(a::x)} ∪ { (a::b)::w | b::w ∈ lines(x)};
- - -nonil([])         <=   true;
- - -nonil(a::x)       <=   a≠[] ∧ nonil(x);
```

The generator function, lines, produces all *possible* paragraphs, while the filter function, nonil, rules out all the *invalid* paragraphs that contain one or more empty lines. This program is not only inefficient, but it is actually non-terminating because the function, lines, returns an infinite set. Also, there is a constructor pattern, b::w with logical variables b and w, in the second equation of function lines. This constructor pattern is used for holding intermediate results of the recursive generator but it violates one of the earlier grammar rules for the desirable generator form.

To extend our transformation technique to cope with the above example, we must first permit a more general grammar rule for the desirable generator form of Definition 3.22. In particular, we must change the grammar rule bt" to allow pattern term p (instead of just a simple variable) to be used with generators as $p \in st'$. This change is reflected in the following two modified grammar rules.

bt"::= $p \in st'$ | bt'

p ::= v | $c(p_1,...,p_n)$ | $(p_1,...,p_n)$

The pattern term, p, could either be a simple variable, v, a constructor pattern, $c(p_1,...,p_n)$, or a tuple pattern, $(p_1,...,p_n)$. With this more general form, we must also modify three $\mathcal{G}$ rules, namely Rules 14, 16 and 17 of Figure 3.21, to handle this wider class of expressions.

The modified rules are given in Figure 3.23. Rule 14 collapses an inner set abstraction and must now apply variable substitutions that are obtained by matching pattern and term. These variable substitutions are obtained from pattern matching by a meta-function called *match*, whose definition is also given in Figure 3.23. Also, the other two rules, Rule 16 and 17, to select appropriate pairs of generators and filters (for fusion) must be modified to handle the more complex pattern link. To help us specify these two rules, we have employed a new context notation, ..w.., for describing an intermediate variable, w, within a pattern term, as shown below.

..w.. ::= w | c(..,...w..,..) | (..,...w..,...)

This more general pattern link is used to replace the simple variable link in the previous $\mathcal{G}$ rules.

(14) $\mathcal{G}$ [[{... | pattern ∈ { term |b_1,...,b_m},...}]]

⇐ $\mathcal{G}$ [[{...[t/$w_{1..n}$] | ... [t/$w_{1..n}$], b_1,...,b_m}]]
where t/$w_{1..n}$ ==*match* [term,pattern]

(16) $\mathcal{G}$ [[{... | ..w.. ∈ $f_G(t_1,...,t_m)$,$g_F(w^+,t_{m+1},...,t_n)$,...}]]

a) If f_G_old(v_1,...,v_n)={ ..w.. | ..w.. ∈ $f_G(v_1,...,v_m)$, $g_F(w,v_{m+1},...,v_n)$} ∈ def_set, **then** (fold)

⇐ $\mathcal{G}$ [[{...| ..w.. ∈ f_G_old(t_1..t_m,t_{m+1}..t_n),...]]

b) **otherwise** (define & unfold) ⇐ $\mathcal{G}$ [[{...| ..w.. ∈ f_G_new(t_1..t_m,t_{m+1}..t_n),...]]

define and add to def_set

f_G_new(v_1,...,v_n)={ ..w.. | ..w.. ∈ $f_G(v_1,...,v_m)$, $g_F(w,v_{m+1},...,v_n)$}

unfold the generator

f_G_new(v_1,...,v_n)=$\mathcal{G}$ [[{ ..w.. | ..w.. ∈ stf, $g_F(w,v_{m+1},...,v_n)$}]]

(17) $\mathcal{G}$ [[{... | ..w.. ∈ $g_G(t_0,t_1,...,t_m)$,$g_F(w^+,t_{m+1},...,t_n)$,...}]]

a) If g_G_old(v_0,v_1,...,v_n)={ ..w.. | ..w.. ∈ $g_G(v_0,v_1,...,v_m)$, $g_F(w,v_{m+1},...,v_n)$} ∈ def_set, **then** (fold)

⇐ $\mathcal{G}$ [[{...| ..w..∈ g_G_old(t_0,t_1..t_m,t_{m+1}..t_n),...]]

b) **otherwise** (define & unfold) ⇐ $\mathcal{G}$ [[{...| ..w.. ∈ g_G_new(t_0,t_1..t_m,t_{m+1}..t_n),...]]

define and add to def_set

g_G_new(v_0,v_1,...,v_n)={ ..w.. | ..w.. ∈ $g_G(v_0,v_1,...,v_m)$, $g_F(w,v_{m+1},...,v_n)$}

narrow on the generator

g_G_new(p_1,v_1,...,v_n)=$\mathcal{G}$ [[{ ..w.. | ..w.. ∈ stg_1, $g_F(w,v_{m+1},...,v_n)$}]]

:

g_G_new(p_r,v_1,...,v_n)=$\mathcal{G}$ [[{ ..w..| ..w.. ∈ stg_r, $g_F(w,v_{m+1},...,v_n)$}]]

WHERE

match [t,v] ⇐ [t/v]

match [c(t_1,...,t_n),c(p_1,...,p_n)] ⇐ *match* [t_1/p_1] <>...<>*match* [t_n/p_n]

match [(t_1,...,t_n),(p_1,...,p_n)] ⇐ *match* [t_1/p_1] <>...<>*match* [t_n/p_n]

Figure 3.23 - More General $\mathcal{G}$ rules which accept Complex Pattern Link

These modified $\mathcal{G}$ rules can be applied to the earlier non-terminating paragraph program in order to obtain the following more efficient (and terminating) program.

```
- - -paragraph([])      <=  {[]};
- - -paragraph(a::x)    <=  {(a::b)::w | b::w ∈ paragraph'(x)};
- - -paragraph'([])     <=  {[]::[]};
- - -paragraph'(a::x)   <=  {[]::w| w ∈ paragraph(x)} ∪ { (a::b)::w | b::w ∈ paragraph'(x)};
```

3.7.2. Fusions between Generators and Processors

After the successful fusion of generators and filters, we can proceed to combine the resulting generator with its outer processor function (where applicable). Typically, such a pair of functions would be of the form:

$$---new_gen(v_1,..,v_n) \Leftarrow \{ process(w,v_{m+1},..,v_n) \mid w \in generate(v_1,..,v_m)\};$$

where the function call, $process(w,v_{m+1},..,v_n)$, is a consumer of intermediate output, w, from the generator, $w \in generate(v_1,..,v_m)$.

However, there is one important pre-condition which must be satisfied before we can consider any fusion between a generator and its processor function. This pre-condition (not needed in the earlier fusion between generators and filters) requires that each intermediate variable should only be present in its generator (as output) and its processor (as input) and nowhere else.

Should the intermediate variable be present in other parts of the set abstraction, complete fusion between the generator and processor functions could not take place. This is because we will not be able to completely eliminate the need for the intermediate output (and thus the generator) by such fusion.

This is why we must always apply filter promotion technique (to reduce the number of references to the intermediate variable) before this form of fusion.

3.7.2.1. Desirable Processor Form

In a similar manner to filters of the previous technique, we must also identify a desirable form for the processor functions which could act as good consumers of intermediate outputs. Each of these processor functions will typically be of the type:

dec process : (Rtype # $\delta_1..\delta_n$) -> Ptype;

which takes an intermediate output of the type, Rtype, together with the other arguments, $\delta_1..\delta_n$, in order to produce a new result of the type, Ptype.

Like the desirable filters, we could also simplify the presentation of our transformation algorithm by restricting the desirable processors to just g-type functions. The f-type processor functions are assumed to be either bad consumers or capable of disappearing by direct unfolding.

With this assumption, we shall consider only g-type functions as our desirable processor functions, denoted with the suffix p, as shown below.

$$---g_p(p_1,v_1,..,v_n) \Leftarrow t_1$$
$$\vdots$$
$$---g_p(p_r,v_1,..,v_n) \Leftarrow t_r \text{ WHERE } p_i=c_i(v'_1,..,v'_j);$$ the suffix p stands for processor function

The RHS terms, $t_1,..,t_r$, of these processor functions will be general (first-order) terms. The only requirement on them is that the desirable g-type function calls must only have variables in their pattern-matching arguments, as defined below.

Definition 3.23 - Desirable Processor Form

A processor function is considered to be of *desirable* form for fusion, if each of its RHS terms satisfies the following grammar.

$$t \;:=\; v \mid c(t_1,..,t_m) \mid k_P(t_1,..,t_n) \mid g_P(v_0^+,t_1,..,t_n)$$

where k_P (either f-type of g-type) is a processor function that does not satisfy the *desirable* processor form.

As for the generator functions, the desirable form remains identical to Definition 3.22 (given in Section 3.7.1.2).

3.7.2.2. Transformation Algorithm

With these desirable expression forms, we can formulate a transformation algorithm for the fusion of desirable generators and processors. However, this transformation algorithm need not be completely new because most of its rules are essentially similar to the previous $\mathcal{G}$ rules for filter promotion. In fact, all the previous rules, except $\mathcal{G}$ 14, 16 and 17 (for dealing with the filter functions), could be re-used for the new fusion algorithm.

(19) $\mathcal{G}\ [\![\{...g_P(c_i(p_1,..,p_j)^+,t_1,..,t_n)... \mid ...\}]\!]$

a) if $t_1..t_n$ **are variables/constants, then** (unfold)

$\Leftarrow \quad \mathcal{G}\ [\![\{...gt_i[p/v'_{1..j},t/v_{1..n}]...\mid ...\}]\!]$

b) **otherwise** (define & unfold) $\Leftarrow \quad f_{G_new}(t_1..t_n,v_{n+1}..v_z)$

define

$f_{G_new}(v_1..v_n,v_{n+1}..v_z)=\{...g_P(c_i(p_1..p_j)^+,v_1..v_n)... \mid ...\}$

unfold the processor

$f_{G_new}(v_1..v_n,v_{n+1}..v_z)=\mathcal{G}\ [\![\{...gt_i[p/v'_{1..j},v/v_{1..n}]...\mid ...\}]\!]$

where $v_{n+1}..v_z$==vars [{... ...|...}]

(20) $\mathcal{G}\ [\![\{...g_P(w^+,t_{n+1},..,t_m)... \mid w \in f_G(t_1,..,t_m),...\}]\!]$! w *must not be present elsewhere* !

a) if $f_{G_old}(v_1,..,v_n)=\{\ g_P(w^+,v_{n+1},..,v_m) \mid w \in f_G(v_1,..,v_m)\} \in$ def_set, **then** (fold)

$\Leftarrow \quad \mathcal{G}\ [\![\{...W...\mid W \in f_{G_old}(t_1..t_m,t_{m+1}..t_n),...]\!]$

b) **otherwise** (define & unfold) $\Leftarrow \quad \mathcal{G}\ [\![\{...W...\mid W \in f_{G_new}(t_1..t_m,t_{m+1}..t_n),...]\!]$

define and add to def_set

$f_{G_new}(v_1,..,v_n)=\{\ g_C(w^+,v_{n+1},..,v_m) \mid w \in f_G(v_1,..,v_m)\}$

unfold the generator

$f_{G_new}(v_1,..,v_n)=\mathcal{G}\ [\![\ \{\ g_C(w^+,v_{n+1},..,v_m) \mid w \in stf\}\ ]\!]$

(21) $\mathcal{G}\ [\![\{...g_P(w^+,t_{n+1},..,t_m)...\mid w \in g_G(t_0,t_1,..,t_m),...\}]\!]$! w *must not be present elsewhere* !

a) if $g_{G_old}(v_0,v_1,..,v_n)=\{\ g_P(w^+,v_{n+1},..,v_m) \mid w \in g_G(v_0,v_1,..,v_m)\} \in$ def_set, **then** (fold)

$\Leftarrow \quad \mathcal{G}\ [\![\{...W...\mid W \in g_{G_old}(t_0,t_1..t_m,t_{m+1}..t_n),...]\!]$

b) **otherwise** (define & unfold) $\Leftarrow \quad \mathcal{G}\ [\![\{...W...\mid W \in g_{G_new}(t_0,t_1..t_m,t_{m+1}..t_n),...]\!]$

define and add to def_set

$g_{G_new}(v_0,v_1,..,v_n)=\{\ g_P(w^+,v_{n+1},..,v_m) \mid w \in g_G(t_0,t_1,..,t_m)\}$

narrow on the generator

$g_{G_new}(p_1,v_1,..,v_n)=\mathcal{G}\ [\![\ \{\ g_P(w^+,v_{n+1},..,v_m) \mid w \in stg_1\}\ ]\!]$

:

$g_{G_new}(p_r,v_1,..,v_n)=\mathcal{G}\ [\![\ \{\ g_P(w^+,v_{n+1},..,v_m) \mid w \in stg_r\}\ ]\!]$

Figure 3.24 - Additional Rules for the Fusion of Generators and Processors

As a consequence, only three more rules (analogous to the three rules $\mathcal{G}$ 14, 16 and 17) are needed for a transformation algorithm to appropriately fuse generators and processors. These three rules are given in Figure 3.24 as $\mathcal{G}$ 19, 20 and 21. In each of these rules, we use a context notation, $...g_P(w,t_{m+1},..,t_n)...$, for selecting a sub-term which has a desirable processor function, g_P, with an intermediate variable, w, as its pattern-matching argument.

Rule 19 is used to consume an intermediate output term by either direct or indirect unfolding of its matched processor function call. Rules 20 and 21 are used to select pairs of generator and processor functions which are suitable for fusion. In both these rules, the intermediate variable, w, must not occur anywhere else in the set expression. Either a *define & unfold* step or a *fold* step is taken in each of these two rules, in order to perform the appropriate fusion.

3.7.2.3. An Example

To illustrate the new transformation algorithm, we shall use the earlier function, all_length, which was transformed in Section 3.7.1.4 to:

```
- - -all_length(x)       <=  { length(w) | w ∈ sub_path(x)};
- - -sub_path([])        <=  { [] };
- - -sub_path(a::x)      <=  sub_path(x) ∪ { a::w | w ∈ sub_path'(x,a)};
- - -sub_path'([],a)     <=  { [] };
- - -sub_path'(b::x,a)   <=  sub_path'(x,a) ∪ { b::w | w ∈ sub_path'(x,b), arc(a,b)};
```

This transformed function contains a generator, sub_path(x), and an outer producer function call, length(w). Such a pair of generator and producer can be fused by applying the new $\mathcal{G}$ rules to the main function, all_length. The transformation sequence for this fusion is shown in Figure 3.25.

This transformation results in a more efficient program (shown below) which does not make use of the previously present intermediate list.

```
- - -all_length([])       <=  { 0 };
- - -all_length(a::x)     <=  all_length(x) ∪ { 1+W | W ∈ all_length'(x,a)};
- - -all_length'([],a)    <=  { 0 };
- - -all_length'(b::x,a)  <=  all_length'(x,a) ∪ { 1+W | W ∈ all_length'(x,b), arc(a,b)};
```

```
---all_length(x)        <=  G [[ { length(w) | w ∈ sub_path(x)} ]]                  ! G narrow sub_path
---all_length([])       <=  G [[ { length(w) | w ∈ { []}} ]]                        ! G 14,substitute
                        <=  G [[ { length([]) } ]]                                  ! G 19,unfold length
                        <=  G [[ {0} ]]                                             ! G 3,singleton
                        <=  {0}
---all_length(a::x)     <=  G [[ { length(w) | w ∈ sub_path(x)                      ! narrow sub_path
                             ∪ { a::w | w ∈ sub_path'(x,a)} }]]                     ! G 8,distribute ∪
                        <=  G [[ { length(w) | w ∈ sub_path(x)}]]                    ! G 21a,12,fold
                             ∪ G [[ { length(a::w) | w ∈ sub_path'(x,a)} ]]         ! G 19, unfold length
                        <=  all_length(x)
                             ∪ G [[ { 1+length(w) | w ∈ sub_path'(x,a)} ]]          ! G 21b,define all_length'
                        <=  all_length(x) ∪ G [[ { 1+W | W ∈ all_length'(x,a)}]]    ! G 18,no more
                        <=  all_length(x) ∪ { 1+W | W ∈ all_length'(x,a)}
define
---all_length'(x,a)     <=  { length(w) | w ∈ sub_path'(x,a)}                       ! narrow sub_path'
---all_length'([],a)    <=  G [[ { length(w) | w ∈ { []}} ]]                        ! G 14,substitute
                        <=  G [[ { length([]) } ]]                                  ! G 19,3,unfold length
                        <=  {0}
---all_length'(b::x,a)  <=  G [[ { length(w) | w ∈ sub_path'(x,a)
                             ∪ { b::w | w ∈ sub_path'(x,b), arc(a,b)}}]]            ! G 14,substitute
                        <=  G [[ { length(w) | w ∈ sub_path'(x,a)} ]]               ! G 21b
                             ∪ G [[ { length(b::w) | w ∈ sub_path'(x,b), arc(a,b)} ]] ! G 19,unfold length
                        <=  G [[ { W | W ∈ all_length'(x,a)} ]]                     ! G 12
                             ∪ G [[ { 1+length(w) | w ∈ sub_path'(x,b), arc(a,b)} ]] ! G 21b,fold
                        <=  all_length'(x,a)
                             ∪ G [[ { 1+W | W ∈ all_length'(x,b), arc(a,b)} ]]      ! G 18,no more
                        <=  all_length'(x,a) ∪ { 1+W | W ∈ all_length'(x,b), arc(a,b)}
```

Figure 3.25 - An Example to Illustrate the Fusion of Generators and Producers

3.8. Summary of Chapter 3

In this chapter, we presented two major transformation algorithms, denoted by $\mathcal{T}$ and $\mathcal{G}$, which can be used for fusing function calls in functional compositions and set abstractions respectively. These transformation techniques improve programs by eliminating the need for intermediate data structures and/or reduce the number of function calls used (whenever possible). We also introduced a producer-consumer model of functions to help us explain the way fusion transformation works. In particular, we stated that successful fusion could only take place when a *good* producer meets a *good* consumer. We gave precise descriptions (in the form of grammar rules) for identifying *good* from *bad* producers and consumers.

In the case of functional compositions, we showed how the original deforestation algorithm of Wadler, could be gradually extended to all first-order programs. This was achieved by two apparatus. The first apparatus is a novel use of the *variable-only* parameter criterion (together with linearity and type information) to mark out those parameters which are *bad consumers*.

Correspondingly, the second apparatus, the **let** construct, was used for identifying (transforming out) *bad producers*. These two simple but effective apparatus enabled us to formulate the Universal Deforestation Algorithm which was proved to be terminating and applicable to all first-order programs[16].

A similar consideration (of the fusion technique) for set abstractions was also formulated but this formulation is more complex. To simplify our presentation, we restricted good consumers (filters or processors) to just g-type functions. This simplification still leaves us with an impressive range of programs which can be successfully fused, as illustrated throughout Section 3.7.

In the next chapter, we shall apply similar algorithmic transformation techniques towards the removal of higher-order features. This removal will help us improve program efficiency. It will also pave the way for an even more general extension of deforestation to higher-order programs.

[16] Remember that programs from the full first-order language can *always* be transformed (by Augustsson and Wadler's pattern-matching translation algorithm) to an equivalent program of the restricted first-order language in this chapter (i.e. single, unnested pattern-matching parameter for each g-type function).

4. Higher-Order Removal

4.1. Introduction

An extremely attractive feature of functional languages is their ability to handle higher-order objects. This feature enables functions to be treated as first-class citizens, where they can be passed around as *arguments* and returned as *results*, in just as the same way as data terms can. It is desirable because of the extra level of program abstraction which can be obtained by parameterising functions. In particular, it encourages and eases the construction of generic functions, which are often the building blocks for reusable software.

Though highly advantageous for ease of programming and software reusability, higher-order functions are inefficient to implement. Expensive closures (on heap space) are frequently needed to support the execution of higher-order programs. Furthermore, the generality of higher-order functions is often a source of complications for both program analysis and transformation techniques. For example:-

- Deforestation can be formulated successfully for first-order programs ([Wadler88] and Chapter 3) but encounters complications to its transformation algorithm when we try to consider it directly for arbitrary higher-order programs.
- Backward analysis techniques work well for first-order programs but face difficulty when considered for higher-order expressions [Hughes88]. This is because function-type arguments, unlike ground-type ones, have to be propagated in a forward manner for analysis.
- Forward analysis techniques (abstract interpretation) [Mycroft81,Burn87a] can be neatly formulated for strongly-typed higher-order functional programs. However, they still experience intractable complexity problem [Hunt89] in their fix-point analyses. This is due primarily to the much larger domain spaces required for function-type parameters.

One method of *partially* addressing this problem is to provide a means by which *most* of the higher-order features can be eliminated. We propose in this chapter a transformation tactic that can eliminate significant classes of higher-order features from source programs.

4.1.1. A Simple Higher Order Language

A simple higher-order language can be provided by making a few extensions to the first-order language given in Chapter 3 (for deforestation). This higher-order language will continue[1] to support the two kinds of functions, denoted by f and g, of the following form:

$$\begin{aligned} &f(v_1,..,v_n) && <= \ tf \\ \\ &g(p_1,v_1,..,v_n) && <= \ tg_1 \\ &&& \vdots \\ &g(p_r,v_1,..,v_n) && <= \ tg_r \ \text{WHERE } p_i = c_i(v'_1,..,v'_j) \end{aligned}$$

In addition, the RHS terms ($tf, tg_1,..,tg_r$) of these functions will now be terms that can be described by the following BNF grammar.

$$\begin{aligned} t &::= t' \mid c(t_1,..,t_n) \\ t' &::= v \mid fid \mid \textbf{lambda}\ (v_1,..,v_n) \Rightarrow t\ \textbf{end} \mid t'\ (t_1,..,t_n) \\ fid &::= f \mid g \end{aligned}$$

This grammar allows functions to be returned as results in the form of function identifiers, (f and g), lambda abstractions (i.e **lambda** $(v_1,..,v_n) \Rightarrow t$ **end**) or applications (i.e. $t'(t_1,..,t_n)$) which return function results. It also allows function-type expressions to be present in any of the arguments, $t_1,..,t_n$, of both applications, $t'(t_1,..,t_n)$, and constructor terms, $c(t_1,..,t_n)$. Furthermore, this grammar describes an *uncurried* language where partial applications are always expressed using lambda abstractions. For example a partially applied function call, $f\ t_1..t_{n-1}$, where the arity is n, can be denoted using **lambda** $v \Rightarrow f(t_1,..,t_{n-1},v)$ **end**. We use this uncurried notation because it helps to simplify the formulations of our transformation algorithms later. In particular, it is easier to consider the unfolding of (uncurried) function calls because they will always have all their arguments present.

Of all the features present, the two most interesting features for the higher-order removal transformations are:

- applications, denoted by $t'(t_1,..,t_n)$.
- function-type arguments in applications and constructor terms.

In the case of the applications, $t'(t_1,..,t_n)$, we can re-classify them into three categories, as shown by the following alternative BNF grammar.

$$\begin{aligned} t'(t_1,..,t_n) &::= fid\ (t_1,..,t_n) \mid va\ (t_1,..,t_n) \mid ga\ (t_1,..,t_n) \\ va &::= v \mid va\ (t_1,..,t_n) \\ ga &::= fid\ (t_1,..,t_n) \mid \textbf{lambda}\ (v_1,..,v_n) \Rightarrow t\ \textbf{end}\ (t_1,..,t_n) \mid ga\ (t_1,..,t_n) \end{aligned}$$

If t' is a function identifier, as in $fid(t_1,..,t_n)$, then its form will be *similar* to the first order function calls, like $f(t_1,..,t_n)$ or $g(t_0,..,t_n)$. We will refer to this application as just a function call or a function application. Also, if $t'(t_1,..,t_n)$ contains a leftmost functional term that is a variable, like $va(t_1,..,t_n)$, we will refer to this application as a variable application. Otherwise, if t' has a leftmost functional term which is either a function call or a lambda abstraction, as in $ga(t_1,..,t_n)$,

---

[1] The distinction between f-type and g-type functions, though useful for universal deforestation, is not essential for higher-order removal transformations. We could avoid it by using the **case** construct language of Section 2.4.1. However, as we intend to provide higher-order extensions to deforestation later (Section 4.5), we shall put up with the slight complication of having to formulate transformation rules separately for f- and g-type functions.

we will refer to it as a general application. These are the only three forms of applications which are allowed in the above language.

The other higher-order feature of interest is the function-type arguments. These arguments may appear inside constructor terms or applications. We shall frequently refer to them as either higher-order or function-type arguments.

In this chapter, we shall see that all *general applications* and a certain class of *higher-order arguments* (from *function applications*) could be eliminated by using just the define/unfold/fold transformation steps. We shall initially give brief overviews of these eliminations through some examples first. After these overviews, we shall present (later in the chapter) the complete transformation algorithms to perform automatic eliminations.

4.1.2. Overview of General Application Elimination

Each *general application* can be viewed as an application of the form:

$$e\ (t'_1,..,t'_m)..(t_1,..,t_n)$$

where the leftmost functional expression, e, is either a *function identifier* or a *lambda abstraction*. This leftmost expression, e, has the type $(\alpha_1,..,\alpha_m) \rightarrow .. (\delta_1,..,\delta_n) \rightarrow \gamma$ where $(\alpha_1,..,\alpha_m)..(\delta_1,..,\delta_n)$ are the types of the arguments, $(t'_1,..,t'_m)..(t_1,..,t_n)$, and γ is the result type from the above general application.

General applications of the above form can always be eliminated by using the **lump uncurrying** technique. This technique can always convert each of these general applications to a function application of the form, $fnew(t'_1,..,t'_m,..,t_1,..,t_n)$ where fnew is a new intermediate function of the type $(\alpha_1,..,\alpha_m,..,\delta_1,..,\delta_n) \rightarrow \gamma$. Uncurrying is the conversion of an application involving a function of the type, $\alpha \rightarrow .. \delta \rightarrow \gamma$, to a function of the type, $(\alpha,..,\delta) \rightarrow \gamma$. Lump uncurrying is simply a more general technique where tuples of parameters are involved.

To illustrate this technique of eliminating general applications, consider a simple program consisting of two functions, prg and inc.

```
- - -prg(u)      <= inc(u)(4);
- - -inc(x)      <= lambda y => y+x end;
```

Here, we have only one general application, inc(u)(4), in the function prg. This general application can be eliminated by the lump currying technique, using a new intermediate function inc', to:

```
- - - prg(u)      <= inc'(u,4);
- - - inc'(u,x)   <= u+x;
```

However, the introduction of new intermediate functions in lump uncurrying is not always necessary. At times, it is also possible to eliminate general applications by just *direct* unfoldings. A *direct* unfold is an unfold operation which does not introduce intermediate functions. With the earlier example, we could also have eliminated the given general application by two direct unfold steps, as shown below.

```
- - -prg(u)        <= inc(u)(4);                        ! unfold inc(u)
                  <= (lambda y => y+u end) (4);        ! unfold lambda application
                  <= 4+u ;
```

The removal of this general application by direct unfolding has managed to transform prg into a more efficient definition, without the need for the intermediate function, inc'.

As we shall see later, the combination of direct unfolding and lump uncurrying could be used for (efficiently) eliminating *all* general applications.

4.1.3. Overview of Higher-Order Argument Elimination

Some *instantiated* higher-order arguments of *function applications* can also be removed by transformations. This elimination of instantiated higher-order arguments can be seen as a form of *function specialisation*, like those used in partial evaluation and mixed computation [Futamura71,Ershov82,JonesN85,PE&MIX88]. In function specialisation, a function call of two arguments, p(k,u), where one argument, k, is known (or grounded) and the other argument, u, is unknown (e.g. a variable), can often be *specialised* to a semantically equivalent but more efficient call, $p_k(u)$. This new call uses a specialised function, $p_k(u)=p(k,u)$, which can often be made more efficient through the specialised context of the known argument, k. Our elimination of higher-order arguments can be viewed as this specialisation process where the *known* or *partially known* arguments are the instantiated function-type arguments.

To illustrate this process, consider a function double which uses a map function call, to double every element in its input list, as shown below.

```
dec double:list(num) -> list(num);
dec map:list(alpha) # (alpha -> beta) -> list(beta);
- - -double(x)      <= map(x,lambda a => 2*a end);
- - -map([],F)      <= [];
- - -map(a::x,F)    <= F(a)::map(x,F);
```

The map call in the function definition of double has a higher-order argument, lambda a => 2*a end. This higher-order argument can be eliminated by initially narrowing the map call, followed by an unfold and then a fold step, as shown below.

```
- - -double(x)      <= map(x,lambda a => 2*a end);                          ! narrow map
- - -double([])     <= [];
- - -double(a::x)   <= (lambda a => 2*a end ) (a) ::map(x,lambda a => 2*a end);  ! unfold lambda
                   <= 2*a::map(x,lambda a => 2*a end);                     ! fold double(x)
                   <= 2*a::double(x);
```

The result of this transformation is a purely first-order definition for double which is sightly more efficient because there are fewer applications needed. We are able to specialise this higher-order map call because of the *unchanged/fixed* parameter, F, in the map function. A parameter of a recursive function is said to be *unchanged/fixed* if it does not change across its successive recursive calls. This property can guarantee successful folding needed to specialise recursive function calls. However, there is a more general parameter criterion, called the *variable-only* parameter criterion (introduced in Section 3.3.1.1 for Extended Deforestation), that could provide a more general way of indicating the arguments which are specialisable.

Consider another higher-order function, with two function-type parameters F and G, which are being *swapped* around with each recursive call.

```
dec alt_map:list(alpha) # (alpha -> beta) # (alpha -> beta) -> list(beta);
- - -alt_map([],F,G)    <= [];
- - -alt_map(a::x,F,G) <= F(a)::alt_map(x,G,F);
```

These two parameters are not fixed but they satisfy the *variable-only* criterion. As a result, we are able to specialise any instantiated use of the function-type parameters of alt_map. For example, the following function with *instantiated* arguments, sqr and cube,

```
- - -every_other(x)    <= alt_map(x,sqr,cube);
```

can be transformed or specialised to the program below (with a similar sequence of define/unfold/fold steps as the previous example).

```
- - -every_other([])        <= [];
- - -every_other(a::x)      <= sqr(a)::every_other'(x);
- - -every_other'([])       <= [];
- - -every_other'(a::x)     <= cube(a)::every_other(x);
```

Notice that we have effectively eliminated both the instantiated function-type arguments from the alt_map call[2].

Apart from the above higher-order arguments which can be (easily) eliminated, it is also possible to have higher-order arguments which are harder to eliminate. These arguments are those *without* the variable-only parameter criterion. As an example, consider the parameter, F, of function acc_map below.

```
dec acc_map:list(num) # (num -> num) -> list(num);
- - -acc_map([],F)    <= [];
- - -acc_map(a::x,F) <= F(a)::acc_map(x,lambda b => b+F(a) end);
```

This parameter acts as a kind of accumulating parameter and could generate infinite versions of recursive calls during unfold/fold transformations. For example, if we try to specialise the function call acc_map(x,id) where id is the identity function, with just the define/unfold/fold transformation steps, we may obtain the following infinite versions of recursive function calls to specialise!

```
acc_map(x,lambda b1 => b1+id(a1) end),
acc_map(x,lambda b2 => b2+(lambda b1 => b1+id(a1) end(a2)) end),
acc_map(x,lambda b3 => b3+(lambda b2 => b2+(lambda b1 => b1+id(a1) end(a2)) end (a3)) end),
.....
```

Such higher-order arguments, without the variable-only criterion, are harder to eliminate. They frequently require either laws or more complex sequence of transformation steps before they can be eliminated. Because of this, we shall ignore these arguments in our higher-order removal tactic.

---

[2] To achieve this higher-order argument removal, we have introduced two new functions, every_other and every_other', to replace the instantiated function call. This increase in program size is frequently a consequence of program transformation techniques. However, as the storage used is statically determined before run-time, it is less expensive than the dynamic heap spaces required by closures for higher-order arguments.

4.1.4. Overview of Full Laziness Technique

An important efficiency concern for higher-order programs is the preservation of *full laziness.* This concern was first recognised in [Wadsworth71], where it was observed that the evaluation of functional expressions may cause losses of efficiency, unless *free expressions* of lambda abstractions are always shared before the lambda bodies are duplicated.

> A **free expression** in a lambda abstraction (or a given expression) is a subexpression which does not contain any of the bound variables of the lambda abstraction (or the given expression).

Wadworth's full laziness technique is a run-time technique which always searches for free expressions (other than simple variables or constants) for sharing, before lambda abstractions are about to be duplicated/copied. This technique carries some run-time overheads.

Subsequently, Hughes proposed a compile-time technique, called *fully lazy lambda lifting* [Hughes82], which provided inherent laziness through the parameter sharing mechanism. This technique combined the need for full laziness into lambda lifting. We discuss the lambda lifting technique next, before looking at how full laziness was built into the lambda lifting technique.

Lambda lifting is a transformation technique for removing *free variables* from lambda abstractions. As lambda abstractions may be nested, lambda lifting must be applied to the inner lambda abstractions before the outer ones. Given an inner lambda abstraction, lambda $x_1..x_n$ => E end, with free variables, $v_1..v_m$ in E, the lambda lifting technique works by converting the given lambda abstraction to (lambda $v_1..v_m,x_1..x_n$ => E end) ($v_1..v_m$). The result of this conversion is a new lambda abstraction; lambda $v_1..v_m,x_1..x_n$ => E end, which does not contain any free variables because the free variables have been added as bounded parameters of the new lambda abstraction. Such lambda abstractions, without free variables, are called *supercombinators* by Hughes. They are useful because compiled codes could be provided to support their *argument instantiations* during program execution. This is more efficient than using the interpretive method (of recursive tree-walk) needed to instantiate lambda abstractions with free variables.

Even though lambda lifting technique is useful, it does not preserve full laziness. To preserve full laziness, Hughes suggested the extraction of maximal free expressions (rather than just free variables) during lambda lifting.

> A **maximal free expression** (or MFE) is a free expression in a lambda abstraction (or a given expression) that is not contained within another free expression.

The technique of extracting out MFEs is known as the *fully lazy lambda lifting* technique. It works by considering the inner lambda abstractions before the outer ones. In addition, it must also deal with single-parameter lambda abstractions. Each multi-parameter lambda abstraction, lambda $x_1..x_n$ => E end, would have to be converted to a nesting of single-parameter lambda abstractions, lambda x_1 => (lambda x_2 => ... (lambda x_n => E end) ... end) end before it could be converted by the technique. Thus, given the above innermost lambda abstraction, lambda x_n => E end, the fully lazy lambda lifting technique will search for all MFEs, say $e_1..e_f$,

in E. These MFEs will be extracted from the lambda body to give (lambda $v_1..v_f\ x_n$ => E [$v_1/e_1,..,v_f/e_f$] end) $e_1..e_f$. The extraction process is repeatedly applied to all lambda abstractions until each of them is made into a fully lazy supercombinator. To illustrate this fully lazy lambda lifting technique, our simple program would have to be expressed in *curried* form (in order to allow the easy extraction of both ground-type and function-type free expressions), as shown below.

```
---p y          <= dup lambda x=> + (sqr y) x end;
---dup v        <= + (v 3) (v 4);
```

This program must also be converted to use single-parameter lambda abstractions, as follows.

```
---p            <= lambda y => dup lambda x=> + (sqr y) x end end;
---dup          <= lambda v => + (v 3) (v 4) end;
---inc'(u,x)    <= u+x;
```

In this form, we can only find one MFE, + (sqr y), which occurs in the inner lambda abstraction of the function p. To extract it, the fully lazy lambda lifting technique would introduce an intermediate function, h (i.e. a supercombinator), to hold the MFE. This extraction and the lifting of all lambda abstractions gives:

```
--p y          <= dup h (+ (sqr y));
--h Y x        <= Y x ;
--dup v        <= + (v 3) (v 4) ;
```

The MFE extracted, + sqr(y), is actually a partial application. The effect of this extraction is the introduction of another function, h, which has a function-type argument, Y. Such extractions of partial applications and the corresponding *introductions* of higher-order functions (with function-type parameters) are the means by which fully lazy lambda lifting preserve full laziness. However, the technique contradicts with the goal of higher-order removal transformations because of its introduction of function calls with higher-order arguments. The goal of our transformation is the removal of higher-order arguments and not their introduction.

To preserve full laziness, we shall propose (later in this chapter) a new full laziness transformation technique which does not need to introduce higher-order features. We give a brief introduction of how the new technique works here.

Instead of extracting out all MFEs, our full laziness technique will only extract out the *ground-type* MFEs.

> A **ground-type MFE** (GT-MFE) of a lambda abstraction (or a given expression) is a *ground-type* free expression, other than a simple variable or constant, which is not contained within another *ground-type* free expression. Also, an expression is *ground-type* if it returns an object-type result, like 2, ['a','b'], as opposed to a function-type result, like lambda x=> x+1 end.

Notice that simple variables or constants are not regarded as GT-MFE because they do not cause any losses of laziness when duplicated.

The extraction of only GT-MFE will help to ensure that we do not introduce higher-order arguments. With the earlier example, this means that we shall extract out the GT-MFE, sqr y,

rather than the MFE, + (sqr y). This extraction would result in the following fully lazy program, without any introduction of functions with higher-order arguments.

```
- - -p(y)        <= dup h'(sqr(y)) ;
- - -h'(Y)       <= lambda x => Y+x end;
- - -dup(v)      <= v(3) + v(4) ;
```

Our alternative technique does not need lambda abstractions to be converted to single-parameter lambda abstractions. This is because it could directly examine each lambda abstraction of the form, lambda $x_1..x_n$ => E end, to find and extract out the GT-MFEs that may be present in E. However, some GT-MFEs may be trapped under function calls. These trapped GT-MFEs must be revealed by other transformation techniques. As an example, consider the following alternative program for p.

```
- - -p(y)        <= dup (lambda x => h(y,x) end);
- - -h(y,x)      <= sqr(y)+x;
- - -dup(v)      <= v(3) + v(4) ;
```

On the surface, the lambda abstraction, lambda x => h(y,x) end in function p, does not contain any *explicit* GT-MFEs. However, if we allow the direct unfold of function call, h(y,x), in the lambda abstraction, this would reveal the *implicit* GT-MFE, sqr(y), trapped under the function h, as shown below.

```
- - -p(y)        <= dup (lambda x => h(y,x) end);        ! unfold h(y,x)
                 <= dup (lambda x => sqr(y)+x end);
```

The use of transformation techniques, like unfolding, to reveal and allow the extraction of implicit GT-MFEs are vital instruments of our new full-laziness technique. We shall see how they are carried out later in this chapter.

4.1.5. Overview of Fully Lazy Higher-Order Argument Elimination

Previously, we illustrated the elimination of higher-order arguments without considering a possible loss in efficiency that is closely related to the full laziness concern. In particular, the primary technique used to eliminate instantiated higher-order arguments is unfolding. This may sometimes duplicate (non-linear) instantiated higher-order arguments and cause possible losses of efficiency.

As an example, consider the fully lazy function, p, from the previous section.

```
- - -p(y)        <= dup (h'(sqr(y)));
- - -h'(Y)       <= lambda x => Y+x end;
- - -dup(v)      <= v(3) + v(4) ;
```

To eliminate the higher-order argument, h'(sqr(y)), in dup call of function p, we could apply the following sequence of transformation steps.

```
- - -p(y)        <= dup (h'(sqr(y)));          ! unfold dup
                 <= h'(sqr y)(3)+h'(sqr y)(4); ! unfold h' twice
                          :
                 <= (sqr(y)+3)+(sqr(y)+4);
```

This sequence of transformation steps has produced a first-order function without any higher-order arguments. However, the transformed function also contains two identical sub-

expressions, sqr(y), that arise from the duplication of non-linear function-type argument of dup call. This redundancy is related to full laziness concern because we have duplicated a non-linear higher-order argument with GT-MFEs.

To avoid this loss of efficiency, we must always extract out all GT-MFEs from the non-linear function-type arguments before they are duplicated by unfolding. The only GT-MFE in h'(sqr(y)) is sqr(y). Note that we are using a more general definition of GT-MFE which does not have to be from a lambda abstraction (see the bracketed alternative in the definition of GT-MFE in previous sub-section). Extracting this GT-MFE out with another intermediate function, pnew, would give us:

```
---p(y)          <= pnew(sqr(y));
---pnew(Y)       <= dup (h'(Y));
---h'(Y)         <= lambda x => Y+x end;
---dup(v)        <= v(3) + v(4) ;
```

The new instantiated higher-order argument, h'(Y) in pnew, does not now contain any GT-MFEs. It could therefore be (efficiently) eliminated by unfolding the dup function call to give:

```
---pnew(Y)       <= (Y+3)+(Y+4);
```

We shall call such a technique for eliminating higher-order arguments, the *fully lazy* higher-order argument elimination transformation.

4.1.6. Outline of Stages for the Higher-Order Removal Tactic

As shown with the earlier examples, the two principal higher-order features, general applications and higher-order arguments, can be eliminated by unfold/fold transformations. In fact, we shall show that these eliminations can be formulated as automatic transformation algorithms, which will be known respectively as:

1) Elimination of General $\mathcal{A}$pplications (denoted by $\mathcal{A}$)

2) Elimination of Higher-Order A$\mathcal{R}$guments (denoted by $\mathcal{R}$)

In addition, we shall also need a new full laziness transformation technique to make programs fully lazy *without* introducing any new higher-order arguments. This technique works by extracting out ground-type MFEs from lambda abstractions, where possible. We shall show that it can also be formulated as a transformation algorithm which will be known as:

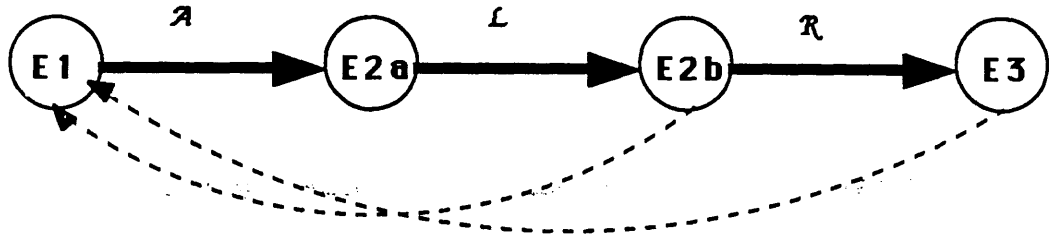
3) Full $\mathcal{L}$aziness Transformation (denoted by $\mathcal{L}$)

Our full laziness algorithm shall be compatible with the higher-order removal transformations. In particular, it will help to float out both *implicit* and *explicit* ground-type MFEs to places, where they could be shared, before the elimination of higher-order arguments via $\mathcal{R}$. This is especially important for the fully lazy elimination of *non-linear* higher-order arguments.

These three transformation algorithms, $\mathcal{A}$, $\mathcal{L}$ and $\mathcal{R}$, together make up the higher-order removal tactic. They are closely inter-related because some of these transformations may introduce features which have to be eliminated by the other transformations. For example, the elimination of higher-order arguments, $\mathcal{R}$, may in its process introduce general applications which have to be eliminated by $\mathcal{A}$, and vice versa.

However, instead of organising it as a single complex transformation, it is better for us to separate them out as three *fairly* independent algorithms which are applied in some appropriate sequence. One suitable application sequence is shown in Figure 4.1. This sequence contains three principal transformation stages (denoted by the thick arrows), which must each be applied exhaustively to an *entire* program before moving on to the next stage. These stages are intended to gradually transform our expressions to more desirable forms - starting with the *elimination of general applications*, then the *preservation of laziness* and finally the *elimination of instantiated higher-order arguments* from function calls. These stages are best illustrated by looking at the expression forms, **E1**, **E2a**, **E2b** and **E3**, before and after each of the above three transformation stages. The various expression forms are:

- **E1** - any expression from the simple higher-order language
- **E2a** - without general applications
- **E2b** - fully lazy and without general applications
- **E3** - fully lazy, without general applications and without instantiated higher-order arguments (that have the variable-only property).



𝒜- Elimination of General Applications

ℒ- Full Laziness Transformation

ℛ- Fully Lazy Elimination of Higher-Order Arguments

Figure 4.1 - Complete Transformation Stages for the Higher-Order Removal Tactic

As mentioned earlier, the three transformation algorithms cannot be made completely independent from one another. This is because it is still possible for the $\mathcal{R}$ and the $\mathcal{L}$ transformations to re-introduce general applications. Nevertheless, these re-appearances can be removed by re-invoking the $\mathcal{A}$ transformation, as shown by the dotted arrows. These re-invocations will not result in non-termination, as we shall see in our termination proofs.

In this chapter, we shall describe all three transformation stages which together constitute our higher-order elimination tactic. We begin by describing, in Section 4.2 and 4.3, the two transformation stages for eliminating general applications and higher-order arguments respectively. In each of these cases, we present the transformation algorithm, target expression form and proof that our transformation is both safe and effective. For simplicity, the presentation of higher-order argument elimination will initially ignore the full laziness concern. This laziness issue is considered retrospectively in Section 4.4, where we present a new approach to full laziness and show how this new approach can be made compatible with higher-

order argument elimination. After describing these three stages for higher-order removal, we shall show in Section 4.5 one very useful result of these eliminations, namely that we are able to extend the previous universal deforestation algorithm of Chapter 3 to higher-order programs! This extension will also have the advantage that it is able to eliminate lists and trees of functions. We shall conclude in Section 4.6 by comparing our tactic to some related research works.

4.2. Elimination of General Applications

The source of the transformation stage, $\mathcal{A}$, is any higher-order expression, while the target is a class of expressions that is devoid of all general applications. We shall call the target expression form, the *applyless* form, and formally define it as follows.

Definition 4.1 - Applyless Form

A term is said to be *applyless* if it does not contain any general applications. This form can be described by the grammar below,

t ::= va | $c(t_1,..,t_n)$ | **lambda** $(v_1,..,v_n)$ => t **end** | $f(t_1,..,t_n)$ | $g(t_0,t_1,..,t_n)$
va ::= v | $va(t_1,..,t_n)$

where g and f are *applyless* functions.

Correspondingly, a function is said to be *applyless,* if its definition is made up of *applyless* terms.

The distinctive feature of this applyless form is the *absence* of general applications. We shall show that general applications can always be eliminated by transformation algorithm, $\mathcal{A}$. This result is stated by the following theorem.

Theorem 4.2 - Elimination of General Applications

Any higher-order expression, which is well-typed can always be effectively transformed by algorithm, $\mathcal{A}$, to an equivalent term that is *applyless*, without any loss of efficiency.

The theorem states that all higher-order expressions can always be converted to applyless form if the expression is well-typed. An expression is *well-typed* if it satisfies the Hindley-Milner's polymorphic type checking algorithm [Hindley69,Milner78]. This requirement is important for proving the termination of the transformation algorithm, $\mathcal{A}$, as we shall see later in Section 4.2.3.3. Also, to convert an entire program into applyless form, we have to apply the transformation algorithm, $\mathcal{A}$, using the bottom-up conversion strategy of Section 3.4.3.1. This will help to ensure that all auxiliary functions are always converted before their parent functions. The proof for this elimination of general applications theorem shall be given in Section 4.2.3.

4.2.1. Transformation Algorithm, $\mathcal{A}$

The twelve transformation rules for $\mathcal{A}$ are given in Figure 4.2. To shorten these rules, we frequently use the abbreviation, $t_{1..n}$, to denote a list of arguments, $t_1,..,t_n$.

The first nine rules ($\mathcal{A}$ 1 to 9) of our algorithm deal with the outer expression structures that are already in applyless form. The structures of these outer expressions are preserved with recursive transformations applied to their sub-expressions, where applicable.

The main general applications elimination rules occur in $\mathcal{A}$ 10, 11 and 12. In each of these three rules, we start our elimination by referring to the leftmost sub-expression of the general application. This leftmost sub-expression, e, is specified with the notation:

$$e\ (t_1,..,t_n)\ (te_1,..,te_a)..(te_{a+b},..,te_s)$$

where e is either a lambda abstraction or a f- or a g-type function identifier. These are the only three types of general applications. We perform their eliminations with either one of the following.

1) a direct unfold step ($\mathcal{A}$10a, $\mathcal{A}$11a), or
2) a define&unfold step ($\mathcal{A}$10b, $\mathcal{A}$11c, $\mathcal{A}$12b), or
3) a fold step ($\mathcal{A}$11b,$\mathcal{A}$12a).

Our first choice step for removing general applications is *direct unfolding* ($\mathcal{A}$ 10a, 11a). This *direct unfolding* can be used for

> *lambda abstractions* and *non-recursive f-type function calls*, whose non-linear as well as function-type arguments are either variables or constant functions (CF).

A constant function is either a constant, like 1 or 2, or a function without parameters that returns a constant. These functions can be assumed to be pre-computed and thus will not cause inefficiency when duplicated.

To identify the parameters which determine if direct unfolding is possible, we introduce an annotation, @, (that can be determined before transformation) to mark all those parameters that are either non-linear and/or of function-type for each lambda abstraction and non-recursive f-type functions. These parameters are assumed to be grouped together as parameters, $v_{m+1}..v_n$ (as shown in $\mathcal{A}$10 and $\mathcal{A}$11).

With this assumption, let us consider a general application, p(3,w)(b*b), where w and b are variables and p is a function as defined below.

```
- - -p(x@,v@)      <=  lambda a => v(x*x+a) end;
```

The function, p, has a non-linear parameter, x, and a function-type parameter, v. Both of these parameters are annotated as @. However, as the arguments of the function call, p(3,w), are just a constant 3 and a variable w, we could still perform a direct unfold using $\mathcal{A}$11a. This could then be followed by a sequence of $\mathcal{A}$ steps, as shown below, in order to transform the given expression to an equivalent applyless expression without any general applications.

```
A[[ p(3,w)(b*b) ]]   is transformed to:                    !A11a
                  =  A[[ lambda a => w(3*3+a) end (b*b) ]]  !A 10a
                  =  A[[ w(3*3+b*b) ]]                      !A 7
                  =  w(A[[ 3*3+b*b]])                       !A 8
                  =  w(A[[ 3*3]]+A[[ b*b]])                 !A 8,8
                  =  w(A[[ 3]]*A[[ 3]]+A[[ b]]*A[[ b]])     !A 2,2,1,1
                  =  w(3*3+b*b)
```

(1) $\mathcal{A}[\![v]\!] \Leftarrow v$

(2) $\mathcal{A}[\![c(t_1,..,t_n)]\!] \Leftarrow c(\mathcal{A}[\![t_1]\!],..,\mathcal{A}[\![t_n]\!])$

(3) $\mathcal{A}[\![$ lambda $(v_1,..,v_n)$ => tf end $]\!] \Leftarrow$ lambda $(v_1,..,v_n)$ => $\mathcal{A}[\![$ tf $]\!]$ end

(4)[3] $\mathcal{A}[\![$ let v== t_1 in $t_2]\!] \Leftarrow$ let v== $\mathcal{A}[\![t_1]\!]$ in $\mathcal{A}[\![t_2]\!]$

(5) $\mathcal{A}[\![f]\!] \Leftarrow f$

(6) $\mathcal{A}[\![g]\!] \Leftarrow g$

(7) $\mathcal{A}[\![va(t_1,..,t_n)]\!] \Leftarrow \mathcal{A}[\![va]\!](\mathcal{A}[\![t_1]\!],..,\mathcal{A}[\![t_n]\!])$

(8) $\mathcal{A}[\![f(t_1,..,t_n)]\!] \Leftarrow f(\mathcal{A}[\![t_1]\!],..,\mathcal{A}[\![t_n]\!])$

(9) $\mathcal{A}[\![g(t_0,..,t_n)]\!] \Leftarrow g(\mathcal{A}[\![t_0]\!],..,\mathcal{A}[\![t_n]\!])$

(10) $\mathcal{A}[\![$ lambda $(v_1..v_m,v_{m+1}{}^{@}..v_n{}^{@})$ => tf end $(t_1..t_m,t_{m+1}{}^{@}..t_n{}^{@})$ $(te_1,..,te_a)..(te_{a+b},..,te_s)$ $]\!]$

Given $^{@}$ annotates each argument which is non-linear and/or function-type

a) if $t_{m+1}..t_n$ are variables or constants; (direct unfold)

$\Leftarrow \mathcal{A}[\![$ tf $[t_1/v_1,..,t_n/v_n]$ $(te_1,..,te_a)..(te_{a+b},..,te_s)$ $]\!]$

b) otherwise (define & unfold) $\Leftarrow$ f_new $(\mathcal{A}[\![t]\!]_{1..n},\mathcal{A}[\![te]\!]_{1..s})$

define and unfold lambda application

f_new$(v_{1..n},ve_{1..s}) = \mathcal{A}[\![$ tf $[v_1/v_1,..,v_n/v_n]$ $(ve_1,..,ve_a)..(ve_{a+b},..,ve_s)]\!]$

(11) $\mathcal{A}[\![f(t_1..t_m,t_{m+1}{}^{@}..t_n{}^{@})$ $(te_1,..,te_a)..(te_{a+b},..,te_s)]\!]$

Given $^{@}$ annotates each argument which is non-linear and/or function-type

a) if $t_{m+1}..t_n$ are all variables/CF and f is non-recursive; (direct unfold)

$\Leftarrow \mathcal{A}[\![$ tf $[t_1/v_1,..,t_n/v_n]$ $(te_1,..,te_a)..(te_{a+b},..,te_s)]\!]$

b) if f_old$(v_{1..n},ve_{1..s})$= f$(v_{1..m},v_{m+1..n})$ $(ve_1,..,ve_a)..(ve_{a+b},..,ve_s) \in$ def_set **then** (fold)

$\Leftarrow$ f_old $(\mathcal{A}[\![t]\!]_{1..n},\mathcal{A}[\![te]\!]_{1..s})$

c) otherwise (define & unfold) $\Leftarrow$ f_new $(\mathcal{A}[\![t]\!]_{1..n},\mathcal{A}[\![te]\!]_{1..s})$

define and add to def_set

f_new$(v_{1..n},ve_{1..s})$= f$(v_{1..m},v_{m+1..n})$ $(ve_1,..,ve_a)..(ve_{a+b},..,ve_s)$

unfold f ***call to give:***

f_new$(v_{1..n},ve_{1..s}) = \mathcal{A}[\![$ tf $[v_1/v_1,..,v_n/v_n]$ $(ve_1,..,ve_a)..(ve_{a+b},..,ve_s)$ $]\!]$

(12) $\mathcal{A}[\![$ g$(t_0,t_{1..n})$ $(te_1,..,te_a)..(te_{a+b},..,te_s)]\!]$

a) if g_old$(v_0,v_{1..n},ve_{1..s})$= g$(v_0,v_{1..n})$ $(ve_1,..,ve_a)..(ve_{a+b},..,ve_s) \in$ def_set **then** (fold)

$\Leftarrow$ g_old $(\mathcal{A}[\![t]\!]_{0..n},\mathcal{A}[\![te]\!]_{1..s})$

b) otherwise(define & unfold) $\Leftarrow$ g_new$(\mathcal{A}[\![t]\!]_{0..n},\mathcal{A}[\![te]\!]_{1..s})$

define and add to def_set

g_new$(v_0,v_{1..n},ve_{1..s})$= g$(v_0,v_{1..n})$ $(ve_1,..,ve_a)..(ve_{a+b},..,ve_s)$

unfold g ***call to give:***

g_new$(p_1,v_{1..n},ve_{1..s}) = \mathcal{A}[\![$ tg$_1$ $[v_1/v_1,..,v_n/v_n]$ $(ve_1,..,ve_a)..(ve_{a+b},..,ve_s)$ $]\!]$

:

g_new$(p_r,v_{1..n},ve_{1..s}) = \mathcal{A}[\![$ tg$_r$ $[v_1/v_1,..,v_n/v_n]$ $(ve_1,..,ve_a)..(ve_{a+b},..,ve_s)$ $]\!]$

Figure 4.2 - Rules for the Elimination of General Applications

If the above condition for direct unfolding cannot be satisfied, we must instead use the define&unfold step. For example, another general application, say p(a*3,sqr)(b), cannot be *directly* unfolded by $\mathcal{A}$11a because of the presence of the non-linear argument, a*3, and the instantiated function-type argument, sqr. To eliminate this general application, we must resort to $\mathcal{A}$11b which would *define* a new function, say p', followed by the extraction of all the arguments before an *unfold*. This transformation is illustrated below.

[3] let is presently not a construct of the simple higher-order language in Section 4.1.1 but this rule represent how it must be handled if it is present.

```
ℛ[[ p(a*3,sqr)(b) ]]   is transformed to:                          ! define p_int by ℛ11b
                       =    p_int(ℛ[[ a*3]],ℛ[[ sqr]],ℛ[[ b]])     ! ℛ 8,5,1
                              :
                       =    p_int(a*3,sqr,b)
---p_int(x,v,b)        <=   p(x,v)(b)                             ! unfold p(x,v)
                       <=   ℛ[[ lambda a => v(x*x+a) end (b) ]]    ! ℛ10a
                       <=   ℛ[[ v(x*x+b) ]]                       ! ℛ 7,..
                              :
                       <=   v(x*x+b)                              ! ℛ 7,..
```

We still get an applyless expression, but it is not as efficient as the earlier case. These two cases illustrate how we would eliminate general applications without losing efficiency by the ℛ rules.

In addition to the direct unfold and the define&unfold steps, we also have the fold steps (ℛ11b, ℛ12a). These fold steps work in tandem with the define&unfold steps for handling recursive function calls which might otherwise cause non-termination. They check to see if there are any re-occurrence of general applications among the previously defined functions in def_set (function definition set). Notice that we do not need a fold step for rule ℛ 10 because lambda abstractions are non-recursive and do not therefore have to be folded.

4.2.2. An Example

To illustrate the application of the transformation algorithm, ℛ, let us consider a recursive higher-order function, app, as shown below.

```
dec app : list(alpha) -> (list(alpha) -> list(alpha));
---app([])          <= lambda w => w end;
---app(a::x)        <= lambda w => a::app(x)(w) end;
```

This function contains a general application, app(x)(w), which can be eliminated. To eliminate it, we apply the ℛ rules to each of the two RHS terms, as shown in Figure 4.3.

```
---app([])          <=  ℛ [[ lambda w => w end ]]                  ! ℛ 3
                    <=  lambda w => ℛ[[ w ]] end                   ! ℛ 1
                    <=  lambda w => w end
---app(a::x)        <=  ℛ [[ lambda w => a::app(x)(w) end ]]       ! ℛ 3
                    <=  lambda w => ℛ[[ a::app(x)(w) ]] end        ! ℛ 2
                    <=  lambda w => ℛ[[ a ]]::ℛ [[ app(x)(w) ]] end ! ℛ 1
                    <=  lambda w => a::ℛ [[app(x)(w) ]] end        ! ℛ 12b, define append
                    <=  lambda w => a::append(ℛ [[x ]] ,ℛ [[w ]]) end ! ℛ 1
                    <=  lambda w => a::append(x ,w) end
define
---append(x,w)      <=  app(x)(w)
---append([],w)     <=  ℛ [[ lambda w => w end (w) ]]              ! ℛ 10a
                    <=  ℛ [[w]]                                    ! ℛ 1
                    <=  w
---append(a::x,w)   <=  ℛ [[ lambda w => a::append(x ,w) end (w) ]] ! ℛ 10a
                    <=  ℛ [[ a::append(x ,w) ]]                    ! ℛ 2,1,9,1,1
                    <=  a::append(x,w)
```

Figure 4.3 - Successful Application of ℛ to a Recursive Function

In this transformation, an intermediate function, append, was introduced. This function, together with function app, was eventually transformed to the following two applyless definitions.

```
---app([])          <= lambda w => w end;
---app(a::x)        <= lambda w => a::append(x,w) end;
---append([],w)     <= w;
---append(a::x,w)   <= a::append(x,w);
```

This example illustrates how recursive g-type functions could be handled by the $\mathcal{A}$ rules.

4.2.3. Proof of Theorem 4.2

The complete proof for the General Application Elimination Theorem follows from the following three separate lemmas.

Lemma 4.3 - On Applyless Form

Each expression which results from algorithm $\mathcal{A}$, under conditions of the General Application Elimination Theorem, will always be applyless.

Lemma 4.4 - On Efficiency

No loss in efficiency occurs with the transformation rules of $\mathcal{A}$

Lemma 4.5 - On the Termination of $\mathcal{A}$

The transformation algorithm, $\mathcal{A}$, will terminate under the conditions of the theorem.

4.2.3.1. Proof for Lemma 4.3 (on applyless form)

The first lemma can be proved by structural induction. By examining each expression which results from the $\mathcal{A}$ rules, we can show that the transformed expressions will always be applyless in form. This is because none of the results (on the RHS of the the $\mathcal{A}$ rules) contain any general applications.

4.2.3.2. Proof for Lemma 4.4 (on efficiency)

The second lemma can also be proved by induction. In the case of the define&unfold step (in $\mathcal{A}$ 10b, 11c and 12b), a subsequent unfold will compensate for the additional intermediate function introduced. No loss of efficiency occurs here. Also, in the direct unfold steps (of $\mathcal{A}$ 10a and 11a), no intermediate functions are introduced. This means that there will be fewer function applications and thus some efficiency improvement. Similarly, folding (of $\mathcal{A}$ 11b and 12a) as it is used here (for recursion) does not cause any loss in efficiency. The other rules, $\mathcal{A}$ 1 to $\mathcal{A}$ 9, just recursively consider their subexpressions. By induction, they do not cause any loss of efficiency.

4.2.3.3. Proof for Lemma 4.5 (on termination)

The third lemma on algorithm termination needs to prove two properties in the recursive rules. These are:

(1) The presence of an upper bound on the size of expression used on the RHS of each new function definition introduced. This would show that there will only be a finite number of define&unfold steps of $\mathcal{A}$ 11b and $\mathcal{A}$ 12b, as any re-occurrences of expressions encountered by these rules could always be folded, in preference over another define&unfold step.

(2) The other recursive rules, $\mathcal{A}$ 2,3,4,7,8,9,10a,10b,11a,11b,12a, must be shown to have a well-founded decreasing measure. This would show that any sequences of steps involving just these rules will always be finite.

If both these properties can be established, then the total number of recursive steps for this transformation algorithm will also be finite (for any given expression that satisfies the conditions of the theorem). We outline the proofs for these two properties separately.

To show the first property, notice that the $\mathcal{A}$ rules use *only* general applications (with variables as arguments) for new function definitions. We could thus define the size of expression by the maximum number of tuples which are present in each general application. This size will always be bounded for any well-typed program. This is because each well-typed program will only have functions whose type descriptions are *finite*. The type description of a function can be described by the following grammar rule.

τ ::= ground_type | $(\tau_1,..,\tau_n)$ | τ_1 -> τ_2

Its description is said to be *finite* if it can be represented using a *finitely*-sized tree. Hindley-Milner's type-checking algorithm can assign a consistent and *finite* typing to each function (or expression) only if the program is well-typed.

Because of this, each well-typed general application, $e(v_1,..,v_n)(ve_1,..,ve_a)..(ve_{a+b},..,ve_s)$, which is used for a new function definition, will have a size (in terms of the number of tuples of arguments) that is never more than size_ga[[type[[e]]]], where type[[e]] returns the type of the expression, e, and size_ga is defined as follows:

size_ga[[ground_type]] = 0
size_ga[[$(\tau_1,..,\tau_n)$]] = 0
size_ga[[τ_1 -> τ_2]] = 1+ size_ga[[τ_2]]

Correspondingly, in any well-typed program with a set of functions[4], $h_1..h_m$, the maximum-sized general application, say $h_i(t_1,..,t_n)(te_1,..,te_a)..(te_{a+b},..,te_s)$, which could result from the functions, $h_1..h_m$, will never be more than max{size_ga[[type[[h_1]]]],..,size_ga[[type[[h_m]]]]}. In other words, the general applications that could result from each well-typed program will always be bounded to a maximum size, namely max{size_ga[[type[[h_1]]]],..,size_ga[[type[[h_m]]]]}, of the original expression. As a result, the size of expressions used for new function definitions of the $\mathcal{A}$ rules will be correspondingly bounded.

For example, the program of the previous section containing only one function, app, whose type is list(alpha) -> (list(alpha) -> list(alpha)), has a bounded size, max{size_ga[[type[[app]]]}=2. As a result, all well-typed general applications that could result from transforming the expressions of this program will never be more than two tuples long, e.g. app(a)(b) and never app(a)(b)(c) !

In contrast to the finitely-typed functions, *infinitely* typed functions may cause non-termination with the $\mathcal{A}$ rules. A fictitious and illegally-typed example is the function, illegal_fn, which has an *infinite* type description, as shown below.

```
dec illegal_fn : alpha -> (alpha -> (alpha -> (.....))) ;     ! infinite type description
- - -illegal_fn(x)        <= illegal_fn(x)(x) ;
```

[4] including lambda abstractions

This infinitely-typed function contains a general application, illegal_fn(x)(x), that can never be completely eliminated! An attempt to eliminate it will simply result in ever larger general applications of the form:

illegal_fn(x_1)(x_2)
illegal_fn(x_1)(x_2)(x_3)
:
illegal_fn(x_1)(x_2)(x_3)..(x_n)
:

causing non-termination to the $\mathcal{A}$ rules. Such illegally typed functions should be excluded from our language. This is assured by Hindley-Milner's type-checking algorithm. Notice that this exclusion does not include the fix-point operator, fix, for primitive recursion. This is because the fix-point operator still has a finite type description, (alpha -> alpha) -> alpha, as shown below.

```
dec fix : (alpha -> alpha) -> alpha ;          ! finite type description
- - -fix(v)          <=  v (fix(v)) ;
```

The other recursive steps, $\mathcal{A}$ 2,3,4,7,8,9,10a,10b,11a,11b,12a, can be shown to have a well-founded decreasing measure. Firstly, there is a set of rules ($\mathcal{A}$ 2,3,4,7,8,9,11b,12a), which always work on successively smaller sub-expressions. This set could rely on the size of expressions for a well-founded measure. Secondly, there is a direct unfold rule, $\mathcal{A}$ 11a, for removing non-recursive f-type function calls that appear as the leftmost term in a general application. We could use the function hierarchy number to show that the resulting expression from this rule has a (well-founded) decreasing measure.

Lastly, there is a direct unfold and a define&unfold rule ($\mathcal{A}$ 10a,10b) for applications involving lambda abstractions. These two rules do not have corresponding fold rules. However, notice that these two rules always reduce the size of the present general application by at least one. This characteristic could be used to define a well-founded decreasing measure. All these separate measures are for the individual rules. They could be *combined* into a single complex measure which decreases with each application of any of the recursive rules. Such a well-founded measure is given and proved in [Chin90]. It is achieved in a manner similar to the proof of Appendix 5.

Because of this well-founded decreasing measure, there could never be an infinite sequence of just these recursive steps. Together with the fact that there is only a finite number of the define&unfold steps of $\mathcal{A}$ 11c,12b, the total number of steps for this algorithm will always be finite. Hence, the termination of algorithm $\mathcal{A}$, under the conditions of its theorem.

4.3. Elimination of Higher-Order Arguments

The second transformation stage, $\mathcal{R}$, to eliminate higher-order arguments is more complicated than the first transformation stage, $\mathcal{A}$. One factor which adds to the complication is the need to provide fully lazy treatment during the elimination of *non-linear* higher-order arguments. This treatment is quite complex but is essential for ensuring no loss of efficiency. In order to simplify our presentation, we shall initially ignore the full laziness concern by assuming that all

higher-order arguments are devoid of ground-type MFEs. This issue will instead be addressed retrospectively in Section 4.4.

4.3.1. Higher-Order Specialised Form

The source expression form to be used by the Elimination of Higher-Order Arguments, $\mathcal{R}$, shall be the *applyless* expression form. The target expression form for $\mathcal{R}$ shall be an expression form which is devoid of all instantiated higher-order arguments (satisfying the variable-only parameter criterion). We shall call this target expression form, the *higher-order specialised,* or *HO-specialised,* form.

To help us define the target form, let us introduce a new parameter annotation, *, to mark each parameter that is *function-type* and satisfies the *variable-only* criterion. (Note that this variable-only parameter criterion is identical to that described and formally defined in Section 3.3.1.1.) This annotation identifies an important group of arguments that are to be eliminated. With this annotation, we can represent the two types of function calls by:

$f(t_1..t_m,t_{m+1}{}^{@}..t_n{}^{@},ht_1{}^{*}..ht_q{}^{*})$ or $g(t_0,t_1..t_n,ht_1{}^{*}..ht_q{}^{*})$

The group of higher-order arguments, $ht_1{}^{*}..ht_q{}^{*}$, annotated as * is the primary group targeted for elimination. Also, we have continued to use the previous annotation, @, for marking all *function-type* parameters (without the variable-only criterion) and all *non-linear* ground-type parameters for the f-type function calls. This could help us identify some of the non-recursive function calls which could be *directly unfolded.*

With this annotation scheme, we can define the higher-order specialised form as follows:

Definition 4.6 - Higher-Order Specialised Form

A term is said to be *higher-order specialised* if it satisfies the grammar below.

t::=va | c (t_1,...,t_n) | **lambda** (v_1,...,v_n) => t **end**
| f(t_1..t_m,$t_{m+1}{}^{@}$..$t_n{}^{@}$,$v_1{}^{*}$..$v_q{}^{*}$) | g(t_0,t_1..t_n,$v_1{}^{*}$..$v_q{}^{*}$)
va::= v | va(t_1,..,t_n)

where $v_1..v_q$ are higher-order variables, and g and f are higher-order specialised functions.

Correspondingly, a function is said to be *higher-order specialised,* if its definition is made up of *higher-order specialised* terms.

Notice that the specialised form does not have any *instantiated* higher-order arguments whose parameters satisfy the variable-only criterion. This is because our transformation will be able to eliminate them, leaving behind only arguments that are made up of variables (i.e. $v_1..v_q$ in the above grammar).

4.3.2. Transformation Algorithm, $\mathcal{R}$

With this higher-order specialised form, we postulate that it should be possible to transform any *applyless* expression to this *specialised* form. This suggestion is stated in the following theorem.

Theorem 4.7 - Elimination of Higher-Order Arguments

> Any well-typed, *applyless* expression, composed using occurrences of functions or lambda abstractions which are *higher-order specialised*, can always be transformed, by algorithm $\mathcal{R}$, to an equivalent term that is also *higher-order specialised*, without loss of efficiency (assuming that full laziness is being taken care of).

The proof of this theorem is given later in Section 4.3.6. In this section, we shall just describe the transformation algorithm, $\mathcal{R}$. This algorithm is represented by twelve rules in Figure 4.4. Rules 1 to 7 are similar to the corresponding ones for the $\mathcal{A}$ rules. Each of these rules preserves the outer structure of the current expression by recursively transforming the inner sub-expressions, where applicable. Rules 8 and 9 deal with both f- and g-type function calls which are already in specialised form, namely with variables as actual arguments of its parameters annotated as *. These function calls are left alone with recursive applications of the rules applied to the (other) arguments.

The main elimination steps appear in $\mathcal{R}$ 10 and $\mathcal{R}$ 11 for each function call that is not yet specialised. The three primary steps used are:

1) the direct unfold step ($\mathcal{R}$ 10a),
2) the define&unfold step ($\mathcal{R}$ 10c,$\mathcal{R}$ 11b),
3) the fold step ($\mathcal{R}$ 10b,$\mathcal{R}$ 11a).

If the function call is a non-recursive f-type function call which can be unfolded without any loss of efficiency, then it is *directly unfolded* (see $\mathcal{R}$ 10a). If this is not possible, then a new function definition is *defined* for the function call, with the instantiated higher-order arguments present, followed by an unfold of the call (see $\mathcal{R}$ 10c and $\mathcal{R}$ 11b). To illustrate these two steps, consider the following simple non-recursive function with a non-linear parameter, x (annotated with $^@$), and a function-type parameter, F (annotated with *).

```
- - -apply_f(a,x@,F*) <= a+F(x*x);
```

To eliminate a function call, apply_f(3,w,sqr) where w is a variable, we can apply the direct unfolding step of $\mathcal{R}$ 10a, as shown below.

```
R[[ apply_f(3,w,sqr) ]]    is transformed, using R 10a, to:
                  = R[[ 3+sqr(w*w) ]]
                        :
                  = 3+sqr(w*w)
```

This step successfully eliminates the previous call with a higher-order argument, sqr. However, if we are given another function call, apply_f(3,w*w,sqr), where there is a non-linear argument, w*w, we cannot use the *direct unfold* step to eliminate the instantiated higher-order argument, sqr. This is because we may duplicate w*w. Instead, the define&unfold step of $\mathcal{R}$ 10c should be used to introduce an intermediate function for holding the non-linear argument, w*w. Thus, the sequence of steps to efficiently remove the higher-order argument is as follows.

```
ℜ[[ apply_f(3,w*w,sqr) ]]  is transformed, using ℜ 10c, to:      ! define apply_f_int
                      =  ℜ[[ apply_f_int(3,w*w) ]]
                             :
                      =  apply_f_int(3,w*w)
- - -apply_f_int(a,x)  <=  apply_f(a,x,sqr)                      ! unfold apply_f
                      <=  ℜ[[ a+sqr(x) ]]
                             :
                      <=  a+sqr(x)
```

Apart from function calls with non-linear (large) arguments, the define&unfold step is also used for *recursive* function calls. This is needed to introduce new function definitions that may be subsequently used by steps ℜ 10b and ℜ 11a for *folding* recursive function calls that re-occur. In all these three steps, the resulting expressions, together with the other arguments, are recursively applied by ℜ in order to convert all expressions to the specialised form.

Lastly, a very special rule for this algorithm is ℜ 12.

(12) $\mathcal{R}[\![\, ga\ (t_1,..,t_n)\,]\!] \quad \Leftarrow \quad \mathcal{R}\,[\,\mathcal{A}[\![\mathcal{R}_x[\![\, ga\,]\!](\mathcal{R}[\![\, t_1]\!],..,\mathcal{R}[\![\, t_n\,]\!])\,]\!]\,]$

This rule deals with the *re-appearances* of *general applications* during transformation. These can re-appear because variable applications can be instantiated to general applications by the unfolding of function calls with instantiated higher-order arguments. These re-appearances must be removed by re-invoking the earlier $\mathcal{A}$ rules but we also need a *nested* application of ℜ rules with:

- an *inner* ℜ rule applied to each sub-expression of the general application (with the help of auxiliary rule $\mathcal{R}_x$) prior to $\mathcal{A}$, and
- an *outer* ℜ rule applied to the entire expression, after $\mathcal{A}$ has converted the general application to *applyless* form.

The outer rule is needed because the applyless expression derived after the $\mathcal{A}$ rule may not yet be HO-specialised. Its use can therefore help to remove further higher-order arguments. The utility of the inner rules is a little more subtle. There is an interplay between the $\mathcal{A}$ and outer ℜ rule which may cause non-termination without the presence of the inner rules. This happens whenever a *recursive sub*-expression, which requires an inner ℜ rule for folding, occurs (or will occur) as the leftmost call of its general application. If this inner fold is not done, a looping condition may result in the transformation. This phenomenon is best illustrated with the help of an example (see Section 4.3.5.3). Furthermore, the termination proof of this algorithm (given in Section 4.3.6) also requires the inner rules to be present.

Given $f(v_1..v_n,vh_1..vh_q) = tf$

and $g(p_1,v_1..v_n,vh_1..vh_q) = tg_1$

$\vdots$

$g(p_r,v_1..v_n,vh_1..vh_q) = tg_r$

(1) $\mathcal{R}[\![v]\!] \Leftarrow v$

(2) $\mathcal{R}[\![c(t_1,..,t_n)]\!] \Leftarrow c(\mathcal{R}[\![t_1]\!],..,\mathcal{R}[\![t_n]\!])$

(3) $\mathcal{R}[\![$ lambda $(v_1,..,v_n)$ => tf end $]\!] \Leftarrow$ lambda $(v_1,..,v_n)$ => $\mathcal{R}[\![$ tf $]\!]$ end

(4)[5] $\mathcal{R}[\![$ let v== t_1 in $t_2]\!] \Leftarrow$ let v== $\mathcal{R}[\![t_1]\!]$ in $\mathcal{R}[\![t_2]\!]$

(5) $\mathcal{R}[\![f]\!] \Leftarrow f$

(6) $\mathcal{R}[\![g]\!] \Leftarrow g$

(7) $\mathcal{R}[\![va(t_1,..,t_n)]\!] \Leftarrow \mathcal{R}[\![va]\!](\mathcal{R}[\![t_1]\!],..,\mathcal{R}[\![t_n]\!])$

(8) $\mathcal{R}[\![f(t_{1..m},t_{m+1..n}^{@},vh_{1..q}^{*})]\!] \Leftarrow f(\mathcal{R}[\![t]\!]_{1..m},\mathcal{R}[\![t]\!]_{m+1..n},vh_{1..q})$

(9) $\mathcal{R}[\![g(t_{0..n},vh_{1..q}^{*})]\!] \Leftarrow g(\mathcal{R}[\![t]\!]_{0..n},vh_{1..q})$

(10) $\mathcal{R}[\![f(t_{1..m},t_{m+1..n}^{@}, ht_{1..q}^{*})]\!]$

a) if $t_{m+1..n}$ **are all variables/CFs and** f **is non-recursive then** (unfold)

$\Leftarrow \mathcal{R}[\![tf\ [t/v_{1..m},t/v_{m+1..n},ht/vh_{1..q}]]\!]$

b) if f_old$(v_{1..n},v'_{1..h})$=f$(v_{1..n},ht_{1..q}) \in$ def_set **then** (fold)

$\Leftarrow$ f_old $(\mathcal{R}[\![t]\!]_{1..m}, \mathcal{R}[\![t]\!]_{m+1..n},v'_{1..h})$

where $v'_{1..h}$=vars$[ht_{1..q}]$

c) **otherwise** (define & unfold) $\Leftarrow$ f_new $(\mathcal{R}[\![t]\!]_{1..m},v'_{1..h})$

where $v'_{1..h}$=vars$[ht_{1..q}]$

define f_new$(v_{1..n},v'_{1..h})$= f$(v_{1..n},ht_{1..q})$ ***add to*** def_set

unfold f ***call to give:***

f_new$(v_{1..n},v'_{1..h})$= $\mathcal{R}[\![$ tf $[v/v_{1..n},ht/vh_{1..q}]]\!]$

(11) $\mathcal{R}[\![g(t_{0..n},ht_{1..q}^{*})]\!]$

a) if g_old$(v_{0..n},vh_{1..h})$ = g$(v_{0..n},ht_{1..q}) \in$ def_set **then** (fold)

$\Leftarrow$ g_old $(\mathcal{R}[\![t]\!]_{0..n},v'_{1..h})$

where $v'_{1..h}$=vars $[ht_{1..q}]$

b) **otherwise** (define & unfold) $\Leftarrow$ g_new $(\mathcal{R}[\![t]\!]_{1..n},v'_{1..h})$

where $v'_{1..h}$=vars $[ht_{1..q}]$

define g_new$(v_{0..n},v'_{1..h})$ = g$(v_{0..n},ht_{1..q})$ ***add to*** def_set

unfold g ***call to give:***

g_new$(p_1,v_{1..n},v'_{1..h})$= $\mathcal{R}[\![$ tg_1 $[v/v_{1..n},ht/vh_{1..q}]]\!]$

$\vdots$

g_new$(p_r,v_{1..n},v'_{1..h})$=$\mathcal{R}[\![$ tg_r $[v/v_{1..n},ht/vh_{1..q}]]\!]$

(12) $\mathcal{R}[\![$ ga $(t_1,..,t_n)$ $]\!] \Leftarrow \mathcal{R}[\![\mathcal{A}[\![\mathcal{R}_x[\![$ ga $]\!](\mathcal{R}[\![t_1]\!],..,\mathcal{R}[\![t_n]\!])]\!]]\!]$

WHERE

$\mathcal{R}_x[\![$ f $(t_1,..,t_n)$ $]\!] \Leftarrow \mathcal{R}[\![$ f $(t_1,..,t_n)]\!]$

$\mathcal{R}_x[\![$ g $(t_0,..,t_n)$ $]\!] \Leftarrow \mathcal{R}[\![$ g $(t_0,..,t_n)$ $]\!]$

$\mathcal{R}_x[\![$ lambda $(v_1,..,v_n)$ => tf end $]\!] \Leftarrow$ lambda $(v_1,..,v_n)$ => $\mathcal{R}[\![$ tf $]\!]$ end

$\mathcal{R}_x[\![$ ga $(t_1,..,t_n)$ $]\!] \Leftarrow \mathcal{R}_x[\![$ ga $]\!](\mathcal{R}[\![t_1]\!],..,\mathcal{R}[\![t_n]\!])$

Figure 4.4 - Transformation Rules for Higher-Order Argument Elimnations

[5] let is presently not a construct of the simple higher-order language in Section 4.1.1 but this rule represent how it must be handled if it is present.

4.3.3. A Simple Example

We illustrate the higher-order argument elimination algorithm, $\mathcal{R}$, with a simple function, twos. This function, shown below, produces a list of 2's that is of the same length as its input list, x.

```
dec twos: list(alpha) -> list(num);
- - -twos(x)          <= map(x,lambda (a) => 2 end);
```

The present definition uses a higher-order map call which can be specialised by applying the $\mathcal{R}$ rule, as shown in Figure 4.5. Notice the re-invocation of the $\mathcal{A}$ rule at step $<=_8$ by $\mathcal{R}$ 12. This step is needed to eliminate a lambda application which has re-appeared from an earlier unfold step. The result of this transformation is a new first-order function for twos.

- - -twos(x)	$<=_1$	$\mathcal{R}$[[map(x,lambda (a) => 2 end)]]	! Rule 11b, narrow map
- - -twos([])	$<=_2$	$\mathcal{R}$[[[]]]	! Rule 2
	$<=_3$	[]	
- - -twos(a::x)	$<=_4$	$\mathcal{R}$[[(lambda (a)=> 2 end) (a)::map(x,lambda (a)=> 2 end)]]	! Rule 2
	$<=_5$	$\mathcal{R}$[[(lambda (a)=> 2 end) (a)]] ::$\mathcal{R}$[[map(x,lambda (a)=> 2 end)]]	! Rule 11a fold
	$<=_6$	$\mathcal{R}$[[(lambda (a)=> 2 end) (a)]] ::twos($\mathcal{R}$[[x]])	! Rule 1,12
	$<=_7$	$\mathcal{R}$[[$\mathcal{A}$[[$\mathcal{R}$[[lambda (a)=> 2 end]] ($\mathcal{R}$[[a]])]]]] ::twos(x)	! Rule 1,3,2
	$<=_8$	$\mathcal{R}$[[$\mathcal{A}$[[(lambda (a)=> 2 end) (a)]]]] ::twos(x)	! Apply $\mathcal{A}$ rule
	$<=_9$	$\mathcal{R}$[[2]] ::twos(x)	! Rule 2
	$<=_{10}$	2::twos(x)	! Rule 2

Figure 4.5 - A Simple Example for $\mathcal{R}$

Though our transformation rules have been formulated (in general) to transform individual expressions, we often apply them to just the RHS of function definitions. Because of this, there is an optimisation which is sometimes possible for the define&unfold steps (e.g. $\mathcal{R}$10c and $\mathcal{R}$11b). In particular, if the new function which is about to be defined is identical (modulo renaming) to the current function whose RHS we are transforming, then there is no need for us to introduce another identical definition. This optimised step of not introducing a new identical function is used in the first step, $<=_1$, of the example in Figure 4.5. We shall use such optimised steps, where applicable, in our examples. To distinguish between the *full* and *optimised* define&unfold steps, we shall indicate on the comment lines whenever new function definitions have to be introduced. This should also be clear from context.

4.3.4. An Optimisation for the Nested Rules

The nested application of rules in $\mathcal{R}$ 12 may actually be a source of redundant transformation. Redundancy can occur because the subexpressions that have been transformed/processed by *inner* rules may be re-transformed again by the *outer* rules!

One way of avoiding this redundancy is to mark each subexpression which has been processed by the *inner* $\mathcal{R}$ rules, with a symbol $^{\rho}$, to indicate that the subexpression is already in higher-

order specialised form. This marking is only needed for the inner rule applications of $\mathcal{R}$ 12 and $\mathcal{R}_X$. We place the markings as follows:

$$(12)\ \mathcal{R}[\![\ ga\ (t_1,..,t_n)\ ]\!] \Leftarrow \mathcal{R}[\![\ \mathcal{A}[\![\mathcal{R}_x[\![\ ga\ ]\!](\mathcal{R}[\![\ t_1\ ]\!]^\rho,..,\mathcal{R}[\![\ t_n\ ]\!]^\rho)\ ]\!]\ ]\!]$$

WHERE

$$\mathcal{R}_x[\![\ f\ (t_1,..,t_n)\ ]\!] \Leftarrow \mathcal{R}[\![\ f\ (t_1,..,t_n)\ ]\!]^\rho$$

$$\mathcal{R}_x[\![\ g\ (t_0,..,t_n)\ ]\!] \Leftarrow \mathcal{R}[\![\ g\ (t_0,..,t_n)\ ]\!]^\rho$$

$$\mathcal{R}_x[\![\ \text{lambda}\ (v_1,..,v_n)\Rightarrow tf\ \text{end}\ ]\!] \Leftarrow \text{lambda}\ (v_1,..,v_n)\Rightarrow \mathcal{R}[\![\ tf\ ]\!]\ \text{end}^\rho$$

$$\mathcal{R}_x[\![\ ga\ (t_1,..,t_n)\ ]\!] \Leftarrow \mathcal{R}_x[\![\ ga\ ]\!](\mathcal{R}[\![\ t_1\ ]\!]^\rho,..,\mathcal{R}[\![\ t_n\ ]\!]^\rho)$$

With these markings, all subexpressions that have been marked with $^\rho$ shall be considered as higher-order specialised, and may therefore be left unchanged when they are re-encountered by the outer $\mathcal{R}$ rules. This *no change* step can be handled by a special rule, $\mathcal{R}$ 0, as shown below.

$$(0)\ \mathcal{R}[\![\ expr^\rho\ ]\!] \Leftarrow expr^\rho$$

The markings in each of these subexpressions will *remain* until some substitutions (say from unfolding) has occurred to change the subexpression and hence its form.

4.3.5. Further Examples

In this section, we will show three examples of higher-order argument eliminations, using the optimised $\mathcal{R}$ rules. One of the examples, the last one, is provided to illustrate the necessity of the inner rules of $\mathcal{R}$ 12.

4.3.5.1. mapreduce

Consider a very general higher-order function, mapreduce, which can be used to define two popular higher-order functions, map and reduce, as follows:

```
dec mapreduce: list(alpha) # beta # (alpha # beta -> beta) -> beta;
dec map: list(alpha) # (alpha -> beta) -> list(beta);
dec reduce: list(alpha) # beta # (alpha # beta -> beta) -> beta;
- - -reduce(x,z,F)      <= mapreduce(x,z,lambda (a,x) => F(a,x) end);
- - -map(x,F)          <= mapreduce(x,[],lambda (a,x) => F(a)::x end);
- - -mapreduce([],z,F)  <= z;
- - -mapreduce(a::x,z,F) <= F(a,mapreduce(x,z,F));
```

Both the above map and reduce definitions each has an instantiated mapreduce call which can be specialised. We show, in Figure 4.6, the steps needed for specialising the call in function map. The transformation for reduce follows in a similar way and is not shown.

Notice that the inner rules at step $<=_5$ have managed to place $^\rho$ markings on some of the transformed subexpressions. These marked subexpressions are more efficiently handled by $\mathcal{R}$ 0. The end result of this transformation is a more efficient direct recursive map function, without the need for the intermediate mapreduce function.

This example illustrates that specialisation can also occur among higher-order functions, and not just with first- or second-order functions.

```
- - -map(x,F)        <=1  ℜ[[mapreduce(x,[],lambda (a,x)=>F(a)::x end) ]]   ! ℜ 11b, unfold mapreduce
- - -map([],F)       <=2  ℜ[[ [] ]]                                          ! ℜ 2
                    <=3  []
- - -map(a::x,F)     <=4  ℜ[[ lambda (a,x). F(a)::x end
                            (a,mapreduce(x,[],lambda (a,x) => F(a)::x end)) ]]   ! ℜ 12
                    <=5  ℜ[[ 𝒜[[ ℜ[[ lambda (a,x) => F(a)::x end ]]^ρ (ℜ[[ a]]^ρ
                            ,ℜ [[ mapreduce(x,[],lambda (a,x)=> F(a)::x end)]]^ρ) ]] ]]
                                                                  ! ℜ 3,..,ℜ 1,ℜ 11a fold
                    <=6  ℜ[[ 𝒜[[ lambda (a,x) => F(a)::x end^ρ (a^ρ,map(ℜ[[ x ]],F)^ρ) ]] ]] ! ℜ 1
                    <=7  ℜ[[ 𝒜[[ lambda (a,x) => F(a)::x end^ρ (a^ρ,map( x ,F)^ρ) ]] ]]   ! 𝒜 rule
                    <=8  ℜ[[ F(a^ρ)::map(x,F)^ρ ]]                          ! ℜ 2
                    <=9  ℜ[[ F(a) ]] :: ℜ[[ map(x,F)^ρ ]]                  ! ℜ 7,1;0
                    <=10 F(a)::map(x,F)
```

Figure 4.6 - Application of the Optimised ℜ Rules to the map function

4.3.5.2. twice_map

The next example, twice_map, will show that the ℜ rules can also be applied to expressions which are already in specialised form. In this case, the rule merely passes over its sub-expressions without making any changes, as shown below.

```
- - -twice_map(x,a,b) <= ℜ [[ map(map(x,a),b) ]]      ! ℜ 9
                     <= map(ℜ[[ map(x,a) ]],b )]]    ! ℜ 9
                     <= map(map(ℜ[[ x ]],a) ,b )]]   ! ℜ 1
                     <= map(map(x,a),b)
```

Notice that the composition of the two map calls can actually be fused together. But this must be dealt with by a later deforestation transformation which can work with HO-specialised expressions (see Section 4.5).

4.3.5.3. fix-point operator and ℜ 12

Our last example uses the primitive recursion operator, fix, to define the following recursive function, inf_int, which produces an infinite list of increasing numbers.

```
dec inf_int : num -> list(num);
dec H : (num -> list(num)) -> (num -> list(num));
- - -inf_int      <= fix(H);
- - -H(v)         <= lambda n => n::v(n+1) end;
- - -fix(v)       <= v (fix(v) );
```

The function definition of inf_int uses a higher-order function call, fix(H). This call can be successfully specialised by the ℜ rules, through the sequence of steps given in Figure 4.7. The outcome of this transformation is two higher-order specialised functions. One of them is a new function, inf_int', that is introduced at step <=8 for the general application, inf_int() (n+1).

This example needs the inner rules of $\mathcal{R}$ 12 to transform successfully. In particular, at step $<=_7$, an inner rule is required to cause the recursive subexpression, fix(H), to be folded against the original function inf_int. This folding is essential for allowing the newly specialised function, inf_int, to be utilised in a subsequent transformation step.

- - -inf_int	$<=_1$	$\mathcal{R}$[[fix(H)]]	! $\mathcal{R}$ 10c, unfold fix
	$<=_2$	$\mathcal{R}$[[H (fix(H))]]	! $\mathcal{R}$ 10a
	$<=_3$	$\mathcal{R}$ [[lambda n => n:: fix(H) (n+1) end]]	! $\mathcal{R}$ 3
	$<=_4$	lambda n => $\mathcal{R}$ [[n:: fix(H) (n+1)]]end	! $\mathcal{R}$ 2
	$<=_5$	lambda n => $\mathcal{R}$ [[n]]:: $\mathcal{R}$ [[fix(H) (n+1)]] end	! $\mathcal{R}$ 1
	$<=_6$	lambda n => n :: $\mathcal{R}$ [[fix(H) (n+1)]]end	! $\mathcal{R}$ 12
	$<=_7$	lambda n => n ::	
		$\mathcal{R}$ [[$\mathcal{A}$ [[$\mathcal{R}$[[fix(H)]] ($\mathcal{R}$[[n+1]])]]]]end	! $\mathcal{R}$ 10b fold,$\mathcal{R}$ 8,1,2
	$<=_8$	lambda n => n :: $\mathcal{R}$ [[$\mathcal{A}$[[inf_int()$^{\rho}$ (n+1$^{\rho}$)]]]]end	! $\mathcal{A}$ Rule, define inf_int'
	$<=_9$	lambda n => n :: $\mathcal{R}$ [[inf_int' (n+1 $^{\rho}$)]]end	! $\mathcal{R}$ 10b,$\mathcal{R}$ 0
	$<=_{10}$	lambda n => n :: inf_int' (n+1) end	
- - -inf_int'(n)	$<=_{11}$	$\mathcal{A}$[[inf_int ()(n)]]	! $\mathcal{A}$ Rules
	$<=_{12}$	$\mathcal{A}$[[lambda n => n::inf_int' (n+1) end (n)]]	
	$<=_{13}$	n::inf_int' (n+1)	

Figure 4.7 - Applying the $\mathcal{R}$ Rule to Inf_Int'

Without this fold step, a non-terminating situation could arise as $\mathcal{R}$ and $\mathcal{A}$ go into a loop; where a general application introduced by $\mathcal{R}$, causes $\mathcal{A}$ to result in a function with higher-order argument; which in turn re-introduces the *same* general application.

Such a sequence of non-terminating transformation is illuminated in Figure 4.8, assuming the use of $\mathcal{R}$ 12 *without* its inner rules. Two new function definitions, G and H', are introduced in this transformation. The non-terminating sequence happens at step $<=_{13}$, where the re-occurrence of a general application, fix(H)(n+1+1), forces a fold with the definition of G. This simply results in a *repetition* of steps from $<=_8$ to $<=_{13}$, again and again, causing the body of function inf_int to be gradually unrolled. The cause of this problem is that no inner rules were present to fold an inner recursive expression, fix(H). In fact, we shall see that the need for the inner rules are also reflected in the termination proof of the next section.

Assume the use of $\mathcal{R}$ 12 without the inner $\mathcal{R}$ rules

- - -inf_int	$<=_1$	$\mathcal{R}$[[fix(H)]]	! $\mathcal{R}$ 10c, unfold fix
	$<=_2$	$\mathcal{R}$[[H (fix(H))]]	! $\mathcal{R}$ 10a
	$<=_3$	$\mathcal{R}$ [[lambda n => n:: fix(H) (n+1) end]]	! $\mathcal{R}$ 3
	$<=_4$	lambda n => $\mathcal{R}$ [[n:: fix(H) (n+1)]]end	! $\mathcal{R}$ 2
	$<=_5$	lambda n => $\mathcal{R}$ [[n]]:: $\mathcal{R}$ [[fix(H) (n+1)]] end	! $\mathcal{R}$ 1
	$<=_6$	lambda n => n :: $\mathcal{R}$ [[fix(H) (n+1)]]end	! $\mathcal{R}$ 12
	$<=_7$	lambda n => n :: $\mathcal{R}$ [[$\mathcal{A}$ [[fix(H) (n+1)]]]] end	! $\mathcal{A}$ rule, define G
	$<=_8$	lambda n => n :: $\mathcal{R}$ [[G(H,n+1)]] end	! $\mathcal{R}$ 10a
	$<=_9$	lambda n => n :: $\mathcal{R}$ [[H(fix(H)) (n+1)]]end	! $\mathcal{R}$ 12
	$<=_{10}$	lambda n => n :: $\mathcal{R}$ [[$\mathcal{A}$ [[H(fix(H)) (n+1)]]]] end	! $\mathcal{A}$ Rule
	$<=_{11}$	lambda n => n :: $\mathcal{R}$ [[H'(fix(H),n+1)]]end	! $\mathcal{R}$ 10a
	$<=_{12}$	lambda n => n :: $\mathcal{R}$ [[n+1::fix(H) (n+1+1)]] end	! $\mathcal{R}$ 2,..
	$<=_{13}$	lambda n => n :: n+1::$\mathcal{R}$ [[fix(H) (n+1+1)]] end	
		: ....keep repeating from step $<=_8$ to $<=_{13}$....	
- - -G(v,n)	$<=_{14}$	$\mathcal{A}$[[fix(v) (n)]]	! $\mathcal{A}$ Rule
	$<=_{15}$	$\mathcal{A}$[[v(fix(v))(n)]]	! $\mathcal{A}$ Rule
	$<=_{16}$	v (fix(v)) (n)	! $\mathcal{A}$ Rule
- - -H'(v,n)	$<=_{17}$	$\mathcal{A}$[[H(v) (n)]]	! $\mathcal{A}$ Rule
	$<=_{18}$	$\mathcal{A}$[[lambda n => n:v(n+1) end (n)]]	! $\mathcal{A}$ Rule
	$<=_{19}$	n::v(n+1)	! $\mathcal{A}$ Rule

Figure 4.8 - A Looping Transformation Without the Inner Rules of $\mathcal{R}$ 12

4.3.6. Proof of Theorem 4.7

The complete proof for the Elimination of Higher-Order Arguments Theorem will again need three lemmas, like those used in the proof of the General Application Elimination Theorem.

Lemma 4.8 - On HO-Specialised Form

Each expression which results from algorithm $\mathcal{R}$, under conditions stated in the Elimination of Higher-Order Argument Theorem, will be in higher-order specialised form

Lemma 4.9 - On Efficiency

Ignoring full laziness concern, no loss in efficiency results from the transformation algorithm $\mathcal{R}$

Lemma 4.10 - On the Termination of $\mathcal{R}$

The transformation algorithm $\mathcal{R}$ will terminate under the conditions of the theorem

The first two lemmas can be trivially proved by structural induction on each of the $\mathcal{R}$ rules. We will not show them here.

However, the most difficult result to prove is still the last lemma which deals with termination. This is especially difficult because the nested rules of $\mathcal{R}$ 12 cause the function set, from which new definitions could be composed, to be *increasing* and *not fixed.* An increasing set of

function symbols used by the expressions occurring during transformation *weakens* one of the normal pre-conditions for such termination proofs.

Nevertheless, we can still prove the termination of our rules by considering the outer rules *separately* from the inner rules. In other words, we regard the outer rules from $\mathcal{R}$ 12 to be completely separate applications of the $\mathcal{R}$ transformation.

With this consideration, we shall provide the termination proof in two parts. Firstly, we shall show that each sequence of the inner rule applications (excluding the outer rules in $\mathcal{R}$ 12) can be shown to be finite. Secondly, we shall prove that there could only be a finite number of times in which the outer $\mathcal{R}$ 12 rule could be (recursively) applied.

The first part of the proof is similar to that of previous transformation algorithms. This involves the determination of an upper bound on the size of expressions used for new function definitions and the presence of a well-founded decreasing measure for those recursive rules, other than the define&unfold rules with corresponding fold rules.

The second part of this proof makes novel use of the type structure of expressions, whose *degree* or higher-order*ness* (to be defined later) will be shown to be strictly decreasing with each encounter by the outer rule. Both these proofs would together support Lemma 4.10 stating that transformation algorithm of $\mathcal{R}$ terminates (for all well-typed programs).

Part One - To show that the inner $\mathcal{R}$ rules will always terminates

In the first part of our proof, we define a measure, S_r, for the expressions to be specialised, as follows:

$$
\begin{array}{lcll}
S_r[\![v]\!] & = & 0 & \\
S_r[\![f]\!] & = & 0 & \\
S_r[\![g]\!] & = & 0 & \\
S_r[\![c(t_1,..,t_n)]\!] & = & \max\{ S_r[\![t_1]\!],..,S_r[\![t_n]\!]\} & \\
S_r[\![\text{lambda } (v_1,..,v_n) \Rightarrow \text{tf end}]\!] & = & S_r[\![tf]\!] & \\
S_r[\![\text{let } v== t_1 \text{ in } t_2]\!] & = & \max\{S_r[\![t_1]\!], S_r[\![t_2]\!]\} & \\
S_r[\![va(t_1,..,t_n)]\!] & = & \max\{S_r[\![va]\!],S_r[\![t_1]\!] ,..,S_r[\![t_n]\!]\} & \\
S_r[\![f(t_{1..n},ht_{1..q}^{*})]\!] & = & \max\{S_r[\![t]\!]_{1..n},S_r[\![ht]\!]_{1..q},\boldsymbol{M}[\![ht]\!]_{1..q}\} & \\
S_r[\![g(t_{0..n},ht_{1..q}^{*})]\!] & = & \max\{S_r[\![t]\!]_{0..n},S_r[\![ht]\!]_{1..q},\boldsymbol{M}[\![ht]\!]_{1..q}\} & \\
S_r[\![ga(t_1,..,t_n)]\!] & = & \max\{S_r[\![ga]\!],S_r[\![t_1]\!] ,..,S_r[\![t_n]\!]\} & \\
\boldsymbol{M}[\![v]\!] & = & 0 & \\
\boldsymbol{M}[\![exp]\!] & = & \textit{size}[\![exp]\!] & \text{! where } \textit{size} \text{ is the actual nesting}
\end{array}
$$

This measure is directly related to the maximum size of expressions used for new function definitions during the $\mathfrak{R}$ transformation. Notice in particular, the use the function, *size*, to measure the actual nesting of each instantiated higher-order argument. Also, by the definition of this measure, every expression which is HO-specialised will have an S_r measure of 0. This could be shown by a simple inductive proof on all the possible forms of the HO-specialised grammar (Definition 4.6).

We can prove in Figure 4.9 that this measure, S_r, is directly bounded by all the *inner* $\mathfrak{R}$ rules of the transformation algorithm. The main reasoning used in the proof is that, the unfolding of higher-order function calls, whose function body is HO-specialised (i.e. S_r=0), will never increase the measure of the resulting expression whenever instantiations are present for only function-type, variable-only arguments.

As the size of expressions which could be encountered (for new function definitions) is bounded, the number of define&unfold steps ($\mathfrak{R}$ 10c,11b) for the inner $\mathfrak{R}$ rules will also be finite. This is because the corresponding fold rules ($\mathfrak{R}$ 10b,11a) will always be applied (in preference over the define&unfold step) whenever an expression re-occurs. As for the other recursive rules, they either operate on successively smaller expressions (e.g $\mathfrak{R}$ 2,3,4,7,8,9,10b,11a) or perform direct unfolds for non-recursive f-type function calls ($\mathfrak{R}$10a). We can thus combine the expression size measure with the function hierarchy number to give a well-founded decreasing measure for these recursive rules. The proof for this is again similar to that given in Appendix 5. For the interested reader, the detailed proof appears in [Chin90].

Prove: For each transformation step, $\mathcal{R}[\![t]\!] \Leftarrow \ldots\ldots \mathcal{R}[\![t_i]\!] \ldots\ldots$, where $\mathcal{R}[\![t_i]\!]$ is an *inner* rule application, then $\forall i.\ \mathcal{S}_r[\![t]\!] \geq \mathcal{S}_r[\![t_i]\!]$

Case for $\mathcal{R}$ 1 - v : nothing to prove

Case for $\mathcal{R}$ 2 - $c(t_1,..,t_n)$:

From the definition: $\mathcal{S}_r[\![c(t_1,..,t_n)]\!] = \max\{\mathcal{S}_r[\![t_1]\!],..,\mathcal{S}_r[\![t_n]\!]\}$

- thus $\forall i \in 1..n \quad \mathcal{S}_r[\![t_i]\!] \leq \mathcal{S}_r[\![c(t_1,..,t_n)]\!]$

Case for $\mathcal{R}$ 3 - lambda $(v_1,..,v_n)$ => tf end :

From the definition: $\mathcal{S}_r[\![$ lambda $(v_1,..,v_n)$ => tf end $]\!] = \mathcal{S}_r[\![tf]\!]$

- thus $\mathcal{S}_r[\![tf]\!] \leq \mathcal{S}_r[\![$ lambda $(v_1,..,v_n)$ => tf end$]\!]$

Case for $\mathcal{R}$ 4 - let v== t_1 in t_2 :

From the definition: $\mathcal{S}_r[\![$let v== t_1 in $t_2]\!] = \max\{\mathcal{S}_r[\![t_1]\!],\mathcal{S}_r[\![t_2]\!]\}$

- thus $\forall i \in 1,2 \quad \mathcal{S}_r[\![t_i]\!] \leq \mathcal{S}_r[\![$let v== t_1 in $t_2]\!]$

Case for $\mathcal{R}$ 5 - f : nothing to prove

Case for $\mathcal{R}$ 6 - g : nothing to prove

Case for $\mathcal{R}$ 7 - $va(t_1,..,t_n)$:

From the definition: $\mathcal{S}_r[\![va(t_1,..,t_n)]\!] = \max\{\mathcal{S}_r[\![$ va$]\!],\mathcal{S}_r[\![t_1]\!],..,\mathcal{S}_r[\![t_n]\!]\}$

- thus $\forall i \in 1..n \quad \mathcal{S}_r[\![t_i]\!] \leq \mathcal{S}_r[\![va(t_1,..,t_n)]\!]$ and $\mathcal{S}_r[\![$va $]\!] \leq \mathcal{S}_r[\![va(t_1,..,t_n)]\!]$

Case for $\mathcal{R}$8 - $f(t_{1..n},vh_{1..q}^*)$:

From the definition: $\mathcal{S}_r[\![f(t_{1..n},vh_{1..q})]\!] = \max\{\mathcal{S}_r[\![t]\!]_{1..n}\}$

- thus $\forall i \in 1..n \quad \mathcal{S}_r[\![t_i]\!] \leq \mathcal{S}_r[\![f(t_{1..n},vh_{1..q})]\!]$

Case for $\mathcal{R}$9 - $g(t_{0..n},vh_{1..q}^*)$: similar to $\mathcal{R}$ 8

Case for $\mathcal{R}$10 - $f(t_{1..m}^{@},t_{m+1..n},ht_{1..q}^*)$:

Step a) $t_{m+1..n}$ are variables/CF

From the definition: $\mathcal{S}_r[\![f(t_{1..m}^{@},t_{m+1..n},ht_{1..q}^*)]\!]$

$= \max\{\mathcal{S}_r[\![t]\!]_{1..n},\mathcal{S}_r[\![ht]\!]_{1..q},\boldsymbol{M}[\![ht]\!]_{1..q}\}$

since f is HO-specialised, $\mathcal{S}_r[\![tf]\!]=0$; $= \max\{\mathcal{S}_r[\![tf]\!],\mathcal{S}_r[\![t]\!]_{1..n},\mathcal{S}_r[\![ht]\!]_{1..q},\boldsymbol{M}[\![ht]\!]_{1..q}\}$

- $\geq \max\{\mathcal{S}_r[\![tf[t/v_{1..n},ht/vh_{1..q}]]\!]\}$

Step b) folding step

From the definition: $\mathcal{S}_r[\![f(t_{1..m}^{@},t_{m+1..n},ht_{1..q}^*)]\!] = \max\{\mathcal{S}_r[\![t]\!]_{1..n},\mathcal{S}_r[\![ht]\!]_{1..q},\boldsymbol{M}[\![ht]\!]_{1..q}\}$

- thus $\forall i \in 1..n, \quad \mathcal{S}_r[\![t_i]\!] \leq \mathcal{S}_r[\![f(t_{1..m}^{@},t_{m+1..n},ht_{1..q}^*)]\!]$

Step c) otherwise

From the definition: $\mathcal{S}_r[\![f(t_{1..m}^{@},t_{m+1..n},ht_{1..q}^*)]\!] = \max\{\mathcal{S}_r[\![t]\!]_{1..n},\mathcal{S}_r[\![ht]\!]_{1..q},\boldsymbol{M}[\![ht]\!]_{1..q}\}$

Each extracted argument is always no bigger than the expression

- thus $\forall i \in 1..n, \quad \mathcal{S}_r[\![t_i]\!] \leq \mathcal{S}_r[\![f(t_{1..m}^{@},t_{m+1..n},ht_{1..q}^*)]\!]$

Also, $\mathcal{S}_r[\![f(t_{1..m},t_{m+1..n},ht_{1..q})]\!] \geq \max\{\mathcal{S}_r[\![ve]\!]_{1..n},\mathcal{S}_r[\![ht]\!]_{1..q},\boldsymbol{M}[\![ht]\!]_{1..q}\}$

since f is HO-specialised , $\mathcal{S}_r[\![tf]\!]=0$; $\geq \max\{\mathcal{S}_r[\![tf]\!],\mathcal{S}_r[\![ve]\!]_{1..n},\mathcal{S}_r[\![ht]\!]_{1..q},\boldsymbol{M}[\![ht]\!]_{1..q}\}$

- $\geq \max\{\mathcal{S}_r[\![tf[ve/v_{1..n},ht/vh_{1..q}]]\!]\}$

Case for $\mathcal{R}$11 - $g(t_{0..n},ht_{1..q}^*)$: Similar to 10b and c.

Case for $\mathcal{R}$12 - $ga(t_1,..,t_n)$: Consider just the inner $\mathcal{R}$ rules.

From the definition: $\mathcal{S}_r[\![ga(t_1,..,t_n)]\!] = \max\{\mathcal{S}_r[\![$ ga $]\!],\mathcal{S}_r[\![t]\!]_{1..n}\}$

- thus $\forall i \in 1..n, \mathcal{S}_r[\![t_i]\!] \leq \mathcal{S}_r[\![ga(t_1,..,t_n)]\!]$

Also for $\mathcal{R}$x - $ga(t_1,..,t_n)$:

From the definition: $\mathcal{S}_r[\![ga(t_1,..,t_n)]\!] = \max\{\mathcal{S}_r[\![$ ga $]\!],\mathcal{S}_r[\![t]\!]_{1..n}\}$

- thus $\forall i \in 1..n, \mathcal{S}_r[\![t_i]\!] \leq \mathcal{S}_r[\![ga(t_1,..,t_n)]\!]$ and $\mathcal{S}_r[\![$ ga $]\!] \leq \mathcal{S}_r[\![ga(t_1,..,t_n)]\!]$

Figure 4.9 - Proof that the Measure, $\mathcal{S}_r$, Is Bounded by All the Inner $\mathcal{R}$ rules

Part Two - To show that for any given expression, there is a *finite* number of recursive applications for the outer $\mathcal{R}$ rule

For the second part of our proof, we must show that the number of times in which the outer $\mathcal{R}$ rules could be (recursively) applied is finite for any given expression. In order to prove this, we must show that there is another measure encountered by each of the outer $\mathcal{R}$ rules, which is *strictly* decreasing (not just bounded!) with respect to the original expression.

To show this, consider a type structure, τ, which can be obtained from each expression, e, as follows, *type* $[\![e]\!]=\tau$. This type structure can be described by the following simple grammar which supports either ground-type (includes constructor types like list and trees), tuple-type or function-type.

$$\tau ::= \text{ground_type} \mid (\tau_1,...,\tau_n) \mid \tau_1 \rightarrow \tau_2$$

With this type, we define the *order* (or *degree*) of types in each expression by:

1+the highest possible degree in the expression's arguments

using the equations:

$$\begin{aligned} &\mathit{degree}\,[\![\,\text{ground_type}\,]\!] &&= 0 \\ &\mathit{degree}\,[\![\,(\tau_1,...,\tau_n)\,]\!] &&= \max\{\mathit{degree}\,[\![\,\tau_1]\!],...,\mathit{degree}\,[\![\,\tau_n]\!]\} \\ &\mathit{degree}\,[\![\,\tau_1 \rightarrow \tau_2\,]\!] &&= \max\{1+\mathit{degree}\,[\![\,\tau_1]\!],\mathit{degree}\,[\![\,\tau_2]\!]\} \end{aligned}$$

This degree of type will always be a finite value for each *well-typed* expression (with a finite type structure). With this finite degree, we can define a new measure, called $\mathcal{S}_t$, which keeps a count of the maximum possible degree in the instantiated higher-order arguments that are to be eliminated. This measure, $\mathcal{S}_t$, will be almost identical to the previous bounded measure, $\mathcal{S}_r$, except for the meta-function, $\boldsymbol{M}$, which is now given as follows:

$$\begin{aligned} &\boldsymbol{M}\,[\![\,v\,]\!] &&= 0 \\ &\boldsymbol{M}\,[\![\,\text{exp}\,]\!] &&= \mathit{degree}\,[\![\mathit{type}\,[\![\,\text{exp}]\!]]\!] \quad \text{! type of expression} \end{aligned}$$

Such a new measure, $\mathcal{S}_t$, can be similarly shown to be bounded by all the inner rule applications. More importantly, we can also show that each of the outer $\mathcal{R}$ rule applications will always have to deal with an expression whose degree that is *strictly smaller* than that of the original expression. This is a very important result for the outer $\mathcal{R}$ rules because the presence of a strictly decreasing sequence implies that the number of recursive outer $\mathcal{R}$ applications is correspondingly finite. As a result, the nested $\mathcal{R}$ algorithm will terminate because this decreasing sequence could only reduce to 0 (i.e. first-order specialised form when the algorithm will definitely terminate).

More specifically, consider an initial expression, e, which contains a number of instantiated higher-order arguments, $ht_1..ht_z$, that are to be eliminated. According to the definition of $\mathcal{S}_t$, the maximum degree of this expression is $\max\{\mathit{degree}\,[\![\,\mathit{type}\,[\![\,ht_1]\!]]\!],...,\mathit{degree}\,[\![\,\mathit{type}\,[\![\,ht_z]\!]]\!]\}$. Now, consider each general application, $ht_i(t''_1,...,t''_m)..(t_1,...,t_n)$, which arises (from an instantiation of a variable application) during the transformation of the initial expression, e. Each of these general applications would have been instantiated by one of the higher-order arguments, $ht_1..ht_z$, as

indicated by ht_i. It will also be handled by $\mathcal{R}$ 12 which will use its inner $\mathcal{R}$ rules to convert all the arguments, $t'_1..t'_m..t_1..t_n$, and the leftmost sub-expression, ht_i, to the specialised form (i.e. S_t=0). With these inner rule applications, the function call for the outer $\mathcal{R}$ rule shall be:

$$\mathcal{A}[\![\mathcal{R}[\![ht_i]\!](\mathcal{R}[\![t'_1]\!],...,\mathcal{R}[\![t'_m]\!])..(\mathcal{R}[\![t_1]\!],...,\mathcal{R}[\![t_n]\!])]\!] \qquad (3.1)$$

after the $\mathcal{A}$ rule has removed its general application. Now, this function call which will be of the form, say $p(at_1..at_n,bt_1..bt_q^*)$, may have a number of instantiated function-type arguments, say $bt_1..bt_q$. These instantiated function-type arguments will have to come from the arguments of the general application whose arguments include $\mathcal{R}[\![t'_1]\!],...,\mathcal{R}[\![t'_m]\!]..\mathcal{R}[\![t_1]\!],...,\mathcal{R}[\![t_n]\!]$, and the arguments of $\mathcal{R}[\![ht_i]\!]$ (if this is a function call rather than a lambda abstraction). The reason for this is that the $\mathcal{A}$ rule never eliminates any of the instantiated function-type arguments. As a result, the S_t measure of (3.1) is always less than degree[[type [[ht_i]]]]. This is because:

$S_t[\![3.1]\!] = S_t[\![p(at_1..at_n,bt_1..bt_q^*)]\!]$

From Definition of S_t: $= \max\{S_t[\![at]\!]_{1..n}, S_t[\![bt]\!]_{1..q}, \boldsymbol{M}[\![bt]\!]_{1..q}\}$

Arguments, $at_1..at_n,bt_1..bt_q$ are in HO-specialised form after processing by the inner $\mathcal{R}$ rules:

$= \max\{\boldsymbol{M}[\![bt]\!]_{1..q}\}$

The instantiated function-type arguments have not been removed by $\mathcal{A}$, thus:

$\leq \max\{\boldsymbol{M}[\![\mathcal{R}[\![t']\!]]\!]_{1..m},..., \boldsymbol{M}[\![\mathcal{R}[\![t]\!]]\!]_{1..n}, \boldsymbol{M}[\![\text{arguments of } ht_i]\!]\}$

According to the definition of degree, the order of each argument is always less than that of its function:

$< \boldsymbol{M}[\![\mathcal{R}[\![ht_i]\!]]\!]$

Now, $\boldsymbol{M}$ (and S_t) is bounded by the application of $\mathcal{R}$:

$< \boldsymbol{M}[\![ht_i]\!]$

From Definition of $\boldsymbol{M}$ $< \text{degree}[\![\text{type}[\![ht_i]\!]]\!]$

$< \max\{\text{degree}[\![\text{type}[\![ht_1]\!]]\!],...,\text{degree}[\![\text{type}[\![ht_z]\!]]\!]\}$

Hence, the degree of the expression (3.1) is strictly less than the degree of the original expression, e, from which expression (3.1) was obtained.

With this result, each outer $\mathcal{R}$ rule application will always have to deal with an expression whose degree is even smaller than the previous outer $\mathcal{R}$ rule application. As there could only be a finite sequence of decreasing S_t measure (to a value of at most 0, before we get to a first-order term), the number of outer $\mathcal{R}$ rule applications will always be finite. This result should not be surprising because the essence of higher-order argument elimination is to reduce the *order* of our expressions and this could not go on forever with well-typed expressions.

Thus, together with the bound on the size of expressions for inner $\mathcal{R}$ rules, we have succeeded in showing that the transformation algorithm, $\mathcal{R}$ (with nested rules), will always terminate for all well-typed expressions.

4.4. Full Laziness Concern

The *fully lazy lambda lifting* technique of Hughes works by extracting out maximal free expressions during lambda lifting, in order to convert all lambda abstractions to fully lazy supercombinators [Hughes82]. This is a useful technique because:

1) It helps to preserve full laziness in higher-order programs by allowing common free expressions to be shared through the call-by-need parameter passing mechanism
2) It removes free variables from lambda abstractions. This allows the parameter instantiation operations for function calls to be *compiled* rather than *interpreted.*

However, the classic fully lazy lambda lifting technique is *not compatible* with the higher-order removal transformation. The reason is that it works by extracting out partial applications for sharing - causing functions with higher-order arguments to be introduced. This contradicts with the goal of higher-order removal transformation.

Apart from this incompatibility, there are also a number of shortcomings in the original fully lazy lambda lifting technique. The main ones are :

- It is dependent on the *positional order* of parameters for maximum laziness. Different orderings of parameters can often cause different degree of laziness. Some of these may not be optimal for all cases (see [PeytonJ87]).
- It is possible for unnecessary intermediate functions to be introduced by extracting out partially applied expressions which are free but *unshared.* This can result in programs that are less efficient (see [Hudak85]).
- It is possible for space-leak to occur for partially applied recursive function calls (see [PeytonJ87] and Section 4.4.1.5 later).

These three shortcomings can be overcome by a new transformational approach to full laziness which extracts out *ground-type* MFEs (GT-MFEs) rather than any MFEs. However, this selective extraction of only GT-MFEs may itself suffer from a minor loss in efficiency under certain circumstances. We shall show two examples of this minor loss in Section 4.4.5 and will argue that the loss can be either *avoided* (by abstracting with **let** constructs), *ignored* (as loss is normally limited) or *partially rectified* (with the help of further transformations).

We begin by presenting an alternative transformational approach for preserving full laziness. This approach is made up of four main techniques which together could convert each *applyless* expression to its *fully lazy* equivalent. We informally illustrate and describe these techniques before presenting a formal transformation algorithm (containing three sets of rules) to implement our full laziness transformation. Once this is done, we shall show how the higher-order argument elimination algorithm, $\mathfrak{R}$, can be appropriately modified to preserve full laziness.

4.4.1. Various Techniques for Full Laziness

Our approach to full laziness will use a collection of four safe techniques to help float out both *explicit* and *implicit* GT-MFEs. These four techniques are:

- **L(a)** Extract out GT-MFEs from lambda abstractions;
- **L(b)** Convert grounded GT-MFEs to constant functions[6] (CFs);
- **L(c)** *Direct* unfolds of non-recursive function calls, where possible, to reveal implicit GT-MFEs;
- **L(d)** *Re-define* certain non-recursive functions, to enable unfolds that could reveal further implicit GT-MFEs.

We shall often refer to these four techniques as **L(a)**, **L(b)**, **L(c)** and **L(d)** respectively. We describe (with the help of examples) each of these techniques in the next few sub-sections.

4.4.1.1. L(a) Extract out GT-MFEs from lambda abstractions.

In the applyless form, the most appropriate places to identify and extract out GT-MFEs are the lambda abstractions. These are the places where free expressions may exist and are most likely to be duplicated during execution. Our transformation should search and extract out all GT-MFEs, whenever possible, from these lambda abstractions.

As an example, consider the function, F, below.

```
dec F : list(num) # num # num -> list(num);
- - -F(x,A,B)          <=   map(x,lambda a => a+A*B end);
```

This function is not fully lazy because of the GT-MFE, A*B, in the lambda abstraction of map call. To make this function fully lazy, we must extract out its free expression through an intermediate function, say int_F, to give:

```
- - -F(x,A,B)          <=   map(x,int_F(A*B));
- - -int_F(Z)          <=   lambda a => a+Z end;
```

We suggest that it is quite acceptable to consider only *ground-type* MFEs for extraction. This is true as long as there is a way of extracting out all such MFEs, that may be *explicitly* present or *implicitly* hidden under levels of function calls. The rationale behind this is that ground-type MFEs are the major source of redundancy for duplicated expressions. In contrast, function-type MFEs require additional inputs before being complete for computation. We shall see in Section 4.4.5 that ignoring the function-type MFEs will only suffer from a minor loss of efficiency.

4.4.1.2. L(b) Convert grounded GT-MFEs to CFs

Ground sub-expressions (i.e. expressions without any free variables) are also considered to be free expressions. Note the distinction between *ground-type* and *ground*. An expression is *ground* if there are no free variables in the expression. This same expression could also be *ground-type* provided it returns an object-type result, like 2, ['a','b'], as opposed to a function-type result, like lambda x=>x+2 end.

---

[6] ground-type functions without any parameters

However, instead of extracting them out and risk the introduction of (unnecessary) intermediate functions for their lambda abstractions, we can introduce *constant functions* (CF) in their places.

As an example, consider the function F2, as shown below.

```
dec F2 : list(num) -> list(num);
- - -F2(x)          <=  map(x,lambda a => a+sqr(4) end);
```

This function has a grounded GT-MFE, sqr(4), in its lambda abstraction. Instead of extracting it out as an argument of an intermediate function, we can simply introduce a constant function, sq4, in its place as follows:

```
- - -F2(x)          <=  map(x,lambda a => a+sq4 end);
- - -sq4            <=  sqr(4);
```

By assuming that this constant function will only be evaluated once, the new program is, by all purpose, fully lazy. This technique is similar to the one described in [PeytonJ87] for introducing CAF[7] during fully lazy lambda lifting. We differ from this primarily in our concern with only expressions of *ground-type.*

4.4.1.3. L(c) Direct Unfolds for non-recursive function calls

Not all GT-MFEs are *explicitly* available for extraction in lambda abstractions. Some of them may be hidden *implicitly* underneath function calls. An example of this is the function F3, as shown below, with a non-recursive G3 function call, in its lambda abstraction.

```
- - -F3(x,A,B)      <=  map(x,lambda a=> G3(a,A,B) end);
- - -G3(a,A,B)      <=  a+A*B;
```

This non-recursive call currently contains an *implicit* GT-MFE, A*B. One way of making this MFE *explicit* is to directly unfold the non-recursive G3 function call, as follows:

```
- - -F3(x,A,B)      <=  map(x,lambda a=> G3(a,A,B) end);      ! unfold G3
                   <=  map(x,lambda a=> a+A*B end);
```

This method of unfolding improves efficiency and helps to reveal GT-MFEs for extraction by the first technique, **L(a)**.

However, we may only apply direct unfolding to *non-recursive* f-type function calls which do not result in any loss of efficiency (from non-linear arguments). This restriction of only permitting the unfolds of *non-recursive* f-type function calls can avoid two problems. Firstly, recursive functions may cause non-termination if they are considered for direct unfolding. Secondly, g-type functions may cause a violation of our language restriction (of one pattern-matching parameter per g-type function) if they are directly unfolded.

4.4.1.4. L(d) Indirect Unfolds by Re-Defining non-recursive functions.

Lastly, not all non-recursive function calls can be directly unfolded to reveal implicit GT-MFEs. Some of these calls may be prevented from direct unfolding by either being g-type function calls or simply contain non-linear bounded arguments which are preventing the direct unfolds. An example is the following function, F4, which has a pattern-matching G4 function call that is not

[7] Constant Applicative Form - an expression (ground or function-type) without any free variables

allowed to be directly unfolded.

```
- - -F4(x,A,B)        <= map(x,lambda a=> G4(a,A,B) end);
- - -G4(true,A,B)     <= A*B;
- - -G4(false,A,B)    <= A;
```

This G4 call hides an implicit GT-MFE, A*B. One way of making this free expression explicit is to *re-define* the non-recursive function, G4, to a semantically equivalent one, G4', as shown below.

```
- - -F4(x,A,B)        <= map(x,lambda a=> G4 '(a,A,B) end);
- - -G4 '(a,A,B)      <= G4 "(a,A*B,A);
- - -G4"(true,AB,A)   <= AB;
- - -G4"(false,AB,A)  <= A;
```

This re-definition floats out the implicit GT-MFE and enables the new semantically equivalent function to become unfoldable again. Unfolding this call will help to reveal the trapped free expression, A*B, as shown below.

```
- - -F4(x,A,B)        <= map(x,lambda a=> G4'(a,A,B) end);   ! unfold G4'
                      <= map(x,lambda a=> G4"(a,A*B,A) end);
```

4.4.1.5. Recursive Functions and the Space Leak Problem.

Both the direct and indirect unfolding techniques for revealing implicit free expressions are only suitable for non-recursive functions. Recursive functions cannot be handled by these techniques because we may run into non-termination problem. As a result, we may sometimes be prevented from maximising the laziness of recursive functions.

However, this may be a blessing in disguise because recursive functions can cause *space leak* problems when they are made fully lazy. This is possible because partially applied recursive function calls may be completely unrolled (unfolded) into a large graph/code which awaits additional arguments. Such unrolled codes may help to reduce computation but they also cause excessive storage spaces to be consumed unexpectedly - hence the space leak problem. We illustrate this phenomenon with the following example from [PeytonJ87].

```
- - -p                <= drop 1000;
- - -drop(n,xs)       <= if n=0 then xs else drop(n-1,tl(xs));
```

The above function, p, contains a partially applied function call, drop 1000, which could be called (or shared) from a number of different places. Assuming the possibility of using partial applications, this program could be converted by the fully lazy lambda lifting technique to:

```
- - -p                <= drop 1000 ;
- - -drop n           <= L n=0 (drop n-1) ;
- - -L bool nxt xs     <= if bool then xs else nxt (tl xs) ;
```

With this fully lazy program, the function call drop 1000 (in p) could be fully reduced or unrolled by one of its invocations, as follows:

```
p                    => drop 1000
                     => L false drop 999
                     => L false L false drop 998
                           :
                     => L false L false L false ..... drop 0
                     => L false L false L false ..... L true drop -1
```

This fully unrolled code (the last line) could be retained for re-use by the other calls of function p. The sharing of this unrolled code helps to avoid some re-computation (involved in building this code/graph). However, it also takes up considerable amount of storage space and may even force the program to fail should insufficient memory space be available.

In contrast to the fully lazy recursive program, the original function of drop is actually tail-recursive and can potentially be optimised into an iterative loop which requires only constant space for execution. This could provide for a space and time-efficient program because of the reduced memory management overheads. Other programs may not be tail-recursive but could still have the space-leak problem avoided through a *sacrifice* of this form of laziness - which is peculiar to partially applied *recursive* function calls.

4.4.2. Full Laziness Transformation Algorithm

The above collection of techniques can be combined together to form the full laziness transformation. In particular, we can formalise it into a transformational algorithm that is made up of three main sets of syntax-directed rules, $\mathcal{L}$, $\mathcal{E}$ and $\mathcal{F}$. The first set of rules, $\mathcal{L}$, is the main set of rules for full laziness transformation. This set of rules performs direct unfolding and invokes the second set of rules, $\mathcal{E}$, at each lambda abstraction for the extraction of GT-MFEs. The last set of rules, $\mathcal{F}$, is used by $\mathcal{E}$ to find the GT-MFEs which may either be *explicitly* present or *implicitly* hidden under nesting of function calls. The $\mathcal{F}$ rules may also be involved in re-defining certain non-recursive functions, which could not be unfolded, in order to extract out the implicit GT-MFEs. These three sets of rules are formally presented and described in the next few sub-sections.

4.4.2.1. The $\mathcal{L}$ rules

The main set of rules for full laziness transformation, $\mathcal{L}$, must be applied to each RHS of our functions. This must be applied in a bottom-up manner to all functions of our program, in order to convert each of them to fully lazy form. The set of $\mathcal{L}$ rules is given in Figure 4.10. It performs two major tasks, namely:

- The direct unfolding of non-recursive f-type function calls, whenever permitted (see $\mathcal{L}$ 8a). This implements the technique of **L(c)**.
- Invokes the second set of rules, $\mathcal{E}$, at lambda abstractions ($\mathcal{L}$ 0 and $\mathcal{L}$ 7a) for the extraction of free expressions. This leads to the techniques of **L(a)**, **L(b)**, as well as **L(d)**.

Note the use of @ parameter annotation (in $\mathcal{L}$ 8) to help us identify all *non-linear* ground-type and all *function-type* parameters. This annotation is needed to avoid the unfolding of function calls which may cause losses in efficiency (or result in further general applications). Also, we avoid the unfolding of functions which contain lambda abstractions because such functions will not contain any GT-MFEs after applying a bottom-up transformation.

In order to achieve these two objectives, the $\mathcal{L}$ rules are structured into three different cases, depending on the first parameter, x, of the rule. The first case, x=0 (see $\mathcal{L}$ 0), is used when we

are dealing with the body of the *outermost* lambda abstraction (the function itself). For example, each f-type and g-type function, shown below:

$$\begin{aligned} &---f(t_1,...,t_n) && \Leftarrow\ tf \\ &---g(p_1,t_1,...,t_n) && \Leftarrow\ tg_1 \\ &\quad\vdots && \quad\vdots \\ &---g(p_r,t_1,...,t_n) && \Leftarrow\ tg_r \end{aligned}$$

must be transformed to its corresponding fully lazy form, as follows:

$$\begin{aligned} &---f(t_1,...,t_n) && \Leftarrow\ \mathcal{L}\ 0\ [\![tf]\!] \\ &---g(p_1,t_1,...,t_n) && \Leftarrow\ \mathcal{L}\ 0\ [\![tg_1]\!] \\ &\quad\vdots && \quad\vdots \\ &---g(p_r,t_1,...,t_n) && \Leftarrow\ \mathcal{L}\ 0\ [\![tg_r]\!] \end{aligned}$$

Even though such outermost lambda abstractions do not contain any free variables, there is still a possibility of finding *ground* GT-MFEs which should be converted to constant functions. This is why the $\mathcal{E}$ extraction rule is invoked by $\mathcal{L}$ 0. The second case, x=1, is used principally to locate each *inner* lambda abstraction (see $\mathcal{L}$ 7a) for free expression extraction; while the last case, x=2, is used purely to search for non-recursive function calls which can be unfolded. This last case is invoked by the $\mathcal{E}$ rule (see Figure 4.11, later). It simply looks for direct unfolds and does not invoke the $\mathcal{E}$ extraction rule when encountering lambda abstractions (see $\mathcal{L}$ 7b).

$$\begin{aligned}
&(0)\ \mathcal{L}\ 0\ [\![expr]\!] && \Leftarrow && \mathcal{E}\ [\![\mathcal{L}\ 1\ [\![expr]\!]]\!]\ vars[expr] \\
&(1)\ \mathcal{L}\ x\ [\![v]\!] && \Leftarrow && v \\
&(2)\ \mathcal{L}\ x\ [\![c(t_1,...,t_n)]\!] && \Leftarrow && c\ (\mathcal{L}\ x\ [\![t_1]\!],...,\mathcal{L}\ x\ [\![t_n]\!]) \\
&(3)\ \mathcal{L}\ x\ [\![\text{let } v==t_1 \text{ in } t_2]\!] && \Leftarrow && \text{let } v==\mathcal{L}\ x\ [\![t_1]\!] \text{ in } \mathcal{L}\ x\ [\![t_2]\!] \\
&(4)\ \mathcal{L}\ x\ [\![f]\!] && \Leftarrow && f \\
&(5)\ \mathcal{L}\ x\ [\![g]\!] && \Leftarrow && g \\
&(6)\ \mathcal{L}\ x\ [\![va(t_1,...,t_n)]\!] && \Leftarrow && \mathcal{L}\ x\ [\![va]\!]\ (\mathcal{L}\ x\ [\![t_1]\!],...,\mathcal{L}\ x\ [\![t_n]\!]) \\
&(7)\ \mathcal{L}\ x\ [\![\text{lambda } (v_1,...,v_n) \Rightarrow tf \text{ end}]\!] && && \\
&\quad \textbf{a) if x=1} && \Leftarrow && \mathcal{E}\ [\![\text{lambda } (v_1,...,v_n) \Rightarrow \mathcal{L}\ 1\ [\![tf]\!] \text{ end}]\!]\ \{\} \\
&\quad \textbf{b) if x=2} && \Leftarrow && \text{lambda } (v_1,...,v_n) \Rightarrow \mathcal{L}\ 2\ [\![tf]\!] \text{ end} \\
&(8)\ \mathcal{L}\ x\ [\![f(t_{1..m}{}^{@},t_{m+1..n})]\!] && && \\
&\quad \textbf{a) if f is a non-recursive, non-lambda function and } t_{m+1..n} \textbf{ are variables/CF then unfold} && && \\
&\quad && \Leftarrow && \mathcal{L}\ x\ [\![tf\ [t/v_{1..m},t/v_{m+1..n}]]\!] \\
&\quad \textbf{b) otherwise} && \Leftarrow && f(\mathcal{L}\ x\ [\![t]\!]_{1..m}{}^{@},\mathcal{L}\ x\ [\![t]\!]_{m+1..n}) \\
&(9)\ \mathcal{L}\ x\ [\![g(t_0,...,t_n)]\!] && \Leftarrow && g(\mathcal{L}\ x\ [\![t_0]\!],...,\mathcal{L}\ x\ [\![t_n]\!])
\end{aligned}$$

Figure 4.10 - Full Laziness Algorithm, $\mathcal{L}$, for Unfolding and Extraction

To illustrate the use of the $\mathcal{L}$ rules, let us consider the following simple program.

```
---P(x)          <= dup(lambda a =>h1(x,a) end);
---h1(x,a)       <= h2(x*x,sqr(a));
---h2(x,a)       <= x*x+(a+a);
---dup(v)        <= v(3)+v(4);
```

The only function with a lambda abstraction is the function P. To make this program fully lazy, we shall apply the $\mathcal{L}$ rule to its RHS term through the following sequence of steps.

```
- - -P(x)      <=  ℒ 0 [[ dup(lambda a =>h1(x,a) end) ]]                          ! ℒ 0
               <=  ℰ [[ ℒ 1 [[ dup(lambda a =>h1(x,a) end) ]] ]] {x}             ! ℒ 8b
               <=  ℰ [[ dup(ℒ 1 [[ lambda a =>h1(x,a) end ]] ) ]] {x}            ! ℒ 7a
               <=  ℰ [[ dup(ℰ [[ lambda a =>ℒ 1 [[ h1(x,a)]] end ]] {} ) ]] {x}  ! ℒ 8a, unfold
               <=  ℰ [[ dup(ℰ [[ lambda a =>ℒ 1 [[ h2(x*x,sqr(a)) ]] end ]] {} ) ]] {x}  ! ℒ 8b,..
                                  :
               <=  ℰ [[ dup(ℰ [[ lambda a => h2(x*x,sqr(a)) end ]] {} ) ]] {x}
```

These steps have accomplished two tasks. The first task is the direct unfolding of function call, h1(x,a), to h2(x*x,sqr(a)). This helps to reveal the implicit GT-MFE, x*x. The second task is the strategic placements of two $\mathcal{E}$ rule applications, ready to extract out all the explicit and implicit GT-MFEs. The outer $\mathcal{E}$ rule application will be used to extract out all grounded GT-MFEs because the complete set of variables, namely {x}, is considered to be bounded. The inner $\mathcal{E}$ rule application will be used to extract out GT-MFEs from the inner lambda abstraction.

We shall look at the $\mathcal{E}$ rule next and will continue (later) with this example in order to illustrate how the GT-MFEs are extracted.

4.4.2.2. The $\mathcal{E}$ rules

Both the explicit and the implicit GT-MFEs must be extracted by the $\mathcal{E}$ rules of Figure 4.11. These rules iteratively extract out GT-MFEs *relative* to a set of bounded variables, bset.

```
ℰ [[ expr ]] bset
     let (expr^, fe1..p,ve1..p)==ℱ [[ expr ]] bset in
1)   if p≥1
     let (expr^', fe'1..q,ve'1..q)==Cℱ [[ (expr^, fe1..p,ve1..p)]] in
     a) if q≥1                    <=   f_new(fe'1..q,vr1..z)
                where vr1..z==vars [expr^' ]- ve'1..q
                define f_new(ve'1..q,vr1..z) = ℰ [[ ℒ 2 [[ expr^' ]] ]] {ve'1..q,vr1..z}
     b) if q=0                    <=   ℰ [[ ℒ 2 [[ expr^' ]] ]] bset
2)   p=0                          <=   expr
WHERE
Cℱ [[ (ex,fe::fe_lt,ve::ve_lt) ]]
     a) if fe is grounded then    <=   Cℱ [[ (ex[f_CF/fe],fe_lt,ve_lt]]
                                  define  f_CF = 𝒜 [[ fe ]]        ! constant function
     b) otherwise                 <=   (ex', 𝒜 [[ fe ]]::fe_lt',ve::ve_lt',)
                                  where (ex',fe_lt',ve_lt')==Cℱ [[ (ex,fe_lt,ve_lt) ]]
```

Figure 4.11- $\mathcal{E}$ Rules for the Repeated Extraction of GT-MFEs

The actual extraction is done with the help of the $\mathcal{F}$ rule. This is applied until no more GT-MFEs are left in the given expression. This iterative step occurs in $\mathcal{E}$ 1a and $\mathcal{E}$ 1b, and terminates with $\mathcal{E}$ 2 when no more GT-MFEs are found. Also, for each GT-MFE extracted, the rule uses an auxiliary meta-function, $C\mathcal{F}$, to check if the MFE is:

1) a ground GT-MFE, and/or
2) a general application.

If it is a *grounded* GT-MFE, the $C\mathcal{F}$ rule will convert it into a constant function. If it is a *general application*, the $\mathcal{A}$ rule will be used to transform it to applyless form. There is a possibility of

getting general applications because we are performing this full laziness transformation before the elimination of function-type arguments. These function-type arguments may cause the instantiations of variable applications to general applications during the extraction process.

The earlier example from the previous section contained two un-evaluated $\mathcal{E}$ rule applications, as shown below.

```
---P(x)          <= E [[ dup(E [[ lambda a => h2(x*x,sqr(a)) end ]] {} ) {x}
```

We are now ready to show the (abbreviated) steps for the extractions of GT-MFEs by the $\mathcal{E}$ rules. These extractions will help to produce a fully lazy program. At each invocation of the $\mathcal{E}$ rule, the $\mathcal{F}$ rule is called to search for both explicit and implicit GT-MFEs. In this illustration, we shall show the results of the $\mathcal{F}$ rule application as brief comments (the details are given in the next section). Each of these $\mathcal{F}$ applications returns a tuple of three items, namely the new expression after extraction, a list of extracted GT-MFEs and a corresponding list of new variables which replace the extracted GT-MFEs. They are important because the presence or absence of GT-MFEs determines precisely what must be done by the $\mathcal{E}$ rule, as illustrated below.

```
---P(x)          <= E [[ dup(E [[ lambda a => h2(x*x,sqr(a)) end ]] {} ) {x}      ! E1a,extract
                      !F [[ lambda a => h2(x*x,sqr(a)) end ]] {}
                      !    returns    (lambda a => h2(A,sqr(a)) end,[x*x],[A])
                 <= E [[ dup( fn1(x*x) ) ]] {x}                                  ! E 2,terminate
                      !F [[ dup( fn1(x*x) ) ]] {x}
                      !    returns    (dup( fn1(x*x) ),[],[])
                 <= dup( fn1(x*x) )
Define fn1 by E
---fn1(A)        <= E [[ lambda a => L 2[[ h2(A,sqr(a)) ]] end ]] {}              ! L rule,try unfold
                 <= E [[ lambda a => h2(A,sqr(a)) end ]] {}                       ! E1a,extract
                      !F [[ lambda a => h2(A,sqr(a)) end ]] {}
                      !    returns    (lambda a => h2'(B,sqr(a)) end,[A*A],[B])
                 <= fn2(A*A)
Define fn2 by E
---fn2(B)        <= E [[ lambda a => L 2[[ h2'(B,sqr(a)) ]] end ]] {}             ! L rule,try unfold
                 <= E [[ lambda a => h2'(B,sqr(a)) end ]] {}                      ! E2,terminate
                      !F [[ lambda a => h2'(B,sqr(a)) end ]] {}
                      !    returns    ( lambda a => h2'(B,sqr(a)) end,[],[])
                 <= lambda a => h2'(B,sqr(a)) end
Re-Define h2 to h2' by F
---h2'(B,a)      <= B+(a+a);
```

The result of the above extractions is the following fully lazy version of P.

```
---P(x)          <= dup( fn1(x*x) );
---fn1(A)        <= fn2(A*A);
---fn2(B)        <= lambda a => h2'(B,sqr(a)) end ;
---h2'(B,a)      <= B+(a+a);
```

Two intermediate functions, fn1 and fn2, were introduced by the $\mathcal{E}$ rules to hold the extracted GT-MFEs. Also, another new function, h2', was introduced by the $\mathcal{F}$ rule to help extract out the implicit GT-MFE from a function call, h2(A,sqr(a)), which could not be directly unfolded. Each of $\mathcal{E}$ rule terminates whenever no more GT-MFEs are found by the $\mathcal{F}$ rule.

4.4.2.3. The $\mathcal{F}$ rules

The search for GT-MFEs is carried out by the $\mathcal{F}$ rules of Figure 4.12. Each of these rules takes an expression, e, together with a set of bounded variables, bset, in order to return a tuple of three items, namely:

(1) a new expression, e^, derived from the original expression, e, by extracting some or all of its GT-MFEs. (Note the use of notation, e^, to denote an expression which has just been processed by $\mathcal{F}$.)

(2) a list of *ground-type* MFEs, say $fe_{1..p}$, extracted from e which are *free* relative to the bound variables in bset. A given expression is *free* relative to a set of bounded variables, if none of the bound variables are contained in the given expression.

(3) a corresponding list of new variables, $ve_{1..p}$, to replace the GT-MFEs of (2)

Every expression examined by $\mathcal{F}$ is initially tested to see if it is a ground-type free expression by $\mathcal{F}$ 1. If it is not, then one of the remaining $\mathcal{F}$ rules is invoked instead.

Of particular interest are rules $\mathcal{F}$ 9 and 10 for dealing with function calls. Initially, the arguments of these calls are tested to see if there are GT-MFEs among the arguments. If there are, then these GT-MFEs are extracted through $\mathcal{F}$ 9a and 10a. But if there are no GT-MFEs in the arguments and if the function call being dealt with is a non-recursive and non-lambda function, then we can search for implicit free expressions that may be present in the body of our functions (see rules $\mathcal{F}$ 9b.2 and 10b.2) via the technique of **L(d)**. In this search for implicit GT-MFEs, we must propagate the function calls arguments' status to indicate if they are *free* or *bounded*, using the meta-function, divide_arg. Successful location of implicit GT-MFEs, will involve the re-definition of these non-recursive functions, according to **L(d)**, in order to float out the located GT-MFEs. This re-definition is done at steps $\mathcal{F}$ 9b.2.2.1 and 10b.2.2.1.

Lastly, an auxiliary meta-function **$\mathcal{F}$_list** is also used in our rules. This rule merely deals with the extraction from a list of expressions rather than from a single expression (as in $\mathcal{F}$).

(1) $\mathcal{F}$ [[e]] bset
If e is a GT-MFE relative to bset (and not a variable or a constant) then
⇐ (ve,[e],[ve]) ! where ve is a new variable

(2) $\mathcal{F}$ [[v]] bset ⇐ (v,[],[])

(3) $\mathcal{F}$ [[$c(t_1,..,t_n)$]] bset ⇐ $(c(t^\wedge_1,..,t^\wedge_n),fe_lt,ve_lt)$
where $(t^\wedge_1,..,t^\wedge_n,fe_lt,ve_lt)==\mathcal{F}$ _list [[$t_1,..,t_n$]] bset

(4) $\mathcal{F}$ [[let $v==t_1$ in t_2]] bset ⇐ (let $v==t^\wedge_1$ in $t^\wedge_2$, $fe_lt_1 <> fe_lt_2, ve_lt_1 <> ve_lt_2$)
where $(t^\wedge_1,fe_lt_1,ve_lt_1)==\mathcal{F}$ [[t_1]] bset
where $(t^\wedge_2,fe_lt_2,ve_lt_2)==\mathcal{F}$ [[t_2]] bset ∪ {v}

(5) $\mathcal{F}$ [[f]] bset ⇐ (f,[],[])

(6) $\mathcal{F}$ [[g]] bset ⇐ (g,[],[])

(7) $\mathcal{F}$ [[$va(t_1,..,t_n)$]] bset ⇐ $(va^\wedge(t^\wedge_1,..,t^\wedge_n),fe_lt' <> fe_lt,ve_lt' <> ve_lt)$
where $(t^\wedge_1..t^\wedge_n,fe_lt,ve_lt)==\mathcal{F}$ _list [[$t_1..t_n$]] bset
where $(va^\wedge,fe_lt',ve_lt')==\mathcal{F}$ [[va]] bset

(8) $\mathcal{F}$ [[lambda $(v_1,..,v_n)$=>tf end]] bset

$\Leftarrow$ (lambda $(v_1,..,v_n)$ => tf^ end,fe_lt,fv_lt)

where (tf^,fe_lt,ve_lt)== $\mathcal{F}$ [[tf]] bset $\cup$ $\{v_1,..,v_n\}$

(9) $\mathcal{F}$ [[$f(t_1,..,t_n)$]] bset

let $(t^\wedge_1..t^\wedge_n,fe_{1..p},ve_{1..p})$==$\mathcal{F}$ _list [[$t_1..t_n$]] bset **in**

a) **if** p≥1 **then** $\Leftarrow$ $(f(t^\wedge_1,..,t^\wedge_n),fe_{1..p},ve_{1..p})$

b) **if** p=0

b.1) **if** f **is a recursive, primitive or a function with a lambda abstraction then**

$\Leftarrow$ $(f(t_1,..,t_n),[],[])$

b.2) **let** (free,bounded) ==divide_arg [[$t_1..t_n$]] bset **in**

b.2.1) **if free={}** $\Leftarrow$ $(f(t_1,..,t_n),[],[])$

b.2.2) **if free≠{}, *given*** $f(v_1,..,v_n)$=tf

let $(tf^\wedge,fe'_{1..q},ve'_{1..q})$=$\mathcal{F}$ [[tf]] $v_{bounded}$ **in**

b.2.2.1) **if** q≥1 $\Leftarrow$ $(f_n(ve'_{1..q},vr_{1..z},v_{bounded})\ [t/v_{1..n}],\ fe'_{1..q}[t/v_{1..n}],ve'_{1..q})$

where $vr_{1..z}$==vars [tf^] - $ve_{1..q}$ - $v_{bounded}$

define $f_n(ve_{1..p},vr_{1..z},v_{bounded})$=tf^

b.2.2.2) **if q=0** $\Leftarrow$ $(f(t_1,..,t_n),[],[])$

(10) $\mathcal{F}$ [[$g(t_0,..,t_n)$]] bset

let $(t^\wedge_0..t^\wedge_n,fe_{1..p},ve_{1..p})$==$\mathcal{F}$ _list [[$t_0..t_n$]]bset **in**

a) **if** p≥1 **then** $\Leftarrow$ $(g(t^\wedge_0,..,t^\wedge_n),fe_{1..p},ve_{1..p})$

b) **if** p=0

b.1) **if** g **is a recursive or lambda fn then**

$\Leftarrow$ $(g(t_0,..,t_n),[],[])$

b.2) **let** (free,bounded) ==divide_arg [[$t_1..t_n$]] bset **in**

b.2.1) **if free={}** $\Leftarrow$ $(g(t_0,..,t_n),[],[])$

b.2.2) **if free≠{}, *given*** $g(p_1,v_1,..,v_n)$=tg_1

:

$g(p_r,v_1,..,v_n)$=tg_r

let $(tg^\wedge_1..tg^\wedge_r,fe'_{1..q},ve'_{1..q})$=$\mathcal{F}$_list [[$tg_1..tg_r$]] vars[$p_1..p_r$] $\cup$ $v_{bounded}$ **in**

b.2.2.1) **if** p≥1 $\Leftarrow$ $(g_n(t_0,ve'_{1..q},vr_{1..z},v_{bounded})\ [t/v_{1..n}],\ fe'_{1..q}\ [t/v_{1..n}],ve'_{1..q}))$

where $vr_{1..z}$==vars [$tg^\wedge_1..tg^\wedge_r$] - $ve'_{1..q}$ - $v_{bounded}$ - vars[$p_1..p_r$]

define $g_n(p_1,ve'_{1..q},vr_{1..z},v_{bounded})$=$tg^\wedge_1$

:

$g_n(p_r,ve'_{1..q},vr_{1..z},v_{bounded})$=$tg^\wedge_r$

b.2.2.2) **if p=0** $\Leftarrow$ $(g(t_0,..,t_n),[],[])$

WHERE

$\mathcal{F}$ _list [[[]]] bset $\Leftarrow$ ([],[],[])

$\mathcal{F}$ _list [[t::rest]] bset $\Leftarrow$ (t^::rest^,fe_lt<>fe_rest,ve_lt<>ve_rest)

where (t^,fe_lt,ve_lt)=$\mathcal{F}$ [[t]] bset

where (rest^,fe_rest,ve_rest)=$\mathcal{F}$ _list [[rest]] bset

divide_arg [[[]]] bset $\Leftarrow$ ([],[])

divide_arg [[t_i::rest]] bset $\Leftarrow$ **If** t_i is free relative to bset **then** (i::freed,bounded)

else (freed,i::bounded)

where (freed,bounded)==divide_arg [[rest]] bset

Figure 4.12 - $\mathcal{F}$ Rules to Find both Explicit and Implicit GT-MFEs

It is important to note that the $\mathcal{F}$ rule will always extract out the *explicit* GT-MFEs before the *implicit* ones. Also, each invocation of the $\mathcal{F}$ rule often extracts out only *some* of the GT-MFEs. This effectively means that we may have to apply the $\mathcal{F}$ rules a number of times before *all* the

GT-MFEs could be extracted. This repetitive extraction of GT-MFEs is, however, handled outside the $\mathcal{F}$ rules (e.g. by the $\mathcal{E}$ rule).

We are now really to illustrate the extraction of GT-MFEs (by $\mathcal{F}$) using the earlier example from the previous section. There were four invocations to the $\mathcal{F}$ rules but only two of them produced any GT-MFEs. We illustrate the two invocations which result in GT-MFEs below.

```
F [[ lambda a => h2(x*x,sqr(a)) end ]]{}                                    !F8
    = (lambda a => tf end,fel,vel) where (tf,fel,vel)==F [[ h2(x*x,sqr(a)) ]]{a}  !F9a
    = (lambda a => tf end,fel,vel) where (tf,fel,vel)==(h2(A,sqr(a)),[x*x],[A])
    = (lambda a => h2(A,sqr(a)) end,[x*x],[A])
F [[ lambda a => h2(A,sqr(a)) end ]]{}                                      !F8
    = (lambda a => tf end,fel,vel) where (tf,fel,vel)==F [[ h2(A,sqr(a)) ]]{a}  !F9b221 define h2'
    = (lambda a => tf end,fel,vel) where (tf,fel,vel)==(h2'(B,sqr(a)),[A*A],[B])
    = (lambda a => h2'(B,sqr(a)) end,[A*A],[B])
Defined by F 9b.221
- - -h2'(B,a)          <=  B+(a+a)
```

The first invocation of the $\mathcal{F}$ rule results in the extraction of an explicit GT-MFE, x*x. The second invocation is slightly more complex because it involves the re-definition of h2 to h2' (by step $\mathcal{F}$ 9b.221). This re-definition results in the extraction of implicit GT-MFE, A*A.

As for the other two invocations which did not result in any GT-MFEs, the sequences of rule applications are as follows.

```
F [[ dup(fn1(x*x)) ]]{x}                                                    !F9
    = let (t1,fe1,ve1)==F_list [[ fn1(x*x) ]]{x} in ?1                      !F9
    = let (t1,fe1,ve1)==let (t2,fe2,ve2)==F_list [[ x*x ]]{x} in ?2 in ?1   !F9b1
    = let (t1,fe1,ve1)==let (t2,fe2,ve2)==(fn1(x*x),[],[]) in ?1
    = let (t1,fe1,ve1)==(fn1(x*x),[],[]) in ?1                              !F9b21
    = (dup(fn1(x*x)),[],[])
F [[ lambda a => h2'(B,sqr(a)) end ]]{}                                     !F8
    = (lambda a => tf end,fel,vel) where (tf,fel,vel)==F [[ h'(B,sqr(a)) ]]{a}  !F9a
    = (lambda a => tf end,fel,vel) where (tf,fel,vel)==
                  let (t1,fe1,ve1)==F_list [[ [B,sqr(a)] ]]{a} in ?1        !F9
    = (lambda a => tf end,fel,vel) where (tf,fel,vel)==
                  let (t1,fe1,ve1)==([B,sqr(a)],[],[]) in ?1                !F9b222
    = (lambda a => tf end,fel,vel) where (tf,fel,vel)==(h'(B,sqr(a)),[],[])
    = (lambda a => h'(B,sqr(a)) end,[],[])
```

To simplify the above presentation, we have used the notations ?1 or ?2 to denote some outstanding operations which depended on the result of the corresponding $\mathcal{F}$_list rule applications. Also, certain steps in the above derivations have been abbreviated.

4.4.3. Fully Lazy Elimination of Higher-Order Arguments

After the application of the full laziness transformation, $\mathcal{L}$, our program will be in a form that is more amenable to fully lazy higher-order argument elimination. In particular, all of the lambda abstractions will not have any explicit or implicit GT-MFEs, as these would have been extracted out. Furthermore, all grounded GT-MFEs would also be converted to CFs.

Based on this fully lazy expression form, it is only necessary to modify two rules of $\mathcal{R}$ to make the higher-order argument elimination algorithm fully lazy. The two rules which need to be

changed are $\mathcal{R}$ 10 and 11 for dealing with function calls with instantiated higher-order arguments. We change these two rules by introducing an *iterative step* in each of these rules. This iterative step searches for all GT-MFEs in its function call's *non-linear* higher-order arguments.

The two iterative steps are shown in Figure 4.13 as $\mathcal{R}$10d and $\mathcal{R}$11c. They must be applied, where possible, in *preference* over the other steps (namely $\mathcal{R}$10a,10b,10c,$\mathcal{R}$11a and 11b of Figure 4.4) of the same rules. If GT-MFEs exist in the non-linear higher-order arguments, they should be extracted out using intermediate functions as shown in $\mathcal{R}$ 10d and 11c. The resulting expression for the intermediate function is then recursively subjected to the same $\mathcal{R}$ rule, until no more GT-MFEs are found. At this stage, the non-linear higher-order arguments can be duplicated, without loss of efficiency, by the other steps of the rules (in Figure 4.4).

Notice that the variable-only higher-order arguments are now divided into two groups, $ht_{1..p}^{*\diamond}$ and $ht_{p+1..q}^{*\circ}$. The first group, with an additional $\diamond$ annotation, is for the *linear* parameters, while that with the additional $\circ$ annotation is for the *non-linear* parameters. Also, we make use of the meta-function, $\mathcal{F}$**_list**, to search for ground-type MFEs in the iterative steps.

(10) $\mathcal{R}$ [[$f(t_{1..m}^{@}, t_{m+1..n}, ht_{1..p}^{*\diamond}, ht_{p+1..q}^{*\circ})$]]

:

d) let $(ht\hat{}_{p+1..q}, fe_{1..a}, ve_{1..a}) == \mathcal{F}$**_list** [[$ht_{p+1..q}$]] {} **in**
if a≥1 **(GT-MFEs exist)**
⇐ f_new($vr_{1..z}$,$\mathcal{R}$ [[fe]]$_{1..a}$)
where $vr_{1..z}$=vars[$t_{1..n}$,$ht_{1..p}$]
define f_new($vr_{1..z}$,$ve_{1..a}$) = $\mathcal{R}$ [[$f(t_{1..m}, t_{m+1..n}, ht_{1..p}, ht\hat{}_{p+1..q})$]]

(11) $\mathcal{R}$ [[$g(t_0, t_{1..n}, ht_{1..p}^{*\diamond}, ht_{p+1..q}^{*\circ})$]]

:

c) let $(ht\hat{}_{p+1..q}, fe_{1..a}, ve_{1..a}) == \mathcal{F}$**_list** [[$ht_{p+1..q}$]] {} **in**
if a≥1 **(GT-MFEs exist)**
⇐ f_new($vr_{1..z}$,$\mathcal{R}$ [[fe]]$_{1..a}$)
where $vr_{1..z}$=vars[$t_{0..n}$,$ht_{1..p}$]
define f_new($vr_{1..z}$,$ve_{1..a}$) = $\mathcal{R}$ [[$g(t_0, t_{1..n}, ht_{1..p}, ht\hat{}_{p+1..q})$]]

Figure 4.13 - Additional Sub-Rules for Fully Lazy Higher-Order Arguments Elimination

4.4.4. An Example

With the modified $\mathcal{R}$ rules, we can now give an example of fully lazy higher-order argument elimination. Our starting point is the following set of three functions, which are applyless in form.

```
dec main : list(num) # num # num -> list(num);
- - -main(x,A,B)      <= map(x,lambda y=> f_1(y,A*B) end);
- - -f_1(y,A)         <= f_2(y*y,A+A);
- - -f_2(y,A)         <= y*y+sqr(A);
```

The full laziness transformation, $\mathcal{L}$, can transform it to a new set of fully lazy functions, as shown below.

```
- - -main(x,A,B)      <= map(x ,ll_x(A*B) ;
- - -ll_x(A)         <= ll_x2(A+A);
- - -ll_x2(A)        <= ll_x3(sqr(A));
- - -ll_x3(A)        <= lambda y => f_2a(y*y,A) end ;
- - -f_2a(y,A)       <= y*y+A;
```

In making the above program fully lazy, three intermediate functions, ll_x, ll_x2 and ll_x3, are introduced to lift out some ground-type MFEs in the lambda argument, **lambda** y=> f_1(y,A*B) **end**. This new fully lazy program also has a new function, f_2a, which arises out of a re-definition of function f_2. This re-definition is needed because the original f_2 call cannot be directly unfolded (as it would duplicate the bounded subexpression y*y). Also, two earlier functions, f_1 and f_2, are not included in our program because these are no longer needed by main.

The above fully lazy program can subsequently be transformed by the modified $\mathfrak{R}$ rules to give the following efficient higher-order specialised program.

```
- - -main(x,A,B)      <= main_1(x,A*B);
- - -main_1(x,A)      <= main_2(x,sqr(A+A)) ;
- - -main_2([],A)     <= [];
- - -main_2(a::x,A)   <= f_2a(a*a,A):: main_2(x,A) ;
- - -f_2a(y,A)        <= y*y+A;
```

The result of this transformation is a set of first-order functions. Two new intermediate functions, main_1 and main_2, are introduced to lift out ground-type MFEs, prior to the elimination of the higher-order map call. Also, all the previous intermediate functions, ll_x, ll_x2 and ll_x3, are eliminated[8] because they have been unfolded out.

4.4.5. Possible Losses of Efficiency

In order to remain compatible with the higher-order argument elimination, we have *constrained* the full laziness transformation to extract out *only* ground-type MFEs. This restriction may, at times, cause a loss of efficiency because it is possible for *function-type* MFEs to cause redundancy when duplicated.

There are two sources of function-type MFEs (FT-MFEs) which may cause some loss in laziness or efficiency in our full laziness transformations. The first source is from the *non-recursive* g-type functions, while the second source is from the recursive functions (both f-type and g-type). They both arise out of our inability to *directly unfold* their respective function calls. We discuss both forms of FT-MFE next and suggest possible solutions to minimise the loss of efficiency.

[8] This example shows that though additional intermediate functions may be created by program transformation, it is also possible for some of the older functions to be eliminated whenever they are no longer referred by the main function. This is particularly true, when we move from one transformation tactic to another.

4.4.5.1. FT-MFEs from g-type functions

Consider the following higher-order program,

```
dec Floss:num # truval -> num;
dec app_twice: (alpha -> alpha) # alpha  -> alpha;
- - -Floss(x,b)        <= app_twice(if(b,sqr,cube),x);
- - -app_twice(f,x)    <= f (f(x)) ;
- - -if(true,f,g)      <= f ;
- - -if(false,f,g)     <= g;
- - -sqr(x)            <= x*x ;
- - -cube(x)           <= x*x*x ;
```

where the app_twice call in function Floss contains a *function-type* MFE, if(b,sqr,cube). This MFE is also a higher-order argument which needs to be eliminated. The fully lazy $\mathcal{R}$ rules cannot extract it out because this would result in another intermediate function with the same higher-order argument. Instead, the present $\mathcal{R}$ rules will simply eliminate this function-type argument through the following sequence of (abbreviated) steps.

```
- - -Floss(x,b)    <= app_twice(if(b,sqr,cube),x);                ! unfold app_twice
                   <= if(b,sqr,cube) (if(b,sqr,cube) (x));        ! eliminate general application
                   <= if(b,sqr(if(b,sqr,cube) (x)),
                        cube(if(b,sqr,cube) (x)) );               ! eliminate general application
                   <= if(b,sqr(if(b,sqr x,cube x)),cube(if(b,sqr x,cube x));
```

This elimination results in a function that is higher-order specialised, but it also incurs a minor loss in efficiency. The conditional if function with the common test variable, b, will be evaluated *twice*, as opposed to *once*, in the newly transformed program.

The main cause of this loss of efficiency is that we did not allowed the non-recursive g-type function calls to be directly unfolded. The reason for this restriction is that we wanted to adhere to the original language specification of one pattern-matching parameter per g-type function. If this language restriction could be lifted, we can avoid the loss of efficiency by a sequence of transformation steps which involve the direct unfolding of g-type function call, if(b,sqr,cube), as shown below.

```
- - -Floss(x,b)      <= app_twice(if(b,sqr,cube),x);     ! unfold if function
- - -Floss(x,true)   <= app_twice(sqr,x);                ! unfold app_twice
                    <= sqr(sqr(x));
- - -Floss(x,false)  <= app_twice(cube,x);               ! unfold app_twice
                    <= cube(cube(x));
```

In this particular example, the only violation of our language is that the pattern-matching parameter of the resulting function did not appear as the first parameter. Additional work would have to be done to transform the resulting program to comply with the original language restriction. Though the language restriction is not essential for the higher-order removal tactic, it is important for the deforestation transformation (of Chapter 3) which could be applied after higher-order removal, as we shall illustrate in Section 4.5.

4.4.5.2. FT-MFEs from recursive functions

Another source of function-type MFEs is from the recursive functions. An example of this is the following program.

```
- - -Floss2(w,x)      <= app_twice(appd(w),x);
- - -appd(w)        <= lambda y => append(w,y) end ;
```

Here, we have a function-type MFE, appd(w), which originates from the partial application of recursive append function (which cannot be directly unfolded). As a result, the higher-order argument elimination transformation would be as follows:

```
- - -Floss2(w,x)      <= app_twice(appd(w),x);          ! unfold app_twice
                      <= appd(w) (appd(w)(x));          ! eliminate general application
                      <= append(w,appd(w)(x));          ! eliminate general application
                      <= append(w,append(w,x));
```

The resulting function will traverse the parameter, w, twice. In contrast, an equivalent function that results from fully lazy lambda lifting need only traverse the parameter once with the help of *code unrolling* technique (as illustrated in Section 4.4.1.5). Code unrolling (from fully lazy lambda lifting) is a form of laziness which is peculiar to partially applied recursive function calls. It is inherently dependent on higher-order facility to support. There is just no way we could support this laziness while purporting to the goal of higher-order removal. The space leak problem associated with this form of laziness may persuade the reader that this form of laziness is not always desirable.

4.4.5.3. Coping with the Loss of Full Laziness

We would like to suggest three possible courses of action which could be taken, in response to the loss of laziness from the two kinds of function-type MFEs. These are:

- ignore the loss of laziness
- avoid the loss of laziness
- recover some of the loss of laziness

The first option is to *ignore* the loss of laziness. This appears to be quite acceptable provided there are not too many intermediate g-type function-type calls which are FT-MFEs. This is because *real* losses of efficiency could occur otherwise. As for function-type *recursive* calls, the loss of laziness may be justified by a trade-off with the space-leak problem. Furthermore, the removal of higher-order features may well compensate this small loss of efficiency.

A second more conservative action is to *avoid* the loss of laziness totally. This can be achieved by using the **let** construct to extract out all FT-MFEs. For example, the first example of function, Floss, can simply avoid its loss of laziness by transforming to:

```
- - -Floss(x,b)      <= let v==if(b,sqr,cube) in app_twice(v,x);
```

This **let** construct does not result in higher-order argument elimination but helps to make the expression appears pseudo-specialised.

A better approach is to try and *recover* the loss of efficiency from the g-type function calls but ignore the loss from the recursive function calls. We have seen how the loss of laziness from g-type function calls could be recovered by permitting direct unfolds to these calls. However, we shall also need to tidy up the program after this elimination. As for the function-type MFEs of recursive functions, the loss of laziness is irrecoverable because this laziness is intrinsically dependent to partially applied expressions of higher-order (curried) programs. This loss is

acceptable if the prevention of space-leaks is also as one of our objectives.

4.4.6. Termination of the Full Laziness Transformation Rules

There are a number of rules involved in the preservation of full laziness transformation. The main rules are $\mathcal{R}$, $\mathcal{L}$ (three cases), $\mathcal{E}$ and $\mathcal{F}$. These rules fall under the following invocation or calling hierarchy, namely $\{\mathcal{L}\ 0 > \mathcal{L}\ 1 > \mathcal{E} > \mathcal{L}\ 2\}$, $\{\mathcal{E} > \mathcal{F}\}$ and $\{\mathcal{R} > \mathcal{F}\}$. Apart from these rules, there are also a number of auxiliary rules, e.g. $\mathcal{F}$_list, C$\mathcal{F}$, divide_arg, but these can be trivially shown to be terminating, as long as the main rules are terminating. We shall therefore be concerned with just the main rules in our outline of the termination proof.

To prove that the entire full laziness transformation algorithm terminates, we have to prove that each of the above recursive rules (except $\mathcal{L}$ 0 which is non-recursive) terminates, and does not affect the termination of the other rules which use it. The full termination proof for each of these rules is given in [Chin90]. In this section, we shall merely outline the essence of these proofs.

Starting, with the $\mathcal{F}$ rule (which lies at the bottom of the hierarchy), we can see that each of its recursive rules either (1) operate on successively smaller sub-expressions or (2) operate on the bodies (RHS) of non-recursive functions (to look for implicit GT-MFEs). These rules can therefore have a well-founded decreasing measure that is made up of the *function hierarchy number* combined with *expression size*. Similarly, the recursive $\mathcal{L}$ 1 and $\mathcal{L}$ 2 rules either operate on successively smaller sub-expressions or perform direct unfolds of non-recursive function calls. These recursive rules could similarly have a well-founded decreasing measure based on the *function hierarchy number* and *expression size*.

Rule $\mathcal{E}$ is also recursive but has a termination property that is tied very closely to the $\mathcal{F}$ rule. In particular, it repeatedly extracts GT-MFEs from its expression and will terminate only when no more GT-MFEs remain. To show that this repeated extraction terminates, we must show that each application of $\mathcal{F}$ extracts out some GT-MFEs, in such a way that there are *fewer* GT-MFEs left in the resulting expression.

To prove this, we simply define a GT-MFE measure to count the maximum number of *extractable* GT-MFEs (by $\mathcal{F}$) that is present in any given expression. The equational definition for this measure will essentially mirror the recursion structure of $\mathcal{F}$. It will always return a finite value because such a recursion structure has been shown (for $\mathcal{F}$) to be terminating. Having defined this GT-MFE measure, we could prove (by induction) that each expression that results from an application of $\mathcal{F}$ will always have a measure that is either (1) *less than* the measure of the original expression, or (2) *zero*, (i.e. *unchanged* from the original expression which also has a measure of zero; no more GT-MFEs). This well-founded decreasing measure is sufficient to prove that the $\mathcal{E}$ rule will terminate.

Rule $\mathcal{R}$ 10d and $\mathcal{R}$ 11c also repeatedly invoke $\mathcal{F}$ until no more GT-MFEs could be found. By the same reasoning as the $\mathcal{E}$ rule, this could not go on forever. As a result, each sequence of repeated extraction of GT-MFEs by $\mathcal{R}$ 10d and $\mathcal{R}$ 11c will always be of finite length. As these repeated extractions are made up of finite sequences, which always come just before the other steps of rules $\mathcal{R}$ 10a,b,c and $\mathcal{R}$ 11a,b, we could never obtain an overall sequence of steps that is

infinite. The reason for this is that there is only a *finite* number of steps for $\mathcal{R}$ 10a,b,c and $\mathcal{R}$ 11a,b (previously proved).

4.5. Deforestation for Higher-Order Expressions

There are two major advantages of the higher-order removal transformation algorithms, $\mathcal{A}$, $\mathcal{L}$ and $\mathcal{R}$. The first advantage is the efficiency gain that results from the absence of higher-order features; while the second advantage is the presence of a much simpler higher-order specialised form that is as close to first-order expressions as our transformations will produce for us.

The simpler expression form is an advantage because it helps to ease the *extensions* of various implementation, analysis or transformation techniques, from *first-order* programs to *higher-order* programs. In particular, it is only necessary to consider the extensions of these techniques to the *specialised* form. Once they work for expressions of this form, we can safely assume that they will also work for any higher-order expressions because these can always be converted to the specialised form by the earlier transformations.

In this section, we shall consider an extension for the Universal Deforestation transformation, which has already been shown in Chapter 3 to work for all first-order programs. Our extension involves the modifications of Universal Deforestation rules to cope with the *vestiges* of higher-order features (e.g. variable applications and lambda abstractions), which remains in the HO-specialised form.

We shall call the suitably modified rules, which could deforest HO-specialised expressions, the **higher-order deforestation** transformation rules. However, before presenting the full modifications required, we shall first present an appropriate parameter annotation scheme and target expression form for this new result.

4.5.1. Parameter Annotations for Higher-Order Deforestation

For the purpose of higher-order deforestation, we need to categorise the parameters of our function calls into three major groupings. These three major groupings will be marked by the following parameter annotation scheme:

Definition 4.11 - Annotation Scheme for Higher-Order Deforestation

Each parameter in a function call will be marked with one of the following annotations depending on whether certain criteria are present. These annotations are:

- ✣ **if** the parameter is linear, variable-only and of tree-like type
- ✱ **if** the parameter is variable-only and function-type
- ▫ **otherwise**

With this annotation scheme, we can present the function calls (from the HO-specialised expressions) in one of the following forms:

$$f(t_1^{\oplus}..t_m^{\oplus},t_{m+1}^{\ominus}..t_n^{\ominus},vh_1^{*}..vh_q^{*})$$

or $$g(t_0^{\oplus},t_1^{\oplus}..t_m^{\oplus},t_{m+1}^{\ominus}..t_n^{\ominus},vh_1^{*}..vh_q^{*})$$

or $$g(t_0^{\ominus},t_1^{\oplus}..t_m^{\oplus},t_{m+1}^{\ominus}..t_n^{\ominus},vh_1^{*}..vh_q^{*})$$

The most important of these parameter groupings is the first group, $t_1..t_m$, for *linear, variable-only* and *recursive-type* parameters. This group is important because they represent intermediate terms (arguments) that are targeted for elimination by the deforestation stage. In addition, we also have the higher-order arguments, $vh_1..vh_q$, which were significant in the previous $\mathfrak{R}$ transformation stage. This group of arguments should be *variables* because their instantiated counterparts would have been eliminated by $\mathfrak{R}$. The remaining arguments, which do not satisfy the criteria of the first two groups, will be lumped into $t_{m+1}..t_n$. Also, as before, there is an additional pattern-matching parameter argument, t_0, for g-type function calls. This pattern-matching argument can either be annotated as $^{\oplus}$ or $^{\ominus}$, depending on whether it is *linear, variable-only* and *recursive-type* or not; thus giving the above two kinds of g-type calls.

With this new categorisation for parameters, we are now ready to propose a possible target expression form for higher-order deforestation.

4.5.2. Higher-Order Treeless Form

We call the target form, which shall result from higher-order deforestation, the *higher-order treeless*, or *HO-treeless*, form. This HO-treeless form is a further specialisation of the HO-specialised form, with all unnecessary intermediate terms eliminated by our transformation. Such a form can be defined as follows:

Definition 4.12 - Higher-Order Treeless Form

A term is said to be *higher-order treeless* if it satisfies the grammar below.

$$ht ::= t \mid \textbf{let } v==t_1 \textbf{ in } t_2$$

$$t ::= va \mid c\,(t_1,..,t_n) \mid \textbf{lambda } (v_1,..,v_n) \Rightarrow tf \textbf{ end} \mid f(v_{1..m}^{\oplus},t_{m+1..n}^{\ominus},vh_{1..q}^{*}) \mid g(va_0^{\oplus},v_{1..m}^{\oplus},t_{m+1..n}^{\ominus},vh_{1..q}^{*}) \mid g(t_0^{\ominus},v_{1..m}^{\oplus},t_{m+1..n}^{\ominus},vh_{1..q}^{*})$$

$$va ::= v \mid va(t_1,..,t_n)$$

where $vh_{1..q}$ are higher-order variables and g and f are *higher-order treeless* definitions.

Correspondingly, a function is said to be *higher-order treeless*, if its definition is made up of *higher-order treeless* terms.

Two distinctive features of this grammar, when compared to HO-specialised form, are:

- The first group of arguments, $v_{1..m}^{\oplus}$, in each f and g-type call must be *variables*. This group represents arguments for intermediate terms which can be eliminated by deforestation.
- The g-type calls, with pattern-matching arguments annotated as $^{\oplus}$, must be of the form $g(va_0^{\oplus},v_{1..m},t'_{m+1..n},vh_{1..q})$, where va_0 is either a *variable* or a *variable application*. We allow variable applications to be present because the presence of leftmost functional variables do not allow these terms to be

unfolded away.

Furthermore, in order to remain compatible with the first-order universal treeless form, we have restricted the use of **let** to be present only as an outermost construct (without loss of generality).

This new treeless form can be used as the target for the following result.

Theorem 4.13 - Higher-Order Deforestation

> Any well-typed *higher-order specialised* expression, whose functions and lambda abstractions are *higher-order treeless*, can always be effectively transformed to an equivalent term which is also *higher-order treeless* without loss of efficiency.

Notice that we require the source expressions to be well-typed HO-specialised form. This could be obtained from the previous higher-order removal transformations. A brief proof for this theorem is given in Section 4.5.6. The effective transformation algorithm to perform this conversion (to higher-order treeless form) is described next.

4.5.3. Modifications to the Transformation Algorithm

The transformation algorithm for higher-order deforestation shall be based on an *extension* of the first-order universal deforestation rules of Chapter 3. This extension must be able to handle:

1) three groups of arguments in each function call, as opposed to two groups in the first-order case;
2) possible presence of variable applications in pattern-matching parameters of g-type function calls;
3) lambda abstractions, variable applications and function identifiers of the HO-specialised form;
4) possible re-appearances of higher-order features.

We present solutions for each of these four issues and look at what rules must be *added* or *modified* in order to handle them.

Previously, the universal deforestation rules need only handle function calls with two groups of arguments, namely those arguments that are either annotated as $^{+}$ (linear,variable-only,recursive-type) or $^{-}$ (others). However, the HO-specialised form has an additional group of arguments, annotated as *, for higher-order arguments. This new group of arguments is fortunately only *variables*, after passing through the $\mathcal{R}$ transformation stage. Thus, we can trivially accommodate them in the modified rules by *treating* them as if they were arguments of the - annotation category.

A more difficult issue to handle is the presence of variable applications as pattern-matching parameters of the g-type function calls. This requires the modification of rules $\mathcal{T}$ 4 and $\mathcal{T}$ 8 (of the original deforestation rules in Figure 3.7) in order to permit the possible *presence* of either variables or variable applications in the pattern-matching parameters of g-type calls, as shown in Figure 4.14.

(4) $\mathcal{T}[\![\ g(va_0^{\Diamond},v_{1..m}^{\Diamond},t_{m+1..n}^{\sim})\ ]\!] \Leftarrow g(\mathcal{T}[\![\ va_0\ ]\!],v_{1..m},\mathcal{T}[\![\ t\ ]\!]_{m+1..n})$

(8) $\mathcal{T}[\![...g(va_0^{\Diamond},t_{1..m}^{\Diamond},t_{m+1..n}^{\sim})...]\!]$

a) if $g_old(vn_0,vn_{1..k},ve_{1..s}) = ...g(vn_0,t^{\sim}_{1..m},ve_{m+1..n})...^{\sim} \in$ def_set; **then** (fold)

$\Leftarrow$ $g_old(\mathcal{T}[\![\ va_0]\!],vo_{1..k},\mathcal{T}[\![te]\!]_{1..s})$

where $(...g(vn_0,t^{\sim}_{1..m},ve_{m+1..n})...^{\sim},te_{1..s}ve_{1..s},$

$v_0vo_{1..k},vn_0vn_{1..k}) ==$ **gen_ren**$[\![...g(v_0,t_{1..m},t_{m+1..n})...]\!]$

b) **otherwise** (define & unfold) $\Leftarrow$ $g_new(\mathcal{T}[\![\ va_0]\!],vo_{1..k},\mathcal{T}[\![te]\!]_{1..s})$

where$(...g(vn_0,t^{\sim}_{1..m},ve_{m+1..n})...^{\sim},te_{1..s}ve_{1..s},$

$v_0vo_{1..k},vn_0vn_{1..k}) ==$ **gen_ren**$[\![...g(v_0,t_{1..m},t_{m+1..n})...]\!]$

define and add to def_set

$g_new(vn_0,vn_{1..k},ve_{1..s}) = ...g(vn_0,t^{\sim}_{1..m},ve_{m+1..n})...^{\sim}$

narrow on call g ***to give:***

$g_new(p_1,vn_{1..k},ve_{1..s}) = \mathcal{T}[\![...tg_1[t^{\sim}/v_{1..m},ve/v_{m+1..n}]...^{\sim}]\!]$

:

$g_new(p_r,vn_{1..k},ve_{1..s}) = \mathcal{T}[\![...tg_r[t^{\sim}/v_{1..m},ve/v_{m+1..n}]...^{\sim}]\!]$

Figure 4.14 - Modified Rules for Higher-Order Deforestation

Apart from these two modified rules, we also need five more *new* rules for higher-order deforestation. These new rules are given in Figure 4.15. The first four rules are provided to deal with the additional constructs of the HO-specialised form. Rules $\mathcal{T}$ 13 and 14 are needed to handle variable applications and lambda abstractions. These two rules preserve their respective expression's outer structure and recursively transform just the inner sub-expressions. Also, rules $\mathcal{T}$ 15 and 16 are needed to skip over function identifiers.

The last rule, $\mathcal{T}$ 17, is <u>not</u> meant for new constructs. Instead, it is required for handling *re-appearances* of higher-order features which may arise under certain circumstances. These re-appearances must be removed by re-invoking the previous transformation algorithm, $\mathcal{R}$. We will describe why these re-appearances of higher-order features can occur under higher-order deforestation in Section 4.5.5 and show an example of how they can be eliminated.

(13) $\mathcal{T}[\![$ lambda $(v_1,..,v_n)$=> tf end $]\!] \Leftarrow$ lambda $(v_1,...,v_n)$=> $\mathcal{T}[\![$ tf $]\!]$ end

(14) $\mathcal{T}[\![\ va(t_1,...,t_n)\ ]\!] \Leftarrow \mathcal{T}[\![\ va\ ]\!](\mathcal{T}[\![\ t_1]\!],..,\mathcal{T}[\![\ t_n\ ]\!])$

(15) $\mathcal{T}[\![\ f\ ]\!] \Leftarrow f$

(16) $\mathcal{T}[\![\ g\ ]\!] \Leftarrow g$

(17) $\mathcal{T}[\![\ ...exp\ (t_1,..,t_n)...\ ]\!] \Leftarrow \mathcal{T}[\![\ ...\mathcal{R}[\![exp\ (t_1,..,t_n)\ ]\!]...\ ]\!]$

where exp $(t_1..t_n)$ does not conform to higher-order specialised form.

Figure 4.15 - New Rules for Higher-Order Deforestation

Apart from extending the deforestation rules, we must also provide a similar extension to the auxiliary meta-function, **gen_ren**. However, only one additional equation, for handling variable applications, is needed. This is because variable applications are the only higher-order feature that could possibly be ground-type and thus appear as arguments with the $^{\Diamond}$ annotation.

gen_ren$[\![\ va(t_1,..,t_n)\ ]\!] \Leftarrow (va^{\sim}(t^{\sim}_1,..,t^{\sim}_n),te_lt'<>te_lt,ve_lt'<>ve_lt,vo_lt'<>vo_lt,vn_lt'<>vn_lt)$

where $(va^{\sim}_1,te_lt',ve_lt',vo_lt',vn_lt')==$**gen_ren**$[\![\ va]\!]$

where $(t^{\sim}_1..t^{\sim}_n,te_lt,ve_lt,vo_lt,vn_lt)==$**gen_list**$[\![\ t_1..t_n]\!]$

The other features of HO-specialised form will only be present as arguments which are annotated as -, and are therefore not processed by **gen_ren**.

4.5.4. An Example

With the help of the above modified rules, we are now ready to present an example of higher-order deforestation. Consider the function, twice_map (from Section 4.3.5.2), which is already in HO-specialised form. This definition has a nested expression, map(map(x,h),k), which can be fused together. Applying the higher-order deforestation rules to this definition, results in a sequence of steps as shown in Figure 4.16. This transformation successfully eliminates the intermediate term of the inner map function call. Note the use of $\mathcal{T}$ 14 at steps $<=_8$ and $<=_9$ for handling the variable application, k(h(a)).

- - -twice_map(x,h,k)	$<=_1$	$\mathcal{T}$ [[map(map(x,h),k)]]	! $\mathcal{T}$ 8, narrow inner map
- - -twice_map([],h,k)	$<=_2$	$\mathcal{T}$ [[map([],k)]]	! $\mathcal{T}$ 6a
	$<=_3$	$\mathcal{T}$ [[[]]]	! $\mathcal{T}$ 2
	$<=_4$	[]	
- - -twice_map(a::x,h,k)	$<=_5$	$\mathcal{T}$ [[map(h(a)::map(x,h),k)]]	! $\mathcal{T}$ 6a
	$<=_6$	$\mathcal{T}$ [[k(h(a))::map(map(x,h),k)]]	! $\mathcal{T}$ 2
	$<=_7$	$\mathcal{T}$ [[k(h(a))]]::$\mathcal{T}$ [[map(map(x,h),k)]]	! $\mathcal{T}$ 8 fold
	$<=_8$	$\mathcal{T}$ [[k(h(a))]]::twice_map(x,h,k)	! $\mathcal{T}$ 14
	$<=_9$	$\mathcal{T}$ [[k]]($\mathcal{T}$ [[h(a)]])::twice_map(x,h,k)	! $\mathcal{T}$ 1; 14,1,1
	$<=_{10}$	k(h(a))::twice_map(x,h,k)	

Figure 4.16 - Applying $\mathcal{T}$ to twice_map

4.5.5. Re-appearance of Higher-Order Features

Under certain circumstances, it may be possible for *general applications* and *instantiated higher-order arguments* to re-appear in higher-order deforestation stage. This is possible because the previous $\mathcal{R}$ algorithm, which eliminates instantiated higher-order arguments, has left the pattern-matching arguments of g-type function calls *undisturbed.* These pattern-matching parameters can be an indirect source of higher-order arguments, in the form of lists or trees of functions.

As a result, the deforestation of intermediate terms from these parameters may cause higher-order features to re-appear again. An example of this phenomenon can be illustrated by the program below.

```
dec main:num # num  -> list(num);
dec add_five:list(num) -> list(num -> num);
dec map_ho:list(alpha->beta) # alpha -> list(beta);
- - -main(x,y)          <= map_ho(add_five(x),y);
- - -map_ho([],y)       <= [];
- - -map_ho(a::x,y)     <= a(y)::map_ho(x,y);
- - -add_five([])       <= [];
- - -add_five(a::x)     <= lambda z=>z+5a end::add_five(x);
```

All the functions of this program are currently in HO-specialised form. The pattern-matching

parameter of map_ho is actually a list of functions which is used to create a corresponding list of applications by applying each of the functions (from this list) to the second argument, y. Also, function add_five takes a list of numbers, x, and produces a corresponding list of functions with 5 times the respective number from x, added to the argument of each of the functions. These two functions, map_ho and add_five, are composed together in the body of function main.

To convert all the functions to HO-treeless form, we must apply the $\mathcal{T}$ rules to each RHS term of the above functions. The definitions of map_ho and add_five are already HO-treeless. The $\mathcal{T}$ rules will make no changes to them. As for function, main, there is an expression, map_ho(add_five(x),y), which can be fused together. The sequence of transformation to fuse it is shown in Figure 4.17.

Notice that the presence of the pattern-matching parameters, which carries functions, has caused a general application, **lambda** x=>x+5a **end** (x), to re-appear during its transformation. This general application occurred in step $<=_6$ of Figure 4.17 and must be eliminated before we are allowed to perform further deforestation. To do this, $\mathcal{T}$ 17 is applied in order to re-invoke the $\mathcal{R}$ rule and subsequently the $\mathcal{A}$ rule to eliminate the general application. This elimination enables us to completely transform the above program to the following first-order treeless function.

```
- - -main([],y)      <= [];
- - -main(a::x,y)    <= y+5a::main(x,y);
```

This example illustrated that the combination of deforestation and higher-order removal can help us eliminate lists and trees of functions. This was *not* achievable by either of the two transformation tactics on their own.

- - -main(x,y)	$<=_1$	$\mathcal{T}$ [[map_ho(add_five(x),y)]]	! $\mathcal{T}$ 8b, narrow add_five
- - -main([],y)	$<=_2$	$\mathcal{T}$ [[map_ho([],y)]]	! $\mathcal{T}$ 6a
	$<=_3$	$\mathcal{T}$ [[[]]]	! $\mathcal{T}$ 2
	$<=_4$	[]	
- - -main(a::x,y)	$<=_5$	$\mathcal{T}$ [[map_ho(lambda x=>x+5a end::add_five(x),y)]]	! $\mathcal{T}$ 6a
	$<=_6$	$\mathcal{T}$ [[lambda x=>x+5a end (y) :: map_ho(add_five(x),y)]]	! $\mathcal{T}$ 2
	$<=_7$	$\mathcal{T}$ [[lambda x=>x+5a end (y)]] :: $\mathcal{T}$ [[map_ho(add_five(x),y)]]	! $\mathcal{T}$ 8a,fold
	$<=_8$	$\mathcal{T}$ [[lambda x=>x+5a end (y)]] :: main(x,y)	! $\mathcal{T}$ 17
	$<=_9$	$\mathcal{T}$ [[$\mathcal{R}$ [[lambda x=>x+5a end (y)]]]] :: main(x,y)	! $\mathcal{R}$ 10
	$<=_{10}$	$\mathcal{T}$ [[$\mathcal{R}$ [[$\mathcal{A}$ [[$\mathcal{R}$ [[lambda x=>x+5a end]] ($\mathcal{R}$ [[y]])]]]] :: main(x,y)	! $\mathcal{R}$ 3,8b,..;1
	$<=_{11}$	$\mathcal{T}$ [[$\mathcal{R}$ [[$\mathcal{A}$ [[lambda x=>x+5a end (y)]]]] :: main(x,y)	! $\mathcal{A}$ rule
	$<=_{12}$	$\mathcal{T}$ [[$\mathcal{R}$ [[y+5a]]]] :: main(x,y)	! $\mathcal{R}$ 8b,..
	$<=_{13}$	$\mathcal{T}$ [[y+5a]] :: main(x,y)	! $\mathcal{T}$ 3
	$<=_{14}$	y+5a :: main(x,y)	

Figure 4.17 - Deforesting a List of Functions

4.5.6. Proof of Higher-Order Deforestation Theorem

The proof for the additional rules of Higher-Order Deforestation follows by extending the proofs of Extended and Universal Deforestation Theorems from Chapter 3 (Section 3.3.3, 3.4.2, 3.4.3.3). Firstly, all the $\mathcal{T}$ rules continue to result in only higher-order treeless terms. This can be trivially shown by structural induction. Secondly, no loss of efficiency could ever arise because each intermediate function introduced is always compensated by an unfold step.

Thirdly, we can prove that the additional rules will not cause the termination property of the $\mathcal{T}$ rules to be lost. This is due to the following three factors.

1) The nesting measure, $\mathcal{N}$, will continue to be bounded for the modified rules, $\mathcal{T}$ 4,8, and the new rules $\mathcal{T}$ 13,14,15,16. This is shown in Figure 4.18 where the definition of $\mathcal{N}$ for the four new constructs (of higher-order treeless form) are also given.
2) The actual size of generalised expressions for newly introduced function will continue to be bounded to the nesting $\mathcal{N}$. This is because:
   (i) we can ignore lambda abstractions and function identifiers because they could only be present in the parameter positions which are annotated as $^{\circ}$. Thus, they will never contribute to the size of generalised expressions for new function definitions.
   (ii) we can regard variable applications as playing a similar role to constructors because they are never unfolded. As a result, they will also have a contiguous bound in the same way as constructors.
3) The $\mathcal{T}$17 rule has an inner $\mathcal{R}$ rule application which could introduce new function symbols. Because of this, we have to treat this rule separately as an independent transformation. However, for well-typed programs, the $\mathcal{R}$ rule always decrease the type degree[9] of the expression it transforms. By virtue of this decrease, there could never be an infinite number of $\mathcal{T}$17 applications for any given well-typed expression.

Hence, the additional rules of Higher-Order Deforestation will continue to be terminating under the conditions of its theorem.

[9] suitably modified (from Section 4.3.6) by taking into account the degree of arguments within constructor terms.

Additional Proof for Lemma 3.14.1

Prove: The nesting measure, $\mathcal{N}$, on expressions being transformed is bounded by modified rules, $\mathcal{T}$ 4,8, and new rules, $\mathcal{T}$ 13,14,15,16.

Define: $\mathcal{N}[\![va(t_1,..,t_n)]\!] = \max\{\mathcal{N}[\![va]\!],\mathcal{N}[\![t_1]\!],..,\mathcal{N}[\![t_n]\!]\}$

$\mathcal{N}[\![\text{lambda } (v_1,..,v_n) \Rightarrow tf \text{ end}]\!] = \mathcal{N}[\![tf]\!]$

$\mathcal{N}[\![f]\!] = 0$

$\mathcal{N}[\![g]\!] = 0$

Case for Rule $\mathcal{T}$ 4 - $g(va_0^{\oplus},v_{1..m},t_{m+1..n})$:

From the definition: $\mathcal{N}[\![g(va_0^{\oplus},v_{1..m},t_{m+1..n})]\!] = \max\{1+\mathcal{N}[\![va_0]\!],\mathcal{N}[\![t]\!]_{m+1..n}\}$

- thus $\forall i \in m+1..n \quad \mathcal{N}[\![t_i]\!] \leq \mathcal{N}[\![g(va_0^{\oplus},v_{1..m},t_{m+1..n})]\!]$
- $\mathcal{N}[\![va_0]\!] \leq \mathcal{N}[\![g(va_0^{\oplus},v_{1..m},t_{m+1..n})]\!]$

Case for Rule $\mathcal{T}$ 8 - $...g(va_0^{\oplus},t_{1..m}^{\oplus},t_{m+1..n}^{\sim})...$:

From Lemma 3.14.2 on **gen_ren**:

$\mathcal{N}[\![...g(v_0,t_{1..m},t_{m+1..n})...]\!] \geq \max\{\mathcal{N}[\![...g(vn_0,t^{\sim}_{1..m},ve_{m+1..n})...^{\sim}]\!],\mathcal{N}[\![te]\!]_{1..s}\}$:

a) apply fold against old expression:

From Lemma 3.14.2: $\forall i \in 1..s$: $\mathcal{N}[\![te_i]\!] \leq \mathcal{N}[\![...g(v_0,t_{1..m},t_{m+1..n})...]\!]$

- By the monotonicity of $\mathcal{N}$: $\mathcal{N}[\![te_i]\!] \leq \mathcal{N}[\![...g(va_0,t_{1..m},t_{m+1..n})...]\!]$
- Also : $\mathcal{N}[\![va_0]\!] \leq \mathcal{N}[\![...g(va_0,t_{1..m},t_{m+1..n})...]\!]$

b) otherwise:

From Lemma 3.14.2: $\forall i \in 1..s$: $\mathcal{N}[\![te_i]\!] \leq \mathcal{N}[\![...g(v_0,t_{1..m},t_{m+1..n})...]\!]$

- By the monotonicity of $\mathcal{N}$: $\mathcal{N}[\![te_i]\!] \leq \mathcal{N}[\![...g(va_0,t_{1..m},t_{m+1..n})...]\!]$
- Also : $\mathcal{N}[\![va_0]\!] \leq \mathcal{N}[\![...g(va_0,t_{1..m},t_{m+1..n})...]\!]$

Also, From Definition: $\mathcal{N}[\![g(vn_0,t^{\sim}_{1..m},ve_{m+1..n})]\!]$

$= \max\{1+\max\{\mathcal{N}[\![vn_0]\!],\mathcal{N}[\![t^{\sim}]\!]_{1..m}\},\mathcal{N}[\![ve]\!]_{m+1..n}\}$

$= 1+\max\{\mathcal{N}[\![t^{\sim}]\!]_{1..m}\}$

For each equation, $g(p_i,v_1,..,v_n)=tg_i$:

Since g is HO-treeless, $\mathcal{N}[\![tg_i]\!] \leq 1 \geq \mathcal{N}[\![tg_i]\!]+\max\{\mathcal{N}[\![t^{\sim}]\!]_{1..m}\}$

$\geq \mathcal{N}[\![tg_i[v'/v'_{1..j},t^{\sim}/v_{1..m},ve/v_{m+1..n}]]\!]$

Thus: $\mathcal{N}[\![...tg_i[v'/v'_{1..j},t^{\sim}/v_{1..m},ve/v_{m+1..n}]...^{\sim}]\!] \leq \mathcal{N}[\![...g(vn_0,t_{1..m},t_{m+1..n})...]\!]$

- By the monotonicity of $\mathcal{N}$:

$\mathcal{N}[\![...tg_i[v'/v'_{1..j},t^{\sim}/v_{1..m},ve/v_{m+1..n}]...^{\sim}]\!] \leq \mathcal{N}[\![...g(va_0,t_{1..m},t_{m+1..n})...]\!]$

Case for Rule $\mathcal{T}$ 13 - lambda $(v_1,..,v_n)$ => tf end :

From the definition: $\mathcal{N}[\![\text{lambda } (v_1,..,v_n) \Rightarrow tf \text{ end}]\!] = \mathcal{N}[\![tf]\!]$

- thus $\mathcal{N}[\![tf]\!] \leq \mathcal{N}[\![\text{lambda } (v_1,..,v_n) \Rightarrow tf \text{ end}]\!]$

Case for Rule $\mathcal{T}$ 14 - $va(t_1,..,t_n)$:

From the definition: $\mathcal{N}[\![va(t_1,..,t_n)]\!] = \max\{\mathcal{N}[\![va]\!],\mathcal{N}[\![t_1]\!],..,\mathcal{N}[\![t_n]\!]\}$

- thus $\forall i \in 1..n \quad \mathcal{N}[\![t_i]\!] \leq \mathcal{N}[\![va(t_1,..,t_n)]\!]$
- $\mathcal{N}[\![va]\!] \leq \mathcal{N}[\![va(t_1,..,t_n)]\!]$

Case for Rule $\mathcal{T}$ 15 - f : nothing to prove

Case for Rule $\mathcal{T}$ 16 - g : nothing to prove

Figure 4.18 - Additional Proof of Lemma 3.14.1 to show that it holds for the New and Modified Rules of HO-Deforestation

4.6. Related Works

The proposed higher-order elimination transformations promise to bring about automatic removal of most of the overheads and complications of higher-order programs. In this section, we shall discuss and compare our result with some related research works.

4.6.1. Restricted Higher-Order Facility

Our higher-order removal tactic is formulated to handle programs from the full higher-order language. In contrast to this, a restricted form of higher-order facility, called *higher-order macros*, was proposed in [Wadler88] as a substitute for a certain class of higher-order functions. These macros can have function-type parameters but are *not* allowed to be recursive. This restriction is imposed to allow the full expansion of *macro calls* to first-order programs.

Even though macros are not allowed to be recursive, they could still use *local* recursion with the **let/where** constructs to support a special kind of recursion where the function-type parameters are *fixed/unchanged*. An example of such a macro is the following definition which corresponds to the popular map function:

```
map(x,F)        =  m(x)      where   m([])=[]
                                     m(a::x)=F(a)::m(x);
```

A similar restricted proposal for higher-order facility was also argued for by [Goguen88] through *parameterised modules*. The main reason why these restricted forms of higher-order facility were suggested, is the ease with which macro applications (or parameterised modules) could be expanded out to first-order equivalent. However, with the formulation of our new higher-order removal transformations, there are now fewer reasons for limiting ourselves to this restricted higher-order facility.

4.6.2. Translation to First-Order Programs using Explicit Closures

In the language implementation front, some researchers [Reynolds72,Bondorf88,Johnsson85] have proposed the use of *explicit closures* for implementing higher-order features directly using first-order programs. This technique can actually convert all higher-order expressions to first-order equivalent. This is done by replacing each lambda abstraction (or partial application) with:

(1) a unique constructor term to act as an explicit closure, and

(2) a new equation to apply the explicit closure.

As an example, consider the following higher-order program.

```
---main(x,y)      <= map(y,lambda a => x<>a end);
---map([],F)      <= [];
---map(a::x,F)    <= F(a)::map(x,F);
```

To convert this higher-order program to its first order equivalent, the explicit closures technique will initially introduce a new unique constructor term, say **cl_1**(x), to represent the lambda abstraction, **lambda** a => x<>a **end**, as follows.

```
---main(x,y)      <= map(y,cl_1(x));
```

This special constructor term acts as an explicit closure. It must always be applied by the

following newly added apply equation.

```
- - -apply(cl_1(x),a)  <=  x<>a;
```

With these two changes (new constructor term and its associated apply equation) for each lambda abstraction, the technique must now change all variable applications (in the program) to use the specially defined apply function. In the above example, a variable application occurs in the second equation of map. This equation must be converted to:

```
- - -map(a::x,F)        <=  apply(F,a)::map(x,F);
```

The result of these changes is a purely first-order program, which uses explicit closures to represent the partially applied higher-order functions. This translation technique has been suggested as a simple means of supporting higher-order facility in an (untyped[10]) first-order language. However, the overheads of higher-order features are still largely present because explicit closures must still be created on the heap space. Furthermore, the complexity of the resulting program (with explicit closures) will continue to hamper both analysis and transformation techniques.

A better use of this technique could perhaps be made for the *residual* higher-order features of *HO-specialised* form, after our higher-order removal tactic has been applied.

4.6.3. Partial Evaluation

The specialisation of function calls, in order to eliminate higher-order arguments, can be viewed as a special form of partial evaluation [PE&MIX88]. Conventionally, the partial evaluation technique has relied on an abstract interpretation technique, called *binding-time analysis* (BTA) [JonesN88b], to help determine the function calls which can be specialised. Those function calls which can be specialised are expected to have one or more arguments which will always be *grounded* (known) or *partially grounded* in the program. Which of these arguments possess this property (of being always known or grounded), is the main analysis performed by BTA for partial evaluation.

In [Nielson89], the use of BTA and the partial evaluation technique was suggested as a way of specialising higher-order function calls with *known* function-type arguments. However, this proposal was only formulated to handle (totally known) function-type arguments without any free variables.

Instead of BTA, we have used a completely different parameter criterion, called the *variable-only* criterion, for specialising higher-order function calls. This criterion is used to help identify both function-type (e.g. higher-order argument elimination) and ground-type (e.g. deforestation) arguments which could be specialised.

---

[10] because of the overloaded apply function

We argue that our criterion is more suitable for specialising *symbolic* arguments. This is because we can :

(1) Ensure the termination of our the transformation process for any given program. In contrast, partial evaluation using BTA, may fail to terminate for non-terminating programs [JonesN88b];

(2) Accept in their natural form higher-order arguments with free variables. In contrast, the partial evaluation technique (using BTA) needs to represent higher-order arguments (with free variables) as explicit closures [Bondorf88] before they can be analysed as partially static structures [Morgensen88].

(3) Support full laziness in our transformation.

(4) Use the simpler variable-only parameter analysis.

(5) Be applied to both *grounded* and *symbolic* arguments (not variables), while BTA could only be used on *grounded* or *partially grounded* arguments.

With these advantages, there appears to be a possibility of providing an alternative to partial evaluation by enlarging the universal or higher-order deforestation transformation to include the following:

- specialisation of atomic-type arguments which are variable-only and linear
- provide reduction and simplification steps which are needed for grounded or partially-grounded terms

Further investigation is necessary to determine how this alternative form of transformation would differ from the conventional partial evaluation through BTA.

4.7. Summary of Chapter 4

In this chapter, we presented two major transformation algorithms, $\mathcal{A}$ and $\mathcal{R}$, for eliminating the following two types of higher-order features:

- general applications,
- instantiated function-type arguments, annotated as *, in function applications

We presented both the source and target expression forms of these two transformation algorithms and also proved that these algorithms will always terminate for all well-typed higher-order programs. Furthermore, to preserve full laziness during higher-order removal, we formulated an alternative transformation algorithm, called $\mathcal{L}$, which differs from the classic fully lazy lambda lifting technique of Hughes. In particular, our technique does not depend on parameter ordering and the extraction of partial applications for full laziness. Instead, our technique simply extracts out all *implicit* and *explicit* ground-type MFEs from lambda

abstractions, whenever possible. This is the way in which full laziness can be made compatible with our eliminations of non-linear instantiated higher-order arguments.

We saw two major benefits of these higher-order removal transformations. The first major benefit is that we get more efficient programs which use fewer applications and less storage heap spaces (for closures). The second major benefit is that the resulting expression form, after higher-order removal, is much simpler. This simpler form makes it easier for us to consider the extensions of other transformation and optimisation techniques (which work for first-order programs) to higher-order programs. In particular, we showed how this was done for the universal deforestation transformation of Chapter 3 by simply extending it to work for the higher-order specialised form. This extension also permitted the removal of function-type arguments in *constructor* terms.

5. Tupling

5.1. Introduction

There is a class of inefficient programs which when invoked result in many subsidiary function calls that are identical and thus cause repeated computations. A classic example of this class of program is the fibonacci function of the following definition:

```
dec fib          : num -> num;
--- fib(0)       <= 1;
--- fib(1)       <= 1;
--- fib(n+2)     <= fib(n+1)+fib(n);
```

A call to this function will normally cause the redundant evaluation of a number of identical subsidiary calls. For example, if we evaluate the function call, fib(5), using the following sequence of reductions:

```
fib(5)   →1   fib(4)+fib(3)
         →2   (fib(3)+fib(2))+fib(3)
         →3   ((fib(2)+fib(1))+fib(2))+fib(3)
         →4   ((fib(2)+fib(1))+fib(2))+(fib(2)+fib(1))
                              :
         →n                   8
```

we will encounter two identical calls to fib(3) at reduction step 2 (denoted by $\rightarrow_2$), and three identical calls to fib(2) at step 4. These redundant function calls cause the fib function to have a computational complexity of $\mathbf{O}((1+\frac{\sqrt{5}}{2})^n)$, where n is the argument value of the initial call.

5.1.1. Memoisation and Tabulation

A very general technique for avoiding the evaluation of redundant function calls is the **memoisation** technique of [Michie68] with his proposal for memo-functions. A memo-function is a special function which remembers/stores either some or all of its previously computed function calls (in a memo-table) so that the values of re-occurring function calls can be retrieved rather than re-computed.

The most general form of memoisation technique may store and remember all its memo-functions' computed calls throughout the life-time of the whole program. However, a more useful sub-class of this technique, called **tabulation** [Bird80], need only be concerned about redundancy from *subsidiary* calls of a specified recursive function (rather than calls from the whole program). This subclass is more useful because it is more amenable to program transformation techniques and contains most of the redundancy (from recursive function calls).

Within this tabulation subclass, there is a very general technique, called *exact tabulation* [Bird80], which simply remembers all previously computed subsidiary calls for re-use. This exact tabulation technique can be applied to any recursive function definition and requires no special conditions from the program. We can even have a functional formulation of the

transformed function by passing its memo-table around as both parameter and result of the new function, as illustrated in [Khoshnevisan87].

Given a schematic function of the form:

```
dec f: alpha -> beta;
- - -f(x)              <= If Base(x) then K(x)
                         else H(x,f(c(x)),f(d(x)));
```

It is possible to transform this function to an equivalent function which uses a fully tabulated function, f_new, as shown below.

```
dec f_new: alpha # table(alpha # beta) -> beta # table(alpha # beta);
- - -f(x)              <= u where (u,_)==f_new(x,empty_table);
- - -f_new(x,tab)      <= If Base(x) then (K(x),tab)
                         else let (flag,ans)==retrieve(tab,x) In
                             If flag=found then (ans,tab)
                             else let (a1,ntab1)==f_new(c(x),tab) In
                               let (a2,ntab2)==f_new(d(x),ntab1) In
                               let ans==H(x,a1,a2) In
                                  (ans,insert(ntab2,x,ans));
```

This tabulated function uses a suitable memo-table of type table(alpha # beta), to store computed subsidiary calls, together with corresponding insertion and retrieval functions for the memo-table.

Though general, exact tabulation suffers from two problems. The first is that the transformed function has a rather complex algorithmic structure. This makes it very difficult for us to subject the tabulated function to further program transformation techniques, e.g. conversion-to-iteration. A second more serious problem is that it suffers from excessive table management overheads. This is because entries inserted in the memo-table are not normally deleted even after they are no longer required. Thus, the performance of such tables, for both insertion and retrieval operations, will deteriorate when large number of subsidiary calls occur during computation. Other means of providing more efficient tables should therefore be favoured, whenever possible, but they do require program analysis.

5.1.2. Tupling Transformation

One class of programs in which we are interested is those programs whose redundancy can be removed by *static-sized* tables[1]. This class of programs can be handled by a transformation technique, known as **tupling**[2] [Burstall77,Pettorossi84]. The tupling transformation technique uses the simple tuple structure (as a static-sized table) for grouping together a collection of potentially re-usable function calls to remove redundancy.

---

1 table whose size is known or deducible (from program structure) at compile-time

2 Tupling can either be used for removing redundancy or for horizontal loop combination (as we saw for function av_list in Section 1.3.3). In this chapter, we will be concentrating on tupling to remove redundant function calls. Those programs which require tupling for loop combination can be considered a special case of this where are no redundancy.

For example, an appropriate tuple structure for removing the redundancy in the earlier fib definition is (fib(n+1),fib(n)). This tuple expression can be used to define a new tuple function:

```
dec fib_tup : num -> (num,num);
- - -fib_tup(n)        <= (fib(n+1),fib(n));
```

which can be transformed by initially narrowing the call, fib(n+1), followed by a sequence of steps involving unfolding, **where** abstraction and folding, as shown below.

```
- - -fib_tup(n)        <= (fib(n+1),fib(n));                    ! narrow fib(n+1)

- - -fib_tup(0)        <= (1,fib(0));                           ! unfold fib(0)
                       <= (1,1);
- - -fib_tup(n+1)      <= (fib(n+1)+fib(n),fib(n+1));           ! abstract fib(n+1),fib(n)
                       <= (u+v,u) where (u,v)==(fib(n+1),fib(n)); ! fold fib-tup
                       <= (u+v,u) where (u,v)==fib_tup(n);
```

Note, in particular, the use of the **where** abstraction to collect together the fib calls, followed by a fold step to ensure that this redundancy is eliminated throughout the entire recursion of the above function. This combination of **where** abstraction and folding has been referred to as *forced folding* [Darlington81].

After the above transformation, we can obtain an efficient version of the fibonacci function by either:

(1) applying another forced fold in the recursive equation for fib, as follows:

```
- - -fib(0)            <= 1;
- - -fib(1)            <= 1;
- - -fib(n+2)          <= fib(n+1)+fib(n);                      ! abstract fib(n+1),fib(n)
                       <= u+v where (u,v)==(fib(n+1),fib(n));   ! fold with fib_tup
                       <= u+v where (u,v)==fib_tup(n);
```

(2) or re-define the fib function as:

```
- - -fib(n)            <= v where (_,v)==fib_tup(n);
```

The second option would give us the following program for fibonacci function which uses the efficient linear recursive tuple function, fib_tup.

```
- - -fib(n)            <= v where (_,v)==fib_tup(n);
- - -fib_tup(0)        <= (1,1);
- - -fib_tup(n+1)      <= (u+v,u) where (u,v)==fib_tup(n);
```

This program is more efficient than an equivalent memoised function because of the use of a static-sized tuple with constant retrieval time. In addition, we can subject the above function to a further transformation using the conversion-to-iteration tactic of Chapter 2. This further transformation can make use of the constant-at-termination and special right-commutative property within the definition of fib_tup to produce a space-efficient, tail-recursive function, fib_it, as shown below.

```
- - -fib_it(0, (u,v))      <= (u,v);
- - -fib_it(n+1,(u,v))    <= fib_it(n,(u+v,u));
```

where:

```
- - -fib_tup(n)          <= fib_it(n,(1,1));
```

Our primary concern in this chapter will be the tupling transformation. The main steps used by this transformation are the define, unfold, **where** abstraction and fold steps. Of all these steps, the most difficult is the step which introduces an appropriate tuple function definition. Due to this difficulty, this step is frequently referred to as a **eureka step** and is the key step for automating the tupling transformation.

In this chapter, we shall present a very general analysis technique which could automatically discover eureka tuples for a large class of programs. We begin our presentation by describing a common tabulation analysis tool, called Dependency Graph (DG for short, see [Bird80]). This tool could be used to analyse for appropriate tables which would remove redundant calls. After that, we discuss in Section 5.3 a range of past analysis techniques. These past techniques can be classified into two main approaches, namely those which rely on algebraic properties of programs and those which perform direct analysis with the DG. One direct DG analysis technique, proposed in [Pettorossi84], provides a very suitable formal framework for tupling analysis but it presently lacks a systematic method for mechanically finding the appropriate eureka tuple. We describe Pettorossi's tupling framework in Section 5.4, before presenting extensions that would give a mechanisable tupling analysis technique.

Our tupling analysis technique employs selection orderings to search for eureka tuples which could be used for tupling transformations. We shall show this new tupling analysis to be mechanisable and widely applicable in Section 5.5. Furthermore, in Section 5.6, we shall show a simple extension which would help to widen our analysis even further. Lastly, in Section 5.7, we examine the application boundaries of the tupling tactic by looking at two inherent limitations.

5.2. Dependency Graph

The **dependency graph** of a function call is basically a compressed representation of the evaluation tree for the function call which shows the calling structure of the resulting subsidiary recursive calls. This compressed representation is made into a graph by merging all function calls with identical arguments together. An example of this compression from a tree to a graph is shown in Figure 5.1 for the function call, fib(4). Each node and leaf in this compressed DG represents a unique function call that is connected by directed arcs which always points from parent function calls to the children (or subsidiary) function calls.

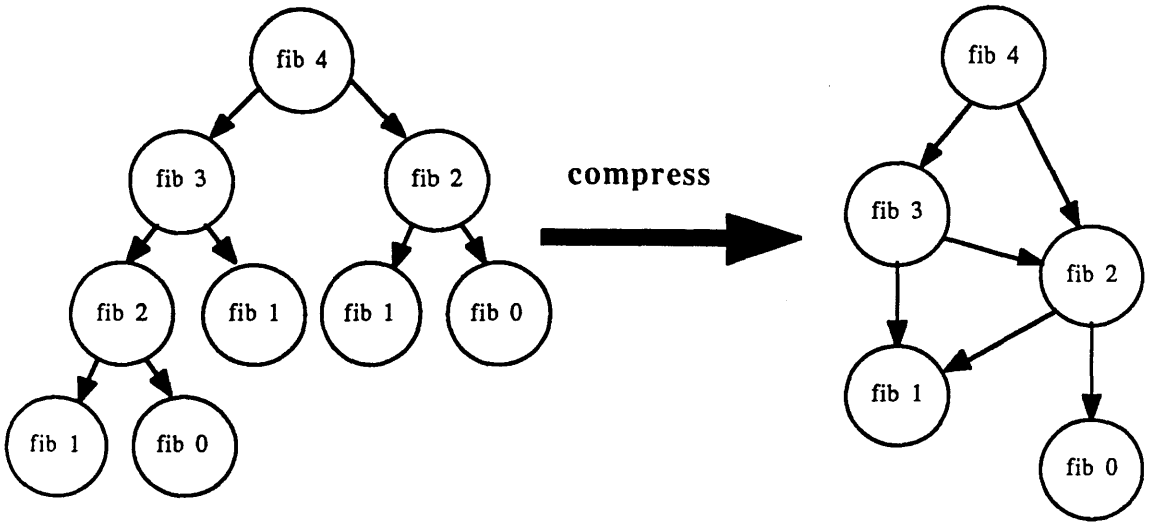


Figure 5.1 - Compression of a Tree of calls to a DG.

This DG for fib(4) is constructed using grounded function calls and is therefore an instance of *grounded DGs* which are normally finite in size. Apart from grounded DGs, we may also have *symbolic DGs*, based on symbolic function calls, which are potentially infinite in size. An example of a symbolic DG is shown in Figure 5.2.

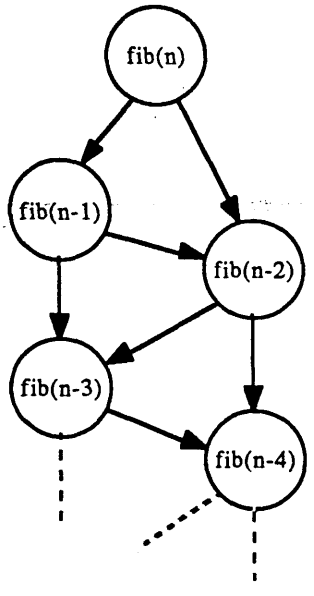


Figure 5.2 - Symbolic DG for `fib(n)`

Both forms of DGs exhibit *redundancy patterns* which exist among subsidiary calls. They are therefore useful for analyses to find suitable tabulation techniques to eliminate redundancy. We shall see how DGs can be used for this purpose later. But first, we examine two important issues of dependency graphs.

5.2.1. Relevant Subsidiary Function Calls

For simplicity, most previous analysis techniques [Cohen83, Bird80] only considered DGs where each node (in the graph) represents a call from the *same* function. This is suitable for functions, like fib, with only self-recursive calls but not for programs which have *mutually* or *auxiliary* recursive function calls among their redundant calls. Previously, these other functions are considered to have been converted to self-recursive definitions before analysis. However, a

slightly better approach is to directly handle both auxiliary and mutually recursive function calls in the DGs. This would avoid the need to convert programs to self-recursive definitions.

We propose a minor extension to DGs which will consider a set of functions, say $f_1..f_n$, for analysis. These functions will be of the form:

$$\text{---}\ f_i(x) \quad \Leftarrow \quad \textbf{if } B_i(x) \textbf{ then } C_i(x) \quad \textbf{else } H_i(x, f_{i1}(N_{i1}\ x), ..., f_{ia}(N_{ia}\ x)); \qquad (5.1)$$

where the subsidiary calls are $f_{i1}(N_{i1}\ x),...,f_{ia}(N_{ia}\ x)$. With some restriction[3] on the possible form of H_i, the function of (5.1) is similar to the *multi-linear* class of functions used in [Khoshnevisan87]

Notice that each of the subsidiary calls, $f_{ij}(N_{ij}\ x)$, of the above function can either be:

- *self-recursive* if $f_{ij} = f_i$
- *mutual-recursive* if f_{ij} is a mutual recursive function of f_i .
- *auxiliary-recursive* if f_{ij} is an auxiliary function of f_i

These subsidiary calls will be the calls which are needed to show redundancy patterns in DGs. We shall refer to them as the **relevant** subsidiary calls. They will include:

1) both self and mutual recursive function calls, and
2) auxiliary recursive calls which share some *common recursion parameters* with the main recursive function under analysis.

The self and mutual recursive calls can be trivially found by examining the recursion structure of function definitions. As for the auxiliary calls, we must be able to tell if it shares a common recursion argument with its parent recursive function. A recursion parameter is a parameter in a recursive function definition which is either increasing or decreasing in some measure (or value). A *common recursion parameter* between an auxiliary and its parent function is an identical parameter which shares this progressive nature (of being either increasing or decreasing in value) over successive recursion calls.

An example of a common recursion argument in the following function is the parameter, n.

```
dec factlist : num -> list(num);
dec fact: num -> num;
--- factlist(n)    <= if n=0 then []
                      else fact(n)::factlist(n-1);
--- fact(n)        <= if n=0 then 1
                      else n*fact(n-1);
```

Notice that the two subsidiary calls, fact(n) and factlist(n-1), in the body of the main function factlist share a common variable, n. This numeric parameter is subjected to a recursion which decreases its value in both the auxiliary function, fact, and the main recursive function, factlist. Thus, it satisfies the requirement of being a common recursion argument. As a result, the auxiliary function call, fact(n), should be considered as a relevant subsidiary call for DG analysis.

---

[3] One simple restriction is that the functions, $f_1..f_n$, must not be a part of H_i.

5.2.2. Requirements of DG analysis

Apart from finding relevant subsidiary calls in DGs, there are two other important requirements which must be satisfied before a given program (a set of recursive functions) is considered as suitable for DG analysis. These two requirements are:

1) *The set of recursive functions must be non-linear*, where a set of functions is considered to be non-linear if there exists two or more relevant subsidiary calls in at least one of the function definitions. If this condition is not satisfied, there will simply be no redundant calls which can be tabulated.

2) *There must be a partial order (transitive relationship) among relevant subsidiary calls.* This is an important requirement because tabulation techniques can only be successfully applied if there are no cyclic dependencies among function calls. If cycles exist, no amount of tabulation is able to remove the redundancy. Because of this, the DGs which are constructed for redundancy analysis must be acyclic. This is why such DGs are also known as *minimal compressed directed acyclic graphs* (m-dag) in [Cohen83].

The first requirement can always be syntactically detected, while the second requirement of detecting for partial ordering would need heuristics based on data domain orderings. The main advantage of having these requirements (or pre-conditions) is that they can provide a means by which *unsuitable* program definitions can be rejected before the DG analysis.

5.3. Past Analysis Techniques

A number of different analysis techniques have been proposed in the past for finding the appropriate table structures for removing redundancy. Most of these techniques either *directly* or *indirectly* analyse the redundancy patterns in function definitions with the help of dependency graphs of function calls.

We classify these techniques into two main approaches:

1) Indirect Analysis using Algebraic Properties of Programs
2) Direct Analysis using DGs

5.3.1. Analysis using Algebraic Properties of Programs

The indirect analysis approach relies on the presence of certain algebraic properties in programs to suggest appropriate tables for removing redundancy. [Cohen83] proposed a technique using this approach by carefully identifying four useful conditions (or relationships) between **descent functions** of certain classes of recursive programs, where the descent functions are those functions which are applied to the arguments of subsidiary recursive calls.

In particular, a schematic function (shown below) with two self-recursive calls, f(c(x)) and f(d(x)), whose descent functions are c and d respectively was employed to illustrate Cohen's four useful descent conditions.

```
---f(x)        <= if b(x) then k(x)
                 else h(x,f(c(x)),f(d(x));
```

Each of these descent conditions corresponds to a class of programs with a particular redundancy pattern. The four classes of redundancy, together with their corresponding descent conditions, are specified below. They fall in a linear hierarchy with each condition progressively more general (or weaker) than the condition before it.

1) Explicit Redundancy, where c=d.
2) Common Generator Redundancy, where there exists a common function, g, such that $g^m=c$ and $g^n=d$, for some m,n, where g^m means g o g o .. o g, m-times.
3) Periodic Commutative Redundancy, where $c^m=d^n$ for some m,n and c o d=d o c.
4) Commutative Redundancy, where c o d=d o c.

The most interesting finding from Cohen's results is that the redundancy from each of the above four categories of programs can be eliminated by tabulating with particular kinds of tables. Cohen discovered the structure of these tables by examining the DG for each of these classes of programs.

For example, each recursive program with the common generator descent condition, where $g^m=c$ and $g^n=d$ (assuming that no common factors exist between m and n), can have a DG of the form shown in Figure 5.3, where the subsidiary recursive calls (represented by just the arguments in the figure) are positioned in a straight line.

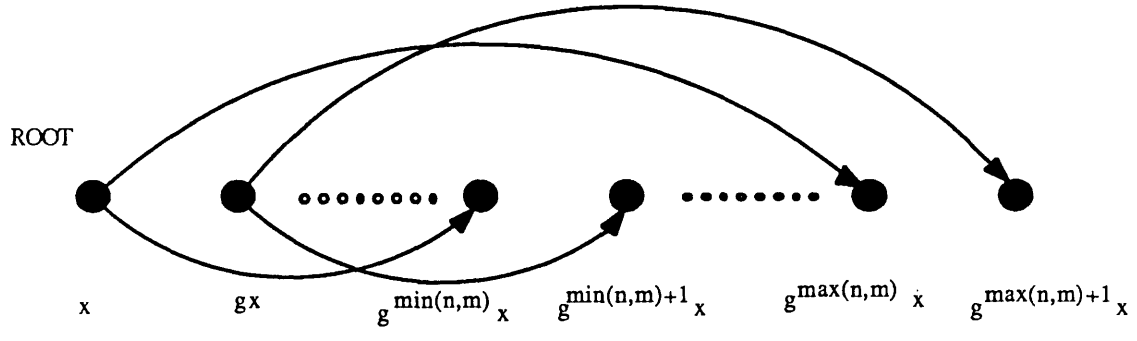


Figure 5.3 - DG for a Program with Common Generator Descent Condition

This DG can be shown to have a redundancy pattern which can be tabulated by using a table of size **max(m,n)**. This is possible because we can always compute the values for a new table, containing the calls:

$$(f(x),f(g\ x),f(g^2x)...,f(g^{max(m,n)-1}\ x))$$

from the previous table (down the recursion):

$$(f(g\ x),f(g^2x)...,f(g^{max(m,n)-1}\ x),f(g^{max(m,n)}\ x))$$

where x is the current recursion argument. This is done by computing the first call, f(x), (for the new table) using the calls $f(g^{min(m,n)}\ x)$ and $f(g^{max(m,n)}\ x)$ and then getting the remaining calls from the previous table. An example of this class of programs is the earlier fib function which needs a table containing two items to remove redundancy.

Similarly, programs with periodic commutative descent condition can also be handled by *fix-sized* tables[4], as the DG has a regular redundancy pattern. (see Section 5.5.5.2 for a detailed look into this class of programs)

However, programs with commutative redundancy require *dynamic-sized* tables[5]. This is because the dependency graph of this class of programs has a width that broadens as it gets further away from the root, as shown in Figure 5.4. An example of this class of programs is the binomial function of the following definition.

```
dec binomial: num -> num;
- - - binomial(n,m)   <=  if n=0 or n=m then 1
                          else binomial(n-1,m-1)+binomial(n-1,m);
```

With a weaker dependency, this class of programs could only have its redundancy removed by a table whose size is big enough to cater for the lowest (and widest) level of the DG. This size is dependent on the initial parameter value of the root function call. In Cohen's approach, this was still acceptable because he used dynamic arrays for tabulation. In contrast, our analysis for tupling transformation would *not* work for this class of programs. This is because we are using static-sized tables.

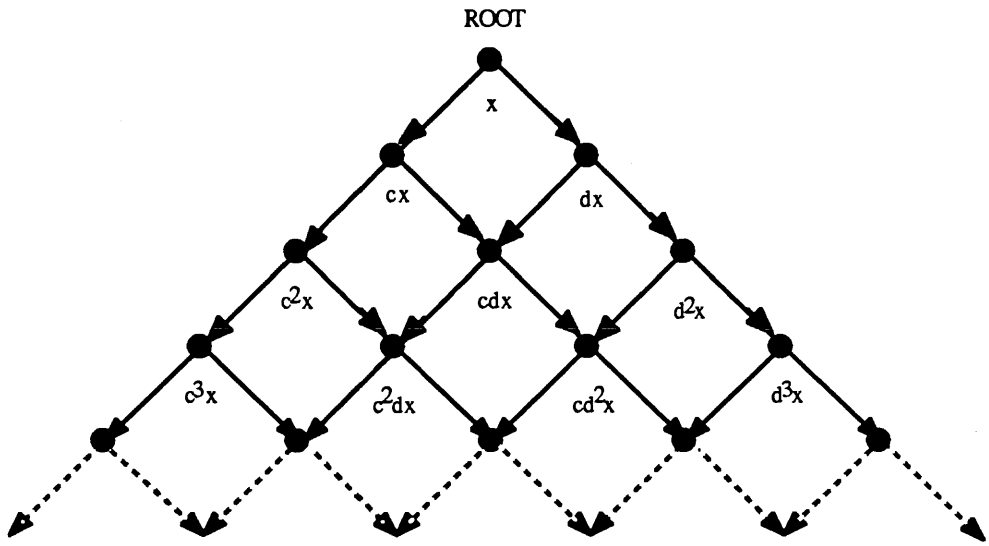


Figure 5.4 - DG for a Program with Commutative Descent Condition.

Closely related to this approach of analysing through relationships between descent functions is the corresponding analysis using function-level reasoning by [Khoshnevisan87]. He used the multi-linear class of self-recursive functions to help identify programs with the common

[4] tables whose size does not change during computation, but may be dependent on initial parameter values of function calls

[5] tables whose size may change during computation

generator property. This analysis was used to provide table managers for a variant[6] of memo-functions, which could automatically eliminate entries (from memo-tables) that are no longer needed. The main advantage of this approach is that full memoisation can still be used when common generator analysis fails to find a suitable table-manager.

Further work in [Harrison88a] used the list construction in FP [Backus78], in a manner similar to tupling, to eliminate redundancy. By using additional analysis conditions (or theorems), Harrison extended his area of applicability to include some auxiliary and mutually recursive functions. These extra analysis conditions essentially specify when *tupling* of auxiliary or mutually recursive functions can be done in order to obtain self-recursive functions (with the common-generator descent condition).

Though fairly extensible and mechanisable, there are still other classes of programs (e.g. Tower of Hanoi) which are not yet addressed by these approaches. This is due primarily to the difficulty of analysing for all possible conditions, without getting ad-hoc.

5.3.2. Analysis using Dependency Graph

A second more general approach for analysing the redundancy pattern of recursive functions is to make direct use of the dependency graph. Richard Bird pointed to the possibility of using *pebbling game* [Pippenger80], which has been extensively studied and used in problem domains involving space-time trade-offs, as a general technique for this purpose. The main objective of this game is to search for a suitable pebble placement sequence, which uses the minimum number of pebbles, to visit the entire graph from the leaves to the root. The basic rules permitted in this game are:

1) remove a pebble from a node or a leaf
2) place a pebble on a leaf
3) place a pebble on a node whose children are all currently pebbled.

The successful discovery of an appropriate pebbling sequence can help to identify a correct tabulation sequence for removing redundancy. However, this game is essentially bottom-up and requires the use of grounded DGs. These DGs must in turn be big enough to exhibit the inherent redundancy pattern of programs. The difficulty of determining an appropriately sized DG together with the choice of a suitable pebbling algorithm casts some doubts on the viability of this approach as a mechanisable technique.

Instead of the bottom-up approach of pebbling games, we can also use a top-down approach based on *symbolic* DGs. Early work in [Darlington81] involved an analysis function, called FindCleverTuple[7], which performs top-down search in the DG for a pair of tuples that matches. This analysis function uses a heuristic search that works for a sub-class of programs but is not generally applicable.

---

[6] known as Self-Garbage-Collecting (SGC) memo-functions
[7] following an earlier suggestion in [Burstall77].

Subsequent work in [Pettorossi84] also used this top-down approach on symbolic DG to find the eureka tuple for removing redundancy. In particular, he rigorously explored this approach and managed to present a formal framework for tupling analysis. This framework is extremely helpful but is incomplete because it still lacks a systematic method for mechanically discovering matching tuples. We examine Pettorossi's framework next and present important extensions which could help us overcome this shortcoming.

5.4. Pettorossi's Framework

Central to Pettorossi's framework for analysing eureka tuples is the notion of a **cut**[8]. This is simply a subset of nodes across a DG that when removed, will divide the graph into two disconnected halves. With this notion, Pettorossi's central thesis is that:

> An appropriate eureka tuple can be found if and only if there exists a **progressive sequence** of cuts, that *match* one another, in the function's dependency graph.

A progressive sequence of cuts is one where each cut in the sequence is in some sense *smaller* than the previous cut. A pair of cuts (each containing a set of function calls) is said to *match* if there is a one-to-one mapping from the first cut to the second cut, such that a *consistent substitution* is obtained when each function call of the first cut is matched with the corresponding function call of the second cut. (We shall give a more formal definition of matching between pairs of cuts later in Section 5.5.)

The terminologies used for this formulation can be more precisely described by the following list of definitions, which we have reproduced (with modifications) from [Pettorossi84].

Definition 5.1 - Dependency Graph

> A dependency graph is a rooted acyclic graph which can be formally defined by the pair, (G,children), where G is the complete set of nodes in the graph, while children is the successor function of the type:
>
> children: node -> set(nodes)

From this basic successor function, children, we can also define the following additional functions between nodes in the DG.

$$\begin{aligned} \text{parents}(x) &= \{y \mid x \in \text{children}(y)\} \\ \text{descendants}(x) &= \text{children}(x) \cup \{z \mid y \in \text{children}(x) \wedge z \in \text{descendants}(y)\} \\ \text{ancestors}(x) &= \{y \mid x \in \text{descendants}(y)\} \end{aligned}$$

Each of these functions presently takes a node in order to return a set of nodes (i.e nodes $\rightarrow$ set(nodes)) However, we may at times consider the corresponding *set* functions of the type, set(nodes) $\rightarrow$ set(nodes), with the same name. Thus given a function, f, we can define its corresponding set function, say f_{set}, by:

$$f_{set}(s) = \bigcup_{\forall a \in s} f(a)$$

Furthermore, as the dependency graph, G, must be acyclic and contain only one root, it must also satisfy the following:

---

[8] called *slice* in Darlington's FindCleverTuple analysis

- there can only be one root node, root, such that
  $root \in G \wedge parents(root) = \{\}$
- the graph must not contain cycles,
  $\forall n \in G \,.\, n \notin descendants(n)$

Inherent within each DG is a transitive ordering among nodes, called the *ancestor-descendant* ordering (AD-ordering). This ordering is defined as follows:

Definition 5.2 - Ancestor-Descendant Ordering

The *ancestor-descendant* ordering (AD-ordering), denoted by $>_{AD}$, is defined by:

$$x >_{AD} y \text{ IFF } x \in ancestors(y)$$

A cut, c, is a subset of nodes from a dependency graph, G, which when removed divides the graph into two disconnected halfs. This definition of a cut is given below.

Definition 5.3 - Cut

Given:

$top_half(x) = ancestors(x) - x$
$bottom_half(x) = descendants(x) - x$

A sub-set of nodes, c, from a dependency graph with nodes, G, is considered to be a **cut** if it satisfies the following:-

$$\forall t \in top_half(c).\ children(t) \cap bottom_half(c) = \{\}$$

This condition essentially states that the two halfs of the DG (separated by the cut) are disjoint.

A progressive sequence of cuts is a series of cuts where each new cut is in some sense *smaller* than the cut just before. A cut is said to be *smaller* than its previous cut if

1) for every node in the current cut which is not in the previous cut, there is a node in the previous cut that is bigger than it; and
2) for every node in the previous cut which is not in the current cut, there is a node in the current cut that is smaller than it.

This can be formally defined as follows:

Definition 5.4 - Progressive Sequence of Cuts

A sequence of cuts, $cut_1, cut_2, ..., cut_N$, is said to be *progressive* if:

$$\forall i \in 1...N\text{-}1.\ cut_{i+1} \not\subseteq cut_i \wedge cut_{i+1} \not\supseteq cut_i$$
$$\wedge \forall m \in cut_i - cut_{i+1}.\ \exists n \in cut_{i+1}.\ m >_{AD} n$$
$$\wedge \forall n \in cut_{i+1} - cut_i\ .\ \exists m \in cut_i.\ m >_{AD} n$$

These definitions provide a formal framework for Pettorossi's thesis on tupling analysis. But his result currently lacks a general technique for *mechanically* discovering progressive sequences of matching cuts. As a consequence, the examples given in [Pettorossi84] have been largely hand-analysed with some non-optimal results given (see Section 5.5.5.2). One heuristic suggested by Pettorossi involved the tupling of calls which share sub-computation (immediate successor calls) during the transformation (symbolic evaluation). However, this heuristic does not appear to be generally applicable.

Our work can be seen as an extension of Pettorossi's framework. In particular, we shall show that it is possible to provide a systematic and general means by which matching tuples can be mechanically discovered.

5.5. Our Tupling Analysis Technique

Our tupling analysis technique is based on a systematic, top-down exploration of the DG, in order to find a pair of matching tuples as its eureka tuple. This exploration starts with the root node as its first cut (or frontier) and then successively builds the next cut by *giving up* at least one node from the current cut, in order to *acquire* corresponding successor nodes for the next cut. The *giving up* of a node (function call) and the *acquisition* of its children (subsidiary calls) corresponds essentially to an *abstract* form of unfolding which reveals only the relevant subsidiary calls (rather than the complete RHS of an equation).

An important requirement of this process is that all the nodes that have been given up must not be subsequently re-acquired. This requirement is important because it can ensure that successive cuts (frontiers) are *progressive* in nature. It could be adhered if at all times we give-up only those nodes in the current cut which are topmost. A **topmost** set of nodes from a cut is defined as the set of nodes containing only calls that are *biggest* or *maximal* with respect to the ancestor-descendant ordering, $>_{AD}$. This topmost set can be formally defined as follows.

Definition 5.5 - Topmost Set

The topmost set, $topmost_i$, from a cut, cut_i, is a set of nodes whose ancestors are not present in the cut itself. This condition is satisfied by the following definition.

$$topmost_i = \{ t \mid t \in cut_i, \neg (\exists x \in cut_i.\ x >_{AD} t) \}$$

With this requirement, the sequence of cuts from this systematic top-down exploration will be a *special case* of the progressive sequence of cuts. This special sequence will systematically cover all nodes in its exploration. In particular, it will visit all nodes between the root and the latest frontier. We call this sequence, the **continuous sequence of cuts**, and define it formally as follows:

Definition 5.6 - Continuous Sequence of Cuts

A *continuous* sequence of cuts, $cut_1, cut_2, ..., cut_N$, is a successive series of cuts which starts with the root node as its first cut. This sequence successively obtains the next cut by giving up a subset of nodes, $selected_i$, from the topmost set, $topmost_i$, of the current cut, cut_i, in order to acquire the children, $newnodes_{i+1}$, for the next cut, cut_{i+1}.

The constructive definition for a continuous sequence of cuts can be stated as follows:

$$topmost_i = \{ t \mid t \in cut_i, \neg (\exists x \in cut_i.\ x >_{AD} t) \}$$

$$selected_i \subseteq topmost_i \wedge selected_i \neq \{\}$$

$$newnodes_{i+1} = \{x \mid t \in selected_i,\ x \in children(t) \}$$

$$cut_1 = \{root \mid parents(root)=\{\} \}$$

$$cut_{i+1} = cut_i - selected_i \cup newnodes_{i+1}$$

This new notion of a *continuous* sequence of cuts provides a more viable technique for systematically finding the eureka tuple. In particular, our tupling analysis is considered to be successful if we can find a pair of cuts in the continuous sequence which matches. This search can be formulated by the following abstract specification.

Find a sequence of continuous cuts, $cut_1, cut_2, cut_3, \ldots cut_N$ such that

$\exists j.\ 1<j<N \wedge match(cut_j, cut_N)$

The matching test between two cuts, each containing a set of function calls, is done by a function, called match. This function performs a kind of force matching, which succeeds when there exists a one-to-one mapping, m, from the first set to the second set, such that their corresponding one-way matchings produce a consistent substitution, σ, as specified below.

$match(cut_j, cut_N)$ **IFF** $\exists\sigma\ \exists m.\forall x \in cut_j\ .\ m(x)=x\sigma$

where m is a one-to-one mapping from cut_j to $cut_N \wedge \sigma$ is a consistent substitution

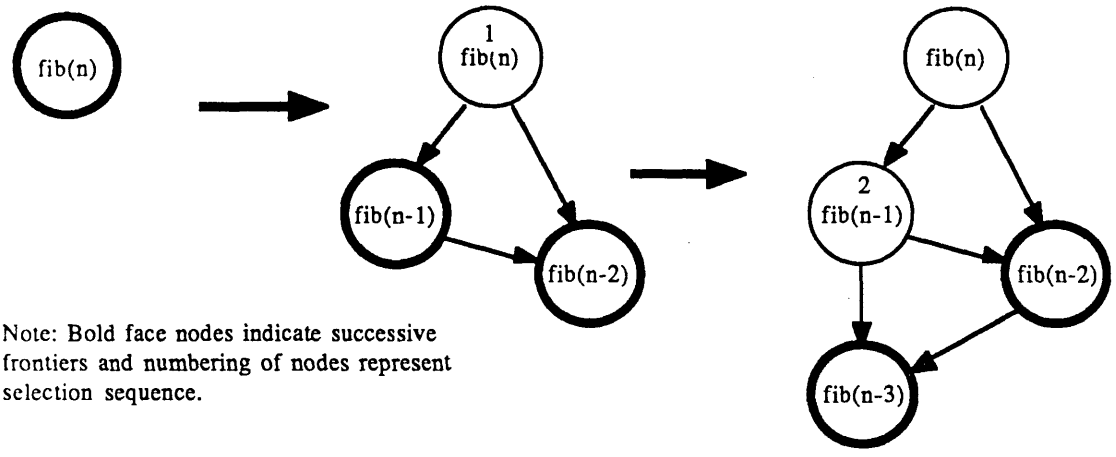
The above specification to construct a continuous sequence of cuts is both *semi-decidable* and *non-deterministic*. It is *semi-decidable* because there are some programs, e.g binomial function, which do not have any static-sized eureka tuple. These programs can cause successive cuts to be increasing in size, resulting in non-termination for our analysis. In order to prevent this, we will need to place an upper bound on our analysis algorithm. This upper bound need not be very large as most programs require only small-sized tuples to eliminate redundancy.

Also, our analysis algorithm is *non-deterministic* because there could be more than one ways in which selected sub-sets of nodes are taken from the topmost set (see Definition 5.6). As a result, it is possible to have more than one continuous sequence of cuts from any given program. We shall see later that this is not a serious problem. This is because there exist good heuristics which could make our analysis deterministic, whilst still finding eureka tuples. But first, we shall illustrate our tupling analysis technique with a simple example where there is a *unique* sequence of continuous cuts.

5.5.1. An Example

We illustrate informally how the systematic construction of continuous sequences of cuts can find us a matching tuple, using the fib function (from Section 5.1) as an example.

Starting from the main function call, fib(n), as the first cut, our analysis will replace this call by its two subsidiary calls, {fib(n-1),fib(n-2)}, to give the second cut. This second cut contains only one topmost item, fib(n-1), because the other call, fib(n-2), is a subsidiary call of this topmost call. It can therefore be unfolded to give the third cut, {fib(n-2),fib(n-3)}, which incidently matches the second cut - thus giving us a matching tuple. This search for a pair of matching tuple can be viewed graphically in Figure 5.5, where the systematic exploration of the DG is represented by a successive sequence of frontiers (shown highlighted).



Note: Bold face nodes indicate successive frontiers and numbering of nodes represent selection sequence.

Figure 5.5 - Successive Frontiers (Cuts) During the Exploration of a DG

Each successive frontier of calls is obtained from the previous frontier by choosing the appropriate calls to unfold. We will frequently represent this systematic exploration[9] by just a list of cuts, as shown below, with calls that are selected for unfolding in **bold** faces.

NO	CUTS
cut_1	**fib(n)**
cut_2	**fib(n-1)**,fib(n-2)
cut_3	**fib(n-2)**,fib(n-3) matches[10] cut_2

Our analysis finds a pair of matching tuples by the *unfolding* of appropriately selected calls and uses *matching* as a means of testing for successful folding. These steps essentially mirrors the necessary unfold/fold steps of the subsequent tupling transformation. Thus, when this analysis succeeds, the major task of the tupling transformation would also have been accomplished. Tupling analysis technique therefore provides us with both a way of determining the right tuple, as well as the essential steps needed in the tupling transformation.

We shall henceforth assume that once the tupling analysis has succeeded, the transformation steps for obtaining a more efficient program automatically follow from the analysis.

5.5.2. Selection Orderings for Deterministic Analysis

One deterministic method for constructing the continuous sequences of cuts is to introduce an appropriate *ordering* for *selecting* nodes to unfold in the construction. Let us call an ordering which is used for this purpose, the **selection ordering** and denote it by $>_{sel}$. With this selection ordering, the nodes which are selected (for unfolding) from each cut, cut_i, shall be all those which are maximal w.r.t. $>_{sel}$, as shown below:

$$selected_i = \{ t \mid t \in cut_i , \neg (\exists x \in cut_i .\ x >_{sel} t) \}$$

---

[9] This systematic exploration can also be viewed as a repeated failure analysis. When a tuple (cut) fails to find a descendant tuple which matches, the resulting tuple (often a larger tuple) is used as the new (generalised) tuple for further analysis. However, we also keep a history of cuts for matching because two matching tuples may be separated by several intermediate stages.

[10] Note that (n-c) is just a short-hand notation for pred(..pred(n)) c times where pred(n)=n-1 and c is a constant. This representation, pred(..pred(n)), is easier for matching than expressions of the form, n-c.

Also, all these selected nodes must belong to the topmost set (of the current cut) if we are to generate *only* continuous sequences of cuts. This means that the chosen selection ordering must always be *at least as strong* as the ancestor-descendant ordering, $>_{AD}$, as specified below:

$$\forall x \forall y.\ (x >_{AD} y) \rightarrow (x >_{sel} y)$$

Even with this restriction, there are still numerous possible orderings which could be used to generate continuous sequences of cuts. In this chapter, we shall only consider two important classes of selection orderings, namely:

. AD-Based Selection Orderings.
. Function-Argument Selection Orderings.

The first class of selection orderings, based on the AD-ordering, is primarily of *theoretical* interest. They will help to illustrate the extent to which our analysis technique could be applied but is not practical for two reasons. The first is that AD-based orderings require, in general, the *complete* construction of *grounded* DGs[11] before the true ordering relationships could be determined. This full construction is expensive and wasteful. A second more serious problem is the difficulty of determining, in advance, the appropriate size of grounded DG to use for tupling analysis. This appropriate size varies according to the program we are dealing with.

The second class of selection orderings is more practical because we are <u>not</u> required to construct complete DGs. This class of orderings is dependent on the possibility of determining a transitive-ordering with the help of function-argument call pairs. Determining such a transitive ordering is, in general, an undecidable problem (cf halting problem). Nevertheless, we shall show that widely-applicable heuristics exist to help us perform this task.

We shall describe these two classes of selection orderings, in the next two sections, using the following factlist definition as our running example.

```
- - - factlist(n)    <= if n=0 then []
                       else fact(n)::factlist(n-1);
- - - fact(n)        <= if n=0 then 1
                       else n*fact(n-1);
```

Redundancy occurs among the auxiliary recursive fact calls and we will look at how these selection orderings can help us find the eureka tuple.

5.5.3. AD-Based Selection Orderings

In this section, we shall present two possible selection orderings which are based on the AD-ordering. One of them is based on the AD-ordering itself (as the selection ordering), while the other is an enhanced version of the AD-ordering.

---

[11] Notice that symbolic DGs cannot, in general, be used to determine AD-ordering relationships. This is because given any set of nodes, $\{n_1,..,n_k\}$, which represents the current frontier (or cut) of a DG, it is in general *undecidable* how far we should explore the DG before we can have sufficient AD-ordering relationships.

5.5.3.1. AD-Ordering

If the AD-ordering is used as the selection ordering, we shall be selecting the maximum permitted subset of nodes (i.e the topmost set) to unfold at each time and this tends to produce a continuous sequence of cuts which is being explored most quickly.

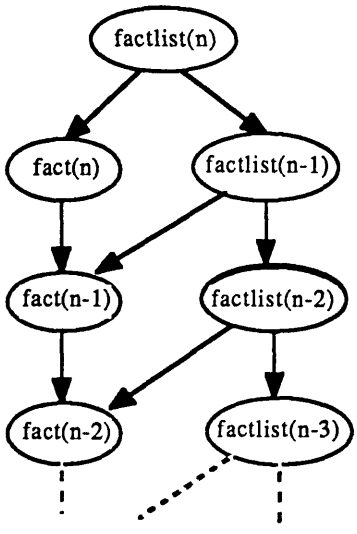


Figure 5.6 - Symbolic DG for factlist definition

To illustrate how the AD-ordering could be used, we give a partial construction of the symbolic DG for factlist in Figure 5.6. We assume that this partially constructed DG is sufficiently large to exhibit all the AD-ordering relationships which will be needed. Using these AD-ordering relationships of factlist, the continuous sequence of cuts that will be generated, is shown below.

NO	CUTS
cut_1	**factlist(n)**
cut_2	**fact(n),factlist(n-1)**
cut_3	**fact(n-1),factlist(n-2)**

In this example, the entire cut is selected at each stage (for unfolding) and the analysis terminates when cut_3 is found to match with cut_2, to give an eureka tuple of (fact(n),factlist(n-1)).

This simple example illustrates how the very weak AD-ordering can be used to successfully find matching tuples. However, not all programs can be successfully analysed by the AD-ordering, even though eureka tuples exist. This is caused by the weakness of the ordering, which may cause part of the frontier of the DG to be explored faster than the rest. We call this phenomenon a **race-ahead** condition. It can be illustrated with a modified version of factlist, called factlist2, which recurses down its argument at twice the rate of the original function. This definition is shown below, together with its DG in Figure 5.7.

```
dec factlist2: num -> list(num);
- - - factlist2(n)      <= if n<1 then []
                           else fact(n)::factlist2(n-2);
```

When the AD selection ordering is used with factlist2, we get a continuous sequence of cuts that grows progressively larger as we explore deeper down its DG, as shown below.

NO	CUTS
cut_1	**factlist2(n)**
cut_2	**fact(n),factlist2(n-2)**
cut_3	**fact(n-1)**,fact(n-2),**factlist2(n-4)**
cut_4	**fact(n-2)**,fact(n-4),**factlist2(n-6)**
cut_5	**fact(n-3)**,fact(n-4),fact(n-6),**factlist2(n-8)**

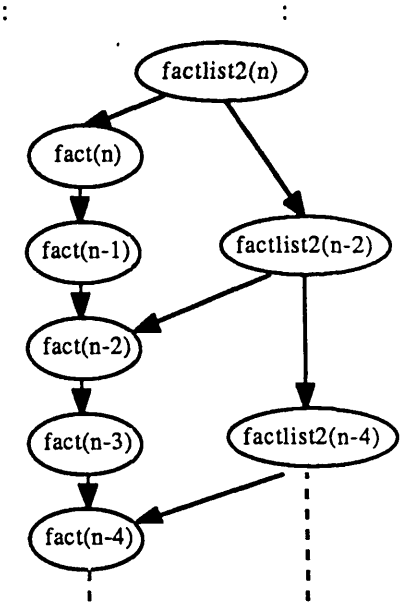


Figure 5.7 - DG for factlist2 definition

The weak selection ordering used and the *auxiliary recursive calling* structure of our example have permitted the factlist2 calls on the right side of DG to be explored (or unfolded) at a much faster rate than the fact calls on the left. As a result, we are unable to find a matching pair in-spite of the fact than an eureka tuple of (fact(n),factlist2(n-2)) is possible.

It is important to note that such race-ahead conditions could only happen in cases where there are *auxiliary* or *independently* recursive calls rather than *self* or *mutual* recursive calls. We make a distinction between recursion structures among the *function calls* (with appropriately instantiated arguments) and the recursion structures of just the *functions* (or *function names*), themselves. This is because it is always possible to change the recursion structure of *functions* without changing the recursion structures of the *function calls*. For example, by using a composite type for num and list,

```
data composite = a_list(list(num)) ++ a_num(num)
```

we could change the (auxiliary recursive) pair of functions, factlist and fact, into the following self-recursive function, f_self:

```
---f_self(1,n)            <= If n=0 then a_list([]);
                             else a_list(get_num(f_self(2,n))::get_list(f_self(1,n-1)));
---f_self(2,n)            <= If n=0 then a_num(1)
                             else a_num(n*get_num(f_self(2,n)));
---get_num(a_num(n))      <= n;
---get_list(a_list(x))    <= x;
```

such that factlist(n)=get_list(f_self(1,n)) and =/.

Though the new function is now self-recursive, the DG of its recursive calls still exhibits an auxiliary recursion structure. This is because function calls of the form f_self(1,n) could invoke recursive calls of the forms, f_self(2,n) and f_self(1,n), while calls of the form f_self(2,n) could only invoke other calls of the form, f_self(2,n). This calling structure is still auxiliary recursive.

A better characterisation for the classes of programs, which can be handled by AD-ordering, is therefore on the recursion structures of function calls. We shall give this characterisation later, after we have look at another AD-based ordering which could partly overcome the race-ahead condition.

5.5.3.2. Descendant-Enhanced Ordering

Race-ahead conditions can cause our analysis to fail and are undesirable. Fortunately, they can be avoided by using stronger selection orderings. One straightforward way of getting a stronger ordering is to introduce *extra* orderings between nodes that were *not* previously ordered (by the AD-ordering).

This stronger ordering can be formulated by adding extra orderings between the children of *unordered* nodes. Specifically, any two unordered nodes, say x and y, can be additionally ordered by checking if children(x) of the first node is *strictly greater* than children(y) of the second node. We define a set of nodes to be *strictly greater* than another set of nodes if there exists a node in the second set that is a descendant of the first set but not vice versa. With this additional ordering, we can define the descendant-enhanced ordering (denoted by $>_{DE}$), as follows:

$$x >_{DE} y = x >_{AD} y \vee \text{strictly_greater}(\text{children}(x),\text{children}(y))$$

$$\text{strictly_greater}(s1,s2) = (\text{descendants}(s1) \cap s2 \neq \{\}) \wedge (\text{descendants}(s2) \cap s1 = \{\})$$

This enhanced ordering can help to prevent the earlier race-ahead condition for auxiliary recursive function calls. In particular, we are able to order auxiliary calls in *front* of main factlist2 calls by this enhanced ordering. As an example, consider a new DG with the extra orderings, in Figure 5.8, for the earlier function, factlist2.

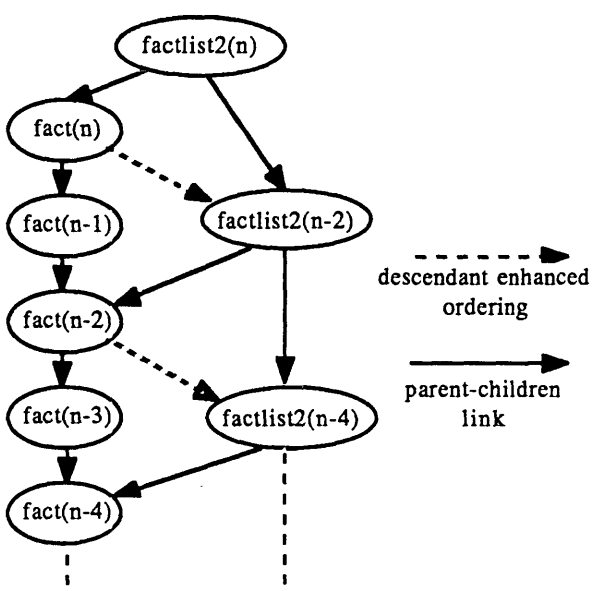


Figure 5.8 - Descendant-Enhanced DG for factlist2 definition

This ordering can be used to produce the following continuous sequence of cuts

NO	CUTS
cut_1	**factlist2(n)**
cut_2	**fact(n)**,factlist2(n-2)
cut_3	**fact(n-1),factlist2(n-2)**
cut_4	**fact(n-2)**,factlist2(n-4) which matches cut_2

where cut_4 is found to match with cut_2. Notice that in cut_2, factlist2(n-2) was prevented from being unfolded because its child node, fact(n-2), is lower in ordering than the child node of the other call, fact(n).

With such enhanced ordering, the tupling analysis is able to correctly handle programs with *auxiliary* recursive calls. However, *independently* recursive calls, which do not have common descendants between them, are still not synchronisable. This should not unduly concern us because independently recursive calls do not have redundancy between them. Nevertheless, there are still stronger orderings which could be used to synchronise independently recursive calls. These stronger orderings (to be examined later) are also more practical because they do not need to construct DGs to obtain ordering relationships. Before turning to this other class of selection orderings, we shall attempt to formally characterise recursion structures of function calls next.

5.5.3.3. A Characterisation for Recursion Structures of Function Calls[12]

In this section, we shall attempt to characterise the recursion structures of the recursive calls from a given program (containing a set of functions). Our characterisation is based on the class of programs with eureka tuples.

Let us assume that in a given program, prog, we have managed to find a continuous sequence of cuts with a pair of matching tuples. We further assume that this pair of tuples are $(a_1,..,a_n)$ and $(b_1,..,b_n)$, with a consistent substitution σ such that $\forall i \in 1..n.\ a_i\sigma = b_i$. As there is a transitive AD-ordering in the acyclic DG, we will assume that the calls in the matched tuples are positioned in a way that is consistent with its (partial) order. That is:

$$\forall i \in 1..n \neg (\exists j > i.\ a_j >_{AD} a_i)$$

With this pair of eureka tuples, we can describe the dependencies of calls in the first tuple on calls from both the tuples by a function, called depends_on. This dependency function essentially identifies the set of function calls (from the two tuples) which are needed to define each call in the first tuple. It can be formally stated by:

[12] The reader may skip this section on first reading.

Definition 5.7 - Dependency in a pair of Matching Tuples

Given a pair of matching tuples, $(a_1,...,a_n)$ and $(b_1,...,b_n)$.

The function, depends_on, which identifies the dependency of each call of the first tuple, in terms of calls in both the first and second tuples, can be defined as follows.

$$\forall i \in 1..n \quad \textbf{let } a_set = \{a_{i+1}..a_n\} \textbf{ in}$$
$$\textbf{let } b_set = \{b_i..b_n\} - a_set \textbf{ in}$$
$$depends_on(a_i) = \{b_j\} \qquad \textbf{If } b_j \in b_set \wedge a_i = b_j$$
$$= unfold(a_i, a_set \cup b_set) \qquad \textbf{otherwise}$$
$$\textbf{WHERE } unfold(a,set) = \{x \mid x \in set, x \in children(a)\} \cup$$
$$\{y \mid y \in unfold(x,set), x \notin set, x \in children(a)\}$$

With this formalisation for the matching tuples, we can now define a graph, called the tuple graph, to characterise the calling structure of the given program. This tuple graph is defined by a set of nodes, labelled {1..n}, corresponding to the calls in the eureka tuple, and the following set of arcs:

$$\{ arc(p,q) \mid p \in 1..n, q \in 1..n, (a_q \in depends_on(a_p) \vee b_q \in depends_on(a_p)) \}.$$

This set of arcs (for the tuple graph) essentially represents the recursion structure of a program where each call in the eureka tuple is 'considered' to be from a separate function. It can be used to characterise three classes of recursion calling structure for our program. We shall refer to a tuple graph (and its corresponding program) as:

1) *mutually recursive* if there is a *complete cycle* which includes every node, {1..n}, in the graph.
2) *independently recursive* if there are two or more *disjoint* sub-graphs present in the tuple graph.
2) *auxiliary recursive* otherwise, i.e. no disjoint subgraphs and no complete cycles.

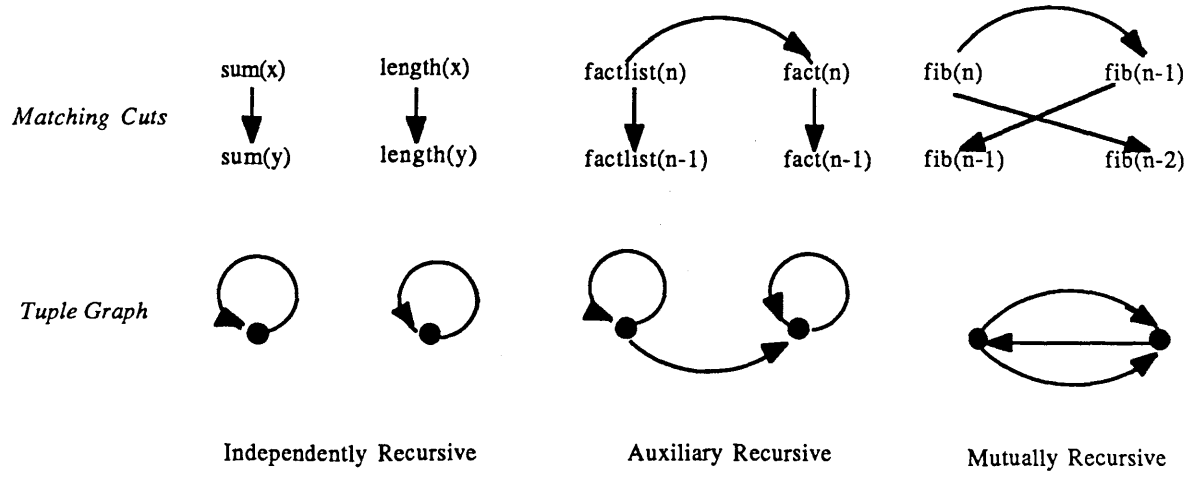


Figure 5.9 - Different Recursion Structures of Function Calls

These three different recursion structures are illustrated by examples in Figure 5.9 Those programs which have calling structures (tuple graphs) that are *mutually recursive* will always be successfully analysed using just the *AD-selection ordering*. This is possible because the cyclic

dependency will ensure that we do not have the race-ahead problem of auxiliary or independent recursions. Correspondingly, those programs with *auxiliary recursive* tuple graph could be handled by the descendant-enhanced ordering. However, programs with *independent recursions* could not be handled by any AD-based orderings.

5.5.4. Function-Argument Selection Orderings

Very often, a stronger selection ordering can be obtained if we are able to determine a (transitive) ordering that is based on the *syntactic representation* of function calls, as a pair <fn,arg>. We shall refer to this class of orderings, which is determined from the representation of function calls, as the **function-argument ordering** and denote it with $>_{farg}$.

An important pre-condition on this class of orderings is that it must be strong enough to preserve the parent-children links, as follows:

$$\forall x\, \forall y.\ x \in \mathbf{G}, y \in \text{children}(x) \rightarrow x >_{farg} y$$

with any two calls, x and y, belonging to the set of calls in DG, **G**, correctly ordered with respect to one another if they are parent-children nodes. This condition ensures that each chosen ordering, $>_{farg}$, will be at least be as strong as the AD-ordering (due to transitivity of descendants), thus:

$$\forall x \forall y.\ x \in \mathbf{G}, y \in \text{descendants}(x) \rightarrow x >_{farg} y$$

Before presenting the technique to suggest appropriate function-argument orderings, let us informally examine a simple example. Consider the previous function, factlist2, and a possible representation for its function calls, v(n-x) where x>= 0 **and** (v = factlist2 **or** v=fact). Such a representation could be used to establish a valid[13] function-argument ordering, $>_{factlist2}$, as follows:

$$v1(n\text{-}x) >_{factlist2} v2(n\text{-}y) \textbf{ IFF } (y>x) \vee (x=y \wedge v1=factlist2 \wedge v2=fact)$$

where v1(n-x) and v2(n-y) are any two function calls in the dependency graph. Notice that the *primary* means of ordering is through the main recursion arguments, n-x and n-y, with the function names, v1 and v2, playing a *secondary* role when both the arguments are identical. This form of ordering is characteristic of most function-argument orderings and can often be determined (see the next sub-section). Using it, we can generate the following continuous sequence of cuts, which finds that cut_5 matches with cut_2, to give an eureka tuple of (fact(n),factlist2(n-2)).

NO	CUTS
cut_1	**factlist2(n)**
cut_2	**fact(n)**,factlist2(n-2)
cut_3	**fact(n-1)**,factlist2(n-2)
cut_4	fact(n-2),**factlist2(n-2)**
cut_5	**fact(n-2)**,factlist2(n-4) which matches cut_2

[13] because it satisfies the parent-children ordering relationship.

This selection ordering has certainly overcome the earlier race-ahead problem of auxiliary recursion. In fact, its ordering through the common recursion parameter is even stronger than the descendant-enhanced ordering because we are able to synchronise calls from independently recursive functions. We suggest a technique (heuristic) for determining function-argument orderings next.

5.5.4.1. A Technique to Determine Function-Argument Orderings

In any given set of functions, $f_1..f_n$, of the form:

$$---f_i(x) \quad <= \quad \textbf{if } B_i(x) \textbf{ then } C_i(x) \textbf{ else } H_i(x,f_{i1}(N_{i1}\ x),..,f_{ia}(N_{ia}\ x));$$

It is, in general, undecidable whether a transitive function-argument ordering, $>_{farg}$, exists to correctly order function calls of the form, $f_j\ (N_{c1} o N_{c2} ... o N_{cr}\ x)$ where o is the function composition operator.

However, there is a general heuristic which could be used to suggest transitive-orderings for a wide class of programs. This heuristic relies on two important conditions. Firstly, we must be able to analyse the parameters of functions individually, where possible. Secondly, we must be able to identify one of the parameters as the recursion parameter for ordering purpose.

At present, the set of functions, $f_1..f_n$, contains only one composite parameter, x, for all the functions. For our technique, we need to be able to split this parameter into a group of *independent* parameters. A parameter, x, of the set of functions, $f_1..f_n$, could be *split* into m independent parameters, $x_1,..,x_m$, if the following condition holds:

$$\forall\, i \in 1..n\ \forall\, j \in 1..a\ \exists\, N_{ij1}..N_{ijm} \textbf{ st } N_{ij}\ (x_1,..,x_m) = (N_{ij1}\ x_1,..,N_{ijm} x_m)$$

This condition essentially implies that each descent function, N_{ij}, could be split into $N_{ij1},..,N_{ijm}$ in order to operate independently on each of the parameters, $x_1,..,x_m$. With this split-up, we could now have the following alternative form for each of the functions, $f_1..f_n$.

$$---f_i(x_1,..,x_m) \quad <= \quad \textbf{if } B_i(x_1,..,x_m) \textbf{ then } C_i(x_1,..,x_m) \textbf{ else } H_i((x_1,..,x_m),f_{i1}(N_{i11}\ x_1,..,N_{i1m} x_m),..,f_{ia}(N_{ia1}\ x_1,..,N_{ia1} x_m));$$

With this alternative form, we can now look for a suitable parameter, say x_j, from the group of parameters, $x_1,..,x_m$, as the *recursion parameter* for ordering purpose. A parameter, x_j, is suitable as a recursion parameter (for ordering) if there exists a *progressive* function, d, for the argument domain such that, for each function, f_i:

$$\forall\, k \in 1..a\ \exists\, z. \quad (N_{ikj} = d^z) \ \wedge\ (z \geq 1) \qquad (5.2)$$

In a sense, d, is a special form of common generator for the argument, x_j, but it is important to note that this function must be *progressive* (i.e. results in either *ascending* or *descending* values when recursively applied to some arguments). A function, d, is said to be *progressive* if

$$\forall\, y\ \forall\, z \in 1..\infty.\ y \neq z \rightarrow d^y \neq d^z$$

Examples of d functions which satisfy this criterion are d(n)=n-1 or d(n)=n+1 or d(a::x)=x. An example which does not satisfy this criterion is d(n)=n mod 2. We can recognise such functions

by associating a set of common d-type functions for each data domain which are likely to provide recursion parameters. Such recursion parameters can be used to order function calls for tupling analysis.

However, this ordering is a little restrictive because it requires the function, d, to be applied at least once (i.e. $z \geq 1$) to each of the parameter. We can relax this requirement to $z \geq 0$ if we could establish a transitive ordering among the set of functions, $f_1..f_n$, whenever $z=0$. The use of functions and recursion arguments for ordering will give us a more general technique for finding transitive-ordering.

Thus, a slightly more general method for selecting an appropriate recursion parameter, x_j, will be based on the following condition where d is a *progressive* function.

$$\forall k \in 1..a \; \exists z. \quad (N_{ikj} = d^z) \; \wedge \; (z \geq 0)$$

In addition, we need to build a function-order graph, call it GF, with the function nodes, $f_1..f_n$, such that:

$$\forall i \in 1..n \; \forall k \in 1..a \quad (N_{ikj} = d^0) \; \rightarrow \; arc(f_i, f_{ik}) \in GF$$

This graph must be *acyclic* before we can select the parameter, x_j, as the recursion parameter. Otherwise, the formulated ordering will not be transitive.

Once such a recursion argument, say x_j, has been identified, together with its corresponding function-order graph, GF, we can define the function-argument ordering, $>_{farg}$, as follows:

For any two calls, $f_{i1}(..,d^{z1}\ x_j,..)$ and $f_{i2}(..,d^{z2}\ x_j,..)$, we have:

$$f_{i1}(..,d^{z1}\ x_j,..) >_{farg} f_{i2}(..,d^{z2}\ x_j,..) \textbf{ IFF } (z2>z1) \vee (z2=z1 \wedge path(f_{i1},f_{i2}))$$

where path(v1,v2)=true if there exists a path from v1 to v2 in the graph, GF

As an example, consider the following function, inf_list, which returns an infinite list.

```
dec inf_list : num -> list(num);
dec merge : list(num) # list(num) -> list(num);
- - - inf_list(n)          <= n::merge(inf_list(n+1),inf_list(n+2)) ;
- - - merge(a::x,b::y)     <= a::b::merge(x,y) ;
```

This function presently involves redundant computation in its subsidiary recursive calls. There is only one relevant recursive function, inf_list, to consider and one parameter, n. This parameter is a suitable recursion parameter for the function-argument selection ordering because there is a common ascending function, d(n)=n+1. With it, we could provide the following function-argument ordering which is based solely on the recursion parameter.

$$inf_list(d^{z1}\ n) >_{farg} inf_list(d^{z2}\ x) \textbf{ IFF } (z2>z1) \textbf{ where } d(n)=n+1$$

This ordering can be used to find an eureka tuple containing two calls, (inf_list(n),inf_list(n+1)). With this eureka tuple, we can define the following tuple function,

```
dec inf_tup : num -> (list(num) # list(num));
- - - inf_tup(n)          <= (inf_list(n),inf_list(n+1)) ;
```

which can be transformed to:

```
- - - inf_list(n)      <= u where (u,v)==inf_tup(n);
- - - inf_tup(n)       <= (n::merge(u,v),u) where (u,v)==inf_tup(n+1) ;
```

This example shows that our programs need not be terminating. The only thing that is needed (before tupling analysis could be successful) is that there exists a recursion parameter, together with its function-order graph, which could give a transitive-ordering among the function calls.

In the next section, we shall consider two more techniques for determining selection orderings for programs that are expressed using pattern-matching equations.

5.5.4.2. Constructor-Based Ordering

Most of the recursive functions presented, so far, are expressed without pattern-matching. However, it is also possible to specify some recursive functions through pattern matching equations, with constructor-based arguments that get *progressively smaller* in the successive recursive calls. This alternative method of specifying functions is often better for tupling analysis. This is because we could use the decreasing sizes of pattern-matching argument as the appropriate selection orderings. We shall call this particular form of selection orderings, the **constructor-based ordering**. It is often the dual of what we get from function-argument ordering of the previous section.

As an example, let us re-express the original definition of factlist, using pattern-matching eqautions, as follows:

```
- - - factlist(zero)     <= [];
- - - factlist(succ n)   <= fact(succ n)::factlist(n);
- - - fact(zero)         <= 1;
- - - fact(succ n)       <= succ n*fact(n);
```

By considering only the pattern-matching arguments which are *unchanged* or *descending*, we could determine an appropriate selection ordering, $>_{cons}$, as follows:

$v1(succ^x\ n) >_{cons} v2(succ^y\ n)$ **IFF** $(x>y)$ or $(x=y \wedge v1=factlist \wedge v2=fact)$

where $v1(succ^x n)$ and $v2(succ^y n)$ are simply any two calls that result from the above factlist definition. The technique to determine this constructor-based ordering is essentially similar to the technique of the previous section for function-argument ordering. In this case, a pattern-matching parameter that is descending (or unchanged) will be selected as the recursion parameter.

With this ordering, we can generate the following continuous sequence of cuts where the 4th cut is found to match with the 2nd cut to give an eureka tuple of $(fact(succ\ n_1),factlist(n_1))$.

NO	INSTANTIATION	SET OF CALLS	
cut_1		factlist(n)	
	n=succ n_1	**factlist(succ n_1)**	
cut_2		**fact(succ n_1)**,factlist(n_1)	
cut_3		fact(n_1),factlist(n1)	
	n_1=succ n_2	fact(succ n_2),**factlist(succ n_2)**	
cut_4		**fact(succ n_2)**,factlist(n_2)	cut_4 matches with cut_2

Notice that the above sequence of cuts contains an additional column for argument instantiations. These instantiation are needed to unfold some of the function calls. For example the first cut, cut_1, containing the call factlist(n), must be instantiated with n=succ n_1, to give factlist(succ n_1), before this call can be unfolded. This is simply a consequence of unfolding pattern-matching (symbolic) function calls. Only minimal instantiations, which enables unfolding, need to be made at each time. This can be achieved by the narrowing mechanism which combines (minimum) instantiation with unfolding.

5.5.4.3. Instantiate and Unfold Ordering

The need to instantiate before unfolding can be combined with some simple restrictions to provide another, even simpler, selection ordering for tupling analysis. We call this technique of selecting calls for unfolding, the **instantiate-and-unfold** ordering technique, and will describe it below.

Let us describe a function call as unfoldable if it can be unfolded *without* any instantiation and those which cannot be so unfolded non-unfoldable. This dichotomous division of calls can be used as a crude means of ordering function calls which are present in a cut. But to use it as a selection ordering, there are two additional pre-conditions which the given function definitions must satisfy first. These two restrictions are:

1) Recursive function definitions must be expressed using pattern matching.
2) All relevant subsidiary calls, on each RHS of definitions, must be in a non-unfoldable state.

The first pre-condition is necessary if we want to use the *unfoldability test* as a crude means of ordering function calls. The second condition is essential to separate out parent calls (represented by LHS of equations) that are unfoldable, from children calls (represented by subsidiary calls in RHS of equations) that are non-unfoldable. These two restrictions make it easier for us to implement tupling analysis but it is less general (e.g. we could not use it for infinite programs, like the earlier inf_list function).

At times, some function calls in the given program definition may not satisfy the second pre-condition. This can occasionally be rectified by applying direct unfolds to the offending subsidiary calls until they are no longer unfoldable. For example, the previous factlist definition has an auxiliary call, fact(succ n), which is unfoldable. This violation can be removed by unfolding the call to give:

```
- - -factlist(0)       <= [];
- - -factlist(succ n)  <= (succ n * fact(n))::factlist(n);
```

With the pre-conditions satisfied, we can generate a continuous sequence of cuts by starting with the main call as the first cut and then successively select all those calls which required the **minimum instantiation** to unfold. The minimum instantiation, $\sigma_{min,i}$, for each cut, cut_i, together with the set of calls that would be selected for unfolding, $selected_i$, is specified as follows:

$selected_i$ = $\{s \mid s \in cut_i, unfoldable(t\sigma_{min,i})\}$

$\sigma_{min,i}$ = σ **st** $\exists a \in cut_i. \forall b \in cut_i.\ min_subs(a) \leq_{subs} min_subs(b)$

$min_subs(s)$ = σ **st** $\exists \sigma.\ unfoldable(s\sigma) \wedge (\forall \alpha\ unfoldable(s\alpha) \rightarrow \sigma \leq_{subs} \alpha)$

Note that unfoldable is a test to see if a call can be unfolded, min_subs finds the minimum instantiation (output substitutions) needed to unfold a given function call and $<_{subs}$ compares two substitutions to see which one is more general. The actual determination of this minimum instantiation (for a cut) can be achieved by using unification (or narrowing) to find the output substitutions required to unfold each of the function calls. These substitutions (from each call) are then compared to see which is the most general (or minimal).

The use of minimum instantiation guarantees that those calls which are made unfoldable will always be from the topmost subset (i.e. parent calls). An application of the instantiate-and-unfold ordering to the above factlist definition would result in the following continuous sequence of cuts.

NO	MIN INSTANTIATION	ITEMS IN CUTS	
cut_1		factlist(n)	
	n=succ n_1	**factlist(succ n_1)**	
cut_2		fact(n_1),factlist(n_1)	
	n_1=succ n_2	**fact(succ n_2),factlist(succ n_2)**	
cut_3		fact(n_2),factlist(n_2)	cut_3 matches with cut_2

At the beginning of each cut, all calls in the cut must be non-unfoldable. Minimal instantiation is then applied to the calls to select all those calls that could be directly unfolded in order to generate the next cut. These unfoldings will only result in children calls which are non-unfoldable[14]; ready for the same process to be repeated for the next cut. The analysis ends when the 3rd cut is found to match with the 2nd cut.

Such an instantiate-and-unfold ordering considerably simplifies the tupling analysis technique. In particular, the selection of calls to unfold is simply achieved through the unfoldability test.

5.5.5. Examples

Our tupling analysis, augmented with an appropriate selection ordering, can find eureka tuples for a wide class of programs. This analysis can be shown to work for common generator and periodic commutative classes of programs; self, auxiliary and mutual recursive programs; and even programs with complex descent conditions like Tower of Hanoi. Furthermore, our analysis technique is capable of producing a better result (with smaller matching tuple) for the periodic commutative class of programs than that reported in [Cohen83] or [Pettorossi84].

In this section, we shall illustrate this capability with some examples, using either the *instantiate-and-unfold* ordering or the *function-argument* ordering.

---

[14] because of minimal instantiation and condition requiring that subsidiary calls in program definitions be in non-unfoldable state.

5.5.5.1. Example 1 - Common Generator

The fibonacci function we showed at the beginning of this chapter is an instance of the common generator class of programs. However, to show that our analysis algorithm is applicable in the general case, let us consider the following schematic function from the common generator class (with two recursive calls),

$$f(x) \Leftarrow \textbf{If } B(x) \textbf{ then } K(x) \textbf{ else } H(x, f(g^n x), f(g^m x));$$

Note that g is the common generator of the two descent functions, g^n and g^m, where n and m are each ≥ 1 and have no common factors between them.

Under such a definition, we can represent the subsidiary calls of this function by a canonical form, $f(g^i x)$, where $i \geq 0$. This canonical form can be used to define a suitable function-argument selection ordering, $>_{farg}$, which satisfies the relationship:

$$f(g^a x) >_{farg} f(g^b x) \textbf{ IFF } b > a$$

where $f(g^a x)$ and $f(g^b x)$ are any two subsidiary calls. Such an ordering could be detected by the technique of Section 5.5.4.1 where the composite parameter of the above function is also the recursion parameter.

The above ordering for this class of programs is total and will only select one largest item to unfold from each cut. With it, we will eventually find a matching tuple, $(f(x), f(gx), f(g^2x), ..., f(g^{max(m,n)-1}x))$, of size max(m,n). This is identical to the result of Cohen's descent function analysis for the common generator class of programs.

It may be of (theoretical) interest to note that the AD-ordering can also be used as the selection ordering for this class of programs. However, this ordering may be a partial ordering which allows the maximal possible items to be selected for unfolding at each cut. As a result, the matching pair of tuples discovered will select min(m,n) calls, rather than just one call, to unfold before getting to the next tuple. This implies that the new transformed program will simply take bigger recursion steps. Nevertheless, an eureka tuple of size max(m,n) can still be obtained.

5.5.5.2. Example 2 - Periodic Commutative

The periodic commutative class of programs can be represented by the following schematic function.

$$p(x) \Leftarrow \textbf{If } B(x) \textbf{ then } K(x) \textbf{ else } H(x, p(c(x)), p(d(x)));$$

where its descent functions, c and d, satisfy the commutative (i.e. $c \circ d = d \circ c$) and periodic (i.e. $c^n = d^m$ for some m,n) conditions. This class of programs was shown by Cohen to have a uniform redundancy pattern whose DGs can be drawn as rectangular blocks of nodes, as shown in Figure 5.10. Notice that there are two kinds of dependency arrows shown. Both are used to denote parent-children dependency but the **bolded** arrows are used to highlight the boundaries of repeated rectangular blocks of nodes. Also, we have abbreviated the calls to show just their arguments with the first root call, p(x), denoted by, x, in the top-leftmost corner.

Using this DG, Cohen showed that a table consisting of two arrays of size m and n, representing the left and top sides of the rectangular blocks, would be sufficient to remove periodic commutative redundancy. This pair of arrays can be represented by the following two groups of calls, $p(x),p(d\,x),p(d^2\,x),...,p(d^{m-1}\,x)$ and $p(x),p(c\,x),p(c^2\,x),...,p(c^{n-1}\,x)$.

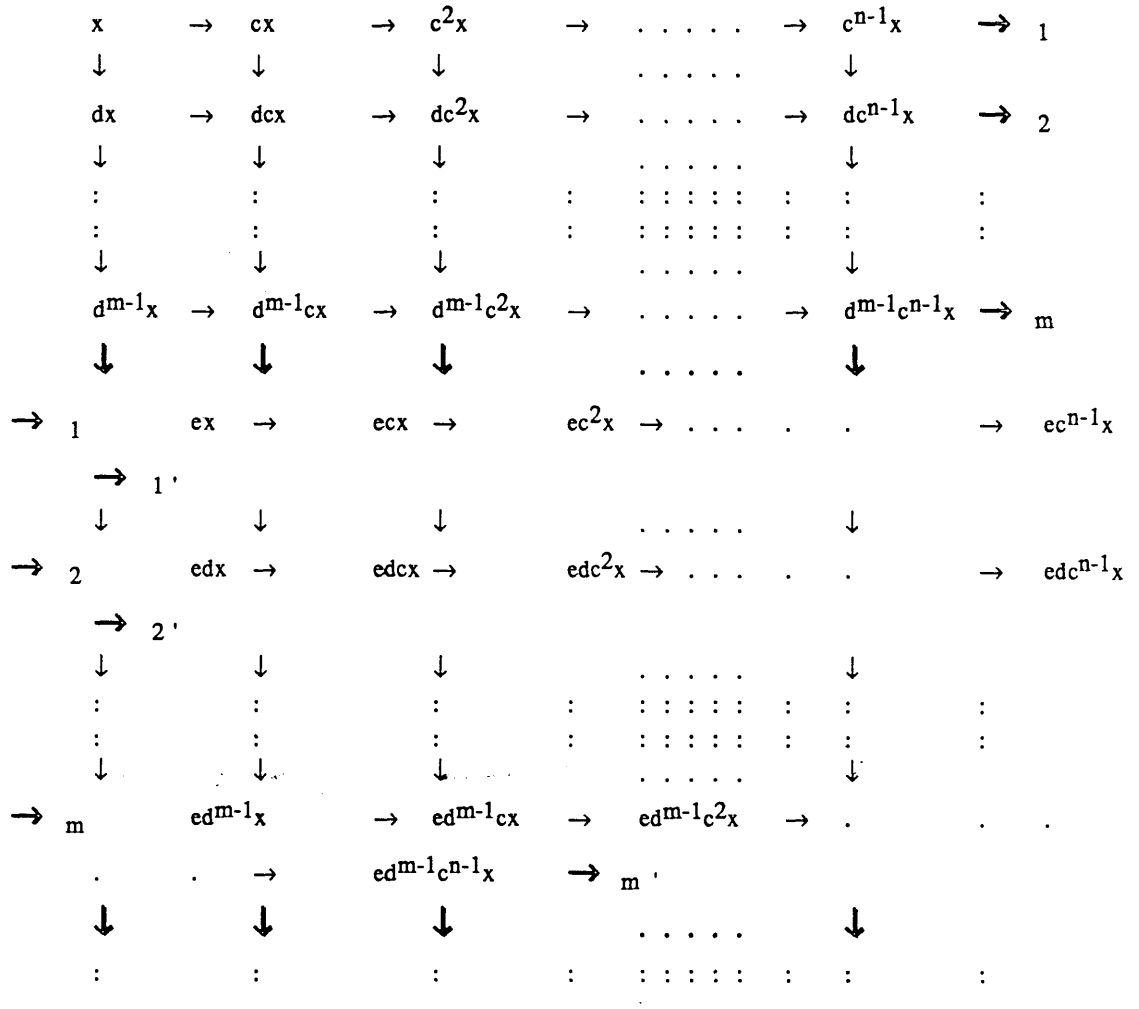


Figure 5.10 - DG for the Periodic Commutative Class of Programs as a Rectangular Array WHERE $e=d^m$ or $e=c^n$

Subsequently, Pettorossi improved this result to **m+n-1**, by using a progressive sequence of matching L-shaped cuts in the DG. These L-shaped cuts are essentially the same two sides of the rectangle with a common overlapping node. However, this result is not yet optimal (minimal). Instead of L-shaped cuts, we can actually use a **diagonalised** cut (as the eureka tuple) which connects the top-right end to the bottom-left end. This is represented by $p(c^{n-1}\,x)$ and $p(d^{m-1}\,x)$ in Figure 5.10 of the rectangular block of nodes. Diagonalised cuts are smaller because each diagonal movement across the DG is equivalent to one horizontal plus one vertical traversals. These cuts result in eureka tuples of size **max(m,n)**. There are a number of different possibilities for such diagonalised cuts. Assuming that $n \geq m$, one of them can be represented by the following set of function calls

$$p(c^{n-1}\,x),p(c^{n-2}d\,x)...,p(c^{n-m}d^{m-1}\,x),p(c^{n-m+1}d^{m-1}\,x),....p(cd^{m-1}\,x),p(d^{m-1}\,x)$$

where the next matching cut can be obtained by applying the descent function, c, to each of these calls. Optimal diagonalised cuts, like the above, can be automatically discovered by our

tupling analysis technique. Even the weakest AD-selection ordering can successfully find an optimal matching cut.

For theoretical interest, each of the above function calls can actually be represented by the canonical form, $p(e^i c^j d^k x)$ where $e=d^m$ and $e=c^n$. In this canonical form, we *could* provide each periodic-commutative program with the following fairly complex selection ordering.

$$p(e^{i1} c^{j1} d^{k1} x) >_{farg} p(e^{i2} c^{j2} d^{k2} x) \textbf{ IFF } (i1>i2) \vee (i1=i2 \wedge ((j1 \geq j2 \wedge k1>k2) \vee (j1>j2 \wedge k1 \geq k2))$$

Our technique will not obtain orderings in this way. This is because we will need to establish the periodic-commutative condition before we are able to use the above canonical form. If we are able to establish the periodic-commutative condition, then there is no need for our tupling analysis at all!

Instead, our technique would try to recognise the presence of some recursion argument which is appropriate for function-argument ordering. Let us illustrate this with the *impatient commuter program*, given in [Cohen83]. This program can be specified by the following schematic function:

```
- - -f(i,h)          <= If i > C then K(i)
                       else S(i,h,f(i+1,h),f(i+2,nh));
```

where its descent functions are c(i,h)=(i+1,h) and d(i,h)=(i+2,nh) respectively and n(n(h))=h. These descent functions satisfy both the periodic and commutative conditions.

Using the earlier technique of Section 5.5.4.1, we could determine a function-argument ordering, based on the first (numeric) recursion parameter, as follows:

$$f(i+a,_) >_{farg} f(i+b,_) \textbf{ IFF } b>a$$

This ordering can be used to produce the following continuous sequence of cuts that finds its eureka tuple, when the 5th cut matches with the 4th cut.

NO	CUTS	
1.	**f(I,h)**	
2.	**f(I+1,h)**,f(i+2,nh)	
3.	**f(I+2,h)**,**f(I+2,nh)**,f(I+3,nh)	
4.	**f(I+3,h)**,**f(I+3,nh)**,f(i+4,h),f(i+4,nh)	
5.	**f(I+4,h)**,**f(I+4,nh)**,f(i+5,h),f(i+5,nh)	matches with Cut 4

This gives us a diagonalised cut of (f(i,h),f(i,nh),f(i+1,h),f(i+1,nh)) which could be used as the eureka tuple. Other kinds of orderings, including the weakest AD-ordering, will also produce diagonalised cuts. They differ mainly in the calls that are selected for the matching tuple. For example, if the AD-ordering was used as the selection ordering, we would get (f(i,h),f(i+1,h),f(i+1,nh),f(i+2,h)) as the eureka tuple. Both these tuples are still *shorter* than the L-shaped cut of five items, (f(i,h),f(i+1,h),f(i+2,nh),f(i+2,h),f(i+3,h)), suggested in [Pettorossi84].

With the first diagonalised cut, we can define a new tuple function

```
- - -f_tup(i,h)      <= (f(i,h),f(i,nh),f(i+1,h),f(i+1,nh));
```

that can be automatically transformed to:

```
- - -f_tup(i,h)      <= If i>C then (K(i),K(i),K(i+1),K(i+1))
```

```
                    else (S(i,h,u,x),S(i,nh,v,w),u,v)  where (u,v,w,x)==f_tup(i+1,h);
```

This tuple function can now be force folded against the original definition of f to give:

```
- - -f(i,h)          <= If i > C then K(i)
                       else S(i,h,u,x) where (u,_,_,x)==f_tup(i+1,h);
```

One possible criticism of the way our analysis produces a matching cut is that it is possible for the resulting tuple function to over-compute (in a strict language[15]) some function calls that are not needed. The above definition of f does not use the two middle items, denoted by don't care variable '_' (from the tuple function call f_tup(i+1,h)). This over-computation is not serious because it only represents a small constant overhead. Nevertheless, it can still be eliminated by simply unfolding the tuple function call and then removing all unreferenced variables, where possible. This is repeated until none of the don't care variables remain. Such a sequence of steps is shown below for the above function, f.

```
- - -f(i,h)          <= If i>C then K(i)
                       else S(i,h,u,x)
                       where (u,x)==If i+1>C then (K(i+1),K(i+2)) else (S(i+1,h,u1,x1),v1)
                       where (u1,v1,_,x1)==f_tup(i+2,h);
                    <= If i>C then K(i)
                       else S(i,h,u,x)
                       where (u,x)==If i+1>C then (K(i+1),K(i+2)) else (S(i+1,h,u1,x1),v1)
                       where (u1,v1,x1)==If i+2>C then (K(i+2),K(i+2),K(i+3))
                                                  else (S(i+3,h,u2,x2),S(i+3,nh,v2,w2),v2)
                       where (u2,v2,w2,x2)==f_tup(i+3,h);
```

5.5.5.3. Example 3 - Tower of Hanoi.

The Tower of Hanoi program can be expressed as a function, seq, which returns a sequence of moves needed to transfer n discs from poles A to B using C, as shown below.

```
dec seq : (int # char # char # char) -> list(char # char);
- - - seq(zero,A,B,C)      <= [];
- - - seq(succ n,A,B,C)    <= seq(n,A,C,B)<>[(A,B)]<>seq(n,C,B,A);
```

Analysis techniques of both Cohen and Khoshnevisan were unable to cope with this class of programs. However, our tupling analysis technique can easily handle this program under its more general framework. In particular, we can analyse the eureka tuple with the instantiate and unfold selection ordering. This analysis will produce the following continuous sequence of cuts, where the 4th cut is found to match with the 3rd cut.

NO	INSTANTIATE	CUTS
1.		**seq(n,A,B,C)**
	n=succ n_1	seq(succ n_1,A,B,C)
2.		**seq(n_1,A,C,B),seq(n_1,C,B,A)**
	n_1=succ n_2	seq(succ n_2,A,C,B),seq(succ n_2,C,B,A)
3.		**seq(n_2,A,B,C),seq(n_2,B,C,A),seq(n_2,C,A,B)**
	n_2=succ n_3	seq(succ n_3,A,B,C),seq(succ n_3,B,C,A),seq(succ n_3,C,A,B)
4.		**seq(n_3,A,C,B),seq(n_3,C,B,A),seq(n_3,B,A,C)** which matches Cut 3

[15] A lazy program would not over-compute for items that are not needed

At each of these cuts, all the calls are selected for instantiation and unfolding. This results in a matching tuple of three calls, ((seq(n_2,A,B,C),seq(n_2,B,C,A),seq(n_2,C,A,B)), which can be used to define the following tuple function, seq_tup.

```
dec seq_tup: num # char # char # char -> (list(char # char) # list(char # char) # list(char # char));
- - -seq_tup(n,A,B,C)        <=  (seq(n,A,B,C),seq(n,B,C,A),seq(n,C,A,B));
```

This function can subsequently be transformed (with the help of the steps taken in analysis) to:

```
- - -seq_tup(zero,A, B,C)    <=  ([],[],[]);
- - -seq_tup(succ n,A, B,C)  <=  (u<>[(A,B)]<>v,w<>[(B,C)]<>u,v<>[(C,A)]<>w)
                                 where (u,v,w)==seq_tup(n,A,C,B);
```

5.5.5.4. Example 4 - Mutual Recursions

We have already seen our analysis technique working for both self and auxiliary recursive functions. The same success can also be shown for mutually recursive functions, where we are able to analyse the function definitions directly without having to convert them to self-recursive definitions first. An example of mutual recursion is the following pair of functions, f and g, which operates on the linear list data structure.

```
dec f,g : list(num) -> num;
- - - f([])          <=  0;
- - - f([a])         <=  a;
- - - f(a::b::x)     <=  a+f(b::x)+g(x);
- - - g([])          <=  0;
- - - g(b::x)        <=  f(x)+g(x);
```

These two functions satisfy the instantiate and unfold ordering's preconditions and can be analysed for a matching tuple by the following sequence of cuts.

NO	INSTANTIATE	CUTS
1.		**f(x)**
	$x=a::a_1:: x_1$	$f(a::a_1::x_1)$
2.		$\mathbf{f(a_1::x_1),g(x_1)}$
	$x_1=a_2:: x_2$	$f(a_1::a_2:: x_2),g(a_2::x_2)$
3.		$\mathbf{f(a_2:: x_2),g(x_2)},f(x_2)$
	$x_2=a_3:: x_3$	$f(a_2::a_3:: x_3),g(a_3:: x_3),f(a_3:: x_3)$
4.		$\mathbf{f(a_3:: x_3),g(x_3)},f(x_3)$
		which matches Cut 3

The matching cut discovered, $f(a_3:: x_3),g(x_3),f(x_3)$, can be used to define the following new tuple function:

```
dec f_tup: num # list(num) -> (num # num # num);
- - -f_tup(a,x)          <=  (g(x),f(a:: x),f(x));
```

which can be subsequently transformed to:

```
- - -f_tup(a,[])         <=  (0,a,0);
- - -f_tup(a,b::x)       <=  (w+u,a+v+u,v) where (u,v,w)==f_tup(b,x);
```

5.6. An Extension to the Tupling Analysis Technique

Thus far, all successful programs chosen to illustrate our analysis technique are based on the following two properties:

1. they use linear (recursive) data structures, e.g. numeric and list type, and
2. they have one recursive equation per function

Other programs which do not have these properties, e.g those which use non-linear[16] binary tree structures or those specified with multiple recursive equations per function, will not be correctly handled by the proposed tupling analysis technique. We look at why this is so for some programs and then suggest ways in which the original analysis technique can be generalised to cater for these other programs.

5.6.1. Non-Linear Data Structure

A program which uses the non-linear binary tree data is the following two functions, deepest and depth, for finding a list of leaves that are furthest away from the root node.

```
dec deepest: tree(alpha) -> list(alpha);
dec depth:tree(alpha) -> num;
- - -deepest(leaf(a))    <= [a];
- - -deepest(node(l,r)) <= if depth(l) > depth(r) then deepest(l)
                           else if depth(l) < depth(r) then deepest(r)
                                else deepest(l)<>deepest(r);
- - -depth(leaf(a))      <= 0;
- - -depth(node(l,r))    <= 1+max(depth(l),depth(r));
```

The redundant function calls in this program *cannot* be correctly analysed by the present analysis technique. This is because the continuous sequence of cuts produced will just continue to grow in size, as shown below,

NO	CUTS	MINIMAL INSTANTIATIONS
1.	**dp(n)**	n=node(l,r)
2.	**d(l),dp(l),d(r),dp(r)**	l=node(ll,lr), r=node(rl,rr)
3.	**d(ll),dp(ll),d(lr),dp(lr),d(rl),dp(rl),d(rr),dp(rr)**	ll=node(lll,llr), lr=node(lrl,lrr), ...
4.	....	

where deepest and depth are abbreviated to dp and d respectively.

The main problem with the above sequence of cuts is the presence of an increasing number of *distinct* recursion variables, e.g **l,r** in the 2nd cut and **ll,lr,rl,rr** in the 3rd cut, which arise from the branches of the original binary tree. These independent variables are a problem because they are not ordered with respect to one another, but more importantly they give rise to descendant calls which belong to completely separate and *disjoint sub-graphs* in the DG. As a consequence, the frontier in this analysis will simply get progressively wider.

---

[16] Lists and peano numbers are considered to be linear data structures but trees are considered as non-linear structures. This is because tree-like structures may contain more than one recursive data structure in their nodes.

To rectify this problem, our analysis algorithm must be extended to recognise groups of function calls in a cut that are *distinctly disjoint*. Such groups of calls should be *split* into sub-cuts which could then be independently analysed. For example, the 2nd cut, d(l),dp(l),d(r),dp(r), should be split into two sub-cuts, d(l),dp(l) and d(r),dp(r) which can then be independently analysed. This modification to our analysis would result in a **continuous tree of cuts**, rather than just a continuous sequence. Each branch of this tree will be considered to be successful if its final cut matches with another cut higher in the tree. Such an extended analysis, when applied to the earlier example, would result in the analysis tree shown in Figure 5.11.

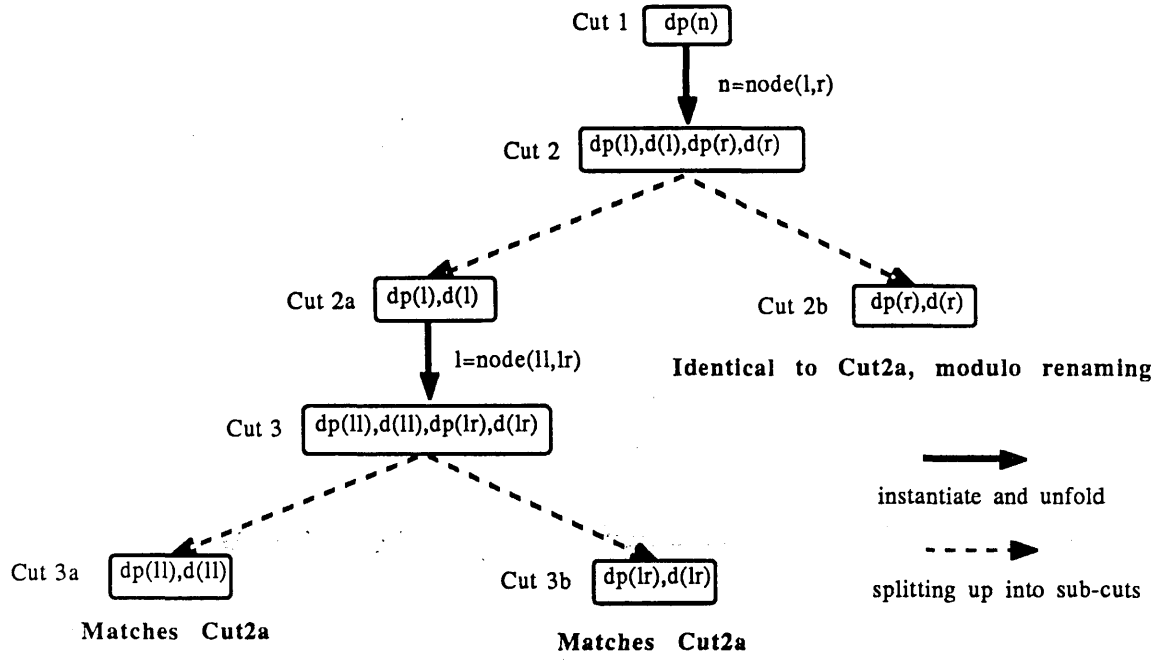


Figure 5.11 - A Continuous Tree of Cuts for dp(n)

We have not produced a *complete* analysis tree in Figure 5.11. This is because we can optimise the analysis by avoiding duplicated analyses of sub-cuts that appear more than once. For example, the two sub-cuts, Cut 2a and Cut 2b, of the above analysis are identical modulo renaming to each other. Hence, only one of them needs to be analysed as both of the analyses will be identical.

The result of this analysis is a pair of matching tuple, (depth(t),deepest(t)), that can be used to define the following new function.

```
dec d_tup: tree(alpha) -> (num,tree(alpha));
- - -d_tup(t)          <= (depth(t),deepest(t));
```

which can then be transformed to:

```
- - -d_tup(leaf a)      =  (0,[a]);
- - -d_tup(node(l,r))   =  if u > a then (md,v)
                           else if u < a then (md,b)
                           else (md,v<>b)
                           where md==1+max(u,a)
                                 (u,v)==d_tup(l); (a,b)==d_tup(r) ;
```

Notice that this extension requires the tupling analysis technique to recognise disjoint groups of calls in a cut. Incidently, this is helped by the use of pattern-matching equations with constructor terms. This is because the presence of independent variables in constructor arguments can be used to detect disjoint function calls. The alternative, without pattern matching, is to specially recognise descent functions, e.g. left and right descent functions for binary trees, which result in disjoint calls.

5.6.2. Multiple Recursive Equations per Function

The fusc function from Dijkstra is an example of a program which uses more than one recursive equation per function definition. This function has two recursive equations, even and odd cases, together with two base case equations, as shown below.

```
dec fusc : num -> num;
- - -fusc(0)       <= 0;
- - -fusc(1)       <= 1;
- - -fusc(2n)      <= fusc(n);
- - -fusc(2n+1)    <= fusc(n+1)+fusc(n);
```

Previously, our analysis only considers one way of unfolding each function call. But the presence of multiple recursive cases has now meant that there could be more than one way of unfolding some of its function calls! This again implies that we should use a continuous tree of cuts, rather than just a sequence of cuts. With this tree structure, the extended tupling analysis should find a matching pair of cuts for each branch before the entire tree is considered to be successfully analysed. This normally results in a *set* of matching cuts (one matching cut from each branch) from which we must select the most *general* matching cut as the eureka tuple. The tree of cuts for the fusc function is shown in Figure 5.12. Notice that this tree structure is slightly different from the earlier tree of Figure 5.11. The earlier tree branches out by splitting a large cut into sub-cuts, while the present tree branches out according to the different ways in which function calls are unfolded.

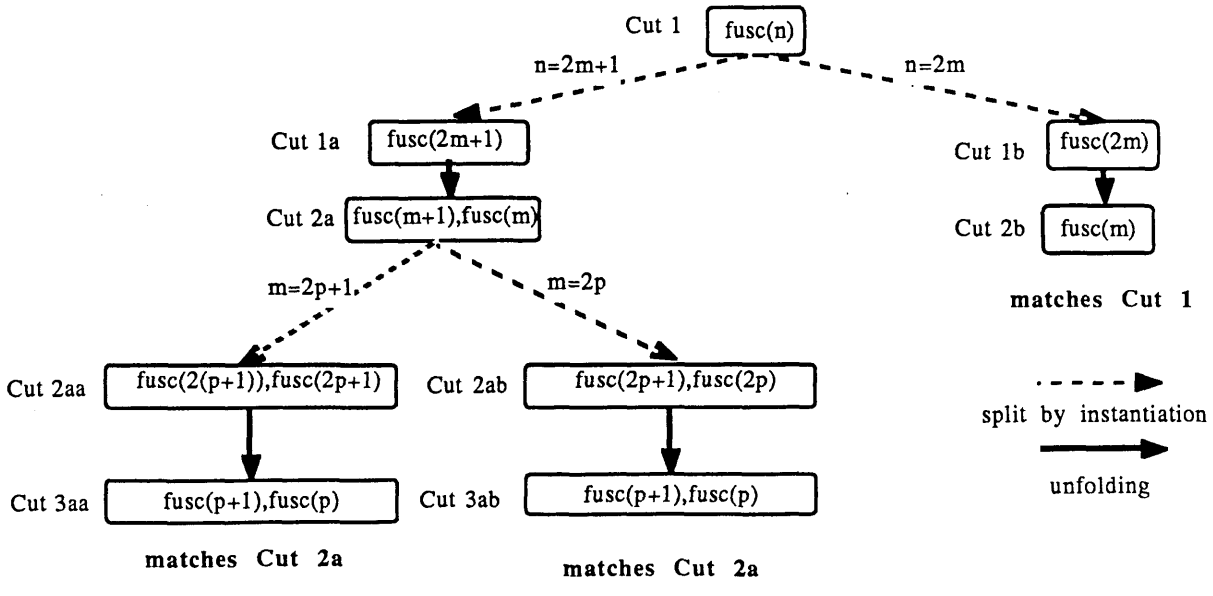


Figure 5.12 - Tree of Cuts for a Multi-Recursive Cases Function

Three matching cuts, Cut 3aa, Cut 3ab and Cut2b, are found from the above analysis. Two of these cuts are actually identical, while the third cut (with only one item) can be viewed as a subset of the other two. Hence, the most general tuple is (fusc(n+1),fusc(n)) which becomes the eureka tuple for transformation.

Such an exhaustive exploration using a tree of cuts can of course be implemented but this is potentially expensive. In particular, redundant analysis can occur because we are interested only in the most general matching cut. Fortunately, there is an alternative method which could find eureka tuples in a much quicker fashion. This method involves dividing up the analysis into a number of consecutive mini-analysis phases - one for each of the recursive equations (e.g. even and odd cases). Each of these mini-analysis phases will only use its assigned recursive case, to instantiate and unfold, in order to find its own matching cut. However, these mini-analysis phases are not independent because we will have to carry over the matching cut from one phase as the starting cut of the next phase. The starting cut of the first phase will be the main call, while the matching cut from the last phase will be our eureka tuple which satisfies all the recursive cases.

This shorter analysis is valid provided two further conditions hold, namely:

1) The entire set of calls, at each cut, is always selected for unfolding; and
2) Each subsequent eureka cut must always be *more general* than the eureka cut before it. We describe a cut, say $(n_1,...,n_{a+b})$, to be *more general* than another cut, say $(m_1,...,m_a)$, if $n_1,...,n_a$ matches $m_1,...,m_a$ and $n_b,...,n_{a+b}$ also matches with $m_1,...,m_a$.

When these two conditions are satisfied, the matching tuple from a latter case will automatically be applicable to the previous cases. However, if either of the above two conditions do not hold, we could still ensure that the last eureka cut will work for all the recursive cases by testing to see if it will match for each of the other cases.

Let us look at how this alternative method, with its pre-conditions, would apply to the fusc function. In this function, there are two recursive cases, namely even and odd, which implies that we must use two corresponding mini-analysis phases. Either of these two phases can be used to start first, but they must use the main call, fusc(n), as its first cut. For this example, we will use the even case as the first phase, where instantiation and unfolding using the even case equation produces a matching cut of one call (see 2nd cut below). This matching cut is now carried over to the second (and final) phase, where further analysis using the odd case instantiation results in a matching cut at Step 4. The final matching cut immediately becomes the eureka tuple.

NO	INSTANTIATE	CUTS	
	start with even instantiations		
1.		**fusc(n)**	
	$n=2n_1$	$fusc(2n_1)$	
2.		**$fusc(n_1)$**	matches with Cut 1
	continue with odd instantiations		
2 (cont)	$n_1=2n_2+1$	$fusc(2n_2+1)$	
3.		**$fusc(n_2),fusc(n_2+1)$**	
	$n_2=2n_3+1$	$fusc(2n_3),fusc(2(n_3+1))$	
4.		**$fusc(n_3),fusc(n_3+1)$**	matches with Cut 3

Notice that the two conditions required by the shorter analysis are satisfied. This is because all calls are unfolded for each cut and the second cut is more general than the first cut.

With the eureka tuple obtained, we could define the following tuple function,

```
dec fusc_tup : num -> (num # num);
- - -fusc_tup(n)      =   (fusc(n),fusc(n+1));
```

that can be transformed to:

```
- - -fusc_tup(0)      =   (0,1);
- - -fusc_tup(1)      =   (1,1);
- - -fusc_tup(2n)     =   (u,u+v) where (u,v)==fusc_tup(n);
- - -fusc_tup(2n+1)   =   (u+v,v) where (u,v)==fusc_tup(n);
```

Incidently, this tupling analysis for programs with multiple recursive equations per function is again made simpler by the use of pattern matching. With it, we are able to mechanise the analysis technique using instantiation, unfolding and matching. Had the fusc function been expressed without pattern matching, like the definition below, more reasoning would be needed to analyse for its eureka tuple.

```
- - -fusc(n)      <=  if x<2 then x
                      else if even(x) then fusc(x/2)
                      else fusc(x/2)+fusc(x/2 + 1);
```

5.6.3. Simplification Rules

In the earlier analysis for the fusc function, we have implicitly used some simplification rules to convert all function calls to their canonical form. This is needed to allow some of these calls to be correctly unfolded. For example, the call $fusc(n_2+1)$ can be instantiated by the odd case to $fusc((2n_3+1)+1)$. This call is not in the right form for unfolding and must be simplified to $fusc(2(n_3+1))$ before we recognise that it is an even case call.

Such simplifications to a canonical form can be performed by a *small set of rules* based on the operators (or constructors) used for the argument domain of the given program. In the case of numeric domain with the three operators/constructors, even(n)=2n, odd(n)=2n+1 and succ(n)=n+1, the complete set of rewriting rules is:

```
succ(even n)      →  odd(n)
succ(odd n)       →  even(succ n)
```

This set of rules merely floats out the even or odd constructors to the outermost position. If in addition, the operator pred(n)=n-1 is used, the following extra rules are also needed.

```
pred(even n)    →  odd(pred n)
pred(odd n)     →  even(n)
pred(succ n)    →  n
succ(pred n)    →  n
```

Such a closed set of simplification rules can be provided (for each commonly used argument domain) to help simplify our calls to their canonical form.

5.6.4. Logarithmic-Time Fibonacci

There is an alternative more parallel definition of fibonacci, which is based on even and odd recursive equations, as given below.

```
- - -fib(0)      <= 1;
- - -fib(1)      <= 1;
- - -fib(2n)     <= fib(n)*fib(n)+fib(n-1)*fib(n-1);
- - -fib(2n+1)   <= fib(n)*fib(n)+2*fib(n)*fib(n-1);
```

The time-complexity of this definition is linear, i.e.**O**(n) where n is the parameter value, but there are still some redundant function calls. If this redundancy can be removed, the time-complexity of the above function can be reduced to logarithmic time. Indeed, the extended analysis from the previous section can be used to analyse for an eureka tuple. This analysis, using the even and odd mini-analysis phases (together with appropriate rewriting of calls to canonical forms) can produce the following result.

NO	INSTANTIATION	CUTS	
	start with even instantiations		
1.		**fib(n)**	
	$n=2n_1$	$fib(2n_1)$	
2.		**$fib(n_1)$,$fib(n_1-1)$**	
	$n_1=2n_2$	$fib(2n_2)$,$fib(2n_2-1)$	
3.		**$fib(n_2)$,$fib(n_2-1)$,$fib(n_2-2)$**	
	$n_2=2n_3$	$fib(2n_3)$,$fib(2n_3-1)$,$fib(2n_3-2)$	
4.		**$fib(n_3)$,$fib(n_3-1)$,$fib(n_3-2)$**	which matches Cut 3
	continue with odd instantiations		
4.		**$fib(n_3)$,$fib(n_3-1)$,$fib(n_3-2)$**	
	$n_3=2n_4+1$	$fib(2n_4+1)$,$fib(2n_4)$,$fib(2n_4-1)$	
5.		**$fib(n_4)$,$fib(n_4-1)$,$fib(n_4-2)$**	which matches Cut 4

The final matching cut from this analysis is a tuple of three calls, (fib(n),fib(n-1),fib(n-2)). This cut is suitable for use as an eureka tuple for eliminating redundancy.

Actually, we can further reduce the size of tuple (required to eliminate redundancy) to just two calls, namely (fib(n-1),fib(n-2)), if another valid equation of fibonacci function, shown below, is either supplied or known by the system.

```
fib(n)=fib(n-1)+fib(n-2)
```

This equation will enable the first call, fib(n), to be derivable from the other two calls. With it, we have a smaller eureka tuple which can be used to define the tuple function:

```
dec fib_log : num -> (num # num);
- - -fib_log (n)        <= (fib(n+1),fib(n));
```

which can be transformed to:

```
- - -fib_log (0)        <= (1,1);
- - -fib_log (1)        <= (2,1);
- - -fib_log (2n)       <= (u*u+2*u*v,u*u+v*v) where (u,v)==fib_log (n-1);
- - -fib_log (2n+1)     <= (z*z+u*u,u*u+2*u*v) where z==u+v; (u,v)==fib_log (n-1);
```

This use of additional equations to find a smaller tuple is optional and is best performed *after* an initial eureka tuple has been found, using just the basic equations. Doing it after an initial eureka tuple has been found can help to avoid the complications of having to decide whether the basic or additional equations are to be applied. In contrast to this approach, Pettorossi suggested the use of DGs which contain alternative dependencies (provided by the extra equations) in order to directly find a smaller eureka tuple. He called DGs constructed with alternative dependencies, **simplified m-dags**, and claimed that their use is indispensable (see [Pettorossi84]) with the help of a more complex linear recurrence function. We would like to suggest otherwise by showing that the more complex function can also have a matching tuple, *without* the help of additional equations.

5.6.5. A Complex Linear-Recurrence Function

The complex linear recurrence function from [Pettorossi82,84] is:

```
- - -G(a,b,0)        <= a;
- - -G(a,b,1)        <= b;
- - -G(a,b,2k)       <= G(1,0,k).G(a,b,k)+G(0,1,k).G(a,b,k+1);
- - -G(a,b,2k+1)     <= G(1,0,k).G(a,b,k+1)+G(0,1,k).G(a,b,k+2);
```

This function is actually a generalised form of fibonacci that can be obtained through the linear recurrence transformation strategy in [Pettorossi82], where fib(n)=G(1,1,n). Our extended analysis algorithm is capable of using just the basic recursive equations to discover a matching tuple for this function definition. This analysis is shown below with a final matching cut of 11 function calls found at the 9th step. (We have abbreviated the calls in the cuts to show just their arguments.)

NO	INSTANTIATION	CUTS
	start with even instantiations	
1.		(a,b,k)
2.	$k=2k_1$	$(a,b,k_1),(1,0,k_1),(0,1,k_1),(a,b,k_1+1)$
3.	$k_1=2k_2$	$(a,b,k_2),(1,0,k_2),(0,1,k_2),(a,b,k_2+1),(1,0,k_2+1),(0,1,k_2+1),(a,b,k_2+2)$
4.	$k_2=2k_3$	$(a,b,k_3),(1,0,k_3),(0,1,k_3),(a,b,k_3+1),(1,0,k_3+1),(0,1,k_3+1),(a,b,k_3+2)$ $,(1,0,k_3+2),(0,1,k_3+2)$
5.	$k_3=2k_4$	$(a,b,k_4),(1,0,k_4),(0,1,k_4),(a,b,k_4+1),(1,0,k_4+1),(0,1,k_4+1),(a,b,k_4+2)$ $,(1,0,k_4+2),(0,1,k_4+2)$ which matches Cut 4
	continue with odd instantiations	
6.		$(1,0,k_4),(0,1,k_4),(a,b,k_4+1),(1,0,k_4+1),(0,1,k_4+1),(a,b,k_4+2)$ $,(1,0,k_4+2),(0,1,k_4+2)$
7.	$k_4=2k_5+1$	$(1,0,k_5),(0,1,k_5),(a,b,k_5+1),(1,0,k_5+1),(0,1,k_5+1),(a,b,k_5+2)$ $,(1,0,k_5+2),(0,1,k_5+2),(a,b,k_5+3),(1,0,k_5+3),(0,1,k_5+3)$
8.	$k_5=2k_6+1$	$(1,0,k_6),(0,1,k_6),(a,b,k_6+1),(1,0,k_6+1),(0,1,k_6+1),(a,b,k_6+2)$ $,(1,0,k_6+2),(0,1,k_6+2),(a,b,k_6+3),(1,0,k_6+3),(0,1,k_6+3)$ which matches Cut 7
	double check to see if it applies to even instantiations	
9.	$k_6=2k_7+1$	$(1,0,k_7),(0,1,k_7),(a,b,k_7+1),(1,0,k_7+1),(0,1,k_7+1),(a,b,k_7+2)$ $,(1,0,k_7+2),(0,1,k_7+2),(a,b,k_7+3),(1,0,k_7+3),(0,1,k_7+3)$ which matches Cut 7

Notice that the second eureka cut of 11 calls (obtained at Step 8) is not *more general*[17] than the first eureka cut of 9 calls (at Step 5). However, we ensure that it applies to the even case instantiation by testing to see if it matches in a further step.

The extra general equation for the G function is:

$$G(a,b,n+k)=G(1,0,k).G(a,b,n)+G(0,1,k).G(a,b,n+1)$$

This could, if desired, be utilised in a further analysis to reduce the size of the eureka tuple just found.

[17] according to criterion in Section 5.6.2.

5.7. Limitations of Tupling Analysis

Our proposed tupling analysis technique, using an appropriate selection ordering, is one of the most general techniques to-date for tabulating programs which require static-sized tables. A version of this technique, using the *instantiate and unfold* selection ordering, has already been implemented. This selection ordering appears to be the easiest to implement and has been used to successfully transform a number of functions, including Tower-of-Hanoi, deepest-depth and factlist functions.

Nevertheless there are two limitations with our analysis when compared to other approaches. These limitations are actually the limitations of tupling tactic itself. The first limitation is that our analysis technique will only work for programs which can be tabulated using static-sized tuples. Other classes of programs which do not satisfy this criterion, e.g. the binomial function, will not be successfully analysed. Apart from those programs which require dynamic-sized tables, our analysis will also fail for programs which require fixed, but not static-sized, tables whose sizes are not known at compile time. An example of this the following generalised fibonacci function from [Khoshnevisan87].

```
dec g_fib : num # num # num -> num;
- - -g_fib(x,a,b)        <=  If x ≤ 1 then 1
                            else g_fib(x-a,a,b)+g_fib(x-b,a,b);
```

The table size required for this function is max(a div g,b div g) where g is the greatest common divisor of parameters a and b. These parameter values are usually known only at run-time. As a result, our tupling analysis technique is unable to cope with it.

A second limitation of our approach is that it is primarily suitable for programs which need small tuples for tabulation. Those programs which need very large tuples for removing redundancy will take a long time to analyse and result in very large programs. An example of this is the following fibonacci-like function.

```
dec fib_1000 : num # num # num -> num;
- - -fib_1000(x)        <=  If x < 1000 then x
                            else fib_1000(x-1)+fib_1000(x-1000);
```

This function needs an eureka tuple of 1000 items. Both long analysis time and correspondingly huge programs are incurred when tupling analysis and transformation are applied to this simple example! Furthermore, the semi-decidable nature of our analysis technique, with its corresponding need for an upper bound, makes it impractical to analyse for programs with large table requirements.

The best solution appears to be the use of a combination of techniques. Programs which need small or medium-sized static tables should be handled by our tupling analysis technique. This has the benefit of both a more efficient program, as well as one which is more amenable to other forms of transformation. If tupling analysis fails, then the descent analysis technique of Cohen, with its use of dynamic arrays or vectors, could be applied to handle programs with the four classes of redundancy. This could help to handle common-generator and periodic-

commutative programs (which need large or unknown-sized tables) and commutative programs (which need dynamic-sized tables). When this analysis fails as well, there is still the generally-applicable full memoisation technique which does not require any special analysis conditions.

5.8. Summary of Chapter 5

In this chapter, we presented a very general analysis technique for the tupling transformation. This analysis technique is able to discover eureka tuples for a large class of programs.

Our analysis technique makes novel use of a selection ordering criterion. This ordering criterion is based principally on the function-argument pairs to help systematically search for eureka tuples. This search will construct a sequence of *continuous* cuts where a matched pair of cuts in the sequence will be taken as the eureka tuple. Though finding an appropriate ordering for function calls is, in general, an undecidable problem, we are still able to give techniques (heuristics) which could determine transitive orderings for a large class of programs.

We showed how the tupling analysis technique could be applied to self, mutual and auxiliary recursive programs. Included within these programs are those with *common-generator*, *periodic commutative* and other forms of more complex redundancy patterns. We also showed how programs, which have multiple-recursive cases and those which use non-linear data structures, could also be analysed through a *tree* rather than a *sequence* of continuous cuts. Furthermore, our analysis algorithm appears to discover minimal-sized tuples where possible. In particular, we illustrated this power by showing how it could discover eureka tuples that are smaller than those reported by Cohen and Pettorossi for the periodic commutative class of programs.

Lastly, we pointed out the theoretical and practical limitations of the tupling analysis technique, namely that it is only suitable for programs with *small* and *static-sized* tupling requirements. To overcome these limitations, we informally suggested the *combined* use of a host of techniques, from tupling analysis (as the most specific technique) to memoisation (as the most general technique) for the removal of redundant calls.

6. Constraint-Based Transformations

6.1. Introduction

The use of assertions to specify invariants (or constraints) between inputs and outputs of programs have always played an important role in program construction and verification of imperative programs. This facility is not available to *pure* functional programs (and the original unfold/fold methodology) but could be provided through the logic and constraint programming language extensions [Darlington86,89a]. The key features that are present in these extensions are *absolute set abstractions*, *logical variables* and *constraints*. These features increase the expressive power of our language by allowing additional properties of programs to be directly specified.

One use of these features is to specify functions in a pre-algorithmic fashion, where the output for a function is specified through a list of constraints between input and output. For example, the integer divide function, div, can be specified in this fashion using the definition:

```
dec div : num # num -> num;
- - -div(a,b)          <=  r st r*b+c=a, 0≤c≤b ;
```

where the constraints, r*b+c=a and 0≤c≤b, are used to define the logical output variable, r, in terms of the inputs, a and b. This use of the constraint facility enables very high-level specifications to be written, which declaratively state the desired relationships between inputs and outputs of programs. Such specifications are clear and concise but may be either unexecutable or very inefficiently executable using the logic and constraint implementation techniques. Nevertheless, they can always be improved by subjecting them to the appropriate unfold/fold transformations.

Apart from this use as a high-level specification aid (via input and output constraints), these features can also be used to state useful *pre-conditions* between just inputs of functions. Such a use is important for exploiting certain categories of transformations which rely on specialised program contexts for improvements. In particular, two transformation techniques which require this capability are **finite differencing** [Paige82] and a new form of **filter promotion**.

Finite differencing's main objective is the replacement of expensive expressions in program loop (or recursion) by equivalent expressions which are *incrementally maintained*. This incremental maintenance is often cheaper than re-calculating the expensive expression in the loop. In the case of functional programs, the incremental maintenance of expressions are normally kept as *new* parameters whose values are specified using constraints. Such constraints represent invariant relationships that must be maintained by the transformation. We describe this technique in Section 6.2 and pay particular attention to the analysis conditions that would permit its application.

Similarly, there is a form of the filter promotion technique which uses constraints, rather than fusion (of Chapter 3), to filter out infeasible sub-spaces of certain generate-and-test programs.

In this case, no new parameters are introduced. Instead, the constraint is used for filtering purpose. This technique requires certain syntactic and semantic conditions to be satisfied before it can be applied. We describe what these various conditions are in Section 6.3 and illustrate its application with examples.

Lastly, we shall also discuss the utility of keeping invariants/constraints for transformation in Section 6.4, and a work that is related to base-case filter promotion in Section 6.5.

6.2. Finite Differencing

Finite differencing has its origin from Briggs' sixteenth century method of polynomial tabulation by difference polynomials [Goldstein77]. This differencing technique underlies both a compiler optimisation technique called *strength reduction* for numeric operators [Allen81], and Earley's proposal for iterator-inversion [Earley76]. Later work by [Paige82,87] considerably generalised this technique by working with high-level set theoretic expressions, where more opportunities for asymptotic program improvements are possible. It was with this generalisation that the name, finite differencing, was coined. These past techniques were proposed principally for imperative programs. We shall therefore begin our description of the general technique by showing how it applies to imperative programs.

Let us consider a schematic, imperative program loop with induction variable, i, that only changes by i:=nxt(i) at each iteration round the loop, together with an expensive expression, costly(i), which uses the induction variable.

```
i:=init;
while cond(i) do
    begin
        i:=nxt(i) ;
        p:=...costly(i)...;
        :
    end;
```

This schematic program may be inefficient because it is often expensive to compute, costly(i), afresh each time round the loop when the changes to the induction variable, i, may only be minor. We could improve the program if the costly expression can be computed incrementally in accordance with the changes in the induction variable. This improvement is possible if there exists a cheaper operator, delta, such that:

```
costly(nxt(i))=delta(costly(i),i)
```

With this incremental relationship, it is possible to change (improve) the earlier program loop to:

```
i:=init;
t:=costly(i);
while cond(i) do
    begin
        t:=delta(t,i);
        i:=nxt(i) ;
        p:=...t...;
        :
    end;
```

The costly expression is now maintained in a new variable, t, which is at all times equivalent to costly(i). This variable is initialised by t:=costly(i) before entering the loop and then incrementally updated by t:=delta(t,i) to maintain its equivalence to the costly expression. The new program is an improvement over the earlier loop because the costly expression, costly, has effectively been replaced by a much cheaper incremental operator, delta. Two concrete examples of the above schematic program loop are:

(1) In the numeric domain, we may have:

```
nxt(i)   =  i+1
costly(i) =  i**3,
```

where the costly expression is the power operator, **. This operator can be replaced by the following cheaper multiplication operator:

```
delta(t,i)   =  t*3,
```

because:

```
costly(i+1)  =  (i+1)**3
             =  i**3+3
             =  costly(i)*3
```

(2) Similarly in the set domain, we may have:

```
nxt(i)       =  i ∪ {x}
costly(i)    =  size(i)
```

where the costly expression is the cardinality function, size, of sets. This operator can be replaced by the following incremental operator:

```
delta(c,i)   =  If x ∈ i then c else c + 1,
```

because:

```
costly(i ∪ {x})  =  size(i ∪ {x})
                 =  If x ∈ i then size(i) else size(i ∪ {x})
                 =  If x ∈ i then size(i) else size(i) + 1
                 =  If x ∈ i then costly(i) else costly(i) + 1
```

This is an improvement if we assume that the set membership test, ∈, can be done in constant time (e.g. by implementing the set with a hash table). Alternatively, if the program context suggests that the element, x, is always a new element of the set, i, then even this membership test can be avoided by using:

```
delta(c,i)   =  c + 1
```

Although this technique was originally developed under the imperative setting, it is even more natural to perform it within the declarative paradigm. Here, in place of program loops and induction variables, we have *recursive functions* together with their *changing (or induction) parameters*. Finite differencing can then be applied to replace costly expressions (in the recursive functions) by incrementally maintained values under *new* function parameters.

The general finite differencing technique is now quite well understood. What we would like to show in this thesis is that this transformation could also be supported under the unfold/fold

framework. We illustrate this with a simple example and follow it up with details of the key transformation steps required.

6.2.1. An Example

A simple function which can be improved by finite differencing is given below. This function attempts to split a list of items into sublists of six items each (except for the last sublist).

```
dec split_list: list(alpha) # list(alpha) -> list(list(alpha));
- - -split_list([],w)     <= w;
- - -split_list(a::x,w)   <= if length(w)=6 then w::split_list(a::x,[])
                             else split_list(x,w<>[a]) ;
```

The costly expression in the above function is length(w), which operates on parameter w. This expression is expensive because it is being re-computed at each recursive call even though the parameter, w, has undergone only a minor incremental change. We can avoid this inefficiency if we are able to introduce a new parameter that can be maintained to the value of length(w).

To achieve this goal, we define a new function, new_split, that would compute the same result as the original function, split_list. This new function will, in addition, have a new parameter, d, whose value is constrained to be equal to the expression, length(w), as shown below.

```
dec new_split: list(alpha) # list(alpha) # num -> list(list(alpha));
- - -new_split(x,w,d)  <= split_list(x,w) st d=length(w);                    (6.1)
```

The invariant condition for the new parameter, d=length(w), is specified through the *constraint* feature. A similar construct, called *qualifier*, was used in [Scherlis80] for such transformations, while [Smith89a] relied on a pre-condition notation. We have opted to use the more general constraint construct (which also includes guards and absolute set abstractions with logical variables) in order to incorporate this transformation under a more general setting.

Having specified what the new function definition should be, we can proceed to transform it into a more efficient function. This involves a sequence of steps that is shown in Figure 6.1. The first step is to unfold the main call, split_list. After this, the resulting equations are immediately subjected to the fold steps in order to obtain a self recursive function. These fold steps are always possible because we can replace split_list(x,w) by new_split(x,w,length(w)), in accordance with definition (6.1) of new_split. Finally, an attempt is made to remove all costly expressions, length(w), by substituting it with the parameter, d. This attempt involves the application of a distributive law and some simplification steps (unfold, reduce, laws) to convert the expressions into a simpler form where the costly expression can be replaced by the parameter, d. The final program is more efficient because the costly expression is now incrementally maintained in a new parameter.

```
- - -new_split(x,w,d)   <= split_list(x,w) st d=length(w);  ! unfold/narrow split_list

- - -new_split([],w,d)  <= w st d=length(w);

- - -new_split(a::x,w,d) <= if length(w)=6 then w::split_list(a::x,[])
                           else split_list(x,w<>[a]) st d=length(w);
                                    ! fold the two calls against new_split definition
                        <= if length(w)=6 then w::new_split(a::x,[],length([]))
                           else new_split(x,w<>[a],length(w<>[a])) st d=length(w);
                                    ! use laws length(w<>x)=length(w)<>length(x); length([])=0
                        <= if length(w)=6 then w::new_split(a::x,[],0)
                           else new_split(x,w<>[a],length(w)+length[a]) st d=length(w);
                                    ! substitute length(w)=d, unfold length [a]
                        <= if d=6 then w::new_split(a::x,[],0)
                           else new_split(x,w<>[a],1+length []) st d=length(w);
                                    ! unfold length []
                        <= if d=6 then w::new_split(a::x,[],0)
                           else new_split(x,w<>[a],1+0) st d=length(w);
                                    ! law x+0=x
                        <= if d=6 then w::new_split(a::x,[],0)
                           else new_split(x,w<>[a],1) st d=length(w);
```

Figure 6.1 - Transformation Sequence for new_split

Of all the steps in this transformation, there are two steps which are most difficult. The first difficult step is the correct identification of the costly expression (for the new definition step), while the second difficult step is a series of rules (distributive laws and simplifications) which enables the costly expression to be incrementally maintained. We shall see how these difficult steps can be systematised later. But first, let us take a general look at the constraints which are to be used for program transformation in this chapters.

The constraints used and introduced in our transformation do not make use of any logical variables. Instead, they appear mainly in expressions of the form, $e(x)$ **st** $\phi(x)$, where x is an input variable. These constraints are used to specify the *invariant* (or pre-condition) relationships which exist for their enclosed expressions, e.g. $e(x)$. They are special forms of constraints for representing additional semantic conditions (or contextual information) that could be attributed to the code of our program.

Their representations as syntactic entities (through the **st** construct) enable us to formally reason and manipulate these invariant constraints for *program transformation purposes*. However, it should be noted that they represent the contextual or semantic conditions, under which expressions appear, and as such are *not* required for *computation purposes*.

But, how are these (special) constraints used in transformation? Very often, they are used to simplify the sub-expressions that are under their context. For example, any sub-expressions of $e(x)$ in the above example, can make use of the constraint $\phi(x)$, to perform simplification transformation.

Also, how are these constraints introduced into our programs? We have already seen one way of introducing constraints, namely through new function definitions. Apart from constrained

function definitions, we could also accumulate the constraint or invariant conditions for certain sub-expressions under appropriate circumstances. For example, if we have an **if** expression,

if B(x) **then** C(x) **else** D(x)

we could assume that C(x) occurs under the context, B(x), while D(x) occurs under the context, not(B(x)). Because of this, we are permitted to add these contexts *explicitly* to the respective sub-expressions by changing the above **if** expression to:

if B(x) **then** (C(x) **st** B(x)) **else** (D(x) **st** not B(x)),

Similarly, if we have a set expression:

{y | y ∈ S(x),ϕ(x)}

where x is an input parameter variable, we consider the sub-expression, y ∈ S(x), to occur under the context/constraint of ϕ(x). As a result, we could add the condition, ϕ(x), to this sub-expression as follows:

{y | (y ∈ S(x) **st** ϕ(x)),ϕ(x)}

We shall refer to these operations as constraint adding laws. The reader should bear in mind that these *explicitly* added constraints are used for transformational purposes only. They do not affect the way expressions are computed.

6.2.2. The Tactic

We are now ready to give a detailed description of the general finite differencing tactic. This tactic is made up of two distinct phases, namely:

1) An analysis phase to *identify* expensive expressions (in the body of recursive functions) which could be *incrementally maintained* across recursion levels.
2) A transformation phase to define a new function with additional parameters to hold the value of each expensive expression, followed by a straightforward sequence of unfold/fold transformations to get a more efficient function.

We give a detailed account for both these phases next.

6.2.3. Phase 1: Analysis for Expensive Expressions

The analysis phase for expensive expressions will involve two tasks. The first task is to *identify* an expensive expression in the recursive function, while the second task must determine if the expression can be *incrementally maintained.* We consider each of these two tasks separately.

To assist us in the presentation, we shall consider the transformation for the following self-recursive (schematic) function, with N subsidiary recursive calls.

```
- - -f(x)    <= if B(x) then C(x)                                        (6.2)
               else H(x,f(nxt1 x),f(nxt2 x),...,f(nxtN x));
```

It is also possible for mutually recursive functions to be considered for finite differencing. To simplify matters, we shall assume that they will be converted to the above self-recursive form before being considered by this tactic.

6.2.3.1. Identification of Expensive Expression

The identification of expensive expressions would have to take into account two important issues. Firstly, we need to know the computational cost of the expensive expression (in relation to its incremental operator) in order to decide if the expression is worth eliminating. Secondly, we need to know if it has the right properties (e.g. incrementally maintainable) which will make it to be suitable for finite differencing. In this section, we shall be concerned mainly with the second issue of the right properties that make expressions suitable for finite differencing. As to the first issue, there are some work currently done on complexity analysis, e.g. [Sands90,Metayer88], which could be used to help decide if an expression is worth replacing or not.

Suitable expensive expressions for finite differencing could occur in any part of the recursive function but must not include the recursive function itself. In the case of function definition (6.2), this means a suitable expensive expression, say costly(x), may appear in either B(x), H(x,?,..,?), C(x) or even one of the $nxt_i(x)$ where $i \leq N$. Should the expensive expression be present in the terminating branch of recursion, i.e. C(x), then the potential improvements will be less significant. We may get linear time speedup, from a kind of loop combination (see Section 6.2.5 for an example), rather than asymptotic improvements that are possible when an expensive expression occurs in the repetitive parts of the recursive function.

The identification task therefore can look for expensive expressions in almost anywhere of the function body. One easy way out of this task is to let the user indicate the expensive expression. Such indications can be considered as decisions made by user in a semi-automatic approach. However, there are still plenty of opportunities for some form of automatic identification. One very useful and general heuristic is to initially identify certain categories of induction/changing parameters which are conducive to finite differencing. These parameters could then be used to search for expensive expressions.

For example, the earlier split_list function has a parameter of list type which accumulates elements across successive recursion by using the append operator, <>. This append operator often allows distributive laws (to be present) which may help incremental maintenance of expressions to be achieved. Similarly, the numeric and set domain also have certain kinds of incremental operations which are more conducive to finite differencing. Some examples of such parameter types, together with the desirable incremental change across successive recursion levels, are given below. In each case, the changing parameter is represented by the variable, w, with a small parameter change across successive recursive calls signified by the variable, c.

types	changes across recursion levels	
numeric	w + c, w - c	WHERE c is a constant
list type	w<> c, c <> w, tail(w)	WHERE c is small list, e.g. [a]
set type	w » c, w - c	WHERE c is small set, e.g. {a}
bag type	w » c, w - c	WHERE c is small bag, e.g. {a}

By specially recognising such parameters with incremental/inductive changes across recursions, we could systematically search for all auxiliary sub-expressions which use these parameters. These auxiliary subexpressions can be suggested as possible costly expressions for finite differencing. In addition to the above incremental change, we may also allow either no change (e.g. w -> w) or re-initialisations (e.g. w -> []).

An example of a parameter which satisfies the above condition is the parameter, w, of splitlist. This parameter could be used to find the sub-expression, length(w), which we already saw could be replaced by finite differencing.

6.2.3.2. Maintenance Laws

For each costly expression that we manage to identify, it is necessary to determine if the expression can be incrementally maintained across successive recursive calls. This is achieved by discovering **maintenance laws** for the expression across each of the subsidiary recursive calls.

Using the earlier schematic function, we must discover one maintenance law, per costly expression for *each* of the N subsidiary recursive calls, $f(nxt_1\ x), f(nxt_2\ x), .., f(nxt_N\ x)$. Each of these maintenance laws must either be one of the following two forms.

$costly(nxt_i\ x) = delta_i(costly(x),x)$ **(1) incremental law**

$costly(nxt_i\ x) = simplified_i(x)$ **(2) simplified or re-initialisation law**

where both $delta_i$ and $simplified_i$ are simple functions.

The first law expresses an incremental maintenance, which uses $delta_i$ to compute the next costly expression in terms of the previous costly expression; while the second law is a simplified law which does not need the previous costly expression. This second law is normally used in recursive calls where re-initialisation of parameter has just taken place. Both forms of maintenance laws can be discovered by applying the *distributive* and *simplification* laws together with expression *reduction* and *unfolding*.

As an example, the two laws (an incremental and a re-initialisation law) for the costly expression of split_list can be derived as follows:

```
length(w<>[a])  =  length(w<>[a])          ! distributive law
                =  length(w)+length([a])   ! unfold length([a])
                =  length(w)+(1+length([])) ! unfold length([])
                =  length(w)+(1+0)         ! reduce 1+0=1
                =  length(w)+1

length([])      =  length([])              ! unfold length([])
                =  0
```

These derivation steps could be handled by an inference sub-system which would attempt to simplify the expressions to either an incremental or re-initialised form. They can be automated

as long as the inference sub-system has an adequate supply of the relevant laws. The laws/rules for unfolding, simplification and reduction are fairly easy to provide, while that for the distributive condition is slightly harder. Nevertheless, some of these distributive laws could be synthesized and we present a technique to do this in Appendix 3.

Before continuing with the transformation phase, we will look at two additional forms of semantic laws. The first form is a *special* law which results from unchanged parameters, while the other form of semantic law is a proper *generalisation* of the incremental law.

6.2.3.3. Special Unchanged Laws

A very simple incremental law can result if we encounter parameters that remain unchanged in some of its recursive calls. When this happens, no incremental maintenance is needed for these calls, as the corresponding costly expression will also remain unchanged. The special maintenance law for this purpose is simply of the form:

$$costly(nxt_i\ x)=costly(x) \qquad \textbf{(3) unchanged law}$$

This *unchanged* law is a special case of the previous incremental law with $delta_i(y,_)=y$. They could be syntactically detected.

Furthermore, should this special situation be repeated for *all* N subsidiary recursive calls, we will have a special case of finite differencing where the costly expression remains unchanged throughout the entire recursive function. This special situation is in fact identical to a compiler optimisation technique known as *constant-code motion* out of loop. We give an example of this special case in Section 6.2.5.

6.2.3.4. Generalisation to Multiple Maintenance Laws

Sometimes, a costly expression may itself be defined in terms of other *auxiliary* or *mutual recursive* (costly) expressions. When this happens, we may need two or more maintenance laws per subsidiary recursive call in order to fully specify the costly expressions incrementally.

This possibility can be considered as a *generalisation* from the earlier incremental law. Instead of one incremental law per recursive call, $f(nxt_i\ x)$, we now have M incremental laws per recursive call, as follows:

$$\begin{aligned}
costly(nxt_i\ x) &= delta_i(x,costly(x),costly_2(x),..,costly_M(x)) \\
costly_2(nxt_i\ x) &= delta_{2i}(x,costly(x),costly_2(x),..,costly_M(x)) \\
&\vdots \qquad\qquad \vdots \\
costly_M(nxt_i\ x) &= delta_{Mi}(x,costly(x),costly_2(x),..,costly_M(x))
\end{aligned}$$

where $M \geq 1$ and $costly_2(x),..,costly_M(x)$ are the auxiliary or mutual recursive expressions that are needed to define costly(x) incrementally.

A simple example of where this is needed is the following function for tabulating a list of sine values.

```
dec lst_sin:real -> list(real);
- - -lst_sin(theta)    <= if theta>2*pi then []
                         else sin(theta)::lst_sin(theta+0.1);
```

The fairly expensive sine expression, sin(theta), uses a numeric induction parameter that changes from theta to theta+0.1, and can be incrementally maintained. However, we require two incremental laws for this purpose, with cos(theta) as the *mutually* recursive costly expression, as shown below.

```
sin(theta+0.1)  =  sin(theta)*cos(0.1)+cos(theta)*sin(0.1)
cos(theta+0.1)  =  sin(theta)*sin(0.1)-cos(theta)*cos(0.1)
```

Another example of the need for multiple incremental laws is a function with an expensive expression, $costly(i)=i^2$, that uses an induction parameter, $nxt(i)=i+1$. This costly expression will need an auxiliary expression, $costly_2(i)=2*i+1$, in order to completely replace both the squaring and multiplication operators by just additions, as shown in the following two incremental laws.

$$(i+1)^2 = (i^2+(2*i+1))$$
$$2*(i+1)+1 = (2*i+1)+2$$

Multiple maintenance laws, like the above, are especially important for complex expressions. Paige found these to be essential for set-theoretic finite differencing and called them **chain rules** [Paige82].

6.2.4. Phase 2 : Transformation

Once the analysis phase is successful, the transformation phase that follows is straightforward. We need only apply (automatically) the following four steps:

1) Define a new function, to be equivalent to the original function, but has an additional parameter for *each* of its costly expressions;
2) Unfold the original call in this new function;
3) Fold away all recursive calls of the original function;
4) Apply maintenance laws obtained from the analysis phase and then remove costly expressions by substituting from the constraint list.

Let us illustrate this sequence of steps with a simpler schematic example, g, which has only one recursive call and a single costly expression, costly(x).

```
- - -g(x)    <= if B(x) then C(x)
              else H(x,costly(x),g(nxt(x)));
```

We further assume that this costly expression must be defined in terms of another auxiliary expression, $costly_2$, using the following two incremental laws.

$$costly(nxt\ x) = delta(x,costly(x);costly_2(x)) \quad (6.3a)$$
$$costly_2(nxt\ x) = delta_2(x,costly_2(x)) \quad (6.3b)$$

The four steps required are now illustrated.

Step 1 - Define a New Function

The first step is to define a new function that would contain two additional parameters, one each for the costly and auxiliary costly expression, as shown below.

$$\text{- - -g_fd}(x,w_1,w_2) \quad \Leftarrow \quad g(x) \textbf{ st } w_1=\text{costly}(x),w_2=\text{costly}_2(x); \qquad (6.3c)$$

We use a list of constraints to associate each new parameter value to its costly expression.

Step 2 - Unfold the Original Function Call

We proceed with a single unfold that would reveal the body of the original function call, g(x), as follows.

```
- - -g_fd(x,w1,w2)   <= if B(x) then C(x)
                       else H(x,costly(x),g(nxt(x)))
                            st w1=costly(x),w2=costly2(x);
```

Step 3 - Fold to Remove the Original Recursive Calls

We then apply the fold rule to each recursive call of the original function, g, in order to remove all references to the old inefficient function as shown below.

```
- - -g_fd(x,w1,w2)   <= if B(x) then C(x)
                       else H(x,costly(x),g_fd(nxt(x),costly(nxt x),costly2(nxt x)))
                            st w1=costly(x),w2=costly2(x);
```

This fold step can always be applied because the implicit equation:

$$\text{g_fd}(x,\text{costly}(x),\text{costly}_2(x))=g(x)$$

is a direct consequence of definition (6.3c) for g_fd. This equality is obtained from the constrained equation of g_fd by substituting the constraint expressions of the parameters, w_1 and w_2, back into the LHS of the equation.

Step 4 - Eliminate Costly Expression

The last step of this transformation concerns the elimination of all costly expression. We achieve this in two sub-steps. Initially, we apply the two incremental laws, (6.3a) and (6.3b), to the costly expressions, in order to express them incrementally in terms of the old costly expressions, as follows:

```
- - -g_fd(x,w1,w2)   <= if B(x) then C(x)
                       else H(x,costly(x),g_fd(nxt(x),delta(x,costly(x),costly2(x)),
                                delta2(x,costly2(x)))
                            st w1=costly(x),w2=costly2(x);
```

The resulting function now uses the old costly expressions, costly(x) and $\text{costly}_2(x)$. Both these expressions can be eliminated by substituting them with corresponding parameters from the constraint list, as shown below.

```
                       <= if B(x) then C(x)
                       else H(x,w1,g_fd(nxt(x),delta(x,w1,w2),delta2(x,w2))
                            st w1=costly(x),w2=costly2(x);
```

The final transformed function is a more efficient function. Remember that the attached constraints are not use for computing results, and merely express some additional semantic conditions which hold for the function.

6.2.5. Special Cases of Finite Differencing

There are two special cases of finite differencing that are normally considered as different categories of transformation. We are able to formulate them as special cases because of our generalisation to constraint-based transformations.

The first special case corresponds to a kind of *loop combination* which arises when expensive expressions occur in terminating branches rather than at the repetitive places of recursive program. An example of this is the following function, rev_sm, which has an expensive expression, sum(w), in its base-case equation.

```
dec rev_sm:list(num) # list(num) -> (num # list(num));
dec double:list(num) -> list(num);
- - -rev_sm([],w)      <= (sum(w),w);
- - -rev_sm(a::x,w)   <= rev_sm(x,a::w);
- - -sum([])          <= [];
- - -sum(a::x)        <= a+sum(x);
```

This program is inefficient because there is a double traversal of its input. It builds up a reversed list in the second parameter, w, and then performs a summation of the reversed list. This double traversal can be removed by defining a new constrained function definition, rev_fd,

```
- - -rev_fd(x,w,s)    <= rev_sm(x,w) st s=sum(w);
```

which can be transformed to:

```
- - -rev_fd([],w,s)   <= (s,w);
- - -rev_fd(a::x,w,s) <= rev_fd(x,a::w,a+s);
```

The above transformed function uses only a single traversal of its input list in order to obtain both the summation and its reversed list.

A second special case of finite differencing arises when an expensive expression is based on the fixed parameters of a recursive function. When this happens, we can simply lift the expensive expressions out of the recursive function in a manner similar to *constant-code motion* out of loop. An example to illustrate this is the function:

```
dec g:list(num) # num -> list(num);
- - -g([],A)          <= [];
- - -g(a::x,A)        <= sqr(A)*a::g(x,A);
```

where the second parameter, A, is unchanged throughout the whole function call. The expensive expression sqr(A) (based on the unchanged parameter) can be moved out of the recursion by including it as a new parameter in the following function.

```
dec g_fd:list(num) # num # num -> list(num);
- - -g_fd(x,A,s)      <= g(x,A) st s=sqr(A);
```

This function can be transformed to:

```
- - -g_fd([],A,s)     <= [];
- - -g_fd(a::x,A,s)   <= s*a::g_fd(x,A,s);
```

without the expensive expression.

Incidently, this transformation causes the unchanged parameter, A, to be useless. This phenomenon is quite common in finite differencing as the addition of new parameters often causes some of the older parameters to become redundant. Such a situation can be easily detected and removed by the useless parameter elimination tactic of Chapter 2.

6.2.6. Comments

As seen in the above presentation, finite differencing tactic can be easily incorporated under the unfold/fold transformational framework. More importantly, the analysis phase which is required to automate the tactic appears to be fairly well-structured. The key steps are the identification of costly expressions followed by the discovery of the appropriate maintenance laws.

The identification of expensive expressions can be automated by exhaustively looking at all expensive expressions (not involving the recursive function itself) that make use of incremental or fixed parameters. The discovery of maintenance law is slightly more complex. This will involve the application of distributive and simplification laws together with expression reduction and unfolding to each costly expression of recursive call, in order to see whether they can be incrementally maintained, re-initialised or simply unchanged. To help us in this process, we will need a collection of useful distributive laws for common operators in numeric, set or list domains. The possibility of synthesizing some of the distributive laws is given in Appendix 3.

6.3. Base-Case Filter Promotion

Apart from finite differencing, constraints are also useful for performing filter promotion in a sub-class of generate-and-test algorithms. Chapter 3 illustrated how fusion techniques can be used to promote filters into generators. This works principally for programs of the form:

```
---prog(s)          <= { p | p ∈ gen(s),test(p)};
```

where the generator, gen(s), and filter, test(p), are two separate recursive function calls that can be merged by fusion.

However, not all filters in generate-and-test functions can be promoted in this way. In particular, there is a sub-class of recursive generate-and-test programs which cannot be improved by fusion. This sub-class of *recursive* functions contains *base-case* filters that are tested at the termination branches of the recursion. A simple schematic example of such a function is given below.

```
---prg_bc(x)        <= {ans(x) | ϕ(x)} ∪ {p_ans(v) | v ∈ prg_bc(nxt x),μ(x)};        (6.4)
```

The generator of this function is present in the recursive portion { p_ans(v) | v ∈ prg_bc(nxt x)}, while its base-case filter, ϕ(x), is applied at the base-case result, { ans(x) | ϕ(x)}, after each solution is fully generated. Notice that this is not the most general schematic function for representing

the class of recursive functions with base-case filters. A more general program scheme is presented in Section 6.3.5. For exposition purposes, we will use this simpler function, prg_bc (with a single base-case and a single recursive case), to illustrate the key features of the base-case filter promotion technique. The reader should note that some of the concrete examples (given in the following sub-sections) are actually instances of the more general scheme of Section 6.3.5, rather than this simple illustrative scheme.

This kind of function is often inefficient because its filter is applied at the very end, after each complete solution has been generated. Nevertheless, there is a possibility of improving the function by bringing forward the base-case filter into the generator. The effect of this promotion would be to filter out sub-spaces of infeasible solutions at much earlier stages. We can perform this by using a constraint-based transformation technique. But before we are allowed to do this, there is one very important semantic condition that must be satisfied.

6.3.1. Continuous Filter Condition

Before a base-case filter can be promoted to its recursive generators, we need to show a condition which states that:

> **IF** the filter condition fails for the present call **THEN** it will also fail for all its subsidiary recursive calls and their descendants.

Using the earlier schematic function, prg_bc, this condition can be precisely stated by the following logical implication,

$$\text{not } \phi(x) \rightarrow \text{not } \phi(\text{nxt } x)$$

which is equivalent to:

$$\phi(\text{nxt } x) \rightarrow \phi(x) \qquad (6.5)$$

where $\phi(x)$ is the base-case filter and nxt is the descent function of the subsidiary recursive call and $\rightarrow$ is the logical implication operator. We call the condition of (6.5), the **continuous filter condition**, because it relates the filter condition of successive recursive calls in a continuous manner.

Why is this continuous filter condition important? It is important, because its presence allows us to promote the base-case filter into the recursive part of the program. This promotion make use of the following law:

$$\text{prg_bc}(x) = \{v \mid v \in \text{prg_bc}(x), \phi(x)\} \qquad (6.5a)$$

which could *only* be obtained if we have the continuous filter condition. The inductive proof (requiring the continuous filter condition) to show this law is as follows:

Case $\phi(x)$=true, trivial because:

$$\begin{aligned} \text{RHS} &= \{v \mid v \in \text{prg_bc}(x), \text{true}\} \\ &= \{v \mid v \in \text{prg_bc}(x)\} \\ &= \text{prg_bc}(x) \\ &= \text{LHS} \end{aligned}$$

Case $\phi(x)$=false, we have:

RHS = {v | v ∈ prg_bc(x),false}
= {}
LHS = prg_bc(x) ! unfold prg_bc
= {ans(x) | ϕ(x)} ∪ {p_ans(v) | v ∈ prg_bc(nxt x),μ(x)}
! ϕ(x) → ϕ(x) & by induction
= {ans(x) | false} ∪ {p_ans(v) | v ∈ {v | v ∈ prg_bc(nxt x),ϕ(nxt x)},μ(x)}
= {p_ans(v) | v ∈ prg_bc(nxt x),ϕ(nxt x),μ(x)}
! ϕ(nxt x) → ϕ(x)
= {p_ans(v) | v ∈ prg_bc(nxt x),false,μ(x)}
= {}
= RHS

This law (obtained with the help of the continuous filter condition) can be used during transformation to promote the base-case filter into the recursive portion of the set function. We shall see how this is used in the coming examples.

An example of a program that satisfies this condition is the sub_less function for obtaining all subsequences (from an input list) which are less than or equal to eight items long.

```
dec sub_less: list(alpha) # list(alpha) -> set(list(alpha));
- - -sub_less([],w)     <= { w | length(w)≤8};
- - -sub_less(a::x,w)   <= sub_less(x,w) ∪ sub_less(x,w<>[a]);
```

This function accumulates its subsequences in the second parameter. It has two recursive calls, sub_less(x,w) and sub_less(x,w<>[a]), and a base-case filter, length(w)≤8. The base-case filter can be promoted into the recursive generator because it satisfies the continuous filter condition for both the recursive calls, as shown below.

```
length(w)≤8          → length(w)≤8  for sub_less(x,w)
length(w<>[a])≤8     → length(w)≤8  for sub_less(x,w<>[a])
```

The analysis to show that the above two conditions are satisfied must be carried out by some inference sub-system (perhaps semi-automatically). Our main concern in this chapter remains the systematisation of the necessary semantic conditions and the steps needed to perform base-case filter promotion.

6.3.2. Transformation

Once the continuous filter condition has been established, we can perform the filter promotion transformation on the function sub_less. This is done by initially defining a new function with the base-case filter as a constraint, as follows:

```
dec new_sub: list(alpha) # list(alpha) -> set(list(alpha));
- - -new_sub(x,w)     <= sub_less(x,w) st length(w)≤8;
```

This definition essentially states that the new function must always have the condition, length(w)≤8, present when its calls are invoked. We can introduce such a definition because of its continuous filter condition. In particular, we can have the following *special fold* rule (a reversed definition!):

```
sub_less(x,w)        =  {v | v ∈ new_sub(x,w),length(w)≤8}
```

that is obtained by the following sequence of transformation steps:

```
sub_less(x,w)   =  sub_less(x,w)                                  ! law from (6.5a)
                =  {v | v ∈ sub_less(x,w),length(w)≤8}             ! constraint adding law
                =  {v | v ∈ sub_less(x,w) st length(w)≤8,length(w)≤8}
                                                                  ! fold against new_sub
                =  {v | v ∈ new_sub(x,w),length(w)≤8}
```

We shall call the end-result of this derivation sequence, a *special fold* rule which required the continuous filter condition. In our examples, we will apply this *special fold* directly without going through the detailed steps.

With this constrained definition, we can proceed to transform the function by three major transformation steps. The first step is to unfold the original function call in order to reveal its function body. This is followed by a special fold operation on each of the recursive call (from the original function) in order to remove all references to the old function. After this, we can simplify all filters in the function body, which are also present in the constraint list, to true. Such a sequence of steps for new_sub is shown in Figure 6.2.

```
---new_sub(x,w)      <= sub_less(x,w) st length(w)≤8 ;      ! narrow sub_less

---new_sub([],w)     <= { w | length(w)≤8} st length(w)≤8;
                              ! simplify with law (& constraint) x → (x=true)
                    <= { w | true} st length(w)≤8;
                              ! law {w | true}=w
                    <= { w } st length(w)≤8;

---new_sub(a::x,w)   <= sub_less(x,w) ∪ sub_less(x,w<>[a])  st length(w)≤8 ;
                              ! law sub_less(x,w)={p|p ∈ new_sub(x,w),length(w)≤8)
                    <= { p | p ∈ new_sub(x,w),length(w)≤8}
                         ∪ { p | p ∈ new_sub(x,w<>[a]),length(w<>[a])≤8} st length(w)≤8;
                              ! simplify with law (& constraint) x → (x=true)
                    <= { p | p ∈ new_sub(x,w)} ∪ { p | p ∈ new_sub(x,w<>[a]),length(w<>[a])≤8}
                         st length(w)≤8;
```

Figure 6.2 - Filter Promotion through Constraint for new_sub

The final outcome of this transformation is the following more efficient function (not showing its constraints), with the base-case filter promoted to the generator portion of the function.

```
---new_sub([],w)     <= { w} ;
---new_sub(a::x,w)   <= { p | p ∈ new_sub(x,w)} ∪ { p | p ∈ new_sub(x,w<>[a]),length(w<>[a])≤8} ;
```

The steps employed for the above transformation are actually quite straightforward, once the analysis phase has successfully established the continuous filter condition. To show this, we will illustrate the above transformation again with the earlier schematic function, prg_bc. This transformation is shown in Figure 6.3 and consists of the following steps:

1) define a new function, constrained by its base-case filter, for the given function
2) unfold the given function call

3) fold away all of the given function's subsidiary recursive call

4) simplify filters, where possible, with the help of the constraint list

```
Assume that φ(nxt x) → φ(x)
Define a new constrained function definition
- - -new_prg(x)        <= prg_bc(x) st φ(x);

Perform the transformation
- - -new_prg(x)        <= prg_bc(x) st φ(x);
                            ! unfold prg_bc
                       <= {ans(x) | φ(x)} ∪ {p_ans(v) | v ∈ prg_bc(nxt x),μ (x)} st φ(x);
                            ! special fold of new_prg (assuming continuous filter cond)
                       <= {ans(x) | φ(x)} ∪ {p_ans(v) | v ∈ new_prg(nxt x),φ(nxt x),μ (x)} st φ(x);
                            ! simplify with law (& constraint) x → (x=true)
                       <= {ans(x)} ∪ {p_ans(v) | v ∈ new_prg(nxt x),φ(nxt x),μ (x)} st φ(x);
```

Figure 6.3 - Filter Promotion using the Continuous Filter Condition

We shall next look at two alternative semantic conditions for base-case filter promotion. One of them is a stronger (more specialised) condition than the continuous filter condition, while the other is a weaker (more general) condition.

6.3.3. Incremental Filter Condition

With certain base-case filters, we are often able to directly express the next filter condition, $\phi(nxt\ x)$, of the recursive call *incrementally* in terms of the original condition, $\phi(x)$, as follows:

$$\phi(nxt\ x) = \phi(x) \wedge \delta(x) \qquad (6.6)$$

We call this condition, the **incremental filter condition**. It simply indicates that the filter condition of each successive recursive call requires more tests than the calls before it. This is actually a stronger result than the continuous filter condition, as the following reasoning shows.

$\phi(nxt\ x) = \phi(x) \wedge \delta(x)$! replace = by the weaker $\rightarrow$

$\phi(nxt\ x) \rightarrow \phi(x) \wedge \delta(x)$! drop $\delta(x)$

$\phi(nxt\ x) \rightarrow \phi(x)$

Thus, all base-case filters with incremental conditions can always be promoted. In addition, we are also able to use the incremental condition, $\phi(nxt\ x) = \phi(x) \wedge \delta(x)$, to improve the program further. More specifically, we can simplify the filter test, $\phi(nxt\ x)$, to just the incremental test, $\delta(x)$, as shown in the sequence of transformation in Figure 6.4.

```
Assuming that φ(nxt x) = φ(x) ∧ δ(x)
Define a new constrained function definition
new_inc(x)          <= prg_bc(x) st φ(x);

Perform the transformation
new_inc(x)          <= prg_bc(x) st φ(x);
                              ! unfold prg_bc
                    <= {ans(x) | φ(x)} ∪ { p_ans(v) | v ∈ prg_bc(nxt x),μ (x)} st φ(x);
                              ! special fold of new_inc (assuming continuous filter condition)
                    <= {ans(x)| φ(x)} ∪ { p_ans(v) | v ∈ new_inc(nxt x),φ(nxt x),μ (x)} st φ(x);
                              ! incremental condition
                    <= {ans(x)| φ(x)} ∪ { p_ans(v) | v ∈ new_inc(nxt x),φ(x) ∧ δ(x),μ (x)} st φ(x);
                              ! simplify with law (& constraint) x → (x=true)
                    <= {ans(x)} ∪ { p_ans(v) | v ∈ new_inc(nxt x), δ(x),μ (x)} st φ(x);
```

Figure 6.4 - Filter Promotion using the Incremental Filter Condition

6.3.4. Weaker Filter Condition

At times, the base-case filter, $\phi(x)$, does not directly satisfy the continuous filter condition. Instead, it may be possible for a *weaker filter*, say $\gamma(x)$, that is a consequence of the base-case filter to have the necessary continuous filter condition, as shown below.

$$(\phi(x) \rightarrow \gamma(x)) \wedge (\gamma(nxt\ x) \rightarrow \gamma(x)) \qquad (6.7)$$

We call this, the **weaker filter continuous condition**. When it occurs, we could use the weaker implied filter, $\gamma(x)$, for promotion. Notice that this is actually a more general situation than our previous continuous filter condition for base-case filters. If we replace γ by ϕ, we have the original continuous filter condition because $\phi(x) \rightarrow \phi(x)$ is trivially true.

Similarly, a slightly stronger condition for weaker filters can be obtained if the incremental condition is present for the weaker filter. This gives us:

$$(\phi(x) \rightarrow \gamma(x)) \wedge (\gamma(nxt\ x) = \gamma(x) \wedge \delta(x)) \qquad (6.8)$$

which will be called, the **weaker filter incremental condition**. When this condition is satisfied, we can promote the weaker filter into the recursive generator. For simplicity, we shall refer to both conditions (6.7) and (6.8) as just the *weaker filter conditions*.

One immediate source of weaker filters is possible if $\phi(x)$ is a conjunctive filter of the form:

$$\phi(x) = \phi_1(x) \wedge \phi_2(x) \wedge ... \wedge \phi_n(x)$$

In this case, any one of $\phi_i(x)$ could be used as the weaker filter provided the continuous condition, $\phi_i(nxt\ x) \rightarrow \phi_i(x)$, or the incremental condition, $\phi_i(nxt\ x) = \phi_i(x) \wedge \delta_i(x)$, holds. However, other sources of weaker filters are also possible and we must be able to search for them. Later in this section, we shall look at an example of weaker filters which does not come from conjunctive expressions.

The weaker filter condition (and some stronger derivatives of it) could also allow filters to be promoted to the recursive parts of our functions. This is possible because we can use the law:

$$prg_bc(x) = \{v \mid v \in prg_bc(x), \gamma(x)\} \quad (6.9)$$

This law is valid because of the following proof:

Case $\gamma(x)$=true, trivial because:

```
RHS  =  {v | v ∈ prg_bc(x),true}
     =  {v | v ∈ prg_bc(x)}
     =  prg_bc(x)
     =  LHS
```

Case $\gamma(x)$=false, we have:

```
RHS  =  {v | v ∈ prg_bc(x),false}
     =  {}
LHS  =  prg_bc(x)                    ! unfold prg_bc
     =  {ans(x) | φ(x)} ∪ {p_ans(v) | v ∈ prg_bc(nxt x),μ(x)}
                                     ! φ(x) → γ(x) & by induction
     =  {ans(x) | false} ∪ {p_ans(v) | v ∈ {v | v ∈ prg_bc(nxt x),γ(nxt x)},μ(x)}
     =  {p_ans(v) | v ∈ prg_bc(nxt x),γ(nxt x),μ(x)}
                                     ! γ(nxt x) → γ(x)
     =  {p_ans(v) | v ∈ prg_bc(nxt x),false,μ(x)}
     =  {}
     =  RHS
```

With the help of this law (obtained by the weaker filter condition), we could define a new constrained function, new_weak:

```
new_weak(x)  =  prg_bc(x) st γ(x)
```

The transformation to promote the weaker filter of this constrained function is given in Figure 6.5. This transformation made use of the earlier law, (6.9), to obtain a *special fold* rule, as follows:

```
prg_bc(x)  =  {v | v ∈ prg_bc(x),γ(x)}          ! add a constraint in context
           =  {v | v ∈ prg_bc(x) st γ(x),γ(x)}   ! fold new_weak
           =  {v | v ∈ new_weak(x),γ(x)}
```

At the end of the transformation, both the original base-case filter and its weaker filter are needed in the final transformed program. The weaker filter, $\gamma(x)$, is there to help cut off *some* sub-spaces of infeasible solutions, while the base-case filter, $\phi(x)$, is needed for selecting only acceptable solutions (from the reduced space of solutions).

```
Assuming that φ(x) → γ(x) ∧ γ(nxt x) → γ(x);
Define a new constrained function definition
new_weak(x)      <= prg_bc(x) st γ(x);

Perform the transformation
new_weak(x)      <= prg_bc(x) st γ(x);
                              ! unfold prg_bc
                 <= {ans(x) | φ(x)} ∪ {p_ans(v) | v ∈ prg_bc(nxt x),μ(x)} st γ(x);
                              ! special fold of new_weak (assuming the weaker filter cond)
                 <= {ans(x) | φ(x)} ∪ {p_ans(v) | v ∈ new_weak(nxt x),γ(nxt x),μ(x)} st γ(x);
```

Figure 6.5 - Filter Promotion using the Weaker Filter Continuous Condition

A concrete example of the presence of two weaker implied filters is the following function with a base-case filter, length(w)=8.

```
dec sub_8: list(alpha) # list(alpha) -> set(list(alpha));
- - -sub_8([],w)      <= { w | length(w)=8};
- - -sub_8(a::x,w)    <= sub_8(x,w) ∪ sub_8(x,w<>[a]);
```

This function finds all sub-sequences from a list that is exactly 8 items in length. Though, the continuous filter condition is not satisfied by this particular base-case filter, there are actually *two* weaker filters which would satisfy the continuous condition instead. These two weaker filters are:

$$length(w) \leq 8 \quad \text{and} \quad length(w)+length(x) \geq 8$$

Both these filters are weaker than the base case filter because:

$$length(w)=8 \rightarrow length(w) \leq 8$$

and

$$length(w)=8 \rightarrow length(w)+length(x) \geq 8$$

In addition, these two filters also satisfy the continuous filter conditions for each of the two recursive calls, sub_8(x,w) and sub_8(x,w<>[a]), through the following provable statements.

$$\begin{aligned}
&length(w<>[a]) \leq 8 && \rightarrow length(w) \leq 8\\
&length(w) \leq 8 && \rightarrow length(w) \leq 8\\
&length(w<>[a])+length(x) \geq 8 && \rightarrow length(w)+length(a::x) \geq 8\\
&length(w)+length(x) \geq 8 && \rightarrow length(w)+length(a::x) \geq 8
\end{aligned}$$

With these two weaker filters, we can improve the original generate-and-test function by the sequence of transformation shown in Figure 6.6. The result of this transformation is to promote the two weaker filters to give:

```
- - -new_sub8([],w)    <= { w | length(w)=8} ;
- - -new_sub8(a::x,w)  <= {p | p ∈ new_sub8(x,w),length(w)+length(x)≥8}
                          ∪ {p | p ∈ new_sub8(x,w<>[a]),length(w<>[a])≤8};
```

The first filter, length(w<>[a])≤8, is there to ensure that the accumulated list has not exceeded a length of 8; while the second filter, length(w)+length(x)≥8, checks to ensure that the sum of the accumulated list, w, together with the remaining items, x, is still big enough to construct subsequences of eight items. This transformed function is more efficient as it avoids some sub-spaces of infeasible solutions.

```
Define a new constrained function definition
dec new_sub8 : list(alpha) # list(alpha) -> set(list(alpha));
- - -new_sub8(x,w)    <= sub_8(x,s) st length(w)≤8,length(w)+length(x)≥8;

Perform transformation (by initially narrowing on sub_8)
- - -new_sub8([],w)   <= { w | length(w)=8} st length(w)≤8,length(w)+length(x)≥8;

- - -new_sub8(a::x,w) <= sub_8(x,w) ∪ sub_8(x,w<>[a])
                          st length(w)≤8,length(w)+length(a::x)≥8;
                             ! special fold of new_sub8 (assuming weak filter condition)
                       <= { p | p ∈ new_sub8(x,w),length(w)≤8,length(w)+length(x)≥8}
                             ∪{p | p ∈ new_sub8(x,w<>[a]),length(w<>[a])≤8,length(w<>[a])+length(x)≥8}
                              st length(w)≤8,length(w)+length(a::x)≥8;
                                 ! simplify with law (& constraint) x → (x=true)
                                 ! also length(w<>[a])=length(w)+1
                       <= { p | p ∈ new_sub8(x,w),true,length(w)+length(x)≥8}
                             ∪{p | p ∈ new_sub8(x,w<>[a]),length(w<>[a])≤8,length(w)+1+length(x)≥8}
                              st length(w)≤8,length(w)+1+length(x)≥8;
                                 ! simplify with law (& constraint) (1+x)>b → (x>b = true)
                       <= { p | p ∈ new_sub8(x,w),length(w)+length(x)≥8}
                             ∪{p | p ∈ new_sub8(x,w<>[a]),length(w<>[a])≤8}
                              st length(w)≤8,length(w)+1+length(x)≥8;
```

Figure 6.6 - Filter Promotion with two Weaker Filters

A short note about the efficiency characteristics of the transformed program when compared to the original program. Except for programs that are transformed via the incremental filter condition (which could be shown to preserve efficiency), the other transformed programs may either cause gains or losses in efficiency. These programs are able to save on computation time by cutting off certain sub-spaces (with infeasible solutions) from their search spaces, but they also require additional filter tests (i.e. the promoted filters) to help perform these cut-offs. There is therefore a trade-off between the reduction in search spaces and the additional filter tests required.

A program can make efficiency gain if the gain from its reduced search space is able to compensate for the additional computations required by its promoted filter. This trade-off is not, however, always clear cut and is often data dependent. For example, the transformed function, new_sub8, could result in considerable efficiency gains if its input list is long (much longer than eight items) but actually causes a loss of efficiency if this list has less than eight items (as no sub-spaces are filtered out)!

In general, we should also prefer transformations which introduce *simple* filters and avoid those which require *complex* filters. One way of ensuring this is to subject our program to further transformations. One transformation tactic that is frequently applicable is finite differencing. This transformation could be used to incrementally maintain the (expensive) expressions, length(w) and length(x), of the earlier example. This will simplify the filter test considerably.

6.3.5. A More General Program Scheme

The schematic function of (6.4), in the beginning of Section 6.3, is not the most general program scheme which could be transformed by the base-case filter promotion tactic. A more general program scheme is the following:

$$\text{---prg}(x) \quad \Leftarrow \bigcup_{i \in 1..n} \{ \text{ans}_i(x) \mid \phi_i(x) \} \cup \bigcup_{j \in 1..m} \{ \text{p_ans}_j(v) \mid v \in \text{prg}(\text{nxt}_j\, x), \mu_j(x) \};$$

which allows multiple base-cases, $\bigcup_{i \in 1..n} \{ \text{ans}_i(x) \mid \phi_i(x) \}$, and multiple recursive cases, $\bigcup_{j \in 1..m} \{ \text{p_ans}_j(v) \mid v \in \text{prg}(\text{nxt}_j\, x), \mu_j(x) \}$. In fact, two of the previous examples (i.e. new_sub8 and sub_less) were actually members of this class of programs because they have multiple recursive function calls.

With this new program form, an appropriate weaker filter, $\gamma(x)$, could be used (for filter promotion) if it satisfies the following.

$$(\forall i \in 1..n\ \phi_i(x) \rightarrow \gamma(x)) \wedge (\forall j \in 1..m\ \gamma(\text{nxt}_j\, x) \rightarrow \gamma(x))$$

Furthermore, programs which use the conditional constructs or pattern matching are <u>not</u> incompatible with the above program scheme. For example, a conditional function of the form:

$$\text{---p}(x) \quad \Leftarrow \textbf{ If } B(x) \textbf{ then } \bigcup_{i \in 1..n} \{ \text{ans}_i(x) \mid \phi_i(x) \}$$

$$\textbf{else } \bigcup_{j \in 1..m} \{ \text{p_ans}_j(v) \mid v \in \text{p}(\text{nxt}_j\, x), \mu_j(x) \};$$

is simply equivalent to :

$$\text{---p}(x) \quad \Leftarrow \bigcup_{i \in 1..n} \{ \text{ans}_i(x) \mid \phi_i(x), B(x) \} \cup \bigcup_{j \in 1..m} \{ \text{p_ans}_j(v) \mid v \in \text{p}(\text{nxt}_j\, x), \mu_j(x), \text{not } B(x) \};$$

where the conditional test, B(x), is added to all the base-cases and not(B(x)) is added to all the recursive cases.

6.3.6. Comments

Our new filter promotion transformation tactic (not implemented yet) again needs an analysis phase that would:

1) syntactically recognise recursive programs with base-case filters; and then
2) semantically check for either the incremental filter condition, continuous filter condition or weaker filter condition.

The syntactic condition can be mechanically checked by parsing but the check for the semantic conditions is non-trivial. In particular, the search for weaker implied filters would, in general, have to come from a fairly sophisticated inference sub-system.

6.4. Keeping and Using the Invariants/Constraints

The constraint-based transformations of finite differencing and base-case filter promotion could accumulate a list of invariants/constraints for each user function they transform. As we saw

earlier, this list of constraints is not needed for computational purpose but it is extremely useful for program transformation.

In formulating the various semantic conditions (or laws) for the two transformation tactics, we have largely ignored the potential use of the invariants that are carried over from the *previous* applications of transformations. These invariants may assist in making subsequent transformations applicable. To show how they could be used, let us reformulate the schematic function (6.4) which contains a base-case filter, to the following:

$$\text{---prg_bc}(x) \quad <= \quad \{\text{ans}(x) \mid \phi(x)\} \cup \{\text{p_ans}(v) \mid v \in \text{prg_bc}(\text{nxt } x), \mu(x)\} \textbf{ st } \text{inv}(x)$$

where inv(x) represents the list of invariants/constraints that were carried over from the previous transformations.

With the presence of these invariants, the weaker filter condition of (6.7) and (6.8) could be re-specified in a more general way, as follows:

$$\text{inv}(x) \rightarrow (\phi(x) \rightarrow \gamma(x) \wedge \gamma(\text{nxt } x) \rightarrow \gamma(x))$$
$$\text{inv}(x) \rightarrow (\phi(x) \rightarrow \gamma(x) \wedge \gamma(\text{nxt } x) = \gamma(x) \wedge \delta(x))$$

This essentially states that we could use the previous invariants to help us establish the weaker filter condition. Similarly, the continuous filter condition of (6.5) and the incremental filter condition of (6.6) could also make use of the previous invariants, as follows:

$$\text{inv}(x) \rightarrow (\phi(\text{nxt } x) \rightarrow \phi(x))$$
$$\text{inv}(x) \rightarrow (\phi(\text{nxt } x) = \phi(x) \wedge \delta(x))$$

Correspondingly, these invariants may also be useful in the finite differencing tactic to help find the appropriate incremental laws.

6.5. A Related Work - Global Search Algorithms

Recently, Smith and Lowry [Smith89a] presented a *global search algorithm* design strategy which can be used to improve high-level generate-and-test program by deriving a filter, called the *necessary filter*, to be promoted. Their design strategy is formulated using (essentially) a schematic approach which relates by theory extensions a problem theory (the *source scheme*) to an algorithm theory (the *target scheme*). The relationship between the problem and algorithm theories may be further constrained by additional axioms (*applicability conditions*).

Let us represent Smith & Lowry's global search algorithm design strategy transformation by the corresponding functional programs of the problem and algorithm theories.

In the case of the problem theory, a source program scheme (used in [Smith89a]) to implement the theory is:

dec F : D -> set(R);
---F(x) <= {z | z ∈ D , O(x,z)} **st** I(x);

where I is an input condition (or pre-condition) for the parameter, x, and O is an output condition for the result, z, in relation to x. Also, the types D and set(R) are the domain and range

of the problem specification. Notice that this is a very high-level way of writing generate-and-test specifications (which may be unexecutable).

The above program scheme can be transformed to the following target program scheme which represents a class of global search algorithms.

dec F : D -> set(R);
dec F_gs : D # $\hat{R}$ -> set(R);
dec Split : D # $\hat{R}$ -> set($\hat{R}$);
dec $\hat{r}_0$: D -> $\hat{R}$;
dec Extract : $\hat{R}$ -> set(R);

$$F(x) \Leftarrow \{z \mid \Phi(x,\hat{r}_0(x)) \ , \ z \in F_gs(x,\hat{r}_0(x))\} \ \textbf{st} \ I(x);$$

$$F_gs(x,\hat{r}) \Leftarrow \{z \mid z \in Extract(\hat{r}) \ , \ O(x,z)\} \cup \{a \mid a \in F_gs(x,\hat{s}) \ , \ \hat{s} \in Split(x,\hat{r}) \ , \ \Phi(x,\hat{s})\}$$
$$\textbf{st} \ I(x) \wedge \Phi(x,\hat{r});$$

where a new type, $\hat{R}$, is used to represent subspace descriptors (sets of solutions) of the original range R. This new subspace descriptor must have three associated functions, $\hat{r}_0$, Split and Extract. The function $\hat{r}_0$ is used to obtain an initial sub-space descriptor; the function Split is used to split a sub-space of solutions into a further set of sub-space descriptors; while the function Extract is used to obtain a set of extractable solutions from the given sub-space. These new functions of the target program scheme must be related to the original program scheme by the following two axioms.

$$\forall x,z. \ O(x,z) \Rightarrow Satisfies(z,\hat{r}_0(x))$$
$$\forall x,\hat{r}. \ Satisfies(x,\hat{r}) \Leftrightarrow \exists k \in Nat, \exists \hat{s} \in \hat{R} \ [\hat{s} \in Split^k(x,\hat{r}) \wedge z \in Extract(\hat{s})]$$

where $Split^k$ is defined by:

$$Split^0(x,\hat{r}) = \{\hat{r}\}$$
$$Split^{n+1}(x,\hat{r}) = \{\hat{t} \mid \hat{s} \in Split(x,\hat{r}) \ , \hat{t} \in Split^n(x,\hat{s})\}$$

The first axiom states that all feasible solutions can be obtained from the initial sub-space descriptor, while the second axiom states that a feasible solution is in a sub-space if it could be extracted after a finite number of applications of Split. The sub-space domain and the corresponding functions, $\hat{r}_0$, Split and Extract, must be made available for each possible target program scheme. However, there is a necessary filter, Φ, of the target program which could be *derived* by ensuring that it satisfies the following condition:

$$\forall \hat{r} \in \hat{R}, \exists z \in R \ [\ Satisfies(z,\hat{r}) \wedge O(x,z) \ ] \Rightarrow \Phi(x,\hat{r})$$

Smith & Lowry proved that the above global-search target algorithm will implement the original problem theory if its specified conditions are satisfied.

The use of theories, containing a set of sorts/types and functions, to represent classes of programs/algorithms is certainly an interesting and generally applicable approach. In fact, the global search design strategy encompassed both the fusion and constraint-based filter

promotion techniques presented in this thesis. In addition, it could also be used to formulate other kinds of search algorithms, e.g. heuristic search or branch-and-bound[1].

However, the approach taken by us differs from Smith & Lowry in several aspects. The global search algorithm design strategy relies on the provision of appropriate target schemes (containing the appropriate sub-space descriptors and their associated functions) for its transformation. In contrast, we make use of more fundamental transformation (symbolic evaluation) techniques, e.g. removal-of-unification, filter promotion by fusion and filter promotion by constraints, which could directly obtain efficient search algorithms from the specification. This has the potential advantage that we could derive implementations without being constrained by the provided target schemes. Furthermore, our fundamental transformation techniques handle both high-level as well as intermediate-level specifications.

6.6. Summary of Chapter 6

This chapter presented two transformation tactics, finite differencing and base-case filter promotion, which required *constraints* for specifying useful invariants to improve certain classes of recursive functions. We have not yet implemented both of these two tactics. However, we have identified that the key requirement in both these two tactics is the successful discovery of laws to maintain specified invariants throughout the recursive functions. If this could be achieved, these programs are improved through either the replacement of expensive expressions by cheaper operators (i.e. finite differencing) or the elimination of sub-spaces of infeasible solutions from the search space (i.e. base-case filter promotion).

We have managed to systematise the main semantic conditions/laws required by the two tactics. In particular, we identified the *unchanged*, *simplified*, *incremental* and *multiple incremental* laws for the finite differencing tactic. We also established the *incremental*, *continuous* and *weaker filter conditions* for the base-case filter promotion tactic.

Finally, we suggested that the search for these semantic conditions should be handled by a sub-systems with more powerful inference or theorem-proving capability.

[1] We need an additional transformation technique (not presented in this thesis) to derive branch-and-bound algorithm.

7. Conclusion

7.1. Summary of Thesis

At the beginning of this thesis, we identified three criteria, namely *correctness*, *generality* and *automatibility*, that are essential for program transformation systems which are aimed at providing formalised routes to software construction. We also identified the unfold/fold methodology as a suitable framework for such transformation systems. This methodology is near-complete, preserves (partial) correctness and is not incompatible with the goal of providing automated methods.

After this introduction, we showed in **Chapter 2** how transformation tactics could be built (as meta-programs) on top of the elementary unfold/fold rules. This arrangement helps to preserve correctness. We also saw that each of these tactics is normally structured into an analysis and a transformation phase. The analysis phase is the key phase for the tactic and is used to determine if the program could be transformed. We illustrated this two-phase structure with two very simple transformation tactics. One of them is an *algorithmic* tactic for eliminating useless parameters. This tactic uses a forward analysis technique to identify useless parameters. The other transformation tactic is a *schematic* tactic for converting programs to tail-recursive form. This tactic uses the Function Equivalence Matching technique to help determine if a given function is an instance of another higher-order one. These two tactics were presented to illustrate the possibility of building meaningful and automatable transformations based on the unfold/fold rules.

Our main contribution in this thesis is the systematisation of four major classes of transformation tactics, namely *fusion*, *higher-order removal*, *tupling* and *constraint-based transformations*. These tactics are widely applicable and systematic enough to be automatable.

In **Chapter 3**, we considered the class of transformations that was characterised by the fusion of composed expressions. We introduced a producer-consumer model to characterise this fusion transformation. We showed how the deforestation algorithm of Phil Wadler could be extended to all first-order programs - using this model to explain the apparatus needed for our extension. We made novel use of the variable-only parameter criterion. We also established an important termination result for this extended transformation algorithm. Lastly, we applied a similar technique to the fusion of expressions in set abstractions and presented the desirable program forms needed to ensure successful transformation.

In **Chapter 4**, we presented a transformation tactic, consisting of three algorithms, to remove certain classes of higher-order features from functional programs. This tactic has two major benefits, namely:

(1) the elimination of overheads associated with higher-order features, and

(2) the provision of an easier route for extending transformation and optimisation techniques from first-order to higher-order programs.

We showed that the higher-order removal transformation algorithms will work and terminate for all well-typed programs. Two of the algorithms, $\mathcal{A}$ and $\mathcal{R}$, are directly used to remove higher-order features. A third transformation algorithm, $\mathcal{L}$, is concerned with preserving full laziness in a way that is compatible with higher-order removal. The classic full laziness technique of Hughes was not compatible and we proposed a new technique, based on algorithm $\mathcal{L}$, to extract out all (implicit and explicit) ground-type MFEs rather than any MFEs. We showed that this did not result in any loss of efficiency, provided we ignore the laziness associated with partially-applied recursive functions. This laziness is, in any case, space-leak related. With the help of this higher-order removal tactic, we showed how the Universal Deforestation Algorithm of Chapter 3 could be extended to higher-order programs.

In **Chapter 5**, we presented a very general analysis technique for the tupling transformation. This analysis, called *tupling analysis*, is based on the presence of appropriate selection orderings to help build up *continuous* sequences of cuts. We showed the various practical techniques which could be used to determine suitable *selection orderings*. These practical ordering techniques include those based on pattern matching (e.g. instantiate and unfold ordering) as well as those based on recursion parameters that are *progressive* (e.g. function-argument orderings). We illustrated the general applicability of our approach by showing that it works for a wide class of programs. Furthermore, by extending the tupling analysis from *sequences* of continuous cuts to *trees* of continuous cuts, we showed how our technique could be extended to programs which use non-linear data structures and multiple-recursive equations.

In **Chapter 6**, we considered how constraints could be used by two transformation tactics. One of them, *finite differencing*, used constraints to introduce new parameters which could replace expensive expressions from recursive programs. We identified four types of laws that could be used in this tactic, namely *unchanged*, *simplified*, *incremental* and *multiple incremental* laws. The other transformation tactic is a new form of filter promotion which promotes *base-case* filters of recursive functions. This class of transformations used constraints to help filter out sub-spaces of infeasible solutions. We identified the semantic conditions that are needed before the tactic is considered to be applicable. These semantic conditions are the *incremental*, *continuous* and *weaker* filter conditions. We also pointed to the need for some inference capability to help establish (by verification or synthesis) these conditions/laws.

7.2. Underlying Computational Structures

In a sense, the classes of transformations presented are actually based on some more fundamental computation structures. These computational structures could be used to classify our transformation tactics. In particular, the three fundamental computation structures used (in this thesis) are:

1) compositions
2) tuples
3) constraints/invariants

The simplest computational structure (from the point of view of transformation) is *composition.* This is used by both fusion and higher-order removal transformations. The essential requirement for this computational structure is to identify the places where fusion could take place. To help us in this task, we used some annotations to indicate the places where fusion can occur and apply generalisations to the places where fusion cannot occur. Furthermore, most of these transformations could be handled by the define, unfold and fold steps together with the appropriate distributive laws (e.g. for set abstractions).

Another computational structure that is useful is the *tuple* structure. This structure is used to group recursive function calls together. An essential requirement for this tuple structure is to get eureka tuples which have descendant tuples that match. We saw how this requirement could be discovered by the tupling analysis technique, which uses appropriate selection orderings to generate continuous sequences of tuples for the matching test.

Similarly, *constraints/invariants* are used to help improve some recursive programs through specialised contexts. The essential transformation task needed by this computational structure is to ensure that the constraints are maintained throughout the recursion.

These computational structures represent more fundamental underlying mechanisms for program transformation techniques. Further types of computational structures are also possible. They include the use of *lambda abstractions* to perform a range of transformations as illustrated in [Wand80,Pettorossi88] and *lists* and *vectors* (in the place of tuples) to provide dynamic-sized tables for certain classes of redundant programs [Bird86,Cohen83]. In addition, it may also be possible to use combinations of the different computational structures. For example, the *tuple* structure may be used with *composed* sub-expressions rather than function calls. To be successful, these more complex structures have to satisfy the simultaneous requirements of each of its constituent structures (e.g. both the requirements of tupling and fusion).

Such computational structures provide us with a more fundamental view on how program improvement by transformation works. Further investigations along this line may help to formulate more fundamental transformation techniques.

7.3. Future Work

The classes of automatable tactics, presented in this thesis, represent substantial contributions to the state-of-art in formalised software construction techniques through program transformation. Despite these contributions, much work remains to be done in order to bring the fruits of transformation technology closer to practical use. Some further work which may be pursued include:

- In an earlier work, a limited number of tactics have been implemented (by the author) as meta-programs (which return scripts) in the Imperial College's Transformation-Based Programming Environment. The implemented tactics include:

(1) a simple form of loop combination (even simpler than pure deforestation!) where functions are self-recursive
(2) a tupling tactic which uses the *instantiate and unfold* selection ordering
(3) a simple partial evaluator.

Much implementation work therefore remains to be done in order to evaluate out the other proposed transformation tactics. These to-be-implemented tactics include higher-order removal, finite-differencing, filter-promotion and more general fusion and function specialisation techniques. An implementation for them will substantially add to the automated capabilities of our transformation system. At the moment, a third year student is helping with the implementation of some of the tactics.

- Another avenue which could be pursued is the incorporation of some of the more automatable tactics (e.g. deforestation and higher-order removal) to functional-language compilers. Present-day functional language compilers are still some way from producing program code with acceptable efficiency. The possibility of providing these transformation tactics as high-level optimisations may dramatically improve this situation and encourage programmers to adopt a reusable and modularised approach to programming. Furthermore, this application could provide a direct route for reaping the benefits of program transformation research. A number of researchers have been pursuing this avenue by incorporating sophisticated program transformation techniques in extended-compilation systems [Paige87,Turchin86].

- Currently, each of the transformation tactics is presented in a stand-alone fashion where the users are expected to give strategic guidances to ensure that tactics are applied in the appropriate sequence. Nevertheless, it may be desirable for the system to spot some of the more obvious sequences of tactic applications. We saw this in the higher-order removal tactic where we indicated to the reader that the three transformation algorithms, $\mathcal{A}$, $\mathcal{L}$ and $\mathcal{R}$, should be applied in sequence. This is done with the possibility of re-invoking prior algorithms if necessary. Also, we noted that these higher-order removal transformations should be applied before deforestation - again with the possibility of re-invoking the earlier transformation algorithms. Other examples of appropriate sequences of applying tactics include the application of useless parameter elimination after finite differencing as a tidying-up stage and the application of higher-order removal after the schematic conversion-to-iteration tactic.

Apart from these straightforward sequences of applications (which could be built into our tactics), we may also wish to provide our system with more sophisticated strategies for ordering transformation tactics. Fickas implemented a semi-automatic system, called Glitter [Fickas82], which used a goal-directed approach to mechanise the planning aspects of transformations. In his system, the user specifies a set of goals and the problem-solver finds an appropriate sequence of transformation methods which could achieve this goal with the help of selection rules (and possibility the user, as well). Corresponding, Alan Bundy et al [Bundy89] reconstructed an extended form of Boyer-Moore's theorem-proving system [Boyer79], which used tactics as planning operators to build proof plans. These tactics are

formulated into methods by adding in pre- and post-conditions to describe the applicability conditions and resulting states of applying the tactics. Further investigations into these two works may yield useful frameworks for formulating application strategies for our transformation tactics.

- In the presentation of some of the our transformation tactics (e.g. constraint-based tactics), we have indicated the need for a more general inference (deductive and inductive) capability that could be used to derive or prove certain laws needed. The KIDS system of Kestrel Institute [Smith89b] uses a general deductive system, called RAINBOW II, which uses *directed inference* for a number of purposes. This could be used for simplification, theorem-proving and derivation of antecedent, consequent or expression bounds. It is achieved by searching via deductive transformation (according to a specified relationship e.g. $\Leftarrow$, $\Rightarrow$, $=$, $\geq$, $\leq$) to obtain a target expression from a source expression which bears the specified relationship. A more complete inference sub-system for our transformation system could perhaps supplement this approach with inductive (e.g. [Bundy89]) and synthesis capability (e.g. distributive lemma of Appendix 3). It may also be organised along tactical lines for more guided searches.

- The tactics presented in this thesis are only a sub-set of the possible transformation techniques which could be systematised. Many other researchers have been working on transformation techniques which apply to various classes of programs. Some examples include the removal-of-unification tactic in [Pull90], the synthesis of divide-and-conquer algorithms in [Smith85,Smith87], the generation of inverse functions in [Harrison88b,Khoshnevisan89], the abstract data-types transformations in [Darlington84], the global search design tactics of [Smith89a]. It would be useful to classify and build up a taxonomy for all these classes of transformation tactics and investigate how they inter-relate.

In conclusion, the program design problem domain, as exemplified by program transformation, contains a rich structure of techniques which could be systematised into automatic or semi-automatic methods. This thesis presented a collection of these methods and developed the necessary analysis for automating them. Our work represents a small but certain step towards formalised and automated processes for software construction. Certainly, machines are destined to play a more significant role in software development.

References

[AbsInterp87] Eds S Abramsky, CL Hankin, *Abstract Interpretation of Declarative Languages* Ellis Horwood, 1987

[Aho77] AV Aho, JD Ullman, *Principles of Compiler Design* Addison-Wesley Publishing Co., 1977

[Allen81] FE Allen, J Cocke and K Kennedy, *Reduction of Operator Strength* In Program Flow Analysis (eds S Muchnick, N Jones), Prentice Hall, 1981 pp 79-101

[Arsac82] J Arsac, Y Kodratoff *Some Techniques for Recursion Removal for Recursive Functions*. ACM TRANS ON PROG LANG AND SYSTEM, Vol 4(2), pp 295-322 1982

[Augustsson85] L Augustsson *Compiling Pattern-Matching*. Conf on Functional Programming and Computer Architecture, Nancy, pp 368-381, LNCS 201 (ed Jouannaud), 1985

[Backus78] J Backus, *Can Programming be Liberated from the von-Neumann Style? A Functional Style and its Algebra of Program*, Comm ACM 21 pp 613-641, 1978

[Balzer81] R Balzer, *Transformational Implementation: An Example*, IEEE Trans on Software Engineering, Vol SE-7, No 1, pp 3-14, 1981

[Bauer79] FL Bauer, *Program Development by Stepwise Transformation: The CIP Project* in Program Constructions (eds FL Bauer, M Broy) pp 237-272, LNCS 69, 1979

[Bauer81] FL Bauer, M Broy et al *Programming in a Wide-Spectrum Language: A Collection of Examples*. Science of Computer Programming 1 pp 73-114, 1981

[Bauer82] FL Bauer, H Wossner, *Algorithmic Language and Program Development*. Springer-Verlag, New York, 1982

[Bird80] RS Bird, *Tabulation Techniques for Recursive Programs* ACM Computing Survey Vol 12 No4, pp 403-417, Dec 1980

[Bird86] RS Bird, *Transformational Programming and the Paragraph Problem* Science of Computer Programming Vol 6, pp 150-189 1986

[Bird87a] RS Bird, *An Introduction to the Theory of Lists*. Logic of Programming and Calculi of Discrete Design, (ed M Broy), Springer-Verlag pp 3-42 1987

[Bird87b] RS Bird, *A Calculus of Functions for Program Derivation*. Technical Report PRG64, Oxford University, Programming Research Group Dec 1987

[Bondorf88] A Bondorf, *Towards a Self-Applicable Partial Evaluator for Term Rewriting Systems* , in [PE&MIX88] pp 22-50

[Boyer79] RS Boyer, JS Moore *A Computational Logic*, Academic Press, ACM monograph series 1979

[Bundy81] A Bundy, LS Sterling, *Meta-Level Inference in Algebra*, Research Paper 163, Dept of AI, Univ of Edinburgh, 1981; also revised as Report 273

[Bundy83] A Bundy, *The Computer Modelling of Mathematical Reasoning*, Academic Press, Inc, 1983

[Bundy89] A Bundy, Fv Harmelen, J Hesketh, A Smaill, A Stevens, *A Rational Reconstruction and Extension of Recursion Analysis*, Intl Joint Conf on Artificial Intelligence, 1989

[Burn87a] GL Burn, S Abramsky, CL Hankin, *The Theory of Strictness Analysis for Higher-Order Functions* DIKU Workshop on Programs as Data Objects, LNCS 217, pp 42-62 1987

[Burn87b] GL Burn, *Abstract Interpretation and Parallel Evaluation of Functional Languages* PhD Thesis, University of London, 1987

[Burstall77] RM Burstall, J Darlington, *A Transformation System for Developing Recursive Programs*. Journal ACM Vol 24, 1 (Jan 1977) pp 44-67.

[Burstall80] RM Burstall, DB MacQueen, DT Sannella *HOPE: An Experimental Applicative Language* Proc 1980 Lisp Conf, Stanford, California pp 136-143.

[Chin89] WN Chin, J Darlington. *Schematic Rules within the Unfold/Fold Program Transformation System* in the TENCON Special Session on Functional Language : Theory and Applications. Bombay, India Nov 89.

[Chin90] WN Chin. *Termination Proofs for Transformation Algorithms* Internal Report, Functional Programming Section, Dept of Computing, Imperial College, May 1990.

[CIP85] Bauer et al, *The Munich Project CIP* Vol 1, Springer Verlag 1985

[Cohen83] NH Cohen, *Eliminating Redundant Recursive Calls* ACM Trans on Prog Lang & Sys, Vol 5 No 3, pp 265-299, Jul 1983

[Cooper66] DC Cooper *The Equivalence of Certain Computations*. Computer J, 9, pp 45-52 1966

[Darlington76] J Darlington, RM Burstall *A System which Automatically Improves Program*. Acta Informatica 6, pp 41-60 1976

[Darlington78] J Darlington *A Synthesis of Several Sorting Algorithms*. Acta Informatica 11, pp 1-30 1978

[Darlington81] J Darlington *An Experimental Program Transformation and Synthesis System*. Artif Intell 16, pp 1-46, 1981

[Darlington84] J Darlington *Design of Efficient Data Representation.* in *Automatic Program Construction Techniques*, eds AW Biermann, G Guiho, Y Kodratoff, Macmillan Publishing Co. pp 139-153, 1984

[Darlington86] J Darlington, AJ Field, H Pull *The Unification of Functional and Logic Languages.* in *Functional and Logic Programming*, eds D DeGroot, G Lindstrom, Prentice-Hall, pp 37-70, 1986

[Darlington88] J Darlington, H Pull *A Program Development Methodology Based on a Unified Approach to Execution and Transformation*, in [PE&MIX88], pp 117-132

[Darlington89a] J Darlington, YK Guo. *Unification of Functional and Logic Languages - Towards Constraint Functional Programming* in the TENCON Special Session on Functional Language : Theory and Applications. Bombay, India Nov 89.

[Darlington89b] J Darlington, et al. *A Functional Programming Environment Supporting Execution, Partial Execution and Transformation*, in PARLE '89, Eindhoven, Netherlands, pp 286-305, Sept 1989

[Darlington89c] J Darlington, P Harrison, AJ Field. *The Exploitation of Parallel Hardware Using Functional Languages and Program Transformations*, Project Summary, Functional Programming Section, Imperial College, 1989

[DeJong86] G DeJong, R Mooney, *Explanation-Based Learning: An Alternative View* Machine Learning Vol 1, pp 145-176, 1986

[Dershowitz79] N Dershowitz, Z Manna *Proving Termination with Multi-Set Ordering.* Comm. ACM Vol 22, No 8, pp 465-476, 1979

[Dijkstra72] EW Dijkstra *Notes on Structured Programming* In Structured Programming. (eds OJ Dahl, EW Dijkstra and CAR Hoare) Academic Press 1972.

[Dijkstra76] EW Dijkstra *A Discipline of Programming* Prentice-Hall 1976.

[Earley76] J Earley, *High Level Iterators and a Method for Automatically Designing Data Structure Representation* Journal of Computer Languages Vol 1,4 pp 554-567. LNCS 204 1976

[Ershov82] AP Ershov *Mixed Computation: Potential Applications and Problems for Study.* Theoretical Computer Science 18, pp 41-67, 1982

[Fairbairn82] J Fairbairn, *Ponder and its Type System* Cambridge University, Computer Laboratory, Technical Report 31 1982

[Feather79] MS Feather *A System for Developing Programs by Transformations* PhD Thesis, Univ of Edinburgh 1979.

[Feather82] MS Feather *A System for Assisting Program Transformation* ACM Trans on Prog Lang & Sys Vol 4, No 1 January 1982, pp 1-20.

[Ferguson88] AB Ferguson, P Wadler, *When will Deforestation Stop?*, Proc of 1988 Glasgow Workshop of Functional Programming, Rothesay, Isle of Bute August 1988 - as Glasgow University, Computer Science Dept Research Report 89/R4, pp 39-56, Feb 1989

[Fickas82] S Fickas, *Automating the Transformational Development*, PhD Dissertation, Univ of California, St Irvine 1982

[Futamura71] Y Futamura, *Partial Evaluation of Computation Process - An Approach to Compiler-Compiler.* Systems Computers Controls Vol 2,5 pp 45-50 1971

[Gelernter63] H Gelernter *Emperical Explorations of the Geometry Theorem-Proving Machine*, In Computers and Thought, pp 153-163, McGraw Hill, 1963

[Goguen87] J Goguen, J Meseguer *Extensions and Foundations for Object-Oriented Programming*, SRI Draft Report, March 1987

[Goguen88] J Goguen, *Higher-Order Functions Considered Unnecessary for Higher-Order Programming*, SRI-CSL-88-1, Project No. 1243, SRI International, Jan 1988

[Goldstine77] Goldstine H, *A History of Numerical Analysis* Springer-Verlag, New York, 1977

[Green83] C Green, D Luckman, R Balzer, T Cheatham, C Rich *Report on A Knowledge-Base Software Assistant.* Kestrel Institute, Report KES.U.83.2, June 1983

[Harrison86] PG Harrison, Khoshnevisan H *Efficient Compilation of Linear Recursive Functions into Object-Level Loops.* SIGPLAN 86 Symp on Compiler Construction Palo Alto June 1986

[Harrison88a] PG Harrison *Linearisation: An Optimisation for Non-Linear Functional Programs* Science of Computer Programming, Vol 10 No 3, pp 281-318, June 1988

[Harrison88b] PG Harrison *Function Inversion* in [PE&MIX88], pp 153-166

[Haskell89] P Hudak, P Wadler (eds) *Report on the Functional Programming Language Haskell* Version 1.0, Dec 1989

[Heckmann88] R Heckmann, *A Functional Language for the Specification of Complex Tree Transformation*, Proc European Symposium on Programming, LNCS 300, pp 175-190, 1988

[Hindley69] R Hindley, *The Principal Type-Scheme of an Object in Combinatory Logic.* Trans Am Math Soc, 29-60, 1969

[Hudak85] P Hudak, B Goldberg *Serial Combinators* Functional Programming Languages and Computer Architecture, Nancy, LNCS 201, pp 382-389 1985

[Hudak88] P Hudak, J Young *A Collecting Interpretation of Expressions (Without Powerdomains).* in Proc of 15th Annual SIGACTSIGPLAN Symp on Principle of Prog Language, San Diego, pp 107-118, 1989

[Huet78] G Huet, B Lang *Proving and Applying Program Transformation Expressed with Second Order Patterns.* Acta Inf 11 pp 31-55 1978

[Hughes82] J Hughes *Supercombinators: A New Implementation Method for Applicative Languages* Proc ACM Symposium on Lisp and Functional Programming, Pittsburgh pp 1-10, 1982

[Hughes88] J Hughes *Backward Analysis of Functional Programs*, in [PE&MIX88] 187-208

[Hunt89] S Hunt, *Frontiers and Open Sets in Abstract Interpretation* 4th Proc of Conference on Functional Programming Languages and Computer Architecture, Imperial College, London Sept, pp 1-13, 1989

[Jaffar87] J Jaffar, JL Lassez, *Constraint Logic Programming* Proc of Conference on Principles of Programming Language, Munich, pp 111-119, 1987

[Johnsson85] T Johnsson, *Lambda Lifting: Transforming Programs to Recursive Equations* in Proc of Workshop on Implementation of Functional Languages (eds L Augustsson, J Hughes, T Johnsson, K Karlsson) Report 17, pp 165-180 Goteberg, Feb 1985

[JonesN85] ND Jones, P Sestoft, H Sondergaard, *An Experiment in Partial Evaluation: The Generation of a Compiler Generator.* First Intl Conf on Rewriting Techniques and Applications, Dijon, France, pp 124-140, May 1985

[JonesN88a] assembled by ND Jones, *Challenging Problems in Partial Evaluation and Mixed-Computations*, in [PE&MIX88], pp 1-14

[JonesN88b] ND Jones, *Automatic Program Specialisation: A Re-Examination from Basic Principles*, in [PE&MIX88], pp 225-282

[JonesSB89] SB Jones, DL Metayer, *Compile-Time Garbage Collection by Sharing Analysis* 4th Proc of Conference on Functional Programming Languages and Computer Architecture, Imperial College, London Sept, pp 54-74, 1989

[Kanamori87] Kanamori, K Horiuchi, *Constructions of Logic Program based on Generalised Unfold/Fold Rules.* Proc of 4th Intl Conference on Logic Programming, Victoria, Australia, pp 743-768,May 87

[KBruckner89] B Kreig-Bruckner, *Algebraic Specification and Functionals for Transformational Program and Meta Program Development*, TAPSOFT '89, Intl Joint Conf on the Theory and Practice of Software Development, LNCS 352, pp 36-59, March 1989.

[Khoshnevisan87] H Knoshnevisan, *Automatic Transformation Systems based on Function-Level Reasoning* PhD Thesis, Imperial College, Univ of London 1987

[Khoshnevisan89] H Khoshnevisan, KM Sephton *InvX - An Automatic Function Inverter.* in Rewriting Techniques and Applications (eds N Dershowitz) LNCS 355, Chapel Hill, North Carolina, 1989

[Kieburtz81] R Kieburtz, J Shultis, *Transformation of FP Program Schemes* Functional Prog Lang and Computer Architecture Conf, New Hampshire, pp 41-48, Oct 1981

[Kott78] L Kott, *About a Transformation System: A Theoretical Study.* Proc 3rd Symposium on Programming, Paris, pp 232-267, 1978

[LCF79] M Gordon, AJ Milner, CP Wadsworth, *Edinburgh LCF*, Springer Verlag, LNCS 78, 1979

[Manna79] Z Manna, R Waldinger, *Synthesis: Dreams => Programs*, IEEE Transaction on Software Engineering, Vol SE5, No 4, pp 90-121, July 1979.

[Manna80] Z Manna, R Waldinger, *A Deductive Approach to Program Synthesis.* ACM Trans on Prog Lang and Sys 2,1, pp 90-121 Jan 1980

[Martin90] C Martin, *Finding Fixpoints in Abstract Interpretation* PhD Thesis, University of London, 1990

[Metayer88] DL Metayer, *An Automatic Complexity Evaluator* ACM Trans on Prog Lang & Sys, Vol 10 (2), pp 69-73, April 1988

[Michalski83] RS Michalski, *A Theory and Methodology of Inductive Learning* Artificial Intelligence, Vol 20, pp 111-161, 1983.

[Michie68] D Michie *"Memo" Functions and Machine Learning.* Nature 218, pp 19-22, Mar 1968

[Milner78] R Milner, *A Theory of Type Polymorphism.* J Comp & Sys Sci, pp 348-375 1978

[Mitchell82] TM Mitchell, *Generalisation as Search* Artificial Intelligence, Vol 18, pp 203-226, Mar 1982.

[Mitchell86] TM Mitchell, RM Keller, ST Kedar-Cabelli, *Explanation-Based Generalisation: A Unifying View* Machine Learning Vol 1, pp 47-80, 1986

[Morgensen88] T Morgensen, *Partially Static Structures in a Self-Applicable Partial Evaluator* in [PE&MIX88] pp 325-347

[Mycroft81] A Mycroft, *Abstract Interpretation and Optimising Transformations for Applicative Programs* PhD Thesis, University of Edinburgh, 1981

[Nielson89] HR Nielson, F Nielson, *Transformations on Higher-Order Functions*, Conference on Functional Programming Languages and Computer Architecture, Imperial College, pp 129-143, Sept 1989

[Paige82] R Paige, S Koenig, *Finite Differencing of Computable Expressions* ACM Trans on Prog Lang & Sys 4,3 July 1982, pp 402-454

[Paige87] R Paige, F Henglein *Mechanical Translation of Set Theoretic Problem Specifications Into Efficient RAM Code - A Case Study.* J. Symbolic Computation Vol 4, 2 pp 207-232 1987

[Parnas72] D Parnas, *On the Criteria to be Used in Decomposing Systems into Modules* Comm. ACM Vol 15, 12 pp 1053-1058 Dec 1972

[Partsch83] H Partsch, R Steinbruggen *Program Transformation Systems*, Computing Survey Vol 15 No 3, pp 199-236, Sept 1983

[PE&MIX88] Eds. D Bjorner, AP Ershov and ND Jones, *Workshop on Partial Evaluation and Mixed Computations* Oct 1987, Gl Avarnes, Denmark, North-Holland, 1988

[Perry89a] N Perry, *Hope+* Report IC/FPR/LANG/2.5.1/7 Imperial College, Oct 1989

[Perry89b] N Perry, *From I/O to Concurrent Programming* Draft Paper, Imperial College, Nov 1989

[Pettorossi82] A Pettorossi, RM Burstall *Deriving Very Efficient Algorithm for Evaluating Linear Recurrence Relations* Acta Informatica 18, pp 181-206, 1982

[Pettorossi84] A Pettorossi, *A Powerful Strategy for Deriving Programs by Transformation* ACM Lisp & Functional Programming Conference, pp 273-281, 1984

[Pettorossi88] A Pettorossi, M Proietti, *Importing and Exporting Information in Program Development* in [PE&MIX88], pp 405-426

[PeytonJ87] S Peyton-Jones *The Implementation of Functional Programming Languages*, Prentice-Hall International, 1987

[Phillips83] J Phillips *Self-Described Programming Environments*. PhD dissertation, Dept of Computer Science, Stanford University, March 1983

[Pippenger80] N Pippenger, *Pebbling* IBM Thomas J Watson Research Centre, Technical Report RC 8258 (#35937) May 1990

[Pull88] H Pull *The Partial Evaluator Script Language Definition - Version 2.0.* Internal Report IC/FPR/LANG/2.3/1 Jul 1988

[Pull90] H Pull *Implementation and Transformation Techniques based on Unification for an Extended Functional Language.* PhD Thesis, Univ of London, to be published 1990

[Reynolds72] RC Reynolds, *Definitional Interpreters for Higher-Order Programming Languages* ACM Annual Conference pp 717-740, 1972

[Runciman89] C Runciman, M Firth, N Jagger, *Transformation in a Non-Strict Language: An Approach to Instantiation* Glasgow Functional Programming Workshop, Aug 1989

[Sands90] D Sands, *Complexity Analysis for a Lazy Higher-Order Language*, to appear in the European Symposium on Programming, Copenhagen, Denmark, May 1990

[Scherlis80] WL Scherlis *Expression Procedures and Program Derivations*. PhD Thesis, Stanford University, Aug 1980

[Sephton89] KM Sephton *Hope+ Compiler*. Int Doc IC/FPR/LANG/2.5.1/14 Dept of Computing, Imperial College, 1989

[Sestoft88] P Sestoft, *Automatic Call Unfolding in a Partial Evaluator*, in [PE&MIX88] pp 485-506

[Sharp90] D Sharp, *Functional Language Program Transformation for Parallel Computer Architectures*. PhD Thesis, Univ of London, to be published 1990

[Shaw90] M Shaw, *Larger Scale Systems Require Higher-Level Abstractions*. 5th Intl Workshop on Software Specification and Design, Pittsburgh, Pennsylvania, pp 143-146 May 1989

[Smith85] DR Smith, *Top-Down Synthesis of Divide-and-Conquer Algorithms*. Artificial Intelligence 6, pp 46-96 1985

[Smith87] DR Smith, *Applications of a Strategy for Designing Divide-and-Conquer Algorithms*. Science of Computer Programming, pp 213-229 1987

[Smith89a] DR Smith, MR Lowry, *Algorithm Theories and Design Tactics*. Proc of Intl Conference on the Mathematics of Program Constructions, LNCS, Springer-Verlag, June 1989

[Smith89b] DR Smith, *KIDS: A Semi-Automatic Program Development System*. Kestrel Institute, Oct 1989

[Sterling82] L Sterling, A Bundy et al, *Solving Symbolic Equations with PRESS*, Research Paper 171, Dept of AI, Univ of Edinburgh, 1982

[Turchin86] VF Turchin, *The Concept of a Supercompiler* ACM Trans on Prog Lang & Sys Vol 8,3, pp 90-121 Jul 1986

[Turchin88] VF Turchin, *An Algorithm of Generalisation in the Supercompiler* in [PE&MIX88] pp 531-549

[Turner85] DA Turner, *Miranda: a Non-Strict Functional Language with Polymorphic Types*. Proc ACM Conf on Functional Programming Languages and Computer Architecture,Nancy, France, pp 1-16, May 1985

[Wadler87a] P Wadler, *Efficient Compilation of Pattern-Matching* as Chapter 5 of [PeytonJ87], pp 78-103

[Wadler87b] P Wadler, *Strictness on Non-Flat Domain* as Chapter 12 of [AbsInterp87]

[Wadler88] P Wadler, *Deforestation: Transforming Programs to Eliminate Trees*, European Symposium on Programming Nancy, France, pp 344-358, March 1988

[Wadsworth71] CP Wadsworth, *Semantics and Pragmatics of the Lambda Calculus*, PhD Thesis, Oxford 1971

[Wand80] M Wand, *Continuation-Based Program Transformation Strategies*, Journal ACM, Vol 27, pp 164-180, 1980

[Wirth71] N Wirth *Program Development by Stepwise Refinement.* Comm. ACM , Vol 14, N0 4, pp 221-227,Apr 1971

[Yourdon75] E Yourdon, L Constantine *Structured Design.* Yourdon Press, New York, 1975

Appendix 1. Higher-Order Useless Parameter Eliminations

The elimination of useless parameters tactic of Chapter 2 can be extended to work with higher-order programs. This is best performed after the elimination of undesirable higher-order features by another transformation tactic in Chapter 4.

The expression form which remains, after the elimination of higher-order features, is called the *higher-order specialised* form (see Section 4.3.1). The corresponding higher-order specialised form for the simple language in Section 2.4.1 can be described by the following grammar rules.

t ::= $va \mid c(t_1,..,t_n) \mid f(t_1..t_n,vh_1^*..vh_q^*)$ | **case** t_0 **in** p_1 => t_1 | .. | p_r => t_r **end**
| **lambda** $(v_1,..,v_n)$ => tf **end**

va ::= $v \mid va(t_1,..,t_n)$

p ::= $v \mid c(p_1,..,p_n)$

Only two additional constructs are present in the above grammar rules when compared to the simple first-order language. These are variable applications, $va(t_1,..,t_n)$, and lambda abstractions, **lambda** $(v_1,..,v_n)$ => tf **end**. In this appendix, we shall present the extensions needed in both the analysis and transformation phases to cope with these two features.

A1.1. Extension to Analysis Phase

For the analysis phase, we shall have to add two new syntax-directed translation rules and a new equation for the least upper bound function, or, as shown below.

use [[lambda $(v_1,..,v_n)$ => tf end]] ⇐ lambda $(v_1,..,v_n)$ => **use** [[tf]] end

use [[$va(t_1,..,t_n)$]] ⇐ **use** [[va]] (**use** [[t_1]],..,**use**[[t_n]])

WHERE

or (lambda $(v_1,..,v_n)$ => tf_1 end, lambda $(v_1,..,v_n)$ => tf_2 end)
⇐ lambda $(v_1,..,v_n)$ => or (tf_1,tf_2) end

These new rules must be added to those rules, given in Section 2.4.2 for first-order expressions. The two **use** rules are for dealing with variable applications and lambda abstractions respectively. Also, we have an additional equation for the semantic function, or, to compute the least upper bound values of any two abstract functions.

Another affected portion of the analysis phase is the appropriate abstract calls needed to identify useless parameters. Previously, each parameter to be identified must be tested with an abstract function call, $f\#(..,P,..)$, where the abstract value **P** is given to the parameter to be identified, while each of the other arguments is given the value **A**. These abstract values are only applicable to ground-type parameters. For function-type parameters, say of type $\alpha_1\#..\#\alpha_n \rightarrow \alpha_{result}$, we will have to use different abstract values. In particular, if this function-type parameter is the parameter of interest, we have to use the abstract value, lambda $(v_1,..,v_n)$ => **P** end, otherwise the abstract value of lambda $(v_1,..,v_n)$ => set_OR$\{v_1,..,v_n\}$ end is needed.

As an example, consider the following higher-order function with a function-type parameter, v, and two other parameters, a and b.

```
- - -ho_prog(v,a,b)    <=  v(a) ;
```

The definition for corresponding abstract function can be obtained using the syntax-directed translation rule, as follows.

```
ho_prog# (v,a,b)    =    use [[v(a) ]]
                    =    v(a)                                    (A1.1)
```

With this abstract function, we can analyse the three parameters by computing the semantic values for the following three calls.

```
ho_prog# (lambda v => P end,A,A)    =  P
ho_prog# (lambda v => v end,P,A)    =  P
ho_prog# (lambda v => v end,A,P)    =  A
```

These three analyses state that the third parameter can be eliminated but not the first two. Notice that we have used lambda v => v end, rather than lambda v => A end, if the function-type parameter is *not* the parameter of interest. This is so because we would like to pass the function-type parameter's possible argument (i.e. a in equation (A1.1)) to the result. If this is not done, we may wrongly identify the second parameter to be useless.

The above example only gives the abstract values needed for a second-order parameter of the type $(\alpha_1,..,\alpha_n)$ -> α_{result}, where $(\alpha_1,..,\alpha_n)$ are ground-type arguments and α_{result} is a ground-type result. For arbitrary higher-order parameters, we need a more complex (and complete) way of obtaining abstract argument values. The appropriate abstract values to supply is, in fact, dependent on the *type* of the parameters. We can describe the type, τ, of our simple expressions by the following simple grammar rule.

τ ::= ground_type | $(\tau_1,..,\tau_n)$ -> τ_{result}

With this type structure, any parameter, say arg_x, whose type is given by Type [arg_x], should be given the abstract value of ABS[Type[arg_x],P] if it is the parameter to be identified. Otherwise, if it is one of the other parameters, an abstract value of ABS[Type[arg_x],A], should be given.

The definition for ABS is given below.

```
ABS[ ground_type,r ]          =  r
ABS[ (τ1,..,τn) -> τresult,P] =  lambda (v1,..,vn) => ABS[τresult,P] end
ABS[ (τ1,..,τn) -> τresult,r ] =  lambda (v1,..,vn) => ABS[τresult
                                      ,set_OR{aux[τ1,v1],..,aux[τ1,v1],r}] end
aux [ ground_type,v ]         =  v
aux [ (τ1,..,τn) -> τresult ,v ] =  aux [τresult, v(ABS[ τ1,A],..,ABS[ τ1,A])]
```

Under this formulation, each ground-type parameter will be given either the value P or A, as before. Also, each higher-order parameter will be given an abstract function value that either returns the value P or simply propagates its argument values to the final result.

A1.2. Extension to Transformation Phase

Apart from the above changes to the analysis phase, some changes must also be made to the transformation rules to enable them to handle higher-order specialised expressions. In

particular, we will need two more new rules, as shown below.

(5) $\mathcal{U}[\![\text{lambda } (v_1,...,v_n) \Rightarrow \text{tf end}]\!] \Leftarrow \text{lambda } (v_1,...,v_n) \Rightarrow \mathcal{U}[\![\text{tf}]\!] \text{ end}$

(6) $\mathcal{U}[\![\text{va}(t_1,...,t_n)]\!] \Leftarrow \mathcal{U}[\![\text{va}]\!](\mathcal{U}[\![t_1]\!],...,\mathcal{U}[\![t_n]\!])$

These two new rules are for handling lambda abstractions and variable applications respectively. Together with the changes to the analysis phase given earlier, we now have a useless parameter eliminations tactic that can be applied to any higher-order program.

Appendix 2. Other Conversion-to-Iteration Schemes

In Chapter 2, we presented a conversion-to-iteration schematic rule that works for all linear recursive functions which have the constant-at-termination and the special right commutative properties. This class of programs is represented by the higher-order function, f1, in Section 2.5.1.

Apart from this special right commutative scheme, there are also a number of other transformation schemes for converting linear-recursive functions to their tail-recursive equivalent. In this Appendix, we shall illustrate three more of these schemes.

For generality, we shall use the following linear recursive function,

```
- - - f(x,B,C,D,H,N)   <= if B(x) then C(x)
                          else H(D(x),f(N(x),B,C,D,H,N));
```

as the common source program for each of the three transformation schemes. These schemes will differ primarily in the additional syntactic or semantic conditions placed on the function, f.

The first scheme to be presented is a general technique for converting any linear recursive function, matching f, to its tail-recursive program using a two-pass iteration and a data stack. The data stack is used (in the first pass) to collect all arguments of the recursive function calls before the second pass utilises it to compute the result. This scheme is very general because it only requires functions to be linear recursive. However, the transformed tail-recursive program contains little improvement.

A second transformation scheme can be formulated if an efficient inverse function, call it invN, is available for the descent function, N, such that invN(N(a))=a. The presence of this inverse function can be used to eliminate the need for the data stack and thus provides a slightly better tail-recursive program.

Lastly, another well-known and useful transformation scheme that will be presented, is based on the associative dual property being present for the outer operator, H, such that H(a,H(b,c))=H(J(a,b),c) for some function, J. This scheme, also known as the re-bracketing technique [Darlington76], will always have an efficient tail-recursive equivalent.

All three schemes are not new. What is new about our formulations is that we have managed to derive these schemes using the constructive unfold/fold transformation methodology. Almost all previous presentations of them, e.g. [Kieburtz81,Harrison86,Bauer82], have relied on the verification approach where both source and target schemes are given before a posterior proof.

A2.1. Two-Pass Iteration with Data Stack

Program transformations, especially those based on elementary rules, typically require some insightful steps before useful results can be obtained. One of these difficult steps, frequently referred to as the eureka step, is the *define step* for obtaining a new function in terms of the other existing functions. This step is considered to be difficult because it often represents the

starting definition for a desirable target program. Choosing the right target definition, from the numerous possibilities, requires intelligent inductive capability.

One useful technique which could be used is to analyse the given function *symbolically* through different evaluation strategies. This analysis could sometimes suggest to us better evaluation strategies that could be applied to compute our functions. More importantly, it could often suggest appropriate new function definitions for program transformation. Let us see how the analysis by symbolic evaluation could suggest an appropriate eureka definition for the conversion-to-iteration transformation.

If we take a snap shot of the evaluation of function call, f(x,B,C,D,H,N), after it has make n+1 recursive invocations, we could get the following intermediate expression:

$$H(D(x),H(D(Nx),H(D(N^2x),...,H(D(N^nx),f(N^{n+1}x,B,C,D,H,N))...)))$$

assuming that the base case conditional test, B(x), is false for each of the n+1 function invocations. This intermediate value represents a general computational history for the recursive call, and can be symbolically altered (using a different evaluation strategy) to the following:

$$H(D(x),?) \circ H(D(Nx),?) \circ H(D(N^2x),?) \circ .. \circ H(D(N^nx),?)\ f(N^{n+1}x,B,C,D,H,N) \qquad (A2.1)$$

where H(D(x),?) is a short hand notation for **lambda** a => H(D(x),a) **end** and o is the function composition operator where a o b = **lambda** c => a(b(c)) **end**.

The expression, $H(D(x),?) \circ H(D(Nx),?) \circ H(D(N^2x),?) \circ .. \circ H(D(N^nx),?)$, to the left of the above recursive call, $f(N^{n+1}x,B,C,D,H,N)$, involves a composition of n+1 functional sub-expressions which differ only in the arguments $x,Nx,N^2x..,N^nx$. This expression pattern can be captured, more generally, by rewriting the expression of (A2.1) to:

$$r_apply([N^nx,...,N^2x,Nx,x],H,D)\ f(N^{n+1}x,B,C,D,H,N)$$

where there is a new function, r_apply, that can be defined as follows:

$$r_apply([N^nx,...,N^2x,Nx,x],H,D) = H(D(x),?) \circ H(D(Nx),?) \circ H(D(N^2x),?) \circ .. \circ H(D(N^nx),?)$$

We have extracted the arguments, $N^nx,...,N^2x,Nx,x$, in reverse order as a data stack for the new function, r_apply. This new function is equivalent (can be synthesized) to the following definition,

```
---r_apply([],H,D)     <= id;
---r_apply(a::s,H,D)  <= r_apply(s,H,D) o H(D(a),?);
```

where id is the identity function.

With this analysis of an alternative expression form which may be taken by function f, we can now define a new function, F2_it, as follows:

```
---F2_it(x,s,B,C,D,H,N)  <= r_apply(s,H,D) f(x,B,C,D,H,N);          (A2.2)
```

This function definition carries an additional data stack, s, and can be transformed to its tail-recursive program, as shown in Figure A2.1. The result of this transformation is a tail-recursive program, which is made up of two functions F2_it and F2a_it, as shown below.

```
- - - F2_it(x,s,B,C,D,H,N)  <=  If B(x) then F2a_it(s,H,D,C(x))
                              else F2_it (N(x),x::s,B,C,D,H,N);
- - - F2a_it([],H,D,w)      <=  w;
- - - F2a_it(a::s,H,D,w)    <=  F2a_it(s,H,D,H(D(a),w));
```

The first function, F2_it, accumulates the argument, x, of each recursive call into the data stack, s. This is done during the main recursion of F2_it, which acts as the first iterative pass. At the end of this recursion (or pass), the auxiliary recursive function, F2a_it, is called as the second iterative pass for computing the results, bottom-up. This second pass uses the argument values from the accumulated stack in its computation.

```
- - - F2_it(x,s,B,C,D,H,N)  <=  r_apply(s,H,D) f(x,B,C,D,H,N);                  ! unfold f
                            <=  r_apply(s,H,D)  If B(x) then C(x)              ! distribute if
                                                else H(D(x),f(N(x),B,C,D,H,N));
                            <=  If B(x) then r_apply(s,H,D) C(x)               ! abstract H(D(x),?)
                                else r_apply(s,H,D) H(D(x),f(N(x),B,C,D,H,N));
                            <=  If B(x) then r_apply(s,H,D) C(x)               ! fold r_apply
                                else r_apply(s,H,D) o H(D(x),?) f(N(x),B,C,D,H,N);
                            <=  If B(x) then r_apply(s,H,D) C(x)               ! fold F2_it
                                else r_apply(x::s,H,D) f(N(x),B,C,D,H,N);
                            <=  If B(x) then r_apply(s,H,D) C(x)               ! define F2a_it
                                else F2_it (N(x),x::s,B,C,D,H,N);
                            <=  If B(x) then F2a_it(s,H,D,C(x))
                                else F2_it (N(x),x::s,B,C,D,H,N);
Define F2a_it
- - - F2a_it(s,H,D,w)      <=  r_apply(s,H,D) w;                              ! narrow r_apply
- - - F2a_it([],H,D,w)      <=  id w;                                          ! unfold id
                            <=  w;
- - - F2a_it(a::s,H,D,w)    <=  r_apply(s,H,D) o H(D(a),?) w;                  ! (f o g) (w) = f (g(w))
                            <=  r_apply(s,H,D)  H(D(a),w));                    ! fold F2a_it
                            <=  F2a_it(s,H,D,H(D(a),w));
```

Figure A2.1 - Transforming to a Two-Pass Iteration with Data Stack

The main schematic rule from this transformation is the following.

Schematic Rule 2 for f2_it

```
f(x,B,C,D,H,N)  =  F2_it(x,[],B,C,D,H,N)
```

This rule can be used to replace any linear recursive function matching f, with the two-pass tail-recursive function F2_it. It is obtained by instantiating s=[] in the original equation (A2.2) of F2_it, followed by the simplification steps, as shown below.

```
F2_it(x,[],B,C,D,H,N)  =  r_apply([],H,D) f(x,B,C,D,H,N)     ! unfold r_apply
F2_it(x,[],B,C,D,H,N)  =  id f(x,B,C,D,H,N)                  ! unfold id
F2_it(x,[],B,C,D,H,N)  =  f(x,B,C,D,H,N)
```

A2.2. Scheme with Function Inversion

The previous tail-recursive result (using a two-pass iteration and a data stack) for f is general but *not* efficient.

If an inverse function, say invN, for N can be found such that invN(N(a))=a, we can at least avoid the need for the data stack, $[N^n x,...,N^2 x,Nx,x]$. With this inverse function, we will only need to keep the two ends of the stack, i.e. $N^n x$ and x, rather than the entire data stack. This is possible because we can always compute the next element, $N^{n-1}x$, by applying invN to the top element, $N^n x$. The bottom element, x, is needed to tell us when we have reached the end of our stack, namely when it (the bottom element) is equal to the top element.

Thus, if we re-examine the generalised expression (A2.1), of the f call, we could also have replaced it by:

i_apply($N^{n+1}x$,x,invN,H,D) f($N^{n+1}x$,B,C,D,H,N) **st** invN(Nx)=x

where i_apply carries the top and bottom element of the stack and has the following definition[1].

```
---i_apply(x,y,invN,H,D)      <= If x=y then id
                                else i_apply(invN(x),y,invN,H,D) o H(D(invN(x)),?);
```

With this alternative form for f, we can define a new function, f3_it, whose existence is dependent on the presence of an inverse function, invN, as shown below.

```
---f3_it(x,y,B,C,D,H,N,invN) <= i_apply(x,y,invN,H,D) f(x,B,C,D,H,N) st invN(N(a))=a;   (A2.3)
```

This new definition can be subjected to a sequence of transformation steps, as shown in Figure A2.2, to obtain a tail-recursive program which consists of the following two functions.

```
---f3_it(x,y,B,C,D,H,N,invN) <= If B(x) then f3a_it(x,y,C(x),invN,H,D)
                               else f3_it(Nx,y,B,C,D,H,N,invN);
---f3a_it(x,y,w,invN,H,D)    <= If x=y then w
                               else f3a_it(invN(x),y,H(D(invN x),w),invN,H,D);
```

We again have a two-pass iteration but this time there is no need for the data stack. All that is needed is a first pass, by f3_it, to obtain the last recursive call's argument and a second pass, by f3a_it, to compute the results backwards. This second pass obtains its successive argument values with the help of function, invN.

The schematic rule for this transformation is shown below, together with a condition which states the need for an efficient inverse function for N.

Schematic Rule 3 for f3_it

f(x,B,C,D,H,N) = f3_it(x,x,B,C,D,H,N,invN) **st** invN(N(a))=a

This schematic rule is obtained from the original equation (A2.3) of F3_it by instantiating y=x, followed by simplification steps, as shown below.

---

[1] In actual fact, the first argument carried by i_apply is $N^{n+1}x$. This is one element above the original top element, $N^n x$, which can be obtained from $N^{n+1}x$ by applying the inverse function, invN, to it.

```
f3_it(x,x,B,C,D,H,N,invN) = i_apply(x,x,invN,H,D) f(x,B,C,D,H,N) st invN(N(a))=a    ! unfold i_apply
f3_it(x,x,B,C,D,H,N,invN) = id f(x,B,C,D,H,N) st invN(N(a))=a                      ! unfold id
f3_it(x,x,B,C,D,H,N,invN) = f(x,B,C,D,H,N) st invN(N(a))=a
```

```
- - - f3_it(x,y,B,C,D,H,N,invN) <= i_apply(x,y,invN,H,D) f(x,B,C,D,H,N);          ! unfold f
                          <= i_apply(x,y,invN,H,D) If B(x) then C(x)             ! distribute if
                                                   else H(D(x),f(N(x),B,C,D,H,N));
                          <= If B(x) then i_apply(x,y,invN,H,D) C(x)             ! abstract H(D(x),?)
                             else i_apply(x,y,invN,H,D) H(D(x),f(N(x),B,C,D,H,N));
                          <= If B(x) then i_apply(x,y,invN,H,D) C(x)             ! apply x=invN(N(x))
                             else i_apply(x,y,invN,H,D) o H(D(x),?) f(N(x),B,C,D,H,N));
                          <= If B(x) then i_apply(x,y,invN,H,D) C(x)             ! fold i_apply
                             else i_apply(invN (Nx),y,invN,H,D) o H(D(invN (Nx)),?) f(N(x),B,C,D,H,N));
                          <= If B(x) then i_apply(x,y,invN,H,D) C(x)             ! fold f3_it
                             else i_apply(N x,y,invN,H,D) f(N(x),B,C,D,H,N));
                          <= If B(x) then i_apply(x,y,invN,H,D) C(x)             ! define f3a_it
                             else f3_it(Nx,y,B,C,D,H,N,invN);
                          <= If B(x) then f3a_it(x,y,C(x),invN,H,D)
                             else f3_it(Nx,y,B,C,D,H,N,invN);
- - - f3a_it(x,y,w,invN,H,D) <= i_apply(x,y,invN,H,D) w;                          ! unfold i_apply
                          <= (If x=y then id
                             else i_apply(invN(x),y,invN,H,D) o H(D(invN x),?)) w; ! law of if
                          <= If x=y then id w                                    ! unfold id
                             else i_apply(invN(x),y,invN,H,D) o H(D(invN x),?) w; ! a o b (x)=a (b(x))
                          <= If x=y then w
                             else i_apply(invN(x),y,invN,H,D) H(D(invN x),w);    ! fold f3a_it
                          <= If x=y then w
                             else f3a_it(invN(x),y,H(D(invN x),w),invN,H,D);
```

Figure A2.2 - Transforming to a Two-Pass Iteration without Data Stack

A2.2.1. Improving the 2-Pass Iteration to a 1-Pass Iteration

Under certain circumstances, it may be possible to improve the two-pass tail-recursive program of f3_it to a single-pass tail-recursive program. In particular, notice that the purpose of the first pass is to find the argument value of the last recursive call. This can be determined, in advance, if the base case test function, B, is of the form B=**lambda** x=> x=co **end** where co is a constant. When this happens, we can simplify f3_it to a single-pass tail-recursive function, as shown below.

```
- - - f3_it(x,y,B,C,D,H,N,invN) <= If B(x) then f3a_it(x,y,C(x),invN,H,D)      ! subs. B(x)=(x=co)
                                  else f3_it(Nx,y,B,C,D,H,N,invN) st B=lambda x=> x=co end;
                               <= If x=co then f3a_it(x,y,C(x),invN,H,D)      ! subs x=co in f3a_it call
                                  else f3_it(Nx,y,B,C,D,H,N,invN) st B=lambda x=> x=co end;
                               <= If x=co then f3a_it(co,y,C(co),invN,H,D)
                                  else f3_it(Nx,y,B,C,D,H,N,invN) st B=lambda x=> x=co end;
                                                   ! outer recursion removal law (see below)
                               <= f3a_it(co,y,C(co),invN,H,D) st B=lambda x=> x=co end;
```

The removal of outer recursion requires a complex law of the following form:

$$Fn(x,N,B,C,y) = C(y) \textbf{ st } \exists i.\ i \geq 0,\ B(N^i\ x,y)=true$$

where:

$$---Fn(x,N,B,C,y) \quad <= \quad \textbf{If } B(x,y) \textbf{ then } C(y) \textbf{ else } Fn(N\ x,N,B,C,y)$$

This complex law can be proved by structural induction. Initially, from the above constraint, $\exists i.\ i \geq 0,\ B(N^i\ x,y)=true$, we can find a minimum i, say min_i, such that:

$$B(N^{min_i}\ x,y)=true \wedge \forall j.\ j<min_i.\ B(N^j\ x,y)=false$$

We could then prove by induction that:

$$\forall j.\ j \leq min_i.\ Fn(N^j\ x,N,B,C,y)=Fn(x,N,B,C,y)$$

using backwards reasoning, as follows:

Base case, j=0:

$$\begin{aligned} Fn(N^0x,N,B,C,y) &= Fn(x,N,B,C,y) \\ Fn(x,N,B,C,y) &= Fn(x,N,B,C,y) \\ &\quad true \end{aligned}$$

Recursive case, where j=j-1, we have:

$$\begin{aligned} &Fn(N^{j-1}x,N,B,C,y) = Fn(x,N,B,C,y) && \text{! unfold } Fn(N^{j-1}x,N,B,C,y) \\ &\textbf{If } B(N^{j-1}x,y) \textbf{ then } C(y) \textbf{ else } Fn(N^j\ x,N,B,C,y),C,y) && \text{! } B(N^{j-1}x,y)=false \\ &\qquad = Fn(x,N,B,C,y) \\ &\textbf{If false then } C(y) \textbf{ else } Fn(N^j\ x,N,B,C,y),C,y) && \text{! law of if} \\ &\qquad = Fn(x,N,B,C,y) \\ &Fn(N^j\ x,N,B,C,y),C,y) = Fn(x,N,B,C,y) && \text{! by induction} \\ &\qquad true \end{aligned}$$

Once this inductive proof is established, we can get the complex law by:

$$\begin{aligned} Fn(x,N,B,C,y) &= Fn(x,N,B,C,y) && \text{! by the above result} \\ &= Fn(N^{min_i}\ x,N,B,C,y) && \text{! unfold Fn} \\ &= \textbf{If } B(N^{min_i},y) \textbf{ then } C(y) && \text{! } B(N^{min_i}x,y)=true \\ &\quad \textbf{else } Fn(N^{min_i}\ x,N,B,C,y) \\ &= \textbf{If true then } C(y) && \text{! if law} \\ &\quad \textbf{else } Fn(N^{min_i}\ x,N,B,C,y) \\ &= C(y) \end{aligned}$$

This additional derivation gives us the following schematic rule.

Schematic Rule 3a for f3_it

f(x,B,C,D,H,N) = f3a_it(co,x,C(co),invN,H,D) **st** invN(N(a))=a, B=**lambda** x=> x=co **end**

It is obtained from Scheme Rule 3 by unfolding the f3_it function. Notice the presence of the additional constraint, B=**lambda** x=> x=co **end.**

A2.2.2. Avoiding the Equality Test on the Recursion Argument

The above tail recursive function f3a_it uses an equality test to check if the end of the second pass has been reached. This equality test may be a source of inefficiency, if the recursion argument used, x, is made up of a large aggregate type.

We can avoid this test by transforming f to a different tail-recursive program whose first iterative pass will obtain the maximum number of recursive calls in this linear recursion. In a way, this count corresponds to the maximum size of the previous data stack, $[N^n x,...,N^2 x,Nx,x]$, from the generalised expression of (A2.1). With this data stack size, we can ignore the bottom element and its associated equality test, and simply concentrate on maintaining the current size of the data stack to indicate when the end of the stack is reached.

Such a formulation would result in the following alternative expression:

$$n_apply(n+1,N^n x,invN,H,D)\ f(N^{n+1}x,B,C,D,H,N)\ \textbf{st}\ invN(Nx)=x \qquad (A2.4)$$

where n_apply is a function of the following definition.

```
- - -n_apply(0,x,invN,H,D)      <= id;
- - -n_apply(n+1,x,invN,H,D)    <= n_apply(n,invN(x),invN,H,D) o H(D(x),?);
```

Using this alternative expression (A2.4), we can define the following new function, f4_it, in terms of f.

```
- - - f4_it(x,n,B,C,D,H,N,invN)  <= n_apply(n,invN(x),invN,H,D) f(x,B,C,D,H,N) st invN(N(a))=a;
```

which can be transformed (as shown in Figure A2.3) to:

```
- - - f4_it(x,n,B,C,D,H,N,invN)  <= if B(x) then f4a_it(invN(x),n,C(x),D,H,invN)
                                    else f4_it(Nx,n+1,B,C,D,H,N,invN);
- - - f4a_it(x,0,w,D,H,invN)     <= w;
- - - f4a_it(x,n+1,w,D,H,invN)   <= f4a_it(invN(x),n,H(D(x),w),D,H,invN) ;
```

The second iterative pass, represented by function f4a_it, no longer uses the equality test on recursion argument. Instead, a count (which is computed by the first pass and decremented in the second pass) is used to determine when to terminate.

The schematic rule from this transformation can be represented as follows:

Schematic Rule 4 for f4_it

$$f(x,B,C,D,H,N) = f4_it(x,0,B,C,D,H,N,invN)\ \textbf{st}\ invN(N(a))=a$$

This rule is obtained by instantiating n=0 in the original equation (A2.4) of F4_it, followed by simplification steps, as shown below.

```
f4_it(x,0,B,C,D,H,N,invN      =  n_apply(0,invN(x),invN,H,D) f(x,B,C,D,H,N) st invN(N(a))=a
f4_it(x,0,B,C,D,H,N,invN)     =  id f(x,B,C,D,H,N) st invN(N(a))=a
f4_it(x,0,B,C,D,H,N,invN)     =  f(x,B,C,D,H,N) st invN(N(a))=a
```

Presently, the target program scheme is still two-pass but it could be converted to a single-pass scheme if the number of recursive calls and the result of the last recursive call could be determined in advance. This could be determined if appropriate conditions exist between B, N and C.

```
- - - f4_it(x,n,B,C,D,H,N,invN)  <= n_apply(n,invN(x),invN,H,D) f(x,B,C,D,H,N);      ! unfold f
                               <= n_apply(n,invN(x),invN,H,D) If B(x) then C(x)    ! distribute if
                                                              else H(D(x),f(N(x),B,C,D,H,N));
                               <= If B(x) then n_apply(n,invN(x),invN,H,D) C(x)    ! abstract H(D(x),?)
                                  else n_apply(n,invN(x),invN,H,D) H(D(x),f(N(x),B,C,D,H,N));
                               <= If B(x) then n_apply(n,invN(x),invN,H,D) C(x)    ! fold n_apply
                                  else n_apply(n,invN(x),invN,H,D) o H(D(x),?) f(N(x),B,C,D,H,N));
                               <= If B(x) then n_apply(n,invN(x),invN,H,D) C(x)    ! x=invN(N(x))
                                  else n_apply(n+1,x,invN,H,D)  f(N(x),B,C,D,H,N));
                               <= If B(x) then i_apply(n,invN(x),invN,H,D) C(x)    ! fold f4_it
                                  else n_apply(n+1,invN(Nx),invN,H,D)  f(Nx,B,C,D,H,N));
                               <= If B(x) then i_apply(n,invN(x),invN,H,D) C(x)    ! define f4a_it
                                  else f4_it(Nx,n+1,B,C,D,H,N,invN);
                               <= If B(x) then f4a_it(invN(x),n,C(x),D,H,invN)
                                  else f4_it(Nx,n+1,B,C,D,H,N,invN);
define  f4a_it
- - - f4a_it(x,n,w,D,H,invN)   <= n_apply(n,x,invN,H,D) w;                          ! unfold n_apply
- - - f4a_it(x,0,w,D,H,invN)   <= id w;                                             ! unfold id
                               <= w;
- - - f4a_it(x,n+1,w,D,H,invN) <= n_apply(n,invN(x),invN,H,D) o H(D(x),?) w;        ! (a o b) c=a(b(c))
                               <= n_apply(n,invN(x),invN,H,D) H(D(x),w);           ! fold f4a_it
                               <= f4a_it(invN(x),n,H(D(x),w),D,H,invN) ;
```

Figure A2.3 - Transforming to a Two-Pass Iteration without the Equality Test on Argument

A2.3. Associative Dual Scheme

Our last transformation scheme is for a sub-class of linear recursive functions which can always be transformed to an efficient single-pass tail-recursive equivalent. This sub-class of functions has the associative dual property for the outer operator, H, such that H(a,H(b,c))=H(J(a,b),c)) and J is an associative dual of H. The associative law, H(a,H(b,c))=H(H(a,b),c)), is a special case of the associative dual where J=H.

With this associative dual property, we can define a new function,

```
- - - f5_it(x,w,B,C,D,H,N,J)<=  H(w,f(x,B,C,D,H,N))  st H(a,H(b,c))=H(J(a,b),c));        (A2.5)
```

which can be transformed to the following tail-recursive equivalent.

```
- - - f5_it(x,w,B,C,D,H,N,J)<=  If B(x) then H(w,C(x))
                               else f5_it(Nx,J(w,D(x)),B,C,D,H,N,J));
```

The transformation to achieve this is shown in Figure A2.4. It uses the associative dual property to help re-order the main expression for an application folding to obtain tail-recursion (for f5_it).

```
- - - f5_it(x,w,B,C,D,H,N,J)<=  H(w,f(x,B,C,D,H,N))   st H(a,H(b,c))=H(J(a,b),c));   ! unfold f
                          <=  H(w, if B(x) then C(x)                                  ! distr. if
                                   else H(D(x),f(N(x),B,C,D,H,N)) ) st H(a,H(b,c))=H(J(a,b),c));
                          <=  if B(x) then H(w,C(x))                                  ! apply assoc. dual
                              else H(w,H(D(x),f(N(x),B,C,D,H,N)) st H(a,H(b,c))=H(J(a,b),c));
                          <=  if B(x) then H(w,C(x))                                  ! fold f5_it
                              else H(J(w,D(x)),f(N(x),B,C,D,H,N)) st H(a,H(b,c))=H(J(a,b),c));
                          <=  if B(x) then H(w,C(x))
                              else f5_it(Nx,J(w,D(x)),B,C,D,H,N,J)) st H(a,H(b,c))=H(J(a,b),c));
```

Figure A2.4 - Transforming to Tail-Recursive Program with the Associative Dual Property

The schematic rule, shown below, is taken directly from the original definition (A2.5) of f5_it.

Schematic Rule 5 for f5_it

H(w,f(x,B,C,D,H,N))=f5_it(x,w,B,C,D,H,N,J) **st** H(a,H(b,c))=H(J(a,b),c))

This rule can be applied to the RHS sub-expression, H(D(x),f(N(x),B,C,D,H,N)), of the linear recursive function, f, as long as the associative dual property is present.

Thus, the earlier definition of function f (given at the beginning of this Appendix):

```
- - - f(x,B,C,D,H,N)   <=  if B(x) then C(y)
                           else H(D(x),f(N x,B,C,D,H,N)        ! apply schematic rule 5
```

can be transformed to the following:

```
- - - f(x,B,C,D,H,N)   <=  if B(x) then C(y)
                           else f5_it(N x,D(x),B,C,D,H,N,H)
```

This last equation could also be used as our schematic rule.

Also, a slightly different rule can be provided if H has a left identity, u, such that, H(u,a)=a. If this extra condition is available, we can have the following simpler rule which gives a direct relationship between f and f5_it.

Schematic Rule 5a for f5_it

f(x,B,C,D,H,N)=f5_it(x,u,B,C,D,H,N,J) **st** H(a,H(b,c))=H(J(a,b),c)), H(u,a)=a

This is obtained from the previous schematic rule by instantiating w=u, and then simplifying the LHS with the identity law, H(u,a)=a.

A2.4. Combining the Schematic Rules into a Single Tactic

The above collection of schematic rules could be combined into a single transformation tactic. Our use of a common linear recursive function, f, is meant to make this integration much easier.

Even the special right commutative transformation scheme, presented in Chapter 2, can be based on this linear recursive function. This is done by demanding that H does not depend on its left parameter, i.e. H=**lambda** (a,b)=>H'(b) **end**, and C be a function which returns a constant, i.e. C=**lambda** a=> c **end**. Both these requirements are syntactic and can be stated through the following possible equations between f and f1_it (from Schematic Rule 1).

```
f1_it(x,c,B,H',N)    =    f(x,B,C,H,N) st C=lambda a=> c end,H=lambda (a,b)=>H'(b) end
```

or:

```
f1_it(x,c,B,H',N)    =    f(x,B,lambda a=> c end,lambda (a,b)=>H'(b) end,N)
```

Based on this common linear recursive function, f, we can devise a conversion-to-iteration tactic around the five major schematic rules. Our tactic would initially match a given function to the common function, f. After that, it will test the various syntactic and semantic conditions in order to determine the most appropriate transformation scheme to apply. A good order for testing and selecting the most appropriate scheme is outlined in Figure A2.5. The first choice rule is Schematic Rule 1, with the special right-commutative and constant-at-termination property. This rule results in an efficient tail-recursive program and is easy to check because its applicability conditions are purely syntactic. The last choice transformation is Schematic Rule 2. This rule is applicable to all linear recursive functions but it hardly results in any program improvement.

```
- - -conversion_to_iteration (g) <=                    ! g is a given function
   If g  is non linear-recursive? then exit
   else
     If g is tail recursive? then exit
     else
       If g is special right-commutative and has constant termination?
         then apply Scheme 1
       else
         If g has an associative dual for H? then
           If H has a left-identity? then apply Scheme 5a
           else apply Scheme 5
         else
           If g has an inverse function for N? then
             If recursion argument is of aggregate type? then apply Scheme 4
             else If B is a constant argument test? then apply  Scheme 3a
             else apply Scheme 3
           else apply Scheme 2 ;
```

Figure A2.5 - An Integrated Conversion-to-Iteration Tactic

A2.5. A Problem with the Lazy Semantics

One crucial problem of the conversion-to-iteration transformations is that it does not always preserve the semantics of programs in a *lazy* language. In particular, it is occasionally possible for non-termination to be introduced into an otherwise terminating lazy program.

As an example, consider the following contrived program which consists of a function flatten for flattening a list of lists; a function first to select the first element in a list; and a function inf to repeatedly return the same input list as elements of an infinite list of lists.

```
- - -main              <= first(flatten(inf(x)));
- - -first(a::_)      <= a;
- - -inf(x)           <= x::inf(x);
- - -flatten([])      <= [];
- - -flatten(a::x)    <= a<>flatten(x);
```

The function, flatten, is linear recursive and can be transformed by the associative dual scheme (Schematic Rule 5a) to the following tail-recursive program.

```
- - -flatten(x)          <= flatten_it(x,[]);
- - -flatten_it([],w)    <= w;
- - -flatten_it(a::x,w)  <= flatten_it(x,w<>a);
```

However, this tail-recursive version of flatten (using flatten_it), causes the function, main, with an expression, first(flatten(inf(x)), to become non-terminating. This is so because the tail-recursive flatten_it needs to traverse the whole of its first input list before it can produce any result. As this input list inf(x) is an infinite list, the entire expression becomes non-terminating. In contrast, the original un-transformed program does not have this problem because function flatten can be lazily evaluated to give partial results. The conversion-to-iteration tactic has thus violated the lazy semantics[2] of this program.

However, not all conversion-to-iteration transformations necessarily result in loss of lazy semantics. For example, the following two functions can be converted to their tail-recursive counterpart without violating the lazy semantics.

```
- - -sum([])         <= [];
- - -sum(a::x)       <= a+sum(x);
- - -rev([])         <= [];
- - -rev(a::x)       <= rev(x)<>[a];
```

The main reason for this is that the arguments for these two linear-recursive functions are of the *same* degree of strictness as their tail-recursive counterparts. In particular, the function sum is *completely strict*[3] in its recursive argument, while the function rev is *tail-strict*[4] in its recursive argument. Corresponding, the tail-recursive counterparts of these two functions are also of the *same* degree of strictness. This implies that lazy semantics will be preserved in the conversion-to-iteration transformations of these two functions.

More specifically, all tail-recursive functions (based on list-type recursion parameter[5]) are at least *tail-strict* in their recursive arguments. As a result, linear recursive functions must also be at least *tail-strict* if they are to be safely transformed to their tail-recursive counterparts. Functions, like sum and rev, satisfy this requirement and can therefore be safely converted to

---

[2] Apart from this loss of lazy semantics, the earlier tail-recursive version of flatten is actually less efficient than its original un-transformed function! This is caused by the inefficient accumulation of list, using the <> operator, in function flatten_it. There appears to be a possible correlation between this loss of efficiency and the loss of lazy semantics. Further investigation is, however, necessary to ascertain this.

[3] need to evaluate every element of the entire list to give a result

[4] need to evaluate only the spine of its input list to give a result

[5] could be analogously formulated for the other types.

their tail-recursive counterparts. In contrast, the function flatten is only *constructor-strict*[6] and not *tail-strict*. It cannot therefore be safely transformed by the conversion-to-iteration tactic.

Such a criterion, based on the appropriate strictness of recursion arguments, could be used to identify programs that could be safely transformed to tail-recursive form (for a lazy language). A number of static analysis techniques which are capable of analysing this strictness information have already been formulated by various researchers [Burn87a,Hughes88]. One technique which is particularly suitable for our purpose, is the strictness analysis technique for list-type arguments using a non-flat abstract domain [Wadler87b].

In [Runciman89], a related investigation was made into how certain unfold/fold operations could be safely carried out for non-strict programs. In particular, they showed that the instantiation step can alter strictness behaviour under certain circumstances. To avoid this change of behaviour, it was suggested that strictness analysis should be used to determine when it is safe to instantiate. If it is not safe to instantiate, there is still a possibility of shifting the instantiation to a safer context with the help of additional intermediate functions.

Runciman et al's investigation is interesting because it is based on a careful consideration of the elementary *instantiation* rule. Using their result, we could probably show that the deforestation and higher-order removal tactics (of Chapter 3 and 4) preserve the non-strict behaviour. This is because we always make use of safer contexts by introducing new intermediate functions before function calls with pattern-matching parameters are instantiated and unfolded.

However, the present strictness consideration for the conversion-to-iteration tactic is slightly different from Runciman's result because laws are involved in the transformation. More work is therefore needed to generalise his result to cover transformations that use laws.

---

[6] requires only a single constructor in its arguments before a result can be produced

Appendix 3. Derivation of Parallel Lemmas

An important part of any program transformation system is the *laws* or *lemmas* that are needed for simplifying and manipulating expressions to more desirable forms. We have already seen the necessity for these laws in a number of places, including the two constraint-based transformation tactics of Chapter 6.

These laws can be classified into two main categories. Firstly, there are *primitive* laws for specifying valid relationships of either language constructs or primitive functions. An example of this is the following law for the conditional construct.

```
v(if b then c else d)    =    if b then v(c) else v(d)
```

Such *primitive* laws represent useful relationships of primitive operators or language constructs, and must be available, a priori, in program transformation systems.

Secondly, there are also laws on *user-defined* functions for specifying additional relationships (other than those given in the basic definitions) which may be useful for transformations. As an example, the expression double(reverse(x)) of the following two functions:

```
- - -double([])        <= [];
- - -double(a::x)      <= 2*a::double(x);
- - -reverse([])       <= [];
- - -reverse(a::x)     <= reverse(x)<>[a];
```

requires an additional law, double(x<>y)=double(x)<>double(y) of function double, for a successful application of the loop combination transformation.

But should such laws or lemmas on user functions be provided? A simple way is to let the users supply the necessary laws together with their programs. This approach is presently used in most program transformation systems. For example, the PROSPECTRA system [KBruckner89] uses additional axioms of algebraic specifications as useful transformation laws, while Kestrel Institute's KIDS [Smith89b] requires users to supply some of the distributive and monotonic laws in order to let its inference sub-system derive (at a higher-level) more complex laws. Such an approach has the advantage of not complicating the transformation systems but we are also placing an unwitting burden on the users to come up with the necessary laws.

An alternative approach is to provide techniques which could *derive/synthesize* (probably with user help) some of these lemmas for us. In particular, it would be nice to have synthesis techniques or methods which are *sound* and will produce correct lemmas by *constructions*.

One important class of useful lemmas (which are frequently needed) is the *parallel* or *distributive* lemmas over simple composition operators, like <> (append for list), + (plus for num) and ∪ (union for set). These lemmas are typically (but not exclusively) of the form f(x<>y)=g(f(x),f(y)) for list type, f(a+b)=g(f(a),f(b)) for numeric type and f(p ∪ q)=g(f(p),f(q)) for set type. They are useful for two purposes. Firstly, they could be used as semantic laws for program transformation. Secondly, they can often act as alternative parallel forms which are suitable for

obtaining divide-and-conquer algorithms.

The original user-functions of these parallel lemmas are normally based on aggregate data types, like list, num, set and bag, whose natural views are sequential. An example of this is:

```
- - -length([])      <= 0;
- - -length(a::x)    <= 1+length(x);
```

From such sequential definitions, we are often expected to obtain the corresponding parallel lemmas, e.g. length(x<>y)=length(x)+length(y).

In this Appendix, we shall outline a *constructive* method for synthesizing such parallel lemmas directly from the user functions' sequential definitions. To illustrate this synthesis method, we shall give eight classes of programs (in schematic form) whose parallel lemmas can be discovered. Finally, we shall also touch on the scope of this method.

A3.1 Three-Stage Synthesis

Our synthesis method for parallel lemmas is based on a *three-stage* procedure. The first of these three stages is a *symbolic evaluation* stage on the given sequential definition, in order to obtain two or more instances of equations with a common parallel form. These instances of equations must then be compared and generalised, in a *matching and second-order generalisation* stage, to obtain a parallel form with unknown functional variables. Using this parallel form, the last stage of our synthesis will apply appropriate instantiations and derivation steps in order to *synthesize the unknown functional variables*. Successful applications of these three stages will produce for us the appropriate parallel lemma.

To simplify the presentation, we shall restrict our attention to synthesizing parallel lemmas for functions with only a *single* self-recursive function call. We shall not be concerned with function definitions that are mutually recursive or have multiple recursive calls in their definitions.

An example, which we would be considering, is the following definition for sum with one recursive call in its second equation.

```
- - -sum([])      <= 0;
- - -sum(a::x)    <= a + sum(x);
```

We shall use this example to illustrate the three stages of our synthesis method.

A3.1.1 Stage 1: Symbolic Evaluation to obtain Comparable Equations.

The main purpose of the first symbolic evaluation stage is to obtain *instances* of an appropriate parallel form directly from the given sequential definition. To do this, we must apply symbolic evaluation to obtain two *comparable* recursive equations from the original recursive equation.

But what is meant by two comparable recursive equations? Two recursive equations are said to be *comparable* if their RHSs each have an identical *skeletal expression* that leads to the recursive function call. For example, the recursive equation of sum,

```
- - -sum(a::x)        <= a + sum(x);
```

has a skeletal expression, _+sum(_), which leads to the recursive call of sum. The symbol, _, is used to represent an arbitrary sub-expression (that is of no immediate concern at this stage).

A second recursive equation, which has an identical skeletal expression to the above recursive equation of sum, can be obtained by the following symbolic evaluation.

```
- - -sum(a::x)        <= a + sum(x);              ! instantiate x=b::x'
- - -sum(a::b::x')    <= a + sum(b::x');          ! unfold sum(b::x')
                     <= a + (b+sum(x'));         ! a+(b+c)=(a+b)+c
                     <= (a + b)+sum(x');
```

This second equation is obtained (from the first equation) by narrowing the recursive call, sum(x), followed by an application of the associative law of +. It is said to be *comparable* to the first equation because it has the *same* skeletal expression[1], _+sum(_).

A3.1.2 Stage 2: Matching and Second-Order Generalisation.

These comparable equations (from the first-stage) are actually instances (or symbolic traces) of a possible parallel form. To obtain this parallel form, we must subject the two comparable equations to a matching and generalisation process.

Our earlier two comparable equations can be re-written as follows,

```
- - -sum([a]<>x)     <= a +sum(x);
- - -sum([a,b]<>x')  <= (a + b)+sum(x');
```

with the LHS altered to take the simple composition operator, <>. These two equations can be compared by matching, with generalisations applied to the places where they mismatch, in order to obtain their common parallel form.

The LHS of both equations are compared to give sum(m<>x), where m is a generalised variable for both [a] and [a,b]. This is obtained by first-order generalisation. Similarly, the RHS of the two equations are also compared (taking into account the matching which has just taken place on the LHS) but this time we need to apply second-order generalisation[2]. In this case, a mismatch occurs between a and a+b, which can only be resolved if we supply a generalised expression, H(m), where H is a functional variable and m is the generalised (or mismatch) variable from the LHS. This comparison gives H(m)+sum(x) as the generalised RHS.

Combining both these generalised LHS and RHS, we obtain the following valid parallel form for the function sum,

```
sum(m<>x)        =  H(m)+sum(x)                                    (A3.1)
```

---

[1] The idea of using skeletal expression is similar to the first-order generalisation technique used by Arsac and Kodratoff [Arsac82] for recursion removal. Their technique identifies a (most general) set of embedded functions (leading to the recursive function call) which could be used as a generalised expression for new function definitions. The recursive call in this generalised expression is then unfolded, followed by manipulations to obtain a matching form - leading to a tail-recursive transformation. Instead of tail-recursion, we are using this technique for finding parallel lemmas where second-order generalisation is required (see later).

[2] First-order generalisation introduces object-type variables, while second-order generalisation introduces both object and function-type variables.

where H is an unknown functional variable.

A3.1.3 Stage 3: Synthesis of Unknown Functions.

The third stage of our method involves both *instantiation* and *derivation* steps to synthesize out the unknown functional variables in the parallel form. Where possible, we will try to derive each of the unknown functions in terms of the original known function.

Let us look at one way of synthesizing the unknown function, H, in the parallel form of sum. To obtain a definition for H(m) in (A3.1), we will need to instantiate its dependent variable, m, to two cases, [] and a::m. This is followed by a sequence of derivation steps in each case, as shown below.

Case m=[], we have:

```
sum([]<>x)      =  H([]) + sum(x)
LHS             =  sum([]<>x)          ! unfold []<>x
                =  sum(x)
RHS             =  H([]) + sum(x)
∴ H([])         =  0
```

Case m=a::m, we have:

```
sum((a::m)<>x)  =  H(a::m)+sum(x)
LHS             =  sum((a::m)<>x)      ! unfold <>
                =  sum(a::(m<>x))      ! unfold sum(a::m)
                =  a+sum(m<>x)         ! inductive step of (A3.1)
                =  a+(H(m) + sum(x))   ! associative law (a+(b+c))=(a+b)+c
                =  (a+H(m)) + sum(x)
RHS             =  H(a::m) + sum(x)
∴ H(a::m)       =  a+H(m)
```

The result of the above synthesis is the definition, H([])=0; H(a::m)=a+H(m), for the unknown function, H. This definition of H is actually syntactically equivalent to the definition for function sum. As a result, the new parallel lemma for function sum is:

```
sum(m<>x)       =  sum(m) + sum(x)
```

Note that this synthesis is normally a direct consequence of the first symbolic evaluation stage. In particular, the laws and unfolding steps used in the first stage are often mirrored in this stage.

The above complete instantiations of the dependent variable, m, is one way of synthesizing the unknown function, H. However, we could also use a different way of synthesizing the unknown function. This technique requires fewer derivation steps and can be used if there is a direct relationship present between the unknown function, H, and known function, sum. It works by instantiating the dependent variable, x, of the known function, sum, to its base case, []. This is followed by a sequence of simplification steps, as shown below.

```
sum(m<>[])      =  H(m) + sum([])
LHS             =  sum(m<>[])          ! m<>[]=m
                =  sum(m)
RHS             =  H(m) + sum([])      ! unfold sum([])
                =  H(m) + 0            ! a+0=a
                =  H(m)
∴ H             =  sum
```

The result of this equational reasoning tells us that H is equal to sum. This method is shorter and simpler to implement. However, it is only feasible if there is a possibility of expressing the unknown function in terms of the known function. This may not always be possible, and we may have to rely on the earlier complete instantiations technique for some of our examples.

A3.1.4 Embedding Synthesis within the Unfold/Fold Framework

The above two derivations, to obtain the value of the unknown function, are based on *equational reasoning* which involves program manipulation in both the LHS and RHS of equations. To perform it under the unfold/fold framework, we must again embed the necessary reasoning in newly defined functions, like what we did for inductive proofs in Chapter 2.

For example, we could embed the equation of (A3.1) into another newly defined function, as follows:

```
- - -synthesize1(m,x,H)   <=  sum(m<>x)=H(m) + sum(x);
```

Direct derivation of H could then be made by instantiating x to [], followed by some simplification steps. The result of this synthesis, shown below, indicates that H is equal to sum.

```
- - -synthesize1(m,[],H)  <=  sum(m<>[])=H(m) + sum([]);   ! m<>[]=m, unfold sum([])
                         <=  sum(m)=H(m) +0;              ! a+0=a
                         <=  sum(m)=H(m);                 ! law (v1(m)=v2(m)) = (v1=v2))
                         <=  sum=H;
```

Alternatively, we could also obtain the definition for H via the complete instantiations of parameter, m. This would similarly have resulted in a definition for H that is identical to sum.

The main attraction for embedding this synthesis within the unfold/fold framework is that it allows us to perform both synthesis and transformations under a single framework.

A3.2 Classes of Programs which are Synthesizable

The three-stage synthesis method just outlined is a very general technique for deriving parallel lemmas. However, it depends crucially on special properties (e.g. associative law) that are needed to help manipulate the recursive equations into common parallel forms.

To illustrate specifically what these properties are and at the same time show the range of applicability of our synthesis, we shall identify eight classes of programs whose parallel lemmas can be synthesized by this method. These classes of programs will be represented by program schemes (i.e. higher-order functions). Such a list of schemes may serve as a knowledge base for guiding our synthesis method.

A3.2.1 Scheme 1- simple homomorphic form

The sum function (given earlier) is an instance of the following program scheme.

Program Scheme 1

```
- - -F1([],g,f,u)      <= u;
- - -F1(a::x,g,f,u)    <= g(f(a),F1(x,g,f,u));
```

WHERE g is associative, i.e. g(a,g(b,c))=g(g(a,b),c), and u is the identity of g, i.e. g(a,u)=a ∧ g(u,a)=a .

Using similar derivation steps as the sum function, we could synthesize the following parallel lemma for the above scheme.

```
F1(m<>x,g,f,u)      = g(F1(m,g,f,u),F1(x,g,f,u))
```

This program scheme, with its parallel lemma, is also known as a *homomorphism* (homomorphic with respect to <> operator) in [Bird87a]. In his paper, Richard Bird showed that such a function is strongly equivalent to a composition of map and reduce, as shown below,

```
F1(x,g,f,u)         = reduce(g,u,map(f,x))
```

where g is associative and u is the identity of g.

This scheme represents one of simplest classes of sequential programs which has a direct parallel lemma. We shall also see other classes of programs with parallel lemmas (many of whom are variants of this basic scheme) in the next few sections.

A3.2.2 Scheme 2 - undefined for the nil or null case

Occasionally, it may be possible to have functions that are *undefined* in the nil or null cases. An example of this is the following function, min, for finding the minimum element in its input list.

```
- - -min([a])        <= a;
- - -min(a::x)       <= min2(a,min(x));
- - -min2(a,b)       <= If a>b then b
                        else a;
```

Notice that the base case equation of this function is defined on a singleton element rather than on the nil case. As a result, it is not considered to be an instance of the first program scheme. Instead, we have to consider it under a variant scheme, as shown below.

Program Scheme 2

```
- - -F2([a],g,f)     <= f(a);
- - -F2(a::x,g,f)    <= g(f(a),F1(x,g,f));
```

WHERE g is associative, i.e. g(a,g(b,c))=g(g(a,b),c).

In [Bird87a], a technique for handling this class of variant functions was suggested. His technique assumes the existence of a *fictitious* identity, say z, for the outer associative operator, g. In the case of the min function, the fictitious identity is the infinite number, ∞.

With this assumption, the second program scheme now resembles the first program scheme because the null case equation, F2([],g,f)=z, is assumed to exist. Hence, a parallel lemma which is identical to the first program scheme can also be obtained.

However, it is also possible to synthesize the same parallel lemma without resorting to the use of fictitious identity. We simply need to perform a synthesis which uses singleton list instantiation rather than nil and recursive case instantiations. We illustrate this synthesis by using the following possible parallel form (obtained from the first two stages of our synthesis method) with an unknown H, as shown below.

```
F2(m<>x,g,f)      =  g(H(m,g,f),F2(x,g,f))
```

With this parallel form, we will initially apply the instantiation, x=[a], as shown below,

```
F2(m<>[a],g,f)    =  g(H(m,g,f),F2([a],g,f))      ! unfold F2([a],g,f)
                  =  g(H(m,g,f),f(a))
```

This gives us:

```
F2(m<>[a],g,f)    =  g(H(m,g,f),f(a))                                    (A3.2)
```

which is unfortunately not a definition for H yet. To obtain a definition of H, we have to perform a further derivation sequence, using another instantiation, x=[b]<>[a], as follows.

```
F2(m<>[b]<>[a],g,f)  =  g(H(m,g,f),F2([b]<>[a],g,f))
RHS                  =  g(H(m,g,f),F2([b]<>[a],g,f))    ! unfold F2
                     =  g(H(m,g,f),g(f(b),F2([a],g,f))) ! unfold F2
                     =  g(H(m,g,f),g(f(b),f(a)))        ! associative law of g
                     =  g(g(H(m,g,f),f(b)),f(a))        ! apply (A3.2) in reverse
                     =  g(F2(m<>[b],g,f),f(a))
LHS                  =  F2(m<>[b]<>[a],g,f)             ! apply (A3.2)
                     =  g(H(m<>[b],g,f),f(a))
∴ H(m<>[b],g,f)      =  F2(m<>[b],g,f)
  H                  =  F2
```

Such a derivation sequence gives the same result as Bird's use of fictitious identity because the new parallel lemma:

```
F2(m<>x,g,f)      =  g(F2(m,g,f),F2(x,g,f))
```

is identical to the previous parallel lemma for F1.

A3.2.3 Scheme 3 - without the identity element

Another possible variant of the first program scheme, F1, is when u is *not* an identity of g. In this case, we need the presence of an extra semantic condition before we are able to synthesize the unknown function in terms of the known function. Specifically, there must a left-inverse of g, say g^{-1}, such that:

$$g^{-1}(g(a,b),b) = a.$$

Some examples of such a left-inverse are minus, -, for plus, +, such that -(+(a,b),b)=a (in pre-fix notation), and divide, /, for multiply, *, such that /(*(a,b),b)=a.

A program scheme for this variant can be specified as follows.

Program Scheme 3

```
- - -F3([],g,f,u)      <= u;
- - -F3(a::x,g,f,u)    <= g(f(a),F3(x,g,f,u));
```

WHERE g is associative, i.e. g(a,g(b,c))=g(g(a,b),c), and there exist a left-inverse, g^{-1}, such that, $g^{-1}(g(a,b),b) = a$.

To derive the corresponding parallel form for this scheme, we must initially obtain the following possible lemma with an unknown function, H.

```
F3(m<>x,g,f,u)    =  g(H(m,g,f,u),F3(x,g,f,u))                    (A3.3)
```

After that, we can apply the instantiation x=[], in order to perform the following derivation sequence.

```
F3(m<>[],g,f,u)   =  g(H(m,g,f,u),F3([],g,f,u))
LHS               =  F3(m<>[],g,f,u)          !  m<>[]=[]
                  =  F3(m,g,f,u)
RHS               =  g(H(m,g,f,u),F3([],g,f,u))   !  F3([],g,f,u)
                  =  g(H(m,g,f,u),u))             !  F3([],g,f,u)
```

Applying $g^{-1}(?,u)$ to both the LHS and RHS, we have

```
g^-1(g(H(m,g,f,u),u)),u) =  g^-1(F3(m,g,f,u),u)   !  simplify LHS using g^-1(g(a,b),b) = a
H(m,g,f,u)               =  g^-1(F3(m,g,f,u),u)
```

The last equality gives a definition for H, in terms of F3. This result can be substituted back into (A3.3) to obtain the following parallel lemma for the program scheme, F3.

```
F3(m<>x,g,f,u)    =  g(g^-1(F3(m,g,f,u),u),F3(x,g,f,u))
```

A3.2.4 Scheme 4 - with accumulating parameter

Our fourth program scheme is meant to cover programs with additional *accumulating parameters*. This is not provided by the earlier three program schemes, where only one main list-type recursion argument and some fixed parameters are present.

An example of this new class of programs is the following function, label, with an accumulating parameter, no, for marking a list of items with consecutive numbers.

```
dec label : list(alpha) # num -> list(alpha # num);
- - -label([],no)      <= [];
- - -label(a::x,no)    <= [(no,a)]<>label(x,no+1);
```

The associativity of the outer <> operator, in the recursive equation, enables us to obtain two comparable equations with the skeletal expression, _<>label(_,_). Second-order generalisation should produce the following parallel form with two unknown variables, H and Aux.

```
label(m<>x,no)    =  H(m,no)<>label(x,Aux(m,no))
```

The second unknown function, Aux, is caused by the presence of the accumulating parameter. This unknown function must also be derived through appropriate synthesis.

Initially, the first unknown function, H, is obtained by instantiating x=[], together with the following derivation steps.

```
label(m<>[],no)     =  H(m,no)<>label([],Aux(m,no))
LHS                 =  label(m<>[],no)              ! apply law, a<>[]=a
                    =  label(m,no)
RHS                 =  H(m,no)<>label([],Aux(m,no))  ! unfold label([],a)
                    =  H(m,no)<>[]                  ! apply law, a<>[]=a
                    =  H(m,no)
∴ H                 =  label
```

This derivation tells us that H is equivalent to label but is unable to provide a definition for Aux. This is because the auxiliary function, Aux, is not directly related to the known function, label, and cannot therefore be obtained by this instantiation. Instead, we must apply complete instantiations of the dependent parameter, m, of Aux in order to synthesize a fresh set of equations for Aux.

Starting from the following partial definition of our parallel lemma,

```
label(m<>x,no)      =  label(m,no)<>label(x,Aux(m,no))                    (A3.4)
```

we could initially apply the instantiations m=[], to obtain a base case equation for Aux.

```
label([]<>x,no)     =  label([],no)<>label(x,Aux([],no))
LHS                 =  label([]<>x,no)              ! unfold []<>x
                    =  label(x,no)
RHS                 =  label([],no)<>label(x,Aux([],no))  ! unfold label([],no)
                    =  []<>label(x,Aux([],no))      ! unfold []<>x
                    =  label(x,Aux([],no))
∴ Aux([],no)        =  no
```

This is followed by the recursive case instantiation, m=a::m, in order to obtain a recursive equation for Aux.

```
label(a::m<>x,no)   =  label(a::m,no)<>label(x,Aux(a::m,no))
LHS                 =  label(a::m<>x,no)                          ! unfold label
                    =  [(no,a)]<>label(m<>x,no+1)                 ! apply (A3.4)
                    =  [(no,a)]<>(label(m,no+1)<>label(x,Aux(m,no+1)))  ! apply associative law of <>
                    =  ([(no,a)]<>label(m,no+1))<>label(x,Aux(m,no+1))
RHS                 =  label(a::m,no)<>label(x,Aux(a::m,no))      ! unfold label
                    =  ([(no,a)]<>label(m,no+1))<>label(x,Aux(a::m,no))
∴ Aux(a::m,no)      =  Aux(m,1+no)
```

The synthesized parallel lemma, together with its auxiliary function, is therefore as follows:

```
label(m<>x,no)      =  label(m,no)<>label(x,Aux(m,no))
- - -Aux([],no)     <=  no;
- - -Aux(a::m,no)   <=  Aux(m,1+no);
```

This example can be more generally formulated under the following program scheme.

Program Scheme 4

```
- - -F4([],w,g,f,j,u)     <=  u;
- - -F4(a::x,w,g,f,j,u)   <=  g(f(a,w),F4(x,j(a,w),g,f,j,u));
```

WHERE g is associative, i.e. g(a,g(b,c))=g(g(a,b),c), and u is the identity of g, i.e. g(a,u)=a ∧ g(u,a)=a.

The accumulating parameter in this program scheme is w. With an associative outer operator, g,

and the identity element, u, we can derive (in a similar manner to label) the following parallel lemma,

```
F4(m<>x,w,g,f,j,u)   =   g(F4(m,w,g,f,j,u),F4(x,Aux(m,w),g,f,j,u))
```

where Aux is defined by:

```
- - -Aux([],w)        <=  w;
- - -Aux(a::m,w)      <=  Aux(m,j(a,w));
```

A3.2.5 Scheme 5 - requiring generalised programs

Certain sequential functions may not have direct parallel lemmas for their (present) definitions. Instead, it may be necessary to obtain generalised versions of these sequential programs before proper parallel lemmas are possible. One class of programs with this characteristic is represented by the following program scheme.

Program Scheme 5

```
- - -F5([],g,f,u)      <=  u;
- - -F5(a::x,g,f,u)    <=  g(f(a::x),F5(x,g,f,u));
```

WHERE g is associative, i.e. g(a,g(b,c))=g(g(a,b),c), and u is the identity of g, i.e. g(a,u)=a ∧ g(u,a)=a .

The primary difference between this program scheme and the first scheme, F1, is that this scheme's function argument, f, is applied to a::x rather than a (in the recursive equation). Our synthesis method could obtain the following parallel form with an unknown function, H.

```
F5(m<>x,g,f,u)       =   g(H(m,x,g,f,u),F5(x,g,f,u))
```

Instantiating x=[] for this equation will produce the following relationship between the unknown function, H, and the original function, F5.

```
F5(m,g,f,u)          =   H([],m,g,f,u)
```

However, this is not a proper definition for H. The proper definition for H can only be obtained by the complete instantiations of m, followed by the appropriate derivation steps. This would result in the following two equations for H.

```
- - -H([],y,g,f,u)      <=  u;
- - -H(a::x,y,g,f,u)    <=  g(f(a::x<>y),H(x,y,g,f,u));
```

Such a definition of H cannot be expressed in terms of F5. As a result, we have actually failed in the attempt to obtain a parallel lemma for F5. This is because the lemma we get is not a genuine parallel form, as H is currently defined sequentially.

Nevertheless, the definition of H is actually a generalised version of function F5. We could use it to try and derive another parallel lemma. Applying the three-stage synthesis method to this generalised function, H, we can obtain the following genuine parallel form.

```
H(m<>x,y,g,f,u)      =   g(H(m,x<>y,g,f,u),H(x,y,g,f,u))
```

This technique of applying the synthesis method a second time, after the first attempt have

failed, is an example of *productive use of failure*[3]. We have made productive use of the generalised program, from the failed first attempt, to try and obtain a genuine parallel lemma in the second attempt.

So far, all the five program schemes presented, are based on the list data type as the main recursion argument. We could also have formulated corresponding program schemes for the numeric, set or bag data types. To illustrate this, we will use a numeric-based example that is similar to the fifth program scheme, F5. This example is a sequential function for calculating factorials, as shown below.

```
- - -fact(zero)       <= 1;
- - -fact(succ(n))    <= succ(n)*fact(n);
```

The synthesis method must initially analyse two comparable equations, in order to obtain the following parallel form with an unknown function, FG.

```
fact(m+n)            =  FG(m,n)*fact(n)                    (A3.5)
```

This unknown function is a generalised definition of fact because instantiating n=zero, followed by some simplification steps, gives:

```
fact(m)              =  FG(m,zero)
```

The proper definition for FG can only be obtained by the complete instantiations of m to zero and succ(m), followed by a further derivation sequence. This would result in the following two equations for FG.

```
- - -FG(zero,n)      <= 1;
- - -FG(succ(m),n)   <= (succ(m)+n)*FG(m,n);
```

A lemma, in terms of f and FG, is now available for fact but this is not a genuine parallel lemma yet. To obtain a proper parallel form, we must apply the same synthesis method to the generalised function, FG. This final application will give:

```
FG(p+m,n)            =  FG(p,m+n)*FG(m,n)
```

as the parallel lemma for FG. With it, we can base the parallel lemma for function, fact, on the generalised function, FG, by replacing fact(n) in (A3.5) with FG(n,0), as shown below.

```
fact(m+n)            =  FG(m,n)*FG(n,0)
```

This last equation (based on the generalised function FG) could be used as a basis for the divide-and-conquer algorithm for the factorial function!

A3.2.6 Scheme 6 - deeper pattern-matching

There is yet another variant of the first program scheme, which uses deeper pattern matching mechanism in its recursive equation. An example of this variant is the following function for testing to see if a list of nodes belongs to a path of a given directed graph. Associated with the given graph is a test function, arc, which determines if any two nodes is an arc of the graph. Notice the deeper pattern, a::b::x, in the third equation of path.

---

[3] Boyer and Moore's theorem proving system is described in [Bundy83] as capable of making productive use of failure. In Boyer-Moore's system, a proof attempt which fails may make use of information from the failure to obtain a stronger (more general) theorem which may succeed.

```
dec path: list(nodes) -> truval;
- - -path([])          <= true;
- - -path([a])        <= true;
- - -path(a::b::x)    <= arc(a,b) ∧ path(b::x);
```

Comparable equations can be generated from the above function in order to obtain the following parallel form with an unknown function, H.

```
path(m<>[b]<>x)   =   H(m<>[b]) ∧ path(b::x)
```

Instantiating x=[], followed by a sequence of derivation steps could derive a definition for H, as shown below.

```
path(m<>[b]<>[])   =   H(m<>[b]) ∧ path(b::[])
LHS                =   path(m<>[b]<>[])          ! a<>[]=a
                   =   path(m<>[b])
RHS                =   H(m<>[b]) ∧ path(b::[])   ! unfold path
                   =   H(m<>[b]) ∧ true          ! a ∧ true = a
                   =   H(m<>[b])
∴ H                =   path
```

This derivation gives us the following parallel lemma.

```
path(m<>[b]<>x)   =   path(m<>[b]) ∧ path(b::x)
```

Because of the deeper pattern matching, the final parallel lemma has an intermediate element, b, which is shared by the two recursive calls of path. This path function is an instance of the following program scheme.

Program Scheme 6

```
- - -F6([],g,f,u)          <= u;
- - -F6([a],g,f,u)         <= u;
- - -F6(a::b::x,g,f,u)     <= g(f(a,b),F6(b::x,g,f,u));
```

WHERE g is associative, i.e. g(a,g(b,c))=g(g(a,b),c), and u is the identity of g, i.e. g(a,u)=a ∧ g(u,a)=a.

Our synthesis method could obtain the following parallel lemma for this scheme.

```
F6(m<>[b]<>x,g,f,u)   <=   g(F6(m<>[b],g,f,u),F6(b::x,g,f,u))
```

The pattern matching, a::b::x, used in the above scheme is presently two constructors deep. Corresponding program schemes could also be devised for even deeper patterns.

A3.2.7 Scheme 7 - tail recursive functions

So far, each of the presented program schemes requires the *associative* property for its outer operator (denoted by the g parameter) to obtain parallel lemma. However, other syntactic or semantic conditions may also be sufficient before parallel lemmas can be synthesized.

One sufficient syntactic condition is the presence of *tail recursion.* However, the parallel lemma for this class of tail-recursive functions is not *strictly* parallel in form because its two recursive calls (in the lemma) will be *nested.* Nevertheless, they are still useful as laws for program transformations.

This class of functions is covered by the following program scheme.

Program Scheme 7

```
- - -F7([],w,f)        <= w;
- - -F7(a::x,w,f)      <= F7(x,f(a,w),f);
```

The generation of two comparable equations, followed by its matching and generalisation, will yield for us the following parallel form with an unknown function, H.

```
F7(m<>x,w,f)          = F7(x,H(m,w,f),f)
```

This unknown function can be found to be equivalent to F7, by instantiating x=[], followed by the derivation steps below.

```
F7(m<>[],w,f)         = F7([],H(f,m,w),f)
LHS                   = F7(m<>[],w,f)        ! m<>[]=m
                      = F7(m,w,f)
RHS                   = F7([],H(m,w,f),f)    ! unfold F7
                      = H(m,w,f)             ! unfold F7
∴ H                   = F7
```

The parallel lemma from this synthesis is:

```
F7(m<>x,w,f)          = F7(x,F7(m,w,f),f)
```

Notice that the two F7 function calls are currently *nested*. Such a lemma is not suitable as a parallel algorithm but is still useful as a law for program transformation.

A3.2.8 Scheme 8 - more complex semantic conditions

Our last example is a program which requires more complex semantic conditions for synthesizing its parallel lemma. This more complex example will hopefully illustrate the point that our synthesis method is quite general and is mainly dependent on the possibility of identifying comparable equations.

Consider the following function, comp, for computing a rather complicated value from a list of numbers.

```
- - -comp([])          <= 0;
- - -comp((a,b)::x)    <= a+(b*comp(x));
```

Using the following associative and distributive laws of + and *,

```
a+(b+c)               = (a+b)+c
a*(b*c)               = (a*b)*c
a*(b+c)               = (a*b)+(b*c)
```

we can obtain a second comparable equation which has an identical skeletal expression, _+(_*comp(_)), to the original recursive equation, as shown below.

```
- - -comp((a,b)::(c,d)x) <= a+(b*(c+(d*comp(x))));     ! distributive law b*(c+d)=b*c+b*d
                        <= a+(b*c+b*(d*comp(x)));     ! associative law of +
                        <= (a+(b*c))+b*(d*comp(x));   ! associative law of *
                        <= (a+(b*c))+(b*d)*comp(x));
```

These two comparable equations can be used to obtain the following parallel form with two unknown functions, H and G.

```
comp(m<>x)      =  H(m)+G(m)*comp(x)
```

By instantiating x=[], we can deduce that the unknown function, H, is in fact equivalent to comp, as follows:

```
comp(m<>[])     =  H(m)+G(m)*comp([])
LHS             =  comp(m<>[])              ! law m<>[]=[]
                =  comp(m)
RHS             =  H(m)+G(m)*comp([])       ! unfold comp([])
                =  H(m)+G(m)*0              ! x*0=0
                =  H(m)+0                   ! x+0=x
                =  H(m)
∴ H             =  comp
```

The other unknown function, G, must also be synthesized. This can only be obtained by applying the complete instantiations of m, followed by the appropriate derivation steps. Thus, using the following partially known parallel lemma,

```
comp(m<>x)      =  comp(m)+G(m)*comp(x)                                  (A3.6)
```

we could perform two sequences of derivation steps in order to derive a definition for this unknown function, G, as follows.

Case m=[],

```
comp([]<>x)     =  comp([])+G([])*comp(x)
LHS             =  comp([]<>x)              ! unfold []<>x
                =  comp(x)
RHS             =  comp([])+G([])*comp(x)   ! unfold comp([])
                =  0+G([])*comp(x)          ! 0+b=b
                =  G([])*comp(x)
∴ G([])         =  1
```

Case m=a::m,

```
comp((a,b)::m<>x)  =  comp((a,b)::m)+G((a,b)::m)*comp(x)
LHS                =  comp((a,b)::m<>x)                  ! unfold comp
                   =  a+(b*comp(m<>x))                   ! apply parallel lemma (A3.6)
                   =  a+(b*(comp(m)+G(m)*comp(x)))       ! distribute b*(c+d)=b*c+b*d
                   =  a+(b*comp(m)+b*(G(m)*comp(x)))     ! associative law of +
                   =  (a+b*comp(m))+b*(G(m)*comp(x))     ! associative law of *
                   =  (a+b*comp(m))+(b*G(m))*comp(x)
RHS                =  comp((a,b)::m)+G((a,b)::m)*comp(x) ! unfold comp((a::b)::m)
                   =  (a+b*comp(m))+G((a,b)::m)*comp(x)
∴ G((a,b)::m)      =  b*G(m)
```

With this synthesis, the parallel lemma for comp, together with a definition for its auxiliary function G, is:

```
comp(m<>x)      =  comp(m)+G(m)*comp(x)
---G([])        <=  1;
---G((a,b)::m)  <=  b*G(m);
```

A corresponding program scheme, with the above semantic conditions, can also be provided.

A3.3 Scope and Limitations of our Synthesis Method

The list of program classes presented in this appendix is fairly extensive but not complete. In particular, we have not even considered the various combinations of the different factors present in the above list of schemes. For example, we may have programs with deeper pattern matching of Scheme 6, combined with say, accumulating parameters of Scheme 4.

However, each of the parallel lemmas (for the program schemes) has been synthesized using the *same* three-stage method. This is a testament to the generality of this method.

Though general, our synthesis method has its own limitations, as well. One of the main difficulties is the symbolic evaluation stage itself (for obtaining two or more comparable equations). This stage is quite difficult because it requires the presence of certain syntactic and semantic conditions (in particular the associative property of the outer operator) before comparable equations and parallel form can be obtained.

Another difficulty, which we may encounter, is that certain sub-expressions (or portions) within our programs may sometimes not *directly* have the required semantic properties. Instead, it may only be possible for the generalised versions of these sub-expressions to have the necessary property. When this happens, our synthesis method will fail. An example of this failure is the following insertion sort function.

```
- - -sort([])        <= [];
- - -sort(a::x)      <= insert(a,sort(x));
- - -insert(a,[])    <= [a];
- - -insert(a,b::x)  <= if a>b then b::insert(a,x)
                       else a::b::x;
```

The outer operator, insert, in the recursive equation of sort does not presently have the necessary associative property for synthesizing a parallel form. As a result, we are unable to synthesize a parallel lemma directly from this sequential function.

Instead, there is actually a more general merge operator (with insert as a special case), which will have the associative property.

```
- - -merge(x,[])      <= x;
- - -merge([],y)      <= y;
- - -merge(a::x,b::y) <= if a>b then b::merge(a::x,y)
                        else a::merge(x,b::y);
```

The use of this merge operator, in place of insert, would have allowed our synthesis method to derive a parallel lemma for the above sort function. The open question here is how to obtain such generalised versions for our sub-expressions. We presently offer no answers to this question.

Notice that this generalisation is not the same as that used in program scheme, F5. In the case of F5, we synthesized a generalised program for the scheme itself, while the sorting example requires a generalised version for one of its sub-expression, i.e. the insert function.

Nevertheless, it is more common for program transformation techniques to derive these more complex parallel algorithms directly from high-level generate-and-test specifications, rather than from the sequential programs. An example of this is the semi-automatic synthesis technique of Douglas Smith (in [Smith85,Smith87]) which uses various design strategies to obtain different forms of divide-and-conquer algorithms from (certain) high-level generate-and-test specifications. Successful derivations of these parallel algorithms could then be used to obtain the corresponding sequential definitions.

In contrast, our synthesis method is not suitable for complex functions, like sorting. It is primary intended for *simpler* functions that are more naturally specified in their sequential definitions (from which we could derive their parallel forms). In this respect, our method can be described as being complementary to the synthesis technique of Douglas Smith.

Appendix 4. Implementation of Tactics

This appendix briefly describes the implementation of some of the transformation tactics presented in this thesis. The implementation is carried out on the IC Transformation-Based Programming Environment [Darlington89b] where each tactic is implemented as a HOPE meta-program of the form:

```
dec tactic: IntHE # add_info -> script;
```

where IntHE is an abstract data type for HOPE expressions and add_info is the additional information supplied from analysis or given by users. Each of these meta-programs must return a transformation script that is supposed to be applicable to the given HOPE expression (or equation).

This appendix presents a top-down description of the implemented tactics. We begin by showing a sample session of how the tactics are used to transform programs. This is followed by brief descriptions of the various implemented tactics. Lastly, we present two principal building blocks for implementing the tactics, namely an intermediate *symbolic tree* data structure (plus its associated support functions) to represent the transformational development, and the low-level *access functions* for performing program manipulations.

A4.1. Overview of the Partial Evaluator and Tactics.

At the heart of the IC Transformation-Based Programming Environment is a transformation processor, called the *partial* (or symbolic) *evaluator* [Pull88]. To invoke this evaluator, there is a command (or function) of the form:

```
dec apply_script : script # program  -> program;
```

which takes a transformation script and a program (in effect a module of functions) in order to return another program (or module).

In each of these programs (modules), there may be a number of functions which could be transformed. To transform these individual functions, the outermost script must be specified as:

```
choose_eqns( list(fname # script) )
```

which takes a list of function names and their corresponding scripts, in order to apply each script to its corresponding function.

This simultaneous application of scripts to individual functions (of a module) is available for general use. However, for our purpose, we shall assume that only one function from a module will be dealt with at a time. This assumption can be handled by a function, called app_tactics, with the following definition,

```
dec app_tactics: (modulename # fname) # (IntHE -> script) -> program;
dec get_one_eqn: (modulename # fname) -> IntHE ;
- - -app_tactics((mn,fid),tactic) <=  apply_script(choose_eqns [ (fid, tactic(get_one_eqn(mn,fid))) ], mn);
- - -get_one_eqn((mn,fid))       <=  let [eqn]==get_eqns(mk_fid(mn,fid)) in eqn;
WHERE  get_eqns -   an access function to retrieve equations from the program repository
       mk_fid   -   converts a module and a function name into a function identifier
```

which takes a module name, mn, and its function name, fid, together with a tactic of the type IntHE -> script, in order to generate a new module where the specified function has been transformed by the tactic. Notice that the given tactic is invoked to generate a transformation script which is then applied (to the specified function) through apply_script.

Let us briefly illustrate the application of a simple tactic, called loop_combine, which has the following declaration.

```
dec loop_combine : IntHE -> script;
```

This tactic is a simplified form of fusion transformation (i.e. deforestation). It applies to expressions where the composed functions are self-recursive.

Consider a program/module, ex1, of the following definition.

```
module ex1;
 dec sum,sumdb : list(num) -> num;
 dec double : list(num) -> list(num);
- - -sumdb(x)          <= sum(double(x));
- - -sum([])          <= 0;
- - -sum(a::x)        <= a+sum(x);
- - -double([])       <= [];
- - -double(a::x)     <= 2*a::double(x);
end ex1;
```

A function of this module, which could be transformed by the loop combine tactic, is sumdb. This function has a composition of two self-recursive functions, sum and double. It could be transformed using the following sequence of issued commands (shown in *italic*).

```
tuple  -app_tactics
                                  ! this invokes the programming environment, called tuple
load  "ex1.hop";
                                  ! this loads in the specified module into the environment
"new_p"  :=  app_tactics(("ex1","sumdb"),loop_combine);
                                  ! this invokes the tactic and partial evaluator to produce a new
                                  !    module, called new_p, which contains the transformed
                                  !    equations
                                  ! Notice that := creates a new module in the environment
display_module   new_p;
                                  ! this displays the contents of the new module
module new_p;
 dec sum,sumdb : list(num) -> num;
 dec double : list(num) -> list(num);
- - -sumdb([])          <= 0;
- - -sumdb(a::x)       <= 2*a+sumdb(x);
- - -sum([])           <= 0;
- - -sum(a::x)         <= a+sum(x);
- - -double([])        <= [];
- - -double(a::x)      <= 2*a::double(x);
end new_p;
```

The first command in the above sequence starts up the programming environment. This is followed by a command to load the module, ex1, into the environment before the function,

sumdb, is subjected to the loop combine tactic (using app_tactics). In the transformation, a new module, new_p, is created to contain the newly transformed function.

This simple session illustrates how tactics are invoked to transform programs in the present programming environment.

A4.1. Description of Implemented Tactics

So far, three simple tactics have been implemented as meta-programs of the programming environment. We give a brief description for each of these tactics below, before proceeding to describe the components used to construct the tactics.

dec tupling:IntHE # set(FunctionId) -> script;

> This tactic takes an equation, containing a single function call, together with a set of mutually or auxiliary recursive functions in order to analyse for a eureka tuple by the instantiate and unfold ordering technique.
>
> If an eureka tuple is found, the tactic will produce a script that would construct a new non-redundant tuple function.

dec loop_combine:IntHE -> script;

> This is a very simplified form of fusion where all the function calls used in the given equation are self-recursive. It narrows the innermost call of the given equation, and performs unfolds (where possible) before a fold step. If this sequence of transformations is successful, a corresponding script is returned; otherwise an empty script is returned.

dec pe: IntHE # (FunctionId -> truval) -> scripts;

> This tactic performs a simplified form of partial evaluation (pre-computation) by unfolding, simplification and reductions. It uses the pattern-matching equations to decide if a call can be unfolded.
>
> To prevent infinite unfolding, this tactic takes a parameter of type, FunctionId -> truval, in order to indicate those function calls which could be safely unfolded (without causing non-termination in the transformation process). Those functions which may cause non-termination are not unfolded by this tactic.

A4.2. Symbolic Tree for Transformation

Program transformation can be viewed as an extended form of execution which operates on symbolic rather than ground terms. This unifying view between execution and transformation was presented in [Darlington88]. A trace for the program transformation steps could thus be recorded symbolically using some form of tree structure. For this purpose, we introduce a *symbolic tree* data structure, called sym_tree, to record the history of the transformation steps performed. This tree structure is represented by the following HOPE data structure.

```
data expr_loc     == (loc # IntHE);
data sym_tree     == st_passive(expr_loc)
                  ++ st_active(expr_loc)
                  ++ st_error(expr_loc,string)
                  ++ st_transform(expr_loc # script # sym_tree)
                  ++ st_branch(list(sym_tree))
                  ++ st_subexpr(expr_loc# list(sym_tree))
                  ++ st_define((IntHE # sym_tree) # sym_tree);
```

The symbolic tree contains enough information to obtain either the final transformed program or the script needed to re-play the transformation. We use it in all our tactics to represent the intermediate stages of transformational development.

The tuple data structure, expr_loc, is used to represent the current sub-expression (of the type IntHE) together with its location (of the type loc) within the present equation. This tuple structure is used in various parts of the symbolic tree structure. The symbolic tree contains three types of *leaves*, namely st_passive, st_active and st_error to represent the latest status (stage) of the transformation. It also contains four types of (recursive) *nodes*, namely st_transform, st_branch, st_subexpr and st_define to represent the intermediate stages of transformation. We give a brief description of the purpose for each of these leaves and nodes, below:

st_passive(ex) - This leaf is used to represent an expression which *could not* to be transformed further (by the present tactic) e.g. a constant or a variable.

st_active(ex) - This leaf is used to represent an expression which *could be* transformed further (by the present tactic).

st_error(ex,msg) - This leaf is used to denote that an erroneous condition has occurred during the present transformation. A common error is when the tactic could not apply successfully, e.g. a required fold step is not possible.

The second argument, msg, is used to contain the error message.

st_transform(ex,sc,tr) - This node represents the application of a transformation script, represented by sc, to an expression, ex, in order to obtain a new symbolic tree, tr.

st_branch(tr_list) - This node contains a list of independent symbolic trees that are being branched out for independent transformations. This could arise from the narrowing (unfold and instantiate) step where each branch is used to denote a different instantiation.

st_subexpr(ex,tr_list) - This node breaks up an expression, ex, into a number of independent sub-expressions, represented by the list of symbolic trees, tr_list, for independent transformations. An example of its use is the separate transformations for each of the arguments of a constructor term.

st_define(eqn,tr_1,tr_2) - This node contains a new function definition, eqn, together with its transformation (or development) in the symbolic tree, tr_1. The other symbolic tree, tr_2, is used to represent further transformation (or development) on the present expression. This further transformation may use the auxiliary function, eqn, just introduced.

The symbolic tree is a fundamental structure used by all our transformation tactics. It is used to contain the original, intermediate and transformed expressions. It also provides a trace of the transformation steps which have been carried out, and could therefore be used to obtain a script to re-play the transformations. Even though there are overheads associated with maintaining the

symbolic tree, it provides us with a re-usable and flexible component for developing tactics.

With this tree, each equation, eqn, which is to be transformed will start of as st_active(([],eqn)). This initial symbolic tree could then be developed (transformed) by the tactic into hopefully a symbolic tree where all the leaves are of the type st_passive. When this is achieved, the application of our tactic will be considered to be successful.

To help us support transformations on the symbolic tree, we have a set of support functions. A sub-set of these functions is presented below.

```
dec one_step_eval: (expr_loc -> sym_tree) # sym_tree -> sym_tree;
dec extract_script: sym_tree -> script;
dec extract_eqn: sym_tree -> list(IntHE);
dec check_error: sym_tree -> truval;
dec finish: sym_tree -> truval;
```

A brief functional description for each of the above set of support functions follows:

one_step_eval(fn_step,tr) - This applies a single transformation step, represented by the function-type parameter, fn_step, to each active leaf of the symbolic tree, tr. The result of this operation is a new tree where each active leaf has been further developed.

Evaluating the symbolic tree, one step at-a-time, allows us to control the transformation. As an example, we may wish to alternate between the application of two different sets of lemmas. This is easy to achieve with the symbolic tree.

extract_script(tr) - This function can be used to extract the entire script from the given symbolic tree, tr. This script would allow the re-play of transformations and is normally invoked at the end of a successfully applied tactic.

extract_eqn(tr) - This function is used to extract the final list of transformed equations from the symbolic tree, tr. This list may include intermediate functions defined and transformed in the tree. It is normally used for display or validation purposes.

check_error(tr) - This function is used to check if one or more errors have occurred during transformation. This is done by checking for st_error leaves in the tree, tr.

finish(tr) - This function checks to see if there are any active leaves left in the present symbolic tree, tr. If there are, then further transformation may be needed, otherwise the transformation will be considered as finished.

The symbolic tree is used to contain the structure of the transformational development. However, to directly manipulate and transform the expressions, we need lower-level access functions. These functions are discussed next.

A4.4. Low-Level Access Functions

There are a number of primitive functions which form the building block for all our transformations. We call these the *access functions*. They are used to retrieve equations from program repository (but not updates); convert between abstract syntax trees (IntHE) to concrete representations (string); manipulate (test, read or build) expressions; and perform matching, apply substitutions, or renaming.

Six of these access functions are:

```
dec get_eqns: FunctionId -> list(IntHE);
```

Given a function's identifier, this primitive function will retrieve the current list of equations for the function from the program repository.

```
dec IntHEtoStr: IntHE -> string;
```

Given an expression, this function will convert it into its string representation. It is useful for displaying expressions to users.

```
dec StrtoIntHE: string -> (truval # IntHE);
```

Given a string (representing the concrete form of an expression), parse it to give the corresponding expression. An error flag may be set if the parsing is not successful

```
dec unify: IntHE # IntHE -> truval # substitutions # substitutions;
```

Given two expressions, make an attempt to see if the two expressions unify. If they do, return a true flag together with an input and/or an output substitution lists. If they don't, return a false flag.

```
dec apply_substitutions: IntHE # substitutions -> IntHE;
```

Apply a list of substitutions to a given expression, in order to obtain a new expression.

```
dec rename: list(IntHE)  -> list(IntHE);
```

Rename a list of expressions with unique variables.

Apart from the above functions, we also have a number of primitive functions which could directly manipulate or access the HOPE abstract syntax tree. These functions are divided into three groups, to either *test*, *read* or *build* expressions, as shown below.

(A) **TEST FUNCTIONS** to check for the various expression constructs.

```
dec Is_HE_eqn: IntHE -> truval;              ! Is the given expression an equation?
dec Is_HE_applic: IntHE -> truval;           ! apply node?
dec Is_HE_ifexpr: IntHE -> truval;           ! if construct?
dec Is_HE_let: IntHE -> truval;              ! let construct?
dec Is_HE_letrec: IntHE -> truval;           ! recursive let construct?
dec Is_HE_lambda: IntHE -> truval;           ! lambda abstraction?
dec Is_HE_tuple: IntHE -> truval;            ! tuple expression?
dec Is_HE_set: IntHE -> truval;              ! absolute set expression?
dec Is_HE_rel_set: IntHE -> truval;          ! relative set expression?
dec Is_HE_equiv: IntHE -> truval;            ! equality?
dec Is_HE_function: IntHE -> truval;         ! function identifier?
dec Is_HE_data_constr: IntHE -> truval;      ! constructor identifier?
dec Is_HE_variable: IntHE -> truval;         ! variable?
dec Is_HE_num: IntHE -> truval;              ! number?
dec Is_HE_real: IntHE -> truval;             ! real?
dec Is_HE_char: IntHE -> truval;             ! character?
dec Is_HE_text: IntHE -> truval;             ! string?
```

(B) **READ FUNCTIONS** to obtain the components of the various constructs;

```
type IntHE2==(IntHE # IntHE);
type IntHE3==(IntHE # IntHE # IntHE);
data vcheck(alpha)==valid(alpha) ++ invalid;
dec Get_HE_eqn: IntHE -> vcheck(IntHE2);      !   Get the LHS and RHS of an eqn.
dec Get_HE_applic: IntHE -> vcheck(IntHE2);   !   Get the function and the argument of an apply node
dec Get_HE_ifexpr: IntHE -> vcheck(IntHE3);
dec Get_HE_let: IntHE -> vcheck(list(IntHE2) # IntHE  # IntHE);
dec Get_HE_letrec: IntHE -> vcheck(list(IntHE2) # IntHE  # IntHE);
dec Get_HE_lambda: IntHE -> vcheck(vector(IntHE));
dec Get_HE_tuple: IntHE -> vcheck(vector(IntHE));
dec Get_HE_set: IntHE -> vcheck(IntHE # list(IntHE) # list(IntHE2));
dec Get_HE_rel_set: IntHE ->  vcheck(IntHE # list(IntHE3));
dec Get_HE_equiv: IntHE -> vcheck(IntHE2);
dec Get_HE_function: IntHE -> vcheck(FunctionId);
dec Get_HE_data_constr: IntHE -> vcheck(FunctionId);
dec Get_HE_variable: IntHE -> vcheck(FunctionId);
dec Get_HE_num: IntHE -> vcheck(num);
dec Get_HE_real: IntHE -> vcheck(real);
dec Get_HE_char: IntHE -> vcheck(char);
dec Get_HE_text: IntHE -> vcheck(string);
```

(C) **BUILD FUNCTIONS** to construct expressions from sub-expressions

```
dec Build_HE_eqn: IntHE2 -> IntHE;     !   Given the LHS and RHS , return an eqn.
dec Build_HE_applic: IntHE2 -> IntHE;  !   Given a function and an argument, return an apply node
dec Build_HE_ifexpr: IntHE3 -> IntHE;
dec Build_HE_let: list(IntHE2) # IntHE  # IntHE -> IntHE;
dec Build_HE_letrec: (IntHE2) # IntHE  # IntHE -> IntHE;
dec Build_HE_lambda: vector(IntHE) -> IntHE;
dec Build_HE_tuple: vector(IntHE) -> IntHE;
dec Build_HE_set: (IntHE # list(IntHE) # list(IntHE2)) -> IntHE;
dec Build_HE_rel_set: IntHE # list(IntHE3) -> IntHE;
dec Build_HE_equiv: IntHE2 -> IntHE;
dec Build_HE_function: FunctionId-> IntHE ;
dec Build_HE_data_constr: FunctionId-> IntHE;
dec Build_HE_variable: FunctionId-> IntHE;
dec Build_HE_num: num -> IntHE;
dec Build_HE_real: real -> IntHE;
dec Build_HE_char: char -> IntHE;
dec Build_HE_text: string -> IntHE ;
```

The *test* and *read* functions are used mainly for analysis. The *build* functions could be used to construct transformation operations or to generate expressions for new function definitions. To assist us in developing more complex functions over the expression type, IntHE, higher-order reduce or map functions (based on the above primitives) have also been written. They include:

```
dec Reduce_IntHE: (IntHE # alpha ) # ((IntHE # alpha) -> alpha) -> alpha;
dec Map_IntHE: (IntHE  ) # ((IntHE') -> IntHE) -> IntHE;
dec Reduceloc_IntHE: (IntHE # alpha # loc) # ((IntHE # alpha # loc) -> alpha) -> alpha;
```

The last higher-order function also maintains a location counter which could be used to indicate the tree-address of the present sub-expression being handled.

Appendix 5. A Termination Proof

For the interested reader, we shall briefly reproduce the main result of Ferguson and Wadler's termination proof (for pure deforestation) in this Appendix. We shall also provide an additional proof to show that the slight modification we suggested for their algorithm will not cause the non-termination property to be lost. This additional proof shows the presence of a well-founded decreasing measure for expressions encountered by those recursive rules which do not have the fold steps, namely T 2, 5 and 6a.

A5.1. Pure Deforestation Termination Proof

As mentioned in Section 3.2.5, Ferguson and Wadler's proof of algorithm termination relies on two major results, namely:

1) Function nesting measure, **N**, is bounded by the T rules
2) Each expression encountered (by the T rules) always satisfies the grammar form of fct.

Proofs for the above two results are reproduced in Figure A5.1 and Figure A5.2.

To Prove: For all steps, $T[\![t]\!] \Leftarrow \ldots\ldots T[\![t_i]\!] \ldots\ldots, \forall i.\ \mathbf{N}[\![t]\!] \geq \mathbf{N}[\![t_i]\!]$

Case for Rule T 1- v : nothing to prove

Case for Rule T 2- $c(t_1,..,t_n)$:

From Definition: $\mathbf{N}[\![c(t_1,..,t_n)]\!] = \max\{\mathbf{N}[\![t_1]\!],.., \mathbf{N}[\![t_n]\!]\}$

- thus: $\forall i \in 1..n \quad \mathbf{N}[\![c(t_1,..,t_n)]\!] \geq \mathbf{N}[\![t_i]\!]$

Case for Rule T 3- $f(v_1,..,v_n)$: nothing to prove

Case for Rule T 4- $g(v_0,v_1,..,v_n)$: nothing to prove

Case for Rule T 5 - $...g(c_i(t'_1,..,t'_j),t_1,..,t_n)...$, where $g(c_i(v_1',..,v_j'),v_1,..,v_n)=tg_i$,

From Definition: $\mathbf{N}[\![g(c_i(t'_1,..,t'_j),t_1,..,t_n)]\!]$

$= \quad 1+\max\{\max\{\mathbf{N}[\![t'_1]\!],.., \mathbf{N}[\![t'_j]\!]\}, \mathbf{N}[\![t_1]\!],.., \mathbf{N}[\![t_n]\!]\}$

$= \quad 1+\max\{\mathbf{N}[\![t'_1]\!],.., \mathbf{N}[\![t'_j]\!], \mathbf{N}[\![t_1]\!],.., \mathbf{N}[\![t_n]\!]\}$

Also, since g is treeless, $\mathbf{N}[\![tg_i]\!] \leq 1$, hence:

$\geq \quad \mathbf{N}[\![tg_i]\!]+\max\{\mathbf{N}[\![t'_1]\!],.., \mathbf{N}[\![t'_j]\!], \mathbf{N}[\![t_1]\!],.., \mathbf{N}[\![t_n]\!]\}$

$\geq \quad \mathbf{N}[\![tg_i[t'/v'_{1..j},t/v_{1..n}]]\!]$

Adding a context to each side:

- $\mathbf{N}[\![...g\ (c_i(t'_1,..,t'_j),t_1,..,t_n)...]\!] \geq \mathbf{N}[\![...tg_i\ [t'/v'_{1..j},t/v_{1..n}]...]\!]$

Case for Rule T 6 - $...f\ (t_1,..,t_n)...$:

a) Similar to Case for T 5
b) Nothing to prove
c) Similar to Case for T 5

Case for Rule T 7 - $...g(v_0,t_1,..,t_n)$:

a) Nothing to prove
b) Similar to Case for T 5

Figure A5.1 - Proof to Show that the Nesting Measure, N, is bounded

To Prove: For each step, T $[\![t]\!] \Leftarrow .....T$ $[\![t_i]\!]$ ....., $\forall$ i. $t \in fct \Rightarrow t_i \in fct$

Case for Rule T 1 - v : nothing to prove

Case for Rule T 2 - $c(t_1,..,t_n)$:

$c(t_1,..,t_n) \in fct \Rightarrow c(t_1,..,t_n) \in c(ft_1,..,ft_n)$

• thus $\forall i \in 1..n \Rightarrow t_i \in ft$

$\Rightarrow t_i \in fct$

Case for Rule T 3 - $f(v_1,..,v_n)$: nothing to prove

Case for Rule T 4 - $g(v_0,v_1,..,v_n)$: nothing to prove

Case for Rule T 5 - $...g(c_i(t'_1,..,t'_j),t_1,..,t_n)...$, where $g(c_i(v_1'..v_j'),v_1..v_n)=tg_i$:

$...g(c_i(t'_1,..,t'_j),t_1,..,t_n)... \in fct \Rightarrow \forall i \in 1..j\ t'_i \in ft, \forall i \in 1..n\ t_i \in ft$

Since g is restricted-treeless, $tg_i \in rt, \Rightarrow ...tg_i\,[t'/v'_{1..j},t/v_{1..n}]... \in fct$

Case for Rule T 6 - $...f(t_1,..,t_n)...$:

a) Similar to Case for T 5

b) Nothing to prove

c) Similar to Case for T 5

Case for Rule T 7 - $...g(v_0,t_1,..,t_n)$:

a) Nothing to prove

b) Similar to Case for T 5

Figure A5.2 - Proof to show that the grammar form of fct is preserved

A5.2. An Additional Proof Step for the Modified Rules

The remaining termination proof of the (slightly) modified pure deforestation rules concerns the recursive rules of T 2, 5 and 6a. This set of three rules do not have corresponding fold rules, and must therefore be shown to have some well-founded decreasing measure. A well-founded decreasing measure is one which starts from some finite value and could always decrease to some smallest (least) value, within a finite number of steps. If such a well-founded measure exists, there could never be an infinite sequence of steps involving just these three rules.

Earlier in Section 3.2.5, we informally argued that there is a decreasing measure present for each of the three rules. In fact, we can provide a multi-set (bag[1]) measure, $\mathcal{M}$, (which takes into account the decreasing measures we informally mentioned) to show that all the three rules will always decrease this measure. Now, this multi-set measure for a given expression, e, will assign a tuple of three numbers to *each sub-expression* that is present in the given expression. The tuple of three numbers is meant to keep track of the following information:

(1) The function nesting depth of the current sub-expression.
(2) The number of constructors since the last function call.
(3) The function hierarchy number of each sub-expression which is a function call (0 if it is a constructor term).

Such a multi-set (of tuples) measure can be defined, as follows:

[1] Bags are sets which may contain duplicate elements.

$$\mathcal{M}(x,y)\ [\![v]\!] = \{\}$$

$$\mathcal{M}(x,y)\ [\![c(t_1,...,t_n)]\!] = \{(x,y+1,0)\} \cup \{ \mathcal{M}(x,y+1)\ [\![t_1]\!], ..., \mathcal{M}(x,y+1)\ [\![t_n]\!] \}$$

$$\mathcal{M}(x,y)\ [\![f(t_1,...,t_n)]\!] = \{(x+1,0,\mathbf{H}[\![f]\!])\} \cup \mathcal{M}(x+1,0)\ [\![t_1]\!] \cup .. \cup \mathcal{M}(x+1,0)\ [\![t_n]\!]$$

$$\mathcal{M}(x,y)\ [\![g(t_0,t_1,...,t_n)]\!] = \{(x+1,0,\mathbf{H}[\![g]\!])\} \cup \mathcal{M}(x+1,0)\ [\![t_0]\!] \cup .. \cup \mathcal{M}(x+1,0)\ [\![t_n]\!]$$

where $\mathbf{H}[\![f]\!]$ returns the function hierarchy number of its input function, f. The pair of parameters, (x,y), of the measure, $\mathcal{M}$, is used to carry the depth of the function nesting for the current sub-expression, and the number of constructors encountered since the last function call.

This multi-set (or bag) measure can be ordered using the classic multi-set ordering, which could be defined as follows:

$$B_1 > B_2 \text{ IFF } \max(B_1) > \max(B_2) \vee (\max(B_1) = \max(B_2) \wedge (B_1 - \max(B_1)) > (B_2 - \max(B_2)))$$

where max(B) selects the largest item from bag B

The items (or tuples) of these bags must also have an (total) ordering defined between them. The ordering which will be used in our proof is a lexicographic ordering of the following definition:

$$(n_1,c_1,h_1) > (n_2,c_2,h_2) \text{ IFF } n_1 > n_2 \vee (n_1 = n_2 \wedge c_1 > c_2) \vee (n_1 = n_2 \wedge c_1 = c_2 \wedge h_1 > h_2)$$

Note that we have used the symbol, >, to represent different versions of the ordering operator (between bags, tuples and numbers). There should be no confusion here because the various versions of the ordering operator should be distinguishable from context.

We shall prove the following lemma which states that the measure, $\mathcal{M}$, provides a well-founded ordering for the recursive rules of ***T*** 2, 5 and 6a.

Lemma A5.1 - Well-Founded Measure for *T* 2, 5 and 6a Rules

The multi-set measure, $\mathcal{M}$, on expressions being transformed by the recursive rules, ***T*** 2, 5 and 6a, can be shown to exhibit a well-founded decreasing ordering. This decreasing ordering can be expressed, as follows:

For rules ***T*** 2, 5 and 6a, $T[\![t]\!] \Leftarrow .....T[\![t_i]\!] ....., \forall i.\ \mathcal{M}(x,y)\ [\![t]\!] > \mathcal{M}(x,y)\ [\![t_i]\!]$

To prove that this measure is well-founded we have to establish three results, namely:

1) the measure is finite for each given finitely-sized expression,
2) the measure has a smallest (bottom) value,
3) each expression encountered (on the RHS of ***T***) by the three recursive rules will always have a measure which is less than the measure of its originating expression (on the LHS of ***T***).

Now, $\mathcal{M}$ is defined recursively on successively smaller sub-expressions. Such a definition is bound to produce a finite value for any given initial expression that is finite. Furthermore, it can be shown that $\mathcal{M}$ has a smallest value of {}, when its given expression is a variable. Most importantly, it can also be shown that the measure $\mathcal{M}$ is decreasing with respect each application of rules, ***T*** 2,5,6a. This result is given as a proof in Figure A5.3.

With this well-founded measure, all sequences of steps, involving just ***T*** 2,5,6a, will always be finite. As a result, they could never extend the overall sequence of the ***T*** rules into an

infinite one (because there are only a finite number of the explicit define&unfold steps). Hence, the slightly optimised rules do not destroy the termination property of Wadler's algorithm.

Proof of Lemma A5.1

To Prove: For steps, T 2,5,6a $T[\![t]\!] \Leftarrow .....T[\![t_i]\!] ....., \forall i.\ \mathcal{M}(x,y)[\![t]\!] > \mathcal{M}(x,y)[\![t_i]\!]$

Case for Rule T 2- $c(t_1,..,t_n)$:

From Definition: $\mathcal{M}(x,y)[\![c(t_1,..,t_n)]\!] = \{(x,y+1,0)\} \cup \mathcal{M}(x,y+1)[\![t_1]\!] \cup .. \cup \mathcal{M}(x,y+1)[\![t_n]\!]$

Also since, $\mathcal{M}(x,y)[\![t_i]\!] < \{(x,y+1)\} \cup \mathcal{M}(x,y+1)[\![t_i]\!]$:

- We have: $\forall i \in 1..n \quad \mathcal{M}(x,y)[\![t_i]\!] < \{(x,y+1,0)\} \cup \mathcal{M}(x,y+1)[\![t_i]\!]$
  $< \{(x,y+1,0)\} \cup \mathcal{M}(x,y+1)[\![t_1]\!] \cup .. \cup \mathcal{M}(x,y+1)[\![t_n]\!]$
  $< \mathcal{M}(x,y)[\![c(t_1,..,t_n)]\!]$

Case for Rule T 5 - $...g(c_i(t'_1,..,t'_j),t_1,..,t_n)...$, where $g(c_i(v_1',..,v_j'),v_1,..,v_n)=tg_i$, and **H** $[\![g]\!]=r$

From Definition: $\mathcal{M}(x,y)[\![g(c_i(t'_1,..,t'_j),t_1,..,t_n)]\!]$

$= \{(x+1,0,r)\} \cup \mathcal{M}(x+1,0)[\![c_i(t'_1,..,t'_j)]\!]$
$\cup \mathcal{M}(x+1,0)[\![t_1]\!] \cup .. \cup \mathcal{M}(x+1,0)[\![t_n]\!]$

$= \{(x+1,0,r),(x+1,1,0)\} \cup \mathcal{M}(x+1,1)[\![t'_1]\!] \cup .. \cup \mathcal{M}(x+1,1)[\![t'_j]\!]$
$\cup \mathcal{M}(x+1,0)[\![t_1]\!] \cup .. \cup \mathcal{M}(x+1,0)[\![t_n]\!]$

Since g is r-treeless, $\mathcal{M}(x,y)[\![tg_i]\!] < \{(x+1,1,0)\}$, hence:

$> \{(x+1,0,r), \mathcal{M}(x,y)[\![tg_i]\!]\} \cup \mathcal{M}(x+1,1)[\![t']\!]_{1..j}$
$\cup \mathcal{M}(x+1,0)[\![t]\!]_{1..n}$

Since $\mathcal{M}(x+1,0)[\![t_i']\!] \leq \mathcal{M}(x+1,1)[\![t_i']\!]$:

$> \{(x+1,0,r)\} \cup \mathcal{M}(x,y)[\![tg_i]\!] \cup \mathcal{M}(x+1,0)[\![t']\!]_{1..j}$
$\cup \mathcal{M}(x+1,0)[\![t]\!]_{1..n}$

Since $t'_{1..j}, t_{1..n}$ (as linear arguments) are each substituted at most once, we also have:

$\mathcal{M}(x,y)[\![tg_i[t'/v'_{1..j}, t/v_{1..m}, t/v_{m+1..n}]]\!] \leq \mathcal{M}(x,y)[\![tg_i]\!] \cup \mathcal{M}(x+1,0)[\![t']\!]_{1..j} \cup \mathcal{M}(x+1,0)[\![t]\!]_{1..n}$

$> \{(x+1,0,r)\} \cup \mathcal{M}(x,y)[\![tg_i[t'/v'_{1..j}, t/v_{1..n}]]\!]$

$> \mathcal{M}(x,y)[\![tg_i[t'/v'_{1..j}, t/v_{1..n}]]\!]$

Adding a context to each side:

- $\mathcal{M}(x,y)[\![...g(c_i(t'_1,..,t'_j),t_1,..,t_n)...]\!] > \mathcal{M}(x,y)[\![...tg_i[t'/v'_{1..j}, t/v_{1..n}]...]\!]$

Case for Rule T 6a - $...f(t_1,..,t_n)...$, where $f(v_1,..,v_n)=tf$, and **H** $[\![f]\!]=r$

From Definition: $\mathcal{M}(x,y)[\![f(t_1..t_n)]\!]$

$= \{(x+1,0,r)\} \cup \mathcal{M}(x+1,0)[\![t]\!]_{1..n}$

Since f is r-treeless, and non-recursive $\mathcal{M}(x,y)[\![tf]\!] < \{(x+1,0,r)\}$, hence:

$> \{\mathcal{M}(x,y)[\![tf]\!]\} \cup \mathcal{M}(x+1,0)[\![t]\!]_{1..n}$

Snce $t_{1..n}$ (as linear arguments) are substituted at most once, we also have:

$\mathcal{M}(x,y)[\![tf[t/v_{1..n}]]\!] \leq \mathcal{M}(x,y)[\![tf]\!] \cup \mathcal{M}(x+1,0)[\![t]\!]_{1..n}$

Thus: $> \mathcal{M}(x,y)[\![tf[t/v_{1..n}]]\!]$

Adding a context to each side:

- $\mathcal{M}(x,y)[\![...f(t_1..t_n)...]\!] > \mathcal{M}(x,y)[\![...tf[t/v_{1..n}]...]\!]$

Figure A5.3 - Proof of Lemma A5.1 ($\mathcal{M}$ is well-founded)