The Use of Proof Plans for Transformation of Functional Programs by Changes of Data Type

Julian D.C. Richardson

ARTIFICIAL INTELLIGENCE LIBRARY
UNIVERSITY OF EDINBURGH
80 South Bridge
Edinburgh EH1 1HN

Ph.D.
University of Edinburgh
1995
Abstract

Program transformation concerns the derivation of an efficient program by applying correctness-preserving manipulations to a source program. Transformation is a lengthy process, and it is important to keep user interaction to a manageable level by automating the transformation steps.

In this thesis I present an automated technique for transforming a program by changing the data types in that program to ones which are more appropriate for the task. Programs are constructed by proving synthesis theorems in the proofs-as-programs paradigm. Programs are transformed by modifying their synthesis theorems and relating the modified theorem to the original. Proof transformation allows more powerful transformations than program transformation because the proof of the modified theorem yields a program which meets the original specification, but may compute a different function to the original program. Synthesis proofs contain information which is not present in the corresponding program and can be used to aid the transformation process.

Type changes are motivated by the presence of certain subexpressions in the synthesised program. A library of possible type changes is maintained, indexed by the motivating subexpressions. I present a new method which extends these type changes from the motivating expressions to cover as much of the program as possible. Once a type change has been chosen, a revised synthesis theorem is constructed automatically.

Search is a major problem for program transformation systems. The synthesis theorems which arise after a type change have a particular form. I present an automated proof strategy for synthesis theorems of this form which requires very little search.
I declare that this thesis has been composed by myself and that the work described is my own.

Julian D. C. Richardson
Acknowledgements

This research was supported by EPSRC studentship 91308481, computing resources from EPSRC GR/J80702 and grant BC/DAAD ARC Project no. 438. I am very grateful for a grant from the Marktoberdorf Foundation enabling me to attend the 1993 Marktoberdorf Summer School. I would like to thank my supervisors, Professor Alan Bundy and Dr Geraint Wiggins, and the members of the Mathematical Reasoning Group for their support.

I would also like to extend my thanks to Edinburgh University Department of Artificial Intelligence for providing me with a grant to attend KBSE-95, and I gratefully acknowledge the past assistance of the Joint Educational Trust.

John Levine kindly helped me with the final printing. Finally, I would like to thank my friends and family for their invaluable support.

This thesis is dedicated to Barbara and Ashley.
# Table of Contents

1. Introduction ........................................ 19
   1.1 Overview ........................................... 19
   1.2 Motivation ......................................... 20
   1.3 Contributions ...................................... 22
   1.4 Organisation ....................................... 23
   1.5 Changing types in programs ....................... 24
      1.5.1 Choosing a transformation .................... 24
      1.5.2 Modifying the program ....................... 24
      1.5.3 Summary ....................................... 26
   1.6 Conclusion ......................................... 26

2. Background ........................................... 27
   2.1 Introduction ...................................... 27
   2.2 Program transformation ............................ 28
      2.2.1 Introduction ................................ 28
      2.2.2 Fold/Unfold ................................ 29
   2.3 Changes of representation ....................... 31
      2.3.1 Introduction ................................ 31
      2.3.2 Representation change in problem solving .... 33
2.3.3 Van Baalen .................................................. 34

2.4 Representation change in programs .......................... 35
  2.4.1 Introduction ............................................. 35
  2.4.2 Definition of an abstract data type ..................... 35
  2.4.3 ADT development ........................................ 36
  2.4.4 Abstraction and Implementation ....................... 37
  2.4.5 Lowry .................................................. 37
  2.4.6 DTRE .................................................. 39
  2.4.7 Reformulation using retrieve/representation functions ... 40
  2.4.8 Automatic Data Structure Selection ................... 42
  2.4.9 Data structure mapping and difference lists .......... 44

2.5 Program synthesis ............................................. 46
  2.5.1 Introduction ............................................. 46
  2.5.2 Knowledge-based program synthesis ..................... 46
  2.5.3 Constructive synthesis in Type Theory ................ 47
  2.5.4 Program transformation vs. program synthesis ........ 49
  2.5.5 Formal program development from algebraic specifications ... 50

2.6 Proof Transformation ......................................... 51
  2.6.1 Applications of proof transformation .................. 51
  2.6.2 Program transformation as proof transformation .......... 51

2.7 Conclusion .................................................. 56

3. Planning Program Synthesis Proofs ............................ 57
  3.1 Introduction ............................................... 57
  3.2 Program construction in type theory ...................... 58
<table>
<thead>
<tr>
<th>3.2.1</th>
<th>Introduction</th>
<th>58</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2.2</td>
<td>The Oyster implementation of Martin-Löf's Type Theory</td>
<td>59</td>
</tr>
<tr>
<td>3.2.3</td>
<td>Tactics</td>
<td>59</td>
</tr>
<tr>
<td>3.3</td>
<td>Proof planning, the meta-level and CLÁM</td>
<td>60</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Proof planning rationale</td>
<td>60</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Methods are partial specifications of tactics</td>
<td>60</td>
</tr>
<tr>
<td>3.3.3</td>
<td>Searching for a proof and CLÁM</td>
<td>62</td>
</tr>
<tr>
<td>3.3.4</td>
<td>A comparison of the object-level and meta-level search spaces</td>
<td>64</td>
</tr>
<tr>
<td>3.4</td>
<td>The proof strategy for induction</td>
<td>65</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Introduction</td>
<td>65</td>
</tr>
<tr>
<td>3.4.2</td>
<td>The form of constructor induction</td>
<td>65</td>
</tr>
<tr>
<td>3.4.3</td>
<td>Wave terms</td>
<td>66</td>
</tr>
<tr>
<td>3.4.4</td>
<td>Wave rules</td>
<td>67</td>
</tr>
<tr>
<td>3.4.5</td>
<td>Rippling and the proof strategy for induction</td>
<td>68</td>
</tr>
<tr>
<td>3.4.6</td>
<td>Ripple analysis</td>
<td>71</td>
</tr>
<tr>
<td>3.4.7</td>
<td>Benefits of rippling</td>
<td>72</td>
</tr>
<tr>
<td>3.5</td>
<td>Extensions to rippling</td>
<td>73</td>
</tr>
<tr>
<td>3.5.1</td>
<td>Directional wave fronts</td>
<td>73</td>
</tr>
<tr>
<td>3.5.2</td>
<td>Wave front unblocking</td>
<td>74</td>
</tr>
<tr>
<td>3.5.3</td>
<td>Proof critics</td>
<td>74</td>
</tr>
<tr>
<td>3.5.4</td>
<td>Difference matching and unification</td>
<td>75</td>
</tr>
<tr>
<td>3.5.5</td>
<td>Middle-out reasoning and meta-variables</td>
<td>75</td>
</tr>
<tr>
<td>3.5.6</td>
<td>Relational rippling</td>
<td>76</td>
</tr>
<tr>
<td>3.6</td>
<td>Program synthesis in CLÁM</td>
<td>76</td>
</tr>
</tbody>
</table>
3.7 A description of the main methods ........................................ 78
  3.7.1 induction/2 .......................................................... 78
  3.7.2 elim_existential/2 ................................................ 79
  3.7.3 step_case/1 ......................................................... 79
  3.7.4 base_case/1 ......................................................... 79
  3.7.5 apply_lemma/2 ..................................................... 80
  3.7.6 existential/2 ...................................................... 80
3.8 Conclusion ........................................................................... 80

4. Type Change as a Proof Development Step ............................... 82
  4.1 Introduction ...................................................................... 82
  4.2 Type changes in Type Theory ............................................ 83
    4.2.1 The correctness of the transformation ......................... 83
    4.2.2 The implementation relation ..................................... 83
  4.3 Correctness in the type theory .......................................... 85
  4.4 Abstract Data Types ....................................................... 89
    4.4.1 Introduction ................................................................ 89
    4.4.2 Formulating ADTs in Type Theory ............................... 90
    4.4.3 Program refinement in type theory ............................. 93
  4.5 An example transformation ............................................... 96
    4.5.1 Definitions ................................................................ 96
    4.5.2 Transformation ........................................................ 98
    4.5.3 Avoiding conversion functions within the recursive loop . 101
    4.5.4 Efficiency ............................................................. 102
  4.6 Making use of the source proof ......................................... 103
5.8.4 Wave rules and equations .................................................. 122
5.8.5 The proof plan ................................................................. 123
5.9 Other ways of motivating type changes ................................. 125
5.10 Direction of transformations ............................................... 125
5.11 Flexibility in type changes ................................................... 126
5.12 An analysis of the nat to nat2 example ................................. 127
5.13 The typechange/2 method ..................................................... 127
5.14 Conclusion ........................................................................ 128

6. A Proof Strategy for Synthesis ............................................... 130
6.1 Introduction .......................................................................... 130
6.2 Branching during synthesis ................................................... 131
6.2.1 Introduction .................................................................... 131
6.2.2 A simple example ........................................................... 131
6.3 Developing a proof strategy ................................................... 133
6.3.1 Restrict search ................................................................. 133
6.3.2 Avoid conversion functions .............................................. 133
6.4 Using rippling to reduce search ............................................. 134
6.4.1 Difference matching ......................................................... 134
6.4.2 Difference matching with the cancellation rule ................... 135
6.4.3 The standard_form/2 method .......................................... 135
6.4.4 The wave rules .............................................................. 137
6.4.5 The benefits of rippling .................................................. 137
6.5 Difference matching in function composition ......................... 138
6.6 Induction vs. standard form .................................................. 139
8.4 Results .......................................................... 179
  8.4.1 Synthesis of functions on binary numbers .......... 180
  8.4.2 Difference list examples ............................... 181
  8.4.3 nat to nat2 examples ................................... 182
  8.4.4 Queue examples ......................................... 183
  8.4.5 An abstraction type change ............................ 183

8.5 Conclusion ..................................................... 184

9. A Comparison With Previous Work ......................... 186
  9.1 Introduction .................................................. 186
  9.2 ADT refinement and translation ......................... 187
    9.2.1 Choice of refinement .................................. 187
    9.2.2 Translation rules ...................................... 189
  9.3 Synthesis of efficient implementations ................. 191
    9.3.1 A comparison of rippling with fold/unfold ......... 191
    9.3.2 Lowry .................................................. 192
  9.4 Proof transformation ...................................... 196
    9.4.1 Advantages of proof transformation ................. 196
    9.4.2 Behaviour-changing transformations ................. 196
    9.4.3 Use of information from proof ....................... 197
    9.4.4 A summary of my use of proof transformation ....... 199
  9.5 Conclusion .................................................. 199

10. Further Work ................................................ 204
  10.1 Introduction ............................................... 204
10.2 Use of algebraic techniques .............................................. 205
10.3 ADT mechanism ............................................................ 205
10.4 Propagation and program efficiency .................................... 206
   10.4.1 Measuring program efficiency ..................................... 206
   10.4.2 Extension of propagation rules .................................... 207
   10.4.3 Controlling search for a type change ............................ 207
   10.4.4 Implementation of the typechange_strat/2 tactic ............. 208
   10.4.5 Use of proof critics ................................................. 208
10.5 Extension of the difference matching proof strategy ............. 208
   10.5.1 Refinement of the strategy ......................................... 208
   10.5.2 Different input and output transformations .................... 209
   10.5.3 Using difference unification ....................................... 211
   10.5.4 Difference matching after type abstraction .................... 211
   10.5.5 Comparison of rippling with fold/unfold ....................... 212
10.6 Abstraction .................................................................... 212
   10.6.1 Allowing output transformations ................................. 212
   10.6.2 Abstraction tactic in Oyster ....................................... 213
   10.6.3 Strategies for abstraction synthesis proofs ................... 214
10.7 Scaling up the proof strategy ........................................... 214
10.8 Use of information in synthesis proofs ................................ 214
10.9 Conclusion ..................................................................... 215

11. Conclusions .................................................................... 216
   11.1 Contributions ............................................................. 216
   11.2 ADTs and type changes in CIMA ...................................... 217
11.3 Heuristics for choosing type changes ........................................ 218
11.4 Synthesis after a type change ..................................................... 219
11.5 Information in synthesis proofs .................................................. 219
11.6 Summary ....................................................................................... 220

A. Glossary .......................................................................................... 221

B. An example synthesis: nat ................................................................. 227
   B.1 Introduction ................................................................................ 227
   B.2 The synthesis of nat ................................................................. 227

C. A session with Pitches ...................................................................... 231

D. Example proofs and their ADTs ....................................................... 238
   D.1 Introduction ................................................................................ 238
   D.2 The list to difference list transformation ................................... 238
      D.2.1 The difference list ADT and definitions ............................... 239
      D.2.2 The transformation of flatten .............................................. 241
      D.2.3 The transformation of rev ................................................... 241
      D.2.4 The transformation of revflatten ....................................... 242
      D.2.5 The transformation of apprevdl ......................................... 244
      D.2.6 The transformation of revappd1 ......................................... 245
      D.2.7 The transformation of app4 ................................................. 246
      D.2.8 The transformation of apprev2 .......................................... 247
   D.3 The list to (min, max) abstraction example .................................. 248
      D.3.1 The genvar synthesis proof ............................................... 249
D.4 The nat to nat2 transformation ........................................... 250
D.4.1 The nat and nat2 ADTs .................................................. 250
D.4.2 The synthesis of mod2sumlist ........................................ 252
D.4.3 The transformation of mod2comp ..................................... 253
D.5 The synthesis of a binary addition function ....................... 254
D.6 Queue examples .......................................................... 256
D.6.1 The queue ADT .......................................................... 256
D.6.2 The transformation of listlastq .................................... 257

E. Descriptions of the new methods ....................................... 259
E.1 Introduction .............................................................. 259
E.2 standard_form(standard_form,Term) ................................... 259
E.3 guess_existential/3 ....................................................... 260
E.4 finitetypeelim/1 ........................................................ 261
E.5 elim_existential/2 ......................................................... 262
E.6 split_implementation/1 .................................................. 262
E.7 existential_subterm/2,existential/2 ................................ 265
E.8 The typechange_strat/2 method ....................................... 266
E.9 The simplify_typechange_goals/2 submethod ....................... 266
  E.9.1 The implies_independence/1 submethod ......................... 267
  E.9.2 The relation_to_function/2 submethod ......................... 268
  E.9.3 The embedded_existential/1 submethod ......................... 268
  E.9.4 prove_output_surjectivity/1 .................................... 268
  E.9.5 substitute_using_cfeq/1 ........................................ 269
F. Code for the main methods and predicates

F.1 The typechange code .................................. 270
F.2 The main methods ..................................... 277
  F.2.1 typechange_strat/2 ................................. 277
  F.2.2 standard_form/2 .................................. 278

G. Bibliography ........................................... 281
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-1</td>
<td>A comparison of different representation change systems.</td>
<td>32</td>
</tr>
<tr>
<td>2-2</td>
<td>Abstract/implement transformations</td>
<td>39</td>
</tr>
<tr>
<td>2-3</td>
<td>Dependency information can allow pruning of parts of the computation.</td>
<td>52</td>
</tr>
<tr>
<td>3-1</td>
<td>The proof strategy for induction.</td>
<td>70</td>
</tr>
<tr>
<td>4-1</td>
<td>The implementation mapping $T_a \sim T_b$.</td>
<td>84</td>
</tr>
<tr>
<td>4-2</td>
<td>Proof obligations arising from a representation change.</td>
<td>87</td>
</tr>
<tr>
<td>4-3</td>
<td>The object level meta-theorem justifying an implementation type change on the input type.</td>
<td>88</td>
</tr>
<tr>
<td>4-4</td>
<td>The object level meta-theorem justifying an implementation type change on the output type.</td>
<td>89</td>
</tr>
<tr>
<td>5-1</td>
<td>The original flatten function.</td>
<td>123</td>
</tr>
<tr>
<td>5-2</td>
<td>The transformed flatten function.</td>
<td>124</td>
</tr>
<tr>
<td>5-3</td>
<td>Derivation of the nat list to nat2 list transformation.</td>
<td>129</td>
</tr>
<tr>
<td>7-1</td>
<td>An abstraction transformation is many-to-one.</td>
<td>148</td>
</tr>
<tr>
<td>7-2</td>
<td>The signature of the mminmax ADT and some of its equations.</td>
<td>163</td>
</tr>
<tr>
<td>7-3</td>
<td>The shape of the mminmax proof.</td>
<td>165</td>
</tr>
<tr>
<td>7-4</td>
<td>The shape of the typechange_strat subplan.</td>
<td>167</td>
</tr>
</tbody>
</table>
7–5 Equations and wave rules for the synthesis of a set analogue of append. ................................................................. 168

7–6 The first half of the mminmax proof. ................................................................. 171

7–7 The second half of the mminmax proof. ................................................................. 172

8–1 The order of the PITCHES methods in Cl\textsuperscript{M}. ................................................................. 174

9–1 The proof plan automatically generated by Cl\textsuperscript{M} for the synthesis of addcirc. ................................................................. 193

9–2 The wave rules which are available during the Cl\textsuperscript{M} synthesis of addcirc. ................................................................. 194

9–3 The fold/unfold search space to depth 4 for the derivation of addcirc. ................................................................. 201

9–4 Equations for the fold/unfold synthesis of addcirc. ................................................................. 202

9–5 Information flow for direct program transformation, e.g. in Goad’s and Madden’s systems. ................................................................. 203

9–6 Information flow in PITCHES. ................................................................. 203

B–1 The proof plan for the synthesis of nat. ................................................................. 230

B–2 The object-level synthesis of nat. ................................................................. 230
Chapter 1

Introduction

1.1 Overview

This thesis develops an automatic technique for program transformation in which data types in a program are exchanged for ones which make the program more efficient. The program is then further transformed in order to make best use of the new data types.

The programs arise from (synthesis) proofs of their specifications in the proofs-as-programs paradigm (§2.5), and the transformation is realised as a proof step in the construction of a revised proof of the specification. This enables us to make use of information which is explicit in the synthesis proof but not in the corresponding program, and to use powerful heuristics which have been developed for constructing inductive proofs. Verifying the correctness of a transformation is simple, because any transformation which produces a proof of the original synthesis theorem is necessarily correct.

The program transformation is automated by encoding proof search heuristics as proof plans (§3.3). There are two aspects to the proof plans we construct:

1. Decide which data type transformation should be made, and where in the proof/program to make it.
2. Given such a transformation, synthesise an *efficient* program on the new type which is equivalent to the original program.

A fairly simple heuristic, which motivates a transformation by the presence of certain subexpressions in the program, is extended by means of *propagation rules* (§5.3) into a sophisticated mechanism for determining where to make a transformation for the maximum benefit.

Once a transformation has been chosen, a revised specification is constructed and related to the original one in order to ensure the correctness of the transformation. Using *difference matching* (§6.4.1) and *rippling* (§3.4.5), a proof of the revised specification can be constructed with very little search.

These techniques are implemented as a proof strategy in the system **PI**T**C**HES ("Planning Type CHanges and Efficient Synthesis"), which comprises a number of proof *methods* which have been written for the CLAM proof planner.

Portions of this thesis are to be published in [Richardson 95].

### 1.2 Motivation

According to modern programming practice, it is more important to write programs which are easy to understand and maintain than ones which are efficient. It should be possible to verify to a high degree of confidence that a computer program performs the task required of it. This is extremely important, especially since computers and computer programs are being given responsibility for our safety: flying aeroplanes and running nuclear reactors, for example. As an example of the dangers, [Thomas 94] analyses what went wrong in a machine for performing radiotherapy when the physical interlocks preventing overdoses were replaced by a computer system.

*Program transformation*, discussed in §2.2, concerns the development of techniques by which programs can be manipulated to improve their efficiency.
without changing their desired behaviour. Program transformation allows programs to be developed in two steps: the production of an inefficient program which can easily be verified, followed by the application of transformations to derive an equivalent, but more efficient, program.

In [Kowalski 79], Kowalski states that "Alteration of data structures is a way of improving algorithms by replacing an inefficient data structure by a more effective one. In a large, complex program, the demands for information made on the data structures are often fully determined only in the final stages of the program design." This suggests a program transformation which changes the data types in a program to make it more efficient.

The major problem in any program transformation system is controlling the vast search space of possible transformations. Except in the case of very limited transformations, some kind of guidance is needed. Guidance can be provided by:

1. user input in an interactive system, for example asking the user to choose between several alternatives, or

2. heuristics incorporated into the system.

It is important not to swamp the user with questions, so even when the user supplies the guidance, heuristics may be necessary to limit the occasions when it is required.

Proof planning provides a framework for the expression of sophisticated heuristics. The domains of proof planning (construction of proofs, in particular inductive proofs) and program transformation (the manipulation of programs, in particular recursive programs) are closely related, and this thesis sets out to exploit existing proof planning techniques to automate a particular kind of program transformation.
1.3 Contributions

This thesis makes the following contributions:

1. I have shown how type changes can be performed and proved to be correct for programs synthesised in Martin-Löf's Type Theory.

2. I have demonstrated that an ADT mechanism can be integrated into the ClAM proof planner.

3. I have developed heuristics for choosing and placing type changes in a program synthesis proof. In order to synthesise efficient programs, I have defined propagation rules which perform a similar kind of analysis in a synthesis proof to that performed by dataflow analysis in programs.

4. I have developed a proof strategy which can often prove automatically the synthesis goals which arise after a type change. This is valuable in itself, but also demonstrates the utility of proof planning: I identified the shape of the synthesis proofs I wanted to construct, and encoded this as a proof strategy.

5. I have demonstrated the generality of existing proof methods, in particular rippling, by using them to automate new types of proof.

6. I have identified information available in the construction of a synthesis proof plan but not in a source program alone, which is useful in the development of an efficient target program (via its synthesis proof). Some of this information is used indirectly by propagation rules. Other information guides the synthesis proofs. I have indicated how this results in a great reduction in the size of the search space when compared to the fold/unfold strategy.
1.4 Organisation

This thesis is organised as follows: the remainder of chapter 1 provides a brief introduction to changing types in programs. Chapter 2 outlines some previous work on program transformation and changes of representation both in general problem solving and in programs. A section on program synthesis introduces the idea of proofs-as-programs, and the transformation of programs by transformation of their synthesis proofs. Chapter 3 introduces the proof planning techniques on which automation of type changes in this thesis is based. The proof strategy for induction is described. Proof in the object-level logic and at the planning level are compared. In chapter 4, I describe how complex data types can be specified as abstract data types. Data type change is formalised as a sound development step in the construction of a synthesis proof, and I present heuristics for choosing data type changes and discuss the efficiency of the resulting program. I argue that proof plans contain most of the information which can be obtained from a proof to guide program transformation. The chapter is illustrated with an example synthesis proof. Chapter 5 formalises the heuristics presented in chapter 4 and describes their implementation as a proof plan. Chapter 6 presents a proof plan for constructing, with very little search, the revised synthesis proofs which arise after a type change. This is demonstrated by synthesising an addition function for binary numbers. Chapter 7 describes the extension of the type change mechanism to deal with type changes which are abstractions. In chapter 8 I discuss the implementation and the examples which have been proved by PITCHES. I compare PITCHES with previous work in chapter 9, and in chapter 10 discuss future avenues of research. Finally, chapter 11 draws the conclusions.

Appendix A is a glossary. Appendix B illustrates the process of program synthesis, going from the production of a proof plan through to the object-level synthesis proof. Appendix C contains the trace of a complete session with PITCHES. Example proofs and the required ADTs are listed in appendix D.
The new methods which are required to construct the proofs in this thesis are described in appendix E, and code for some of the main predicates and methods is given in appendix F.

1.5 Changing types in programs

1.5.1 Choosing a transformation

Commonly, the most convenient data types are used when initially writing a program or a theorem. In many cases, however, there are better representations which will make the program more efficient, or lead to a "better" proof of the theorem. For example, using difference lists instead of lists in a Prolog program can yield dramatic speed improvements [Hansson & Tärrlund 82], but care is needed to ensure that the transformation is correct [Marriott & Sondergaard 88]. The change of data type requires a change of the program, replacing functions on lists (hd, tl, append etc.) by their equivalents on difference lists. The main effect of this transformation is to improve the efficiency of append. We can exploit this fact by using the presence of append in a program to motivate a change from lists to difference lists.

1.5.2 Modifying the program

The essentials of this transformation technique are most easily introduced in programming terms.

In a typed language, every expression, say \( f(x) \), in a program can be assigned a type. For example, \( f \) could be of type \( t \rightarrow t \), where \( t \) is the type of \( x \). Sometimes, the operation performed by \( f \) may be done more efficiently by some other operation \( f' \) defined on a different type \( t' \), say \( f':t' \rightarrow t' \). For example, many operations on sets, such as intersection or deleting an element, are more efficient on ordered lists.
In order to replace $f$ by $f'$ in the program we must relate the types and functions in question. We do this by requiring a *conversion function* which maps each element of $t'$ to the element of $t$ it represents.

$$\rho : t' \to t$$

Here, $\rho$ stands for "representation". Now throughout the program we can replace

$$f(x) \text{ by } \rho(f'(\rho^{-1}(x)))$$

For example, sets may be implemented by ordered lists to make certain operations more efficient, e.g. finding the least element, or checking for set membership. This type change is encoded by a conversion function $\rho : \text{set} \to \text{olist}$ to map each set to an ordered list such that:

$$\forall s : \text{set}. \text{ordered}(\rho(s)) \land (\forall x : \text{obj}. \text{member}(x, \rho(s)) \leftrightarrow x \in s)$$

To ensure that this transformation is correct, we must verify that for all $x$, $f(x)$ and $\rho(f'(\rho^{-1}(x)))$ are equal. We do this by proving that:

$$\forall x' : t'. \rho(f'(x')) = f(\rho(x'))$$

Although $f'$ may be more efficient than $f$, the overhead introduced by the insertion of conversion functions means that the composite $\rho f' \rho^{-1}$ may not be. This simple replacement of $f$ by $\rho f' \rho^{-1}$ is a very local transformation. Conversion functions are only applied at the interface between a function which has been transformed and one which has not. Using *propagation rules* the transformation can be extended to eliminate the conversion function overheads in a large part of the program by making it operate entirely on the new type.

The types of the arguments and outputs of a function can be simultaneously transformed. The transformation can be applied to functions of any type, e.g. $f : t_1 \to t_2 \to \ldots \to t_n$. 

25
1.5.3 Summary

There are two strands to this work:

1. Determining which transformation to make and where to make it in the program. This is achieved by selecting transformations from a library, and using propagation rules to move the transformation to cover as much of the program as possible. This is discussed in sections 4.8.2 and 5.3.

2. Finding a new function $f'$ on the new type $t'$ which is equivalent to $f$ on $t$. This can be achieved by synthesis (§2.5) of $f'$ from a specification determined by the type change selected in (1). Difference matching (§6.4.1) and rippling (§3.4.5) are used to guide the proof. This is illustrated in § 6.8.

1.6 Conclusion

This chapter outlines a program transformation technique in which the types in the program are changed, and then the program modified to make best use of the new type. Search is always a problem for program transformation systems, so instead of transforming programs, synthesis proofs are transformed in the proofs-as-programs framework. This makes available certain information which is contained in proofs but not in the corresponding programs, and allows the use of proof planning to automate the transformation and reduce the problem of search.
2.1 Introduction

This chapter summarises previous work which is relevant to this thesis. First of all I give an overview of previous program transformation work. Most of it is based on the fold/unfold paradigm, which defines a number of very basic transformation steps which can be composed to form complex transformations. Very tight control over the transformation process is needed to reduce search to a minimum.

I then review previous work in problem reformulation and representation change. Work in this field ranges from quite general principles for solving problems expressed in logic through to very specific techniques for changing representations of data in programs.

I briefly discuss program synthesis, and introduce the logic which is used in this thesis for program synthesis and verification.

Finally, I discuss the notion of program transformation as proof transformation within the proofs-as-programs framework. The extra information which is present in a proof but not in the extracted program can be used to guide the transformation process, and we can make effective use of the techniques which
have been developed for constructing proofs. These techniques are discussed in chapter 3.

2.2 Program transformation

2.2.1 Introduction

*Program transformation* concerns the development of techniques by which programs can be manipulated to improve their efficiency without changing their desired behaviour. Program transformation allows programs to be developed in two steps: the production of an inefficient program which can easily be verified, followed by the application of transformations to derive an equivalent, but more efficient, program.

Much of the work in the field concerns the automation of program transformations.

In order to judge the effectiveness of a program transformation, we must define what is meant by "efficiency". Usually, one program is considered to be more efficient than another if it can be executed in fewer steps, i.e. it is faster. It is also possible to consider the space efficiency of a program, and this can be important. For example, modern functional languages often produce programs which consume huge amounts of heap space, and program transformations which can alleviate this problem can be useful.

Formal program development can be seen as program transformation: starting with an unexecutable specification of the program, a series of transformation steps is carried out until an executable specification (i.e. a program) is derived.

Most program transformation work is based on the fold/unfold principle of [Burstall & Darlington 77] which is outlined in the next section.
2.2.2 Fold/Unfold

Outline

[Burstall & Darlington 77] describes a system for specifying and transforming programs, expressed as recursive equations. Similar foundational work carried out around the same time is also described in [Manna & Waldinger 74]. Transformation consists of a series of applications of folds, unfolds, laws and abstraction. Once a function definition has been transformed, it is usual to delete the original version from the equation set.

An unfold step rewrites an expression using a function definition by replacing an instance of a function call in the expression with the body of the corresponding function definition after substituting the parameters used in the call for the formal parameters of the definition. For example, if

\[ f(x) := g(h(x), x) \] (2.1)

then unfolding the expression \( g(f(a + b)) \) using this definition produces:

\[ g(g(h(a + b), a + b)) \]

A fold step replaces an instance of a function body with the function head, again after substitution of actual for formal parameters. For example, the expression \( g(h(10), 10) \) can be folded using (2.1) to give \( f(10) \).

Application of a law rewrites an expression using a non-definitional equation, e.g. the associativity of +.

Finally, abstraction consists of the insertion of where clauses, replacing a term \( t[s] \) which contains a distinguished subterm \( s \) by a term containing a where clause (\( x \) is a new free variable which must not occur in \( t \) or \( s \)):

\[ t[x] \text{ where } x = s \]
See (§2.4.7) for an example illustrating development of a recursive definition using fold/unfold.

The fold/unfold technique has been adapted to the transformation of logic programs in [Tamaki & Sato 84], where much care is taken to ensure that termination of the transformed program is preserved.

Search in fold/unfold

The search space of applications of the fold and unfold steps is enormous. When application of laws is introduced the problem becomes still worse. Abstraction compounds the problem.

Care must be taken in order to avoid creating circular function definitions. For example, folding the right hand side of a function definition with itself creates an equation such as \( g(x) = g(x) \) which cannot be used as part of a program.

Fold/unfold can be used to perform a large variety of program transformations. With this flexibility comes the problem of making reasoning tractable. The original system of [Burstall & Darlington 77] was partly automated. The user selects which heuristics may be used, and occasionally the system asks the user to choose between several alternatives. In large problems, however, the user quickly becomes swamped with choices.

In the ZAP system [Feather 79], a simple meta-language is used to indicate which functions can appear in the final solution, which equations can be used to unfold, what lemmas are available, and to specify the final form of the solution. Given a starting expression to transform and a final form, the system fully expands both using the allowed unfolds and laws. If at some point the two expansions match, then a fold/unfold transformation from the starting expression to the final form can be performed by unfolding the starting expression up to the point of the match, then folding back along the path towards the final form. The final form may contain function variables which become determined during the matching process. This system is used in [Darlington & Feather 80] to develop executable programs from (possibly unexecutable) NPL specifications.
The fold/unfold process is still only loosely constrained, and the size of the search space is a problem.

The major problem faced by fold/unfold systems is control of the search space. The proof planning framework (chapter 3) allows the expression of powerful heuristics. Rippling is a very strong heuristic, accurately modelling the shape of inductive proofs, and often eliminating search entirely. In chapter 6 I show how rippling is used to guide part of the transformation process.

2.3 Changes of representation

2.3.1 Introduction

There are at least three reasons why one may wish to change the representation of some data/operations:

1. To facilitate reuse of existing problem-solving methods.

2. For integration of data (e.g. converting integers to real numbers for performing real number addition)

3. To improve the efficiency, since for given tasks some representations can be manipulated more efficiently than others.

The general principle that different problem-solving methods are suitable for different problems has been explored by several authors. Figure 2–1 illustrates the spectrum covered by work in this area.

At one end of the spectrum is work concerning general principles of representation change. The main technique is to alter the solution search space by incorporating problem properties into the representation and forming macro-operators. At the other end there are systems for changing data types in programs. An implementation is chosen for a data type according to the way it
## Incorporation of Constraints

<table>
<thead>
<tr>
<th>Logic</th>
<th>Translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pure problem reformulation. Macro-operators.</td>
<td>Korf, Amarel, Riddle</td>
</tr>
<tr>
<td>Composing predefined reformulations.</td>
<td>Van Baalen</td>
</tr>
<tr>
<td>Logic program transformation. Incorporation of problem properties into ADTs.</td>
<td>Lowry</td>
</tr>
<tr>
<td>Applying predefined data type reformulations to programs. Synthesis of efficient functions. Proof transformation.</td>
<td>Richardson</td>
</tr>
<tr>
<td>Algebraic data type reformulations. Translations.</td>
<td>DTRE</td>
</tr>
<tr>
<td>Fixed data type refinements in an Algol-like language.</td>
<td>Low</td>
</tr>
<tr>
<td>SETL set refinements. Data flow analysis</td>
<td>Schonberg</td>
</tr>
</tbody>
</table>
is used in the program, and operations on the original data type are translated into operations on the new data type.

When an implementation (or refinement) is chosen for an abstract data type, a translation mapping is given which maps elements of the old data type to their equivalents in the new type and functions on the old type to functions on the new. The extra information available in the implementation can often be used to create efficient implementations for compositions of functions from the original type. Systems based on translation usually translate only the primitive functions specified in the original abstract data type and so miss such efficient implementations of compositions (§9.2.2).

Lowry (§2.4.5) combines the two approaches, generating a new data representation by determining and incorporating properties of the original representation, and then applying a translation to the program.

2.3.2 Representation change in problem solving

There is a large body of work concerned with finding ways to reformulate a problem to use a better representation. A typical problem from this field is the "Missionaries and Cannibals" problem discussed in [Amarel 68]. A simple representation for this problem consists of 3 pairs of variables — specifying how many missionaries and cannibals there are at any time in the boat and on each bank of the river. Better representations are found by incorporating some of the problem constraints into the representation.

In [Amarel 68], it is shown how a problem representation can be improved by exploiting properties such as regularities of the problem. It is suggested that a way to form improved representations is by identifying "easily traversable areas" of the problem search space. Points in the search space which are on the edge of two easily traversable areas ("narrow") are critical. Macro-operators are formed which move between narrows. Then an abstract representation is found in which the points are the narrows, and the basic operators are the macro-operators of the original representation.
Automation of these techniques is developed in [Riddle 90]. In this work, reformulation is based on determining the path in the problem state space of a training example: a single problem query and its solution. This path is decomposed into a preparation subproblem, a critical subproblem, and a remainder subproblem. Both the preparation subproblem and the remainder subproblem are defined in such a way that they are trivially solvable. The critical subproblem is then further decomposed, eventually yielding a parse tree of the solution path, in which the leaves are either trivially solvable, or are irreducible subproblems. Macro-operators are formed to solve these irreducible subproblems. Subsequent problems are solved by using the macro-operators, which are not however compiled into the problem representation. Much of this work is concerned with generalising from the training example.

Korf [Korf 80] defines two aspects of representation change:

1. Isomorphisms — information-preserving restructuring of the data

2. Homomorphisms — information-destroying

He shows how even drastic representation changes can be decomposed into minor representation changes along these two axes.

An important way of finding good representations is to find problem properties and incorporate them into the representation.

2.3.3 Van Baalen

The DRAT system of [Van Baalen 92] attempts to solve problems formulated in first order predicate calculus by incorporating constraints into a representation. It uses a library of simple representation schemes, for example representing a concept as a symmetric-relation. The library is organised hierarchically using specialisation links. For example, equivalence-relation is linked to symmetric-relation, reflexive-relation and transitive-relation. The system attempts to push a representation as far down the scheme hierarchy as
possible. Each representation scheme has associated methods for performing inference with it. The reformulated problem is solved using a combination of these methods and a general theorem prover. This results in good solutions for constraint satisfaction problems, but the system cannot reason about inductive properties of the problem, and it cannot solve problems which require proof by induction.

2.4 Representation change in programs

2.4.1 Introduction

Programs perform a particular kind of problem-solving activity, so the general principles discussed above are applicable. More specific techniques have also been developed, mostly based on the selection of implementations for abstract data types.

In this section, I briefly describe what an abstract data type (ADT) is and outline previous approaches to ADT refinement.

2.4.2 Definition of an abstract data type

An important theme in modern programming languages is modularity. An abstract data type (ADT) achieves modularity by packaging a data type with its properties and the functions which access it.

An ADT is composed of the following:

1. A set S of sorts. This set may be divided into observable and unobservable sorts. Observable sorts are accessible outside the ADT. Unobservable sorts can only be used within the ADT definition.
2. A signature $\Sigma$. This is a set of operation declarations with domain $s_1 \rightarrow \ldots \rightarrow s_n$, $s_i \in S$, and range $s \in S$. If the domain is empty then the operation is a constant.

3. A set $E$ of equations, which are equalities between terms over the sorts and operations of the ADT.

To each ADT there corresponds a class of models of the ADT. A model is a mathematical structure which provides the sorts and functions required by $S$ and $\Sigma$, and which satisfies the equations in $E$, together with an interpretation mapping elements of the ADT to objects in the model. For example, binary numbers and unary numbers are both models of the ADT for the natural numbers.

### 2.4.3 ADT development

A model of the development of an abstract data type is the ENRICH—FORGET—IDENTIFY process [Broy et al 86]. Briefly, these steps are:

1. ENRICH — structure is added to the data type by adding sorts to $S$, functions to $\Sigma$, or equations to $E$. The original data type can easily be extracted from the new one by deleting the extra structure.

2. FORGET/HIDE — structure (for example redundancy introduced by (1)) can be removed from the type.

3. IDENTIFY — elements of the type can be identified, for example identifying permutations of lists to create bags. This corresponds to an enrichment of the equality of the type.

In general, dramatic changes in a data type can be achieved using these steps. Enrichment of a type restricts the class of models of the type since it requires the models to have more sorts/functions or to satisfy more equations. Forgetting extends the class of models of the type.
In this formalism, program development starts with an ADT as a specification. Development consists of the gradual refinement of the specification, enriching and forgetting, until a satisfactory executable specification is arrived at.

2.4.4 Abstraction and Implementation

In the ADT literature there is a standard notion of implementation. An abstract data type $T_2$ implements an abstract data type $T_1$ if the models of $T_2$ are a subset of the models of $T_1$. So enrichment is implementation.

The inverse of implementation is abstraction. A more restrictive model of data type development is of abstraction followed by implementation. This is the model followed in [Lowry 89]. For example, we can move from a decimal to a binary representation of the natural numbers by first abstracting decimal to the ADT for $\text{IN}$, then implementing this ADT using binary.

In practical application, it is desirable for the data type obtained after abstraction and implementation to be an implementation of the original data type, but this need not be the case.

According to [Bidoit et al 91], more than twenty notions of data type refinement have been defined. In [Sannella & Tarlecki 92], a constructor implementation $SP \xrightarrow{\rho} SP'$ is defined such that $\rho$ maps models of $SP'$ to models of $SP$. $\rho$ has some computational content. If the algebra $A'$ is a model of $SP'$ then $\rho(A')$ is a model of $SP$. This implies that $\rho(A')$ has sorts and functions corresponding to those of $A'$, and that the axioms of $SP'$ are provable in $SP$.

2.4.5 Lowry

[Lowry 89] addresses the problem of automatically generating an improved problem representation given a logical specification of the problem, in order to make the design of an algorithm for solving the problem easier. A problem reformulation can be split into two parts:
1. Abstract the data types in the original specification.

2. Implement these abstracted data types in a concrete domain.

The system has a limited library of pre-defined implementations, which are mappings from an abstract representation (data + functions and predicates over those data) to a concrete one.

Problems are specified as equational theories. Lowry's system works from a specification in several phases:

1. Abstract the theory. This is done in one of two ways:

   (a) Apply the inverse of an implementation map, or

   (b) Incorporate problem properties:

      i. A variety of methods are used to hypothesise some behavioural properties of the problem. Behavioural properties are expressed as equations, e.g. \( p(x, y) = p(y, x) \).

      ii. Verify the property using a theorem prover.

      iii. Abstract the domain theory by incorporating the problem property, such that it becomes a property of the entire theory, not just the problem being reformulated. This leads to a more abstract domain theory. For example, if the problem specifies the length of a list, then one problem property is that \( \text{length}(a :: l) = \text{length}(b :: l) \). Incorporating this problem property abstracts the theory of lists to the theory of natural numbers because it is the lengths of lists, not the identities of their elements, which matter in the new theory. A theorem prover carries out reasoning to determine when it is possible to add a particular equation without over-abstracting the theory.

2. Design (synthesize) an algorithm to solve the abstracted problem. For this Lowry uses the RAINBOW system of Douglas Smith [Smith 85]. Smith's technique attempts to transform the problem using conditional rewrite
rules and rules for decomposing a composite specification into one to which a predefined design strategy can be applied.

3. Implement the abstract data types with concrete ones.

The abstraction/implementation model of reformulation allows complex transformations to be broken into simpler components. For example, a small representation library could contain the transformations in figure 2–2.

2.4.6 DTRE

The DTRE system of [Blaine & Goldberg 91] is intended as a module of the KIDS system [Smith 90]. It performs data type refinement in programs and can generate code in a variety of programming languages.

A parameterised theory is an ADT coupled with a theory for that ADT. An interpretation from theory $T_1$ to theory $T_2$ is a translation from the sorts, functions etc. of $T_1$ to the sorts, functions etc. of $T_2$ such that the translations of the axioms of $T_1$ are provable from those of $T_2$. The translations of terms of $T_1$ to terms of $T_2$ can be extended naturally to translations of arbitrary propositions. The interpretation also induces a mapping from models of $T_2$ to models of $T_1$. Correctness is established once for each implementation.

Partial implementation is allowed, which may impose conditions on each use of a data type.
The user annotates a program or specification to indicate how a particular data type should be implemented. The system then uses data and value flow analysis, operations analysis, symbolic containment and size bounds on objects to select further implementations.

Transformation consists of the exhaustive application of transformation rules: normalisation rules, translation rules derived from the interpretation and back-stop rules which attempt to create a new rewrite rule when a translation rule does not exist.

Additional refinement steps produce code in C, Common LISP or ADA. Development has concentrated on implementations of sets.

2.4.7 Reformulation using retrieve/representation functions

[Darlington 80] describes how functions defined over an abstract data type can be transformed into corresponding functions operating over an implementation of that abstract data type. ADTs are specified equationally within the programming language NPL.

Given two such abstract data types, abs and imp, an implementation relation is established between them by defining a function \( \text{rep}: \text{imp} \rightarrow \text{abs} \). The equations defining a function over abs can then be transformed by folding and unfolding them with the equations for rep and any other laws (lemmas) which may be available. The objective is eventually to remove references to the representation function from the equations and arrive at definitions entirely in terms of the implementation. The technique is illustrated with examples showing implementation of queues as circular lists, and queues as arrays.

As an example, I reproduce here the development of a function addcirc which is the implementation on circular lists of the function add on queues.

The function \( \text{addcirc} : \text{circ.list} \rightarrow \text{int} \rightarrow \text{circ.list} \) is specified using the representation function:
\[
\text{rep}(\text{addcirc}(c, i)) = \text{addq}(\text{rep}(c), i) \quad (2.2)
\]

The following equations are available in the definition of the abstract data types for queues and circular lists:

\[
\text{appendq}(q, \text{addq}(r, i)) = \text{addq}(\text{appendq}(q, r), i) \quad (2.3)
\]

\[
\text{rep}(\text{insert}(\text{circ}, i)) = \text{appendq}(\text{addq}(\text{emptyq}, i), \text{rep}(\text{circ})) \quad (2.4)
\]

Recursive equations for addcirc are now developed by folding and unfolding with the equations defining rep and the functions on queues. First, an equation defining addcirc for the empty circular list is developed. This yields equation (2.5) below.

Then the recursion equation is developed:

\[
\text{rep}(\text{addcirc}(\text{insert}(c, i_1), i_2)) = \text{addq}(\text{rep}(\text{insert}(c, i_1)), i_2)
\]

Unfold with (2.4) \[= \text{addq}(\text{appendq}(\text{addq}(\text{emptyq}, i_1), \text{rep}(c)), i_2)\]

Fold with (2.3) \[= \text{appendq}(\text{addq}(\text{emptyq}, i_1), \text{addq}(\text{rep}(c), i_2))\]

Fold with (2.2) \[= \text{appendq}(\text{addq}(\text{emptyq}, i_1), \text{rep}(\text{addcirc}(c, i_2)))\]

Fold with (2.4) \[= \text{rep}(\text{insert}(\text{addcirc}(c, i_2, i_1)))\]

Since \(\text{rep}(x) = \text{rep}(y) \iff x = y\), it is consistent to drop the occurrences of rep to give equation (2.6) and define addcirc by the following equations:

\[
\text{addcirc}(\text{create}, i) = \text{insert}(\text{create}, i) \quad (2.5)
\]

\[
\text{addcirc}(\text{insert}(c, i_1), i_2) = \text{insert}(\text{addcirc}(c, i_2), i_1) \quad (2.6)
\]

The entire development takes place in a system based on unfold/fold. No particular control strategy is used in the development, and so development suffers from the search space control problems of fold/unfold. For example several of the equations from the ADTs used in the example above can be used to fold arbitrarily many times. The search space for this example is analysed further in §9.3.1. In §9.3.1, I compare the search space for a fold/unfold proof derivation with that for the proof strategy I define in §6.
Although recursive equations are developed, there is no control to ensure that a well-founded definition has been created. In the proofs-as-programs paradigm (§2.5.3), the conditions which guarantee the soundness of induction as a proof step do ensure that well-founded definitions are created.

Darlington notes that not only functions which are in the signature of the original data type, but also other functions which are compositions of the signature functions, can be transformed in this way. This can produce more efficient programs than could be obtained simply by transforming the signature functions and then using those implementations in the program (see §9.2).

### 2.4.8 Automatic Data Structure Selection

**Low**

In [Low 78], Low presents a system for choosing suitable representations for the abstract data types in an Algol-60-like programming language. The starting point is a large library of representations — each representation has associated definitions of its primitive operations (access/update functions etc.) and cost functions which estimate the time and space costs of applying each primitive to the given representation given the size etc. of the data elements.

The approach taken is an empirical one — statistics are gathered from a run of the program on a typical user input using the default representations for the abstract data types. Static flow analysis determines which variables should have the same representation. The system chooses appropriate data structures by minimising the cost functions. No attempt is made to ensure that the implementations are formally correct. Complex run-time verification conditions, e.g. bounds checking, probably cannot be checked.

Low makes some good points:

1. When choosing a representation for an abstract data type, it is necessary to know how the data structure is used within the program, for example, which primitives are used and how often.
2. An abstract data structure may be implicitly represented. For example, the set of all odd integers may be best represented by a predicate.

3. Several different representations for a data structure may be used in different parts of the program.

4. A structure can be represented redundantly by storing the same data in more than one representation.

These points are discussed further in §9.2.1.

Selection of set implementations in the SETL system

SETL [Schwartz 86] is a high-level programming language supporting set-theoretic syntax and semantics. Efficiency of set operations is a major concern. SETL provides several ways of implementing sets, known as bases:

- bit strings,
- indexed extensible bit strings,
- linked hash tables.

The programmer may specify which representation is to be used by declaring a set appropriately. [Schonberg et al. 81] describes how the implementation of a set may also be selected automatically. Provisional representations are selected for occurrences of set variables based on the local context in which they appear in the program. These provisional representations are propagated globally and merged.

The algorithm consists of the following phases:

1. base generation,

2. representation merging and base equivalencing,
3. base pruning and representation adjustment,

4. conversion optimisation.

*Base generation* performs a linear pass through the code introducing representations for the arguments of each program instruction so that the execution of the instruction with the chosen representations is no slower than the execution assuming a generic representation.

*Representation merging and base equivalencing* applies data flow analysis to propagate the chosen representations and merge them where necessary, so if a function call \( f(s) \) indicates that \( s \) should be implemented by a bit string \( "x_1x_2...x_n" \) of fixed size \( n \), and \( g(s) \) indicates that it should be implemented by a bit string \( "y_1y_2...y_m" \) of size \( m \), then a bit string of size \( k \leq m + n \) which is a substring of \( "x_1...x_ny_1...y_m" \) would be chosen.

The representation merging phase may result in some sets being implemented in a way which is no more efficient than the default representation. These useless representations are removed and the sets are given a baseless implementation in the *base pruning and representation adjustment* phase.

*Conversion optimisation* uses a simple dataflow analysis to determine where in the program to convert a set \( s \) between the different representations chosen for it in the program. This includes moving conversions out of a loop if there is at least one conversion which takes place inside the loop and is made redundant by making the conversion before entry to the loop.

The authors of the system note that in general run-time information such as frequency and bounds information could be used to further refine the choice of set representation. However, this is not used because the aim is to produce a fully automatic system.

### 2.4.9 Data structure mapping and difference lists

In Prolog, a *difference list* (see [Sterling & Shapiro 86]) is a pair \((X, Y)\), whose second component is a suffix of the first, i.e. \( X = x_0...x_n.Y \). The pair
\(<x_0,\ldots,x_n,Y,Y)> \) represents the list \([x_0,\ldots,x_n]\). The pair \(<x_0,\ldots,x_n,X,Y)> \) where \(X\) and \(Y\) are unequal uninstantiated variables represents a variable-length list \([x_0,\ldots,x_n]|\). Difference lists are especially useful when a program needs to append to the end of the list, because the second element of the pair acts as a pointer to the end of the first element. A particular example of this use is the definition of the append/3 predicate. For difference lists, this is reduced to the single fact:

\[\text{append}(\langle X,Y\rangle, \langle Y,Z\rangle, \langle X,Z\rangle).\]

In [Hansson & Tärnlund 82], logic programs manipulating difference lists are derived from the corresponding programs on lists, and equations specifying translations from lists to difference lists and vice versa.

[Zhang & Grant 88] describes an algorithm for automating the replacement of lists with difference lists in a Prolog program. It is based on a tightly-constrained fold/unfold transformation combined with partial evaluation of the clauses which are generated and application of the associativity of append. Only lists which appear as arguments of recursive predicates are transformed. This is a restriction since lists which appear as arguments of functors in the predicate head will not be transformed. Non-recursive calls are not transformed. It is noted in the paper that the algorithm may produce a program which is only partially equivalent to the original. This point is illustrated by considering the naïve reverse program supplemented by the additional fact \(\text{rev}(0,1)\). This would be transformed into a program equivalent to the naïve reverse program, i.e. the solution \(\text{rev}(0,1)\) would be lost.

[Marriott & Sondergaard 88] examines the pitfalls of such a simple transformation. There are several sources of problems which mean that a simple transformation from lists to difference lists does not always yield an equivalent program. Unification for difference lists does not correspond to unification for lists, for example, \((a.nil, nil)\) and \((a.b, b)\) both represent the list \([a]\), but do not unify, whereas \((X,X)\) and \((a.Y,Y)\) do unify but the lists they represent \((\text{nil} \text{ and } a.Y)\) do not. In addition, some difference lists do not represent any
list, e.g. (a.nil, b.nil). The paper develops a more sophisticated transformation which introduces additional predicates to the program, for instance an extended equality test, which yields a safe transformation. It shows how for most simple programs these extra predicates can be removed by an analysis based on data-flow analysis. For more pathological programs, some non-logical runtime tests must be included to preserve program equality.

2.5 Program synthesis

2.5.1 Introduction

Program transformation allows the development of an efficient program by a sequence of manipulations of an original, inefficient program.

Program synthesis addresses the problem of developing an executable program from a specification of its behaviour, and verifying that the program does meet the specification. This can be seen as an extreme form of program transformation, in which an (unexecutable) specification is transformed into an executable one.

2.5.2 Knowledge-based program synthesis

One approach to program synthesis is to use a large library of refinement steps, which specify how a complex specification can be broken into several simpler ones. KIDS [Smith 90] is one such system. The knowledge base contains general algorithm design tactics, specifying algorithms such as divide-and-conquer and global search algorithms. The degree of automation is largely dependent on the extent of the domain theory for the problem being solved, and construction of such a theory is the main task in performing a synthesis.
Program synthesis is naturally a theorem proving activity. Constructive logic, described in the next section, allows programs to be developed as a by-product of a constructive proof that the specification is satisfiable.

2.5.3 Constructive synthesis in Type Theory

The logic of Constructive Type Theory provides a formal system in which specifications, programs, proofs and transformations can be expressed in a uniform way.

Constructive logics in general do not admit the Law of the Excluded Middle ($a \lor \neg a \equiv \text{true}$) as an axiom. In addition, from a proof of a proposition $\exists x . p(x)$ it is always possible to determine an $x$ such that $p(x)$ holds.

Martin-Löf's Constructive Type Theory is a typed logic. Every proposition is a type. A proof of a proposition is a (lambda-calculus) program of that type. For example, a proposition $a \supset b$ is a function type $a \rightarrow b$, and a proof of such a proposition is a function of this type.

A sequent is a term $H \vdash G$ in which $G$ is a well-formed formula of the type theory, and $H$ is a (possibly empty) list of well-formed formulae, the hypotheses. A proof of a sequent is a demonstration using the inference rules of the logic that $G$ holds if all the formulae in $H$ hold.

Proofs in Martin-Löf's Type Theory typically contain a large number of sub-goals verifying that terms have the appropriate type.

Each rule of inference has a corresponding rule of program construction. For example, in the two inference rules below, $\land$ introduction and $\forall$ introduction, the sequents are subscripted by program fragments extracted from their proofs:

\[
\begin{align*}
\Gamma, \phi \vdash A & \quad \Gamma, \psi \vdash B \\
\Gamma, \phi \psi \vdash A \land B & \quad \text{\textit{$\wedge$-intro}} \\
\Gamma, \phi \psi x : t \vdash A & \quad \text{\textit{$\forall$-intro}} \\
\end{align*}
\]

Consider a proposition such as the following:
\[ \forall x:t. \exists y:u. \text{spec}(x,y) \] (2.7)

A proof of this specification ensures that every \( x \) has a corresponding \( y \) such that \( \text{spec}(x,y) \) (but in general we do not know what this \( y \) is). Since we are using a constructive logic, however, we can extract from the proof a program which will compute such a \( y \) for any given \( x \). Thus we can view (2.7) as a specification of such a program, \( f(x) \), which will satisfy:

\[ \forall x:t. \text{spec}(x,f(x)) \]

In proving that the specification is satisfiable we produce, or synthesise a program which satisfies the specification. This proofs-as-programs paradigm [Constable 82] has recently been adapted to the synthesis of logic programs [Wiggins et al 91].

For example, the following is a specification of a function to find the greatest element of a list of natural numbers:

\[ \forall l: \text{nat list}. \exists m: \text{nat}. (\forall n: \text{nat}. \text{member}(n,l) \rightarrow n \leq m) \] (2.8)

Induction in the proof corresponds to recursion in the synthesised program. For example, structural induction on the list \( l \) in (2.8) produces a proof with two branches:

1. A base case:

\[ \exists m: \text{nat}. (\forall n: \text{nat}. \text{member}(n,\text{nil}) \rightarrow n \leq m) \]

A proof of this branch yields the value of the extracted function on \( \text{nil} \),

2. A step case:

\[ l: \text{nat list} \]
\[ h: \text{nat} \]
\[ \exists m: \text{nat}. (\forall n: \text{nat}. \text{member}(n,l) \rightarrow n \leq m) \]
\[ \exists m: \text{nat}. (\forall n: \text{nat}. \text{member}(n,h::l) \rightarrow n \leq m) \]
A proof of this branch yields a recursion equation calculating the value of the extracted function on \( h \) given its value on \( l \).

When a term is explicitly introduced for an existentially quantified variable, \( y \), it is called the *existential witness* for \( y \).

The program synthesis rule associated with the induction proof rule puts these two building blocks together to give a recursive function.

Only total, terminating functions can be verified or synthesised.

### 2.5.4 Program transformation vs. program synthesis

When a program is synthesised or verified in Martin-Löf's Type Theory, application of a proof rule produces subgoals whose proof ensures that the extracted program is total and terminating. In the fold/unfold framework, recursion is only introduced into the transformed program by folding, but this takes place without the program termination guarantees given by induction in proofs-as-programs.

In the compiling control work of [Bruynooghe et al 89], a novel method of transforming Prolog programs is described which involves two phases: recording the execution path of a Prolog program using an enhanced evaluation strategy, followed by compilation of this trace into a new program which has the same execution pattern when executed with the standard Prolog execution strategy as the original did with the extended strategy.

Much of the difficulty in the transformation process is in spotting loops in the program trace and encoding them as recursive predicates. In my opinion, [Wiggins 90] shows how the duality in constructive type theory (§2.5.3) between the proof of a specification and the extracted program can be applied to compiling control: the initial execution phase is analogous to proving the program specification (possibly guided by program annotations), and the second phase to the extraction of a program from the proof. The problem of spotting loops
becomes the more controllable and better understood problem of performing inductive proof.

Inductive proof provides a powerful and appropriate framework for reasoning about recursive structures, e.g. programs, and the mechanisms which have been developed for guiding inductive proof are very valuable in tackling the search problems inherent in program transformation and synthesis.

2.5.5 Formal program development from algebraic specifications

[Sannella & Tarlecki 92] discusses the derivation of correct programs from algebraic specifications in general terms. The primary concern is correctness. Efficiency is a secondary issue, and no formal language is provided for discussing it.

Specifications are viewed as algebras, and development/refinement steps as functionals between algebras.

The paper outlines a theory which defines specification refinement operations which are powerful and have desirable mathematical properties such as compositionality. In common with some previous work in the field, a powerful set of kernel operations on specifications is defined, which can, in turn, be used to define operations which are more specific and amenable to practical application.

A formulation in a type theory of this method of program development is outlined in §4.4.3.
2.6 Proof Transformation

2.6.1 Applications of proof transformation

In [Kawai 81], a graph-marking algorithm is verified by first verifying the algorithm on a simple tree data structure, and then transforming (by hand) both the algorithm and the proof to derive and verify a more efficient algorithm. Similarly, in [Bidoit & Corbin 83], an inefficient unification algorithm on trees is modified to manipulate graphs, and the correctness proof of the original is similarly modified to a correctness proof for the new, efficient algorithm. These two papers illustrate one possible use of data type transformation: the derivation and proof of a complex efficient algorithm by transforming an initial simple but inefficient algorithm and proof.

Another application of proof transformation is in turning verbose and unnatural (computer generated) proofs into proofs which are better structured and more amenable to human understanding [Lingenfelder & Pracklein 90].

2.6.2 Program transformation as proof transformation

In the following sections I outline some of the merits of proof transformation. Program transformation often requires detailed analysis, for example, identifying recursive functions, or performing dataflow analysis. Much of this information is present in a synthesis proof, but not in the extracted program. It is possible to make more profound changes to a program by transformation of its synthesis proof than are allowed by the application of correctness-preserving program transformations.

For these reasons, amongst others discussed below, I have chosen to automate the type change transformation as a proof transformation rather than a program transformation.
Goad’s program specialisation

Goad [Goad 80] develops the notion of proof transformation in order to automate the specialisation of programs. Instead of transforming a program, a synthesis proof is transformed. This allows the transformation process to use extra information from the synthesis proof which is not contained in the corresponding program. Much of this information consists of identification of dependencies between the computation being performed and certain hypotheses/assumptions accumulated during execution. In a program, only data is passed through the program execution. In the proof, some of its properties are carried down too. In particular, a case split in the proof (which corresponds to an \texttt{if}... \texttt{then} statement in the extracted program) adds a hypothesis to each of the subsequent branches of the proof.

The simple example of figure 2–3 shows how a case split on $\forall a, b, n : \text{nat}. a + b < n \lor a + b \geq n$ allows part of the subsequent execution to be avoided.

\begin{figure}[h]
\centering
\begin{tikzpicture}
  \node {a+b<n \lor a+b>n}
  \node[below] {a+b<n}
  \node[below] {Prune!}
  \node[below left] {a+b<n \lor a+b>n}
  \node[below] {a>n}
\end{tikzpicture}
\caption{Dependency information can allow pruning of parts of the computation.}
\end{figure}
Goad demonstrates that the program extracted from the transformed proof may compute a \textit{different function} to the original, though it must still satisfy the original specification. This makes possible transformations not allowed by correctness-preserving program transformations. Goad demonstrates his method by automatically optimising a bin-packing algorithm by fixing certain parameters, for example the number of bins, and pruning. A simplex algorithm uses the dependency information in the proof to prune certain branches. This typically produces a function which is faster than the original unpruned function by a factor of two, and computes a different function while still satisfying the specification. For example, a specialised bin-packing algorithm for the case of packing two blocks, of size \(i_1 \geq i_2\) respectively into two bins of size \(n\) is:

```plaintext
if \(i_1 \leq n\) then
  if \(i_1 + i_2 \leq n\) then <1,1>
    else
      if \(i_2 \leq n\) then <1,2>
        else
          if \(i_1 \leq n\) then
            if \(i_2 \leq n\) then <2,1>
              else
                if \(i_1 + i_2 \leq n\) then <2,2> else FAIL
              else
                \text{FAIL}
          else
            \text{FAIL}
      else
        if \(i_1 \leq n\) then
          if \(i_2 \leq n\) then <2,1>
            else
              if \(i_1 + i_2 \leq n\) then <2,2> else FAIL
            else
              \text{FAIL}
```  

Application of pruning and simplex optimisation produces the following expression, which satisfies the original specification but computes a different function:
if \( i_1 \leq n \) then \(<1,2>\) else FAIL

This function is the fastest which satisfies the specification, and cannot be derived by correctness-preserving program transformations on the original function.

Madden

In [Madden 91], Madden exploits the extra information available in a synthesis proof to automate a tupling transformation. This is illustrated by its use to transform the naive Fibonacci definition, which executes in exponential time, into a linear-time version. In a conventional fold/unfold program transformation system, finding the correct tuple requires extensive analysis of the program's execution path. In Madden's work, the induction step witness provides the information needed to determine the tuple.

Madden extracts the following information from the source proof:

1. The branching structure of the proof.

2. The rules applied, with their arguments.

3. An account of dependencies between facts in the proof:

   (a) interrelations between subgoals and

   (b) interrelations between subgoals and hypotheses.

It is this abstraction which is transformed, and then used to construct a target proof. Some useful information is also available in the extracted program, partly because the act of synthesis forces the extracted program to have a certain form.
Automatic generation of DELAY declarations

DELAY declarations can be generated for a logic program by identifying the recursion variables. This requires dataflow analysis, and is not always possible. [Wiggins 92] describes how information contained in logic program synthesis proofs (which is absent in programs) can be used to generate DELAY declarations automatically. DELAY declarations are attached to the induction variables in the synthesis proof.

Advantages of proof transformation

Proof transformation has several advantages over program transformation:

1. Extra information contained in the synthesis proof can be used to guide the transformation process.

2. Correctness of the transformed synthesis proof is guaranteed by the soundness of the proof system.

3. Programs derived from synthesis theorems are firmly grounded in logic, thus avoiding contact with extra-logical features such as Prolog's cut and var/1, as well as more subtle logical anomalies. For example, this avoids the unsoundness which was identified in [Marriott & Sondergaard 88] of the difference-list transformation.

4. There is always access to the original program specification, which allows the construction of a new proof of the specification which yields an extracted function with different behaviour to the original. I exploit this in §7.10, for example.

5. If the entire synthesis process is seen as a theorem proving task, then we can bring the full power of the theorem prover to bear on the problem. In particular, we can use proof planning techniques.
2.7 Conclusion

In this chapter I have introduced some of the literature on program transformation, on changes of representation both in problem solving in general and in programs in particular.

Program transformation is mainly based on the fold/unfold paradigm. The central problem is one of guidance, since the space of possible fold/unfold transformations is large.

I briefly discussed program synthesis and Martin-Löf’s Type Theory.

Some of the early work on change of representation in problem solving identified the incorporation of problem properties as a powerful technique for changing the problem solving search space. This idea was exploited by Lowry, who uses a theorem prover to identify and incorporate properties of a data representation. Lowry also allows predetermined representation changes to be made. Several other works develop limited transformation systems which apply such predetermined data type changes.

When a synthesis proof is transformed, the program extracted from the new synthesis proof is correct as long as it satisfies the original specification, i.e. as long as the new synthesis proof really is a proof of the original specification. It may compute a different function to the program extracted from the original synthesis proof. This allows transformations which are not possible by the application of correctness-preserving program transformations.

Proof transformation has other applications apart from program transformation, and these were discussed.

In order to ease the problem of proving correctness of the transformation, allow the interleaving of the type change method with more general theorem proving techniques, and make use of information available in synthesis proofs, I prefer to transform program synthesis proofs instead of programs themselves.
Chapter 3
Planning Program Synthesis
Proofs

3.1 Introduction

A major problem in program transformation and synthesis is search control. The techniques described in this chapter give us the tools to tackle this problem effectively by providing us with powerful techniques such as rippling (§3.4.5) and with a way of encoding heuristics.

When constructing a proof, there are typically many inference rules which can be applied at any given point during the proof, and proofs are normally quite deep. The search space of possible proof attempts is therefore very large.

Proof planning (§3.3) reduces the size of the proof search space because the steps (methods) from which a proof plan is constructed are larger than those from which the object level proof is constructed, and because sequents are annotated to provide guidance for the theorem proving process.

The proof planning techniques introduced in this chapter provide the basis for the automation of type changes. Heuristics for carrying out the proof are encoded by methods.
Of particular interest is the proof strategy for induction, because induction in a synthesis proof corresponds to recursion in the extracted program, and this allows us to reason effectively about programs. The rippling strategy is very effective for planning inductive proofs, often eliminating search entirely.

Once a successful proof plan has been found, a procedure which will prove the theorem in the object level logic is automatically constructed using tactics. The logic used is Martin-Löf’s Type Theory.

This chapter is organised as follows: I give a description of the Oyster theorem prover, and compare the sizes of the proof spaces at the level of individual inference rules and of tactics. I then describe how the object level logic is abstracted at the meta level. CLAM constructs proofs at the meta level. I describe the proof strategy for induction, which is vital to the work in this thesis, and finally describe some of the methods which make up this proof strategy.

### 3.2 Program construction in type theory

#### 3.2.1 Introduction

An essential requirement of any program transformation system is that the transformations it applies must be provably correct. In most previous program transformation work this means that the transformed programs are equivalent to or specialisations of the originals. This criterion is generalised by the transformation work described in §2.6 to a requirement that the transformed synthesis proof obeys the same specification as the original.

The proof/program duality of Martin-Löf’s Constructive Type Theory (§2.5.3) makes it very well suited to program synthesis, verification and transformation. The correspondence between induction in the proof, and recursion in the extracted program, makes the study of techniques for inductive proof vital for the production of recursive programs.
3.2.2 The Oyster implementation of Martin-Löf’s Type Theory

Oyster [Horn & Smaill 90] is a goal-directed theorem prover which implements a variant of Martin-Löf’s Constructive Type Theory [Martin-Löf 79]. Oyster originated as a Prolog version of NuPRL [Constable et al 86].

The state of a partial proof is maintained as a proof tree, with the original conjecture at the root. Each node in the tree is labelled with a goal which is to be proved. The rules of inference of the logic are used to break these goals down into simpler subgoals, and each node is also labelled with the rule of inference applied which produced its children. When a rule of inference proves the goal at a node, there are no subgoals and this node is labelled as completed. Leaves which are not yet complete are open subgoals. Figure B-2 (appendix B) shows part of an Oyster proof tree.

Proofs and definitions are stored in a library, and can be loaded into Oyster for use in proofs, e.g. as lemmas.

3.2.3 Tactics

The proof rules for Martin-Löf’s Type Theory are quite low-level, and proofs constructed solely from the basic rules of inference are large and hard to understand.

A tactic is a procedure which constructs a part of the proof. When a tactic is applied to a goal in the proof, one branch is created for each subgoal which has not been proved by the tactic, and the node is labelled with the tactic, not with the individual proof rules the tactic applied.
3.3 Proof planning, the meta-level and CL\textcopyright M

3.3.1 Proof planning rationale

A discussion of the ideas behind proof planning is contained in [Bundy 91]. Some of the arguments in this section are based on those in that paper.

Any proof can eventually be reduced to a sequence of applications of the rules of inference of the logic and the theory in which the proof is taking place.

Such a sequence of inferences has the desirable property that one can ascertain whether or not it constitutes a proof. It is, however, extremely low-level and does not generally satisfy other desirable criteria, namely that it be:

1. Easy to generate.

2. Easy to understand.

Proof planning addresses these issues by providing a higher-level view of the proof.

3.3.2 Methods are partial specifications of tactics

A method is a partial specification of a tactic. It consists of several slots:
<table>
<thead>
<tr>
<th>Input</th>
<th>Meta-level sequent to which the tactic applies</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preconditions</td>
<td>Conditions which must hold for the method to apply.</td>
</tr>
<tr>
<td>Postconditions</td>
<td>Conditions which must hold after the method has been applied.</td>
</tr>
<tr>
<td>Outputs</td>
<td>Subgoals generated. This is a list of meta-level sequents.</td>
</tr>
<tr>
<td>Tactic</td>
<td>The name of the tactic which constructs the piece of the object-level proof corresponding to this method.</td>
</tr>
</tbody>
</table>

**Definition 3.1** A meta-level sequent is an abstraction of an object level sequent which may be annotated (§3.4.5), contain meta-variables (§3.5.5), and may have some hypotheses added or deleted.

Tactics operate on object-level sequents. Methods operate on meta-level sequents.

A method’s preconditions determine the context in which the method will apply. The same object-level tactic may be represented by several methods, each operating in different circumstances. This allows us to distinguish situations in which it is desirable to apply a tactic from those in which it is possible. It also has explanational value, since applying the tactic in different circumstances may have a different intuitive meaning.

Most terminology for describing object level proofs can be abstracted to the meta-level.

**Definition 3.2** A partial proof plan of a specification is a tree with the following properties:
1. Each node is labelled with a meta-level sequent and a method which is applied to that sequent to produce the node's children.

2. The specification is the sequent at the root of the tree.

3. If a method application generates no subgoals, then the leaf at which it is applied is labelled as complete.

4. If no method is specified for a leaf, then the subgoal at that node is open.

**Definition 3.3** A proof plan is a partial proof plan with no open subgoals.

When attempting to apply a method to a meta-level sequent, first the sequent is unified with the input slot. If this succeeds, the preconditions are evaluated. If they succeed, the postconditions are evaluated, leading to instantiation of the output slot. The method fails unless each of these steps succeeds. Pre- and post-conditions are pieces of Prolog code. A set of predicates, called the meta-language is provided for accessing and manipulating meta-level sequents.

### 3.3.3 Searching for a proof and CLAM

**Construction of the plan**

The planner, CLAM [Bundy et al 91], constructs a proof plan by linking together methods. A completed plan has a tree structure. See §3.4.5 for a simple proof plan, and §B-1 for a much larger example.

Initially, the list of open subgoals is the singleton consisting of the proposition to be proved. For each remaining open subgoal, CLAM tries to apply the methods in the order they appear in method database. When a method succeeds, its output slot is instantiated to a list of subgoals. A node is created in the proof plan which is labelled with this method. If the output list is empty, this branch of the proof plan is complete, and the node is a leaf of the proof plan. If the list is non-empty, it is merged with the list of open subgoals. Children of this node
are created for each of the new subgoals. The planner is called recursively with the new list of open subgoals.

A proof plan is successfully constructed when there are no remaining open subgoals. If no methods are applicable to a sequent, then the method which generated it fails. CI\textsc{am} then backtracks, first trying to satisfy the method in other ways (for example, the induction/2 method may apply alternative induction schemes upon backtracking), then trying to apply other methods. There are various strategies for exploring the search space: depth first, breadth first, best first and iterative deepening.

A procedure for constructing the object level proof is formed by replacing each method in the tree with its associated tactic.

Applying a method at the meta-level is usually significantly faster than applying the corresponding tactic at the object-level for several reasons:

1. Well-formedness goals (§2.5.3) which must be proved at the object level are suppressed at the meta-level.

2. Oyster maintains the entire proof tree as part of a global state, whereas proof planning only requires the local state of the proof.

The modular structure of proof strategies — methods and their order in the method database, available lemmas, and the proof planner used (depth-first, breadth-first, best-first or iterative deepening) — makes it very easy to experiment with different heuristics for guiding the proof process.

Organisation of the theory

When a theorem is loaded into CI\textsc{am}, it is passed to Oyster for incorporation into the internal database, and also parsed by CI\textsc{am} to extract and record certain meta-level properties, such as function definition and wave rule records (§3.4.4). CI\textsc{am} also loads any lemmas or methods which are required for a proof of the theorem. These are specified by entries in the "needs" file.
Methods are divided into two classes: methods and submethods, and they are stored internally in two separate databases. Methods may be applied explicitly by CLAM as it searches for a plan. Submethods may only be applied explicitly by other methods or submethods.

Several methods or submethods can be combined together as an iterator. The iterator recursively applies the specified methods or submethods to its input sequent and to any resulting output sequents until no more method applications are possible.

### 3.3.4 A comparison of the object-level and meta-level search spaces

Table 2 of [Bundy et al 91] gives various measures of the size of the proofs of several inductive theorems. For example, in a proof of the commutativity of addition, the tactic for performing the step case of the induction (see §3.4 for more on proof by induction) applies 139 Oyster proof steps. Search using iterative deepening for a proof plan of the commutativity of multiplication visits 3078 nodes. It was estimated that an iterative deepening search for a proof at the level of Oyster inference rules would visit about $10^{32}$ nodes.

From the same paper, the shortest proof plan found for the commutativity of multiplication contains 17 tactic applications, which expands to 665 Oyster rule applications.

Figure B-2 of appendix B contains a part of the object-level proof which is generated when the plan of figure B-1 is executed. Proofs of the well-formedness goals have not been expanded but left marked as proved by the wfftacs tactic. This object-level proof contains more proof steps, more branches and is deeper than the corresponding proof plan.

At the object level, whenever an existentially quantified goal is encountered it is possible to introduce any well-formed object-level term as a witness, thus creating an infinite branching point in the search space. At any point in the
proof it is possible to cut in a lemma. A thorough analysis of the properties of
search at the level of Oyster inference rules is beyond the scope of this thesis,
but the empirical evidence I have presented strongly suggests that search at the
level of Oyster proof steps is likely to be completely impracticable.

The different levels at which a proof can be constructed have the following
characteristics:

- **Object level** — Small proof steps, lots of well-formedness goals.
- **Tactic level** — Large proof steps, well-formedness goals.
- **Meta-level** — Large proof steps, annotations, no well-formedness goals.

### 3.4 The proof strategy for induction

#### 3.4.1 Introduction

Proof by induction can be used to reason about and synthesise programs. A
variety of sophisticated heuristics have been implemented in ClAM to automate
it.

The proof strategy for induction is described in the following subsections.
One of its cornerstones is the rippling strategy, which greatly reduces search
when term rewriting. This strategy is exploited in chapter 5 to automate pro-
gram synthesis after a type change.

#### 3.4.2 The form of constructor induction

ClAM has been used extensively to mechanise proofs by induction. As an ex-
ample, consider structural induction over the natural numbers:

\[
\begin{array}{c|c}
\text{Base case} & \text{Step case} \\
\hline
P(0) & P(n) \vdash P(s(n)) \\
\forall n \; P(n) \\
\end{array}
\]

65
When this rule of inference is applied it generates two subgoals: a base case to prove that the proposition is true for zero, and a step case to prove that the proposition is true for \( s(n) \) if it is true for \( n \).

Other induction principles are also available, e.g. \( \text{hd} :: \text{tl} \) structural induction on lists, or \( s(s(x)) \) two-step induction on natural numbers. In all the cases we shall consider, the difference between the induction conclusion and the induction hypothesis is the presence of constructor functions in the former. This is constructor induction.

The proof strategy for induction proceeds in the step case by trying to rewrite the induction conclusion until it matches the induction hypothesis. The following sections describe how this rewriting process is guided.

### 3.4.3 Wave terms

The definitions in sections 3.4.5 and 3.4.4 are taken from [Basin & Walsh 94a]. They provide us with the necessary language for annotating rewrite rules and terms to ensure that rewriting a term preserves some parts of the term while it may change others.

**Definition 3.4** A wave front is a term with at least one distinguished proper subterm. It is represented by marking a term with annotations, where wave fronts are enclosed in boxes and the distinguished subterms, called wave holes are underlined.

**Definition 3.5** A wave term is an annotated term which contains wave fronts.

These are wave terms: \( \boxed{f(x)} \), \( h(\boxed{g(x,x)}) \), \( s(s(\text{plus}(x,y))) \).

These are not wave terms: \( f(x) \), \( f(\boxed{)} \), \( f(x) \), \( f(x) \).
Definition 3.6 The parts of a term which are not in a wave front are called the skeleton. Formally, the skeleton is a non-empty set of terms defined as follows:

1. \( \text{skeleton}(t) = \{t\} \) for \( t \) a constant or variable.
2. \( \text{skeleton}(f(t_1, ..., t_n)) = \{f(s_1, ..., s_n) | \forall i. s_i \in \text{skeleton}(t_i)\} \).
3. \( \text{skeleton}(f(t_1, ..., t_n)) = \text{skeleton}(t_1) \cup ... \cup \text{skeleton}(t_n) \) for the \( t_i \) in wave holes.

Definition 3.7 The erasure of an annotated term is the term with its annotations removed. Formally:

1. \( \text{erasure}(t) = t \) for \( t \) a constant or variable.
2. \( \text{erasure}(f(t_1, ..., t_n)) = f(\text{erasure}(t_1), ..., \text{erasure}(t_n)) \).
3. \( \text{erasure}(f(t_1, ..., t_n)) = f(\text{erasure}(t_1), ..., \text{erasure}(t_n)) \).

For example, the erasure and skeleton of \( p(s(n)) \) are \( p(s(n)) \) and \( \{p(n)\} \) respectively.

3.4.4 Wave rules

Rewrite rules are stored in the external library as theorems with associated proofs. They are annotated using the same notation as wave terms. Annotated rewrite rules are called wave rules. A wave rule is written \( L \Rightarrow R \), and may only be applied in the specified direction.

In order to ensure that wave rule application terminates, a measure on annotated terms is defined, \( \mu : \text{annotated\_term} \rightarrow \text{IN} \). Wave rules must decrease this measure. Termination of rippling is proved in [Basin & Walsh 94a].
Definition 3.8 A wave rule is an annotated rewrite rule $L \Rightarrow R$ such that:

1. $\text{erasure}(L) = \text{erasure}(R)$ is a proved theorem,
2. $\text{skeleton}(L) \supseteq \text{skeleton}(R)$,
3. $\mu(L) > \mu(R)$, i.e. the rewrite is measure-decreasing.

Definition 3.9 (Wave rule application) A wave rule $L \Rightarrow R$ may be applied to an annotated term $T$ to yield an annotated term $T'$ if:

1. $S$ is a subterm of $T$,
2. there is a substitution $\sigma$ such that:
   (a) $\text{erasure}(S)\sigma = \text{erasure}(L)\sigma$,
   (b) $\text{skeleton}(S)\sigma = \text{skeleton}(L)\sigma$,
   (c) $T' = T[R/S]\sigma$.

3.4.5 Rippling and the proof strategy for induction

When the induction method applies (3.1), the step case is annotated to indicate the differences between the induction hypothesis ($P(n)$) and the induction conclusion ($P(s(n))$). These differences are usually nested deep within the term structure of $P$.

Rippling — the successive application of wave rules — moves the differences closer and closer to the root of the term structure. Wave rules are skeleton-preserving, so during an inductive proof, the skeleton of the induction conclusion is always equal to the induction hypothesis. Eventually, either rippling becomes stuck, or the induction hypothesis can be used in a process called fertilisation, which is divided into two kinds:

---

1In fact, the theorem need not be an equality. We also allow implication $\text{erasure}(R) \rightarrow \text{erasure}(L)$. Note the direction of implication is the opposite to the direction of rewriting because ClAM is a goal-directed (backward-chaining) system.
1. **Strong fertilisation**: the induction conclusion has been rippled until it contains a copy of the induction hypothesis (when all the wave fronts will have been rippled away), which can then be used to prove the induction conclusion immediately, completing this branch of the proof.

2. **Weak fertilisation**: The induction hypothesis is an equality \( L = R^2 \), and the induction conclusion contains a subterm which is a copy of either \( L \) or \( R \). The induction hypothesis is then used as a rewrite rule, replacing \( L \) with \( R \) in the goal or vice versa. This step removes any further annotations from the goal.

Either way rippling terminates. In order to achieve termination we insist that wave rules not only preserve the skeleton, but also move the wave fronts out.

The basic proof strategy for induction is outlined in figure 3–1. It consists of application of induction/2, followed by application of base_case/1 in the base cases and step_case/1 in the step cases. Rippling is implemented by the ripple/1 submethod, which is applied as part of step_case/1. The individual methods are described in §3.7. The base_case/1 and step_case/1 methods may leave some subgoals which require further planning to prove. For example there may be applications of base_case/1 or induction/2 in the step case of the induction.

An example of the use of rippling is in the proof of the associativity of addition:

\[
\forall x, y, z. x + (y + z) = (x + y) + z
\]

Applying induction on \( x \), we get the step case:

\[2\] More generally, weak fertilisation can be used as long as the main connective is transitive. If the connective is not symmetric, then some care must be taken over the direction of rewriting.
\[ \forall y, z. x + (y + z) = (x + y) + z \quad \text{(induction hypothesis)} \]
\[ \vdash \forall y, z. [s(x)] + (y + z) = ([s(x)] + y) + z \quad \text{(induction conclusion)} \]

This is proved by rippling using the following wave rule, which constitutes part of the recursive definition of +:

\[ s(u) + v \Rightarrow s(u + v) \quad (3.2) \]

The induction conclusion is successively rewritten:

\[ \vdash \forall y, z. [s(x)] + (y + z) = ([s(x)] + y) + z \]
\[ \vdash \forall y, z. s(x + (y + z)) = s(x + y) + z \]
\[ \vdash \forall y, z. s(x + (y + z)) = s((x + y) + z) \]

One more ripple is possible, using the wave rule:

\[ s(x) = s(y) \Rightarrow x = y \]
The goal is now:

\[ \forall y, z. x + (y + z) = (x + y) + z \]

This can now be proved by strong fertilisation.

Below is the proof plan generated automatically by Cl\^\text{AM}:

\begin{verbatim}
induction([s(v0)], [x:nat]) then
  [base_case([sym_eval(...), elementary(...)]),
   step_case(ripple(...) then [fertilize(strong, v1)])]
\end{verbatim}

Sometimes the arguments of a method are replaced in the printed representation by ellipsis for simplicity.

In appendix B, the entire program synthesis process is illustrated. I give a proof plan for a synthesis theorem, the corresponding object-level proof, and the extracted program.

### 3.4.6 Ripple analysis

In a given formula, it may be possible to apply induction in several different ways, and one must choose between different induction schemes and induction variables. As an example, consider the goal:

\[ \forall x, y : \text{nat}. x + (y + y) = (x + y) + y \] (3.3)

Possibilities include single or two-step induction on \( x \), single or two-step induction on \( y \), simultaneous single step induction on \( x \) and \( y \), three-step induction on \( x \) and many others.

We rely heavily on rippling because it is such a strong technique for inductive proof. Most of the possible inductions above do not lead to a proof because there
are no wave rules which can ripple out the wave fronts introduced. Using this insight, we choose between possible inductions with a technique called ripple analysis. Ripple analysis ranks possible induction schemas according to how well the resulting wave terms could be rippled.

In (3.3), if the only available wave rule is (3.2), then both single step induction on x, and single step induction on y are suggested. The second of these two suggestions is flawed: it introduces a wave front in a position from which it cannot immediately be rippled. The first suggestion is therefore ranked higher and is the one which is selected by the induction/2 method. The subsequent proof does in fact succeed.

3.4.7 Benefits of rippling

The rippling technique has several benefits:

1. It restricts rewriting so that termination of the rewriting process is guaranteed and search is reduced. Often search in the rewriting process is completely eliminated.

2. It has some explanatory power, making proofs involving rewriting easier to understand.

3. The applicability of other proof methods (e.g. induction) can be restricted by taking into account the ability of subsequent rippling to make progress in the proof.
3.5 Extensions to rippling

3.5.1 Directional wave fronts

In the presentation above, wave fronts move differences out through the term structure. There are occasions when we want to move differences in through the term structure. Usually this is done to move wave fronts into a sink: a universally quantified variable in the induction hypothesis which can be instantiated when the induction conclusion is fertilised.

We modify the annotations on both the wave rules and wave terms to specify a direction, either outwards:

\[ \text{plus}(s(x), y) \Rightarrow s(\text{plus}(x, y)) \]

...or inwards:

\[ s(\text{plus}(x, y)) \Rightarrow \text{plus}(s(x), y) \]

Often a left-to-right wave rule can be used as a right-to-left wave rule by reversing the directions of the wave fronts.

Rippling into a sink directs an outward-bound wave front down into a sink (marked \([y]\)):

\[ \text{plus}(s(x), [y]) \Rightarrow \text{plus}(x, [s(y)]) \]

To maintain termination of rippling, we allow outward bound wave fronts to become inward bound, but not vice versa. This entails some modification of the associated measure on annotated terms.
3.5.2 Wave front unblocking

The two essential requirements of wave rule application are that it preserves the skeleton, and is measure decreasing.

In a wave term \( f(h(x) + g(x)) \), for example, we can rewrite \( h(x) \) without changing the skeleton. This is one kind of unblocking: we are allowed to apply a directed rewrite rule \( L \Rightarrow R \) to a term \( L \) which is not in the skeleton of an annotated term. For example, applying the rule \( h(x) = x \) reduces the term above to \( f(x + g(x)) \). This may allow us to subsequently apply a wave rule where no wave rule application was possible before unblocking. Unblocking can also split and join wave fronts, since there are many annotated terms which are equivalent in that they have the same skeleton and erasure and the wave fronts are in the same direction, e.g. \( f(g(x)) \) and \( f(g(x)) \).

3.5.3 Proof critics

Recently the idea of a proof critic [Ireland & Bundy 95] has been developed. This comes from the notion of a critic in conventional A.I. planning, a procedure which oversees the planning process and may attempt to patch failed plans.

Critics exploit the high-level nature of proof plans, and have been used to patch unsuccessful proofs and to speculate lemmas or generalisations during the course of a proof. This provides another good motivation for using rippling and reducing search as much as possible: it creates the opportunity for critics to speculate needed lemmas automatically.

---

\(^3\)I use the notation \( L \Rightarrow R \) here because we allow any rewrite rule, not just wave rules.
3.5.4 Difference matching and unification

I define difference matching in §6.4.1. Difference matching and an extension, difference unification, have been used in proof plans for finding closed forms for sums of series. They have also been used to automate proof in LF [Negrete & Smaill 95].

3.5.5 Middle-out reasoning and meta-variables

In middle-out reasoning, meta-variables are used to stand for unknown existential witnesses, which are instantiated by subsequent proof planning steps. Meta-variables are represented by Prolog variables. As the proof proceeds, more information becomes available which can be used to choose a concrete term for the existential variable. Middle-out reasoning was used extensively in [Kraan 94] to synthesise logic programs. Middle-out reasoning allows, for example, speculative rippling, in which a meta-variable is partially instantiated to allow a wave rule application, e.g. a speculative ripple with the wave rule \( \text{plus}(s(x), y) \Rightarrow s(\text{plus}(x, y)) \) can be applied to (3.4) to give (3.5), partially instantiating \( z \) to \( s(z') \).

\[
\begin{align*}
\vdash & \exists z. \text{plus}(z', x) = s(\text{plus}(x, y))' \\
\vdash & \exists z'. s(\text{plus}(z', x)) = s(\text{plus}(y, x))'
\end{align*}
\]

[Kraan 94] makes extensive use of middle-out reasoning to synthesise logic programs. Kraan identifies several problems with the use of middle-out reasoning:

1. It is nontrivial to ensure that meta-variables are only instantiated to well-formed object-level terms.

2. Speculative rippling leads to proof search and non-termination of rippling.
Kraan suggests that the nontermination introduced by middle-out reasoning requires some kind of global control, such as that provided by proof critics (§3.5.3), over speculative steps such as speculative rippling.

3.5.6 Relational rippling

Rippling controls rewriting so that differences between the goal and some other formula, for example, an induction hypothesis, move outwards through the term structure. In logic programming, instead of nesting functions, complex expressions are built by forming conjunctions of relational expressions, which are implicitly linked by existentially quantified variables. Relational rippling [Bundy & Lombart 95] is an adaptation of rippling to the relational case. Relational rippling is still under development, and not yet integrated into CIAM.

3.6 Program synthesis in CIAM

In order to permit proof plans to be constructed for program synthesis, some extension of the proof strategy for induction was necessary.

Programs are specified by conjectures of the form (3.6), and from a proof of such a conjecture, a program satisfying the specification can be extracted, as described in §2.5.3.

\[
\forall \text{args} : t. \exists \text{output} : t'. \text{spec(args, output)} \tag{3.6}
\]

Most of the theorems which CIAM has been used to prove in the past have not contained existential quantifiers, so the main modification which was necessary in order to synthesise programs was extension of the proof strategy for induction to handle existentially quantified goals.
This has involved the creation of an `elim_existential/2` method (§3.7.2) which allows weak fertilisation with existentially quantified induction hypotheses.

A proof of (3.6) could proceed by immediately introducing an explicit lambda term for the existential quantifier, a function of the form \( \lambda \text{args}. f(\text{args}) \). The remainder of the proof is then a verification that \( f \) satisfies the specification and is of the correct type. In practice, it is not possible to determine such an \( f \) at the start of the proof. As the proof proceeds, the conjecture is broken down into simpler conjectures. At some point, however, it becomes necessary to explicitly introduce a term for an existential quantifier.

The standard `existential/2` method replaces any existentially quantified variable in the goal with a Prolog variable (a meta-variable). The meta-variable is instantiated as a side-effect of the application of other methods later in the proof. This greatly increases the scope for further method application, since the methods may instantiate the meta-variable in order to allow their application, but by the same token it leads to serious search problems. Since there is no check on the well-formedness of the substitution which is applied, it can easily result in false subgoals (which cannot subsequently be proved, of course).

In order to reduce proof search, several versions of the `existential/2` method have been written which do not introduce meta-variables into a proof plan. The existential witnesses which these methods introduce are restricted. There are two types of restrictions:

1. `existential_subterm/2` tries to prove a goal of the form \( \vdash \exists x. L = R \) by substituting subterms of \( L \) or subterms of \( R \) for \( x \) in \( L = R \) to obtain a tautology. This solves many goals without recourse to the existential method.

2. Without any further restrictions, a goal of the form \( \vdash \exists z. z = \text{term} \) could be trivially solved by supplying the witness term for \( z \). Often, we would like to apply further methods to break this goal down into simpler subgoals and so synthesise a compound expression for \( \text{term} \). In particular,
we would like to do this when we are trying to obtain a revised synthesis
proof which will yield a more efficient extracted function. Such a restric-
tion is achieved by only allowing certain terms to be used as existential
witnesses. These terms are specified by the user in the “needs” file. Both
existential_subterm/2 and existential/2 restrict the witnesses which
they supply for existential variables to those which are declared by the user
to be acceptable.

3.7 A description of the main methods

This section describes the most frequently used CLAM methods. For full docu-
mentation see [vanHarmelen et al 93].

3.7.1 induction/2

The induction/2 method applies induction on a variable $x$ which is either
universally quantified or free in the goal. The induction schema is chosen by
ripple analysis: this means that there must be some rippling possible in the step
cases(s).

The output of the method is a list of sequents:

1. Base cases $H \Rightarrow G[\text{base}_i/x]$ where $\text{base}_i$ are ground terms in $t$.

2. Step cases $H, G \Rightarrow G[c_i(x)/x]$ where $c_i(x)$ is a constructor term.$^5$

$^4$More generally, it can apply simultaneous induction on several variables.

$^5$More generally, destructor induction is also possible, in which $c_i$ is a destructor
function such as hd for lists. The subsequent rippling is more complicated than in the
constructor case.
3.7.2 elim_existential/2

In order to extend the step_case/1 method in a modular way, I have written a new method, elim_existential/2.

Preconditions:

1. There is an existentially quantified induction hypothesis,

   \( ih : \exists x : t . p(x) \)

2. This method has not already been applied to that hypothesis.

Postconditions:

1. \( \text{nv} \) is a newly generated object-level variable which does not occur in the goal or in the hypotheses.

Outputs:

1. Two new induction hypotheses:

   \( \text{nv} : t \)

   \( ih+ : p(\text{nv}) \)

   The new induction hypothesis can be subsequently be used in fertilisation.

3.7.3 step_case/1

The step_case/1 method applies rippling and unblocking to the annotated terms in the induction conclusion. The method terminates after strong or weak fertilisation (§3.4.5), or leaves an annotated output sequent when neither kind of fertilisation is possible and no further rippling or unblocking can be performed.

3.7.4 base_case/1

The base_case/1 method is sometimes sufficient for proving induction base cases. It does not rewrite annotated terms. It can prove some tautologies,
and tries to reduce the goal to one of these tautologies by repeatedly using symbolic evaluation, applying the following kinds of rewrites: function definitions, hypotheses which are equalities, reduction rules (which reduce an expression in the goal to one which is "more canonical"), and trying to find instantiations for existential variables.

3.7.5 apply_lemma/2

The apply_lemma/2 method applies when the current goal matches a currently loaded lemma, after substitution of appropriate values for the universally and existentially quantified variables. The guidance for application of this method is quite weak, so normally, this method is not loaded. When it is used, it is placed low down in the method hierarchy, so it is only applied when other methods have failed.

3.7.6 existential/2

The existential/2 method introduces explicit witnesses for existentially quantified variables in the goal. The standard ClAM method does this by introducing a Prolog meta-variable which is instantiated by later planning. In order to avoid the serious proof search problems this introduces, I have implemented more controlled ways of determining existential witnesses, which are outlined in §3.6 above.

3.8 Conclusion

Constructing proofs is a search problem. The search space of possible proof attempts built from object-level rules of inference is huge. Proof planning allows the expression of powerful heuristics and searches for proofs in a much smaller search space.
The programs which we manipulate are synthesised in a constructive logic: Martin-Löf's type theory. The CL\text{AM} system constructs proof plans by chaining together heuristics, encoded as methods. A successful proof plan can then be translated into a compound tactic which proves the theorem at the object level.

The synthesis proofs are generated and represented as proof plans in the CL\text{AM} system. Methods allow heuristics for constructing a proof to be expressed. Proof by induction is crucial to reasoning with programs, and rippling is a powerful strategy for constructing such proofs with very little search. The most-used proof strategy in CL\text{AM} is the proof strategy for induction. Some extensions to this proof strategy have been made to allow program synthesis proofs to be constructed.

The techniques described in this chapter provide us with the means to express heuristics for carrying out type changes in synthesis theorems, and to control search in the subsequent proofs.
Chapter 4

Type Change as a Proof Development Step

4.1 Introduction

This chapter builds on the technology of the previous chapters to formalise type changes. First, the transformation is introduced at the level of type theory, and proven to be a correct proof development step.

I describe how abstract data types are encoded in Martin-Löf’s Type Theory, and compare the approach taken in this thesis to one based on stepwise refinement using refinement operators defined in a type theory. The principal difference is that I concentrate on efficiency and automation rather than the ease with which (possibly inefficient) programs can be derived.

Instead of making explicit reference in the transformation to a source program, I assume that a proof plan for the source program has already been constructed. The methods and lemmas which were used in its construction will also be of use in the construction of a proof plan for the target. Information which can aid choice and execution of the transformation is available in the hypotheses and goal during the construction of a proof plan.
After an example transformation, I discuss a simple heuristic for deciding which type changes to make in a program. The initial motivation for a type change — the presence of certain function expressions in the program — is refined by heuristics which reduce the need for conversion of data from one representation to another.

4.2 Type changes in Type Theory

4.2.1 The correctness of the transformation

An advantage of program transformation as proof transformation [Madden 91] within the proofs-as-programs framework is that any transformation which produces a proof of the original specification is necessarily correct, and transformations which are not correct will not produce synthesis proofs.

4.2.2 The implementation relation

In §2.4.4 I defined implementation between ADTs. The notion of implementation between types which I use in this thesis is a notion of simulation, going back to [Hoare 72].

I use abstract data types primarily as a means of expressing complex types, and consider implementation more as a transformation between programs than as a relation between ADTs, so when implementing an ADT \( I \) by an ADT \( I' \), I do not insist that models of \( I' \) are mapped to models of \( I \), but only that the types and functions of \( I' \) which are required for the program being transformed are mapped to types and functions of \( I \). I call \( \rho \) a conversion function, and write \( I \xrightarrow{\rho} I' \). The arrow indicates the direction of refinement; the function \( \rho \) maps in the opposite direction. This is illustrated in figure 4–1.

After a type change \( I \xrightarrow{\sim} I' \) in a function \( f \), we have to verify that the original function \( f : I \rightarrow I \) can be replaced with a new function, \( f' : I' \rightarrow I' \):
The proof/program duality of type theory means that a proof of (4.2) produces an existential witness $z$ which is a function $f'$ obeying (4.1) as required.

$$\vdash \forall x' : T'. \exists z : T'. \rho(z) = f(p(x'))$$  \hspace{1cm} (4.2)$$

There are two main reasons for writing a specification like (4.2) instead of the following:

$$\vdash \forall x' : T'. \exists z : T'. z = \rho^{-1}(f(p(x'))))$$  \hspace{1cm} (4.3)$$

1. Often the "inverse" is not a true inverse in the sense that we do not have $\forall x : T'. \rho^{-1}(\rho(x)) = x$. This is the case for the retrieve function.
nat : bin → IN used in the specification of binary addition in §6.8 because any leading zeros in the binary number are stripped by this composition. The specification (4.3) unnecessarily restricts the functions which satisfy it. When synthesising functions which return binary numbers, for example, it would restrict us to functions which return binary numbers free of leading (most significant) zeros.

2. As I show in §4.5.3, we do not wish to end up with an extracted program which contains applications of the inverse, $\rho^{-1}$. Writing the goal in the form of (4.2) allows us to apply a strategy in which we aim to cancel $\rho$ on each side of the equality, yielding a witness for $z$, and hence an extracted program, which is free of $\rho$ and its inverse.

Since $\rho$ represents an implementation, there may be more than one representation for any given abstract term, for example the set $\{a, b\}$ may be represented as a list by $[a, b]$ or $[b, a]$ or $[a, a, b, a, b]$. Although this is the case, we can effectively write an inverse for $\rho$, $\rho^{-1}$, which maps each abstract term to some canonical concrete term. The proof/program duality of type theory ensures that a proof of the following implementation lemma produces such an inverse.

$$\forall x : T. \exists x' : T'. \rho(x') = x$$

The inverse is one-sided in the sense that $\forall x : T. \rho(\rho^{-1}(x)) = x$, but it can be the case that $\rho^{-1}(\rho(x')) \neq x'$.

4.3 Correctness in the type theory

If $f$ is synthesised from a specification:

$$\vdash \forall x : T. \exists y : T. \text{spec}(x, y) \quad (4.1)$$

then $f'$ must meet the following specification:
This implies the original specification, (4.4). The first conjunct states that $\rho$ is surjective, which is true if and only if the representation change is an implementation. Note that in (4.5), there is no explicit mention of $f$. $f$ and the composite $\rho \circ f \circ \rho^{-1}$ must meet the same specification, but it is quite possible that they may compute different functions.

Figures 4–3 and 4–4 show the object-level theorems which prove the correctness of the transformation for single type changes on input or output variables.

Figure 4–2 illustrates this justification process. The new synthesis proof, and a side condition to ensure that the representation is an implementation, are cut in. When the two left-hand branches have been proved, the bottom right hand branch is a logical consequence, and the entire proof tree constitutes a proof of the original specification.

The function transformed need not be of type $t \rightarrow t$. In general if we are transforming from a function of type $t_1 \rightarrow t_2 \rightarrow t_3 \rightarrow \ldots \rightarrow t_n$ to one of type $t'_1 \rightarrow t'_2 \rightarrow t'_3 \rightarrow \ldots \rightarrow t'_n$ we will require $n$ conversion functions $\rho_i : t'_i \rightarrow t_i$. At the object level this is justified by performing the transformations one by one on the (curried) function $f$. The change of output type is justified by the lemma of figure 4–4.

**Theorem 4.1 (Correctness of multiple transformations)** If we are changing the input and output types in a specification:

$$\forall x_1 : t_1, \ldots, \forall x_n : t_n, \exists z : t_0. \text{spec}(x_1, \ldots, x_n, z)$$

with transformations:

$$t_1 \xrightarrow{\rho_1} t'_1, \ldots, t_n \xrightarrow{\rho_n} t'_n, t_0 \xrightarrow{\rho_0} t'_0$$

then we obtain the following proof obligations:
Original specification: \( \forall x : T . \exists z : I . \text{spec}(x, z) \)

![Proof obligations diagram]

Lemma to prove the representation is an implementation

\( \forall x : T \exists x' : T' . \rho(x') = x \)

\( x : T \)

\( x', z' : T' \)

\( \rho(x') = x \)

\( \text{spec}(\rho(x'), \rho(z')) \)

\( \vdash \exists z : I . \text{spec}(x, z) \)

**Figure 4–2**: Proof obligations arising from a representation change. Variables have been introduced as convenient to reach the final (bottom right) step of the proof.

1. We must prove that each of the \( n + 1 \) conversion functions: \( \rho_0, \ldots, \rho_n \), is surjective:

   \( \forall x_i : t_i \exists x'_i : t'_i . \rho_i(x'_i) = x_i \)

2. We must synthesise a new function from the modified specification:

   \( \forall x'_1 : t'_1, \ldots, x'_n : t'_n . \exists z' : t'_0 . \text{spec}(\rho_1(x'_1), \ldots, \rho_n(x'_n), \rho_0(z')) \)

**Proof:**

In order to simplify this proof I assume that all of the input and output types are being transformed. The identity transformation \( T \xrightarrow{\lambda x.x} T \) is trivially surjective, thus allowing us to leave some of the types unchanged if we wish.
Consider the following definitions:

\[ u = t_1 \times \ldots \times t_n \]

\[ u' = t'_1 \times \ldots \times t'_n \]

\[ \sigma : < x'_1, \ldots, x'_n > \mapsto < \rho_1(x'_1), \ldots, \rho_n(x'_n) > \]

\[ \forall < x_1, \ldots, x_n > : u \cdot \text{spec}_u(< x_1, \ldots, x_n >, z) \leftrightarrow \]

\[ (\forall x_1 : t_1, \ldots, \forall x_n : t_n \cdot \text{spec}(x_1, \ldots, x_n, z)) \] (4.6)

\[ \sigma : u' \rightarrow u \text{ is surjective if each of the component conversion functions } \rho_i : t'_i \rightarrow t_i \text{ is:} \]

\[ \forall x : u \cdot \exists x' : u' \cdot \sigma(x') = x \leftrightarrow \bigwedge_{i=1}^{n} \forall x_i : t_i \cdot \exists x'_i : t'_i \cdot \rho_i(x'_i) = x_i \] (4.7)

Making one single input transformation \( u \xrightarrow{\sigma} u' \) gives the following goals:

\[ \forall x' : u' \cdot \exists z : t_0 \cdot \text{spec}_u(\sigma(x'), z) \] (4.8)

\[ \forall x : u \cdot \exists x' : u' \cdot \sigma(x') = x \] (4.9)

Subsequently making an output transformation \( t_0 \xrightarrow{\rho_0} t'_0 \) on (1) produces the goals:

\[ \forall x' : u' \cdot \exists z' : t_0 \cdot \text{spec}_u(\sigma(x'), \rho_0(z')) \] (4.10)
output_trn: [] complete autotactic(idtac)

\[ \Rightarrow t:u(1)\Rightarrow \]
\[ t1:u(1)\Rightarrow \]
\[ \rho:(t1=>t)\Rightarrow \]
\[ p: (x:t=>x1:t1#\rho \text{ of } x1=x \text{ in } t)\Rightarrow \]
\[ \text{spec}:(t=>u(1))\Rightarrow \]
\[ q: (x1:t1#\text{spec \ of } (\rho \text{ of } x1))=>x:t#\text{spec \ of } x \]

Figure 4–4: The object level meta-theorem justifying an implementation type change on the output type.

\[ \forall z : t_0 . \exists z' : t'_0 . p_0(z') = z \quad (4.11) \]

Applying (4.7) to (4.9) gives us \( n \) surjectivity goals for the inputs, which with (4.11) makes the \( n + 1 \) surjectivity goals as required.

Applying (4.6) to (4.10) gives us the transformed synthesis goal as required.

\[ \Box \]

4.4 Abstract Data Types

4.4.1 Introduction

The notion of an abstract data type (ADT) was introduced in §2.4.2. ADTs allow the expression of complex types while hiding the details of their implementation. In chapter 2, I outlined some previous work concerning refinement of ADTs. As well as having such theoretical advantages, ADTs allow theories to be organised in a modular way.

Although Martin-Löf's Type Theory is extremely powerful, definitions of new types tend to be long and complicated. Use of such definitions leads to large,
difficult well-formedness goals in a proof. When the type is packaged as an ADT, the tricky proof obligations are hidden from the body of the theorem and need only be proved once.

This has several benefits:

1. In line with Kowalski's statement (§1.2), an implementation for the abstract data type can be chosen after program development has been completed.

2. Facilities can be provided for structuring ADTs, such as the ENRICH—FORGET—IDENTIFY language described in §2.4.3.

3. At the proof planning level, the ADT appears just like one of the built-in types such as nat or nat list. Induction schemes can be defined and easily processed by the induction/2 method.

4.4.2 Formulating ADTs in Type Theory

Contrast with adding types directly to Type Theory

The PICT [Hamilton 93] implementation of Martin-Löf's Type Theory allows new recursive data types to be defined.

In Martin-Löf's Type Theory, every type has the following associated inference rules:

1. two formation rules,

2. introduction rules (two for each canonical object constructor),

3. two elimination (induction) rules,

4. computation (recursion) rules (one for each canonical object constructor).
In PICT, the user supplies a specification of a new recursive data type, and the system automatically generates inference rules (including those above) for the new type and adds them to the logic.

This approach of adding inference rules directly to the logic carries the advantage that the types which are added enjoy exactly the same status as the primitive types of pnat, tlist etc., allowing uniform proof mechanisms to be used for all types. Such an approach has the following disadvantages:

1. The new inference rules will be complicated for all but the simplest types, and lead to long and difficult proof obligations. PICT, for example, only allows the definition of recursive data types. By contrast, an ADT can be used to specify only those properties of a complex data type which are relevant, and hide the irrelevant details from the user and the theorem prover.

2. Mistakes made in the new inference rules can lead to unsoundness in the resulting logic. PICT automatically generates the new inference rules, but it is not clear whether the procedure by which it does this has been verified to be correct for the recursive data types it allows. By contrast, an ADT must be proved correct in the existing logic by supplying an implementation (model).

Abstract data types in NuPRL

[Basin & Constable 93] describes an ADT mechanism for NuPRL. ADTs are formulated as Σ-types (existential types):

\[ \exists t : \text{Type}. \exists \text{functions} \]

\[ \exists \text{proof of equations} \]

\[ \exists \text{proof of induction principles} \]

\[ \exists \text{proof of computation rules} \]
To prove such a theorem at the object level it is necessary to provide witnesses for the existentials, i.e. give concrete structures for the type and functions, and give proofs that the equations and induction principles are valid in this type. For each induction principle (elimination rule in the terminology of type theory), there is a corresponding computation rule which specifies the corresponding recursion schema.

Conjectures using ADTs in this way are existentially quantified with the ADT.

\[ \vdash \exists t : \text{Type} \cdot \exists \{\text{functions, proofs}\} . \text{spec}(t) \]

Induction schemes for non-free data types are complicated. It is necessary to include a condition in the step case(s) of the induction schemes, and in the corresponding computation rules, which guarantees that the value of the extracted (recursive) function does not depend on the way in which members of the type are represented. For example, if sets are represented using a constructor \( \text{add}: \text{obj} \rightarrow \text{set}(\text{obj}) \rightarrow \text{set}(\text{obj}) \), then we must prove \((4.13, 4.14)\) because the type theory requires that \( \forall f : t \rightarrow t', \forall x, y : t. x = y \rightarrow f(x) = f(y) \).

\[ \forall a, b : \text{obj} . \forall s : \text{set}(\text{obj}) . f(\text{add}(a, \text{add}(b, s))) = f(\text{add}(b, \text{add}(a, s))) \] \( (4.13) \)

\[ \forall a : \text{obj} . \forall s : \text{set}(\text{obj}) . f(\text{add}(a, \text{add}(a, s))) = f(\text{add}(a, s)) \] \( (4.14) \)

Abstract data types in Oyster-\text{ClAM}

I have extended \text{ClAM} with methods which implement an abstract data type mechanism in which ADTs are formulated as existential types like \((4.12)\). When planning the proof of a theorem which uses an ADT, the \text{split implementation}/1 method, described in §E.6, assumes the existence of an implementation (model) for the ADT, i.e. it assumes that object-level terms exist for all the existential quantifiers in the ADT. Only the equations which are declared in the ADT are
relevant during the proof; particular properties of the object-level model should not be used. The split_implementation method extracts from the ADT the function declarations, equations and induction schemes. A note is made of the type of any function declared in the ADT. A separate theorem is made for each of the equations in the ADT, and these theorems are run through CLM's wave rule parser to add any wave rules to the internal database. The induction schemes are processed and stored away to be used by an extended version of the induction/2 method.

This leaves a difficult task for the split_implementation tactic: to provide a correct implementation for the ADT, and to pull out from this implementation proofs of the theorems which have been created from the equations in the ADT. This tactic has not yet been written. This is suggested as further work in §10.3.

4.4.3 Program refinement in type theory

The power of type theories to provide a program development framework based on ADTs is illustrated in [Luo 91]. The higher-order features of type theory are used to implement specification refinement operations similar to those discussed in [Sannella & Tarlecki 92] (§2.5.5). The type theory used is the Extended Calculus of Constructions (ECC) [Luo 89]. ECC is similar to Martin-Löf’s Type Theory, except that the universe of types is split into an impredicative universe of propositions, Prop, and an indexed family of predicative universes of types, Type.

Specifications are defined as Σ-types. Luo is careful to separate the built-in equality of the type theory from that of an abstract data type (a congruence relation which must be supplied in any implementation of the ADT). As in my work, the data type invariant is incorporated into the equality of the ADT.

A Σ-type, Σx:A.B(x) intuitively represents a set of pairs1 (a, b) such that a:A, b:B(a). A specification is split into computational and axiomatic parts:

---

1In Martin-Löf’s Type Theory, such a Σ-type corresponds intuitively to a proposition
\[
\text{SPEC} \overset{\text{def}}{=} \sum \text{Str} : \text{Type} \cdot \sum (\text{Ax} : \text{Str} \rightarrow \text{Prop})
\]

\text{Str} is a structure (types and functions), and \text{Ax} is a predicate which is true if \text{Str} satisfies the required properties. An implementation of this specification consists of providing a structure for the type, which includes implementations of the functions in the type, and a proof that the axioms are satisfied.

The implementation relation is constructor implementation (§2.4.4). A specification \text{SP} is implemented by a specification \text{SP}', written \text{SP} \overset{\rho}{\rightarrow} \text{SP}', if there is a refinement map from \text{SP}' to \text{SP}. This is a function \( \rho : \text{Str}[\text{SP}'] \rightarrow \text{Str}[\text{SP}] \) such that:

\[
\forall s' : \text{Str}[\text{SP}'] . \text{Ax}[\text{SP}'](s') \supset \text{Ax}[\text{SP}](\rho(s'))
\]

This is a slightly weaker requirement than (4.1). If \text{spec}' is an axiom in \text{SP}' specifying \( f' \) and \text{spec} is an axiom in \text{SP} specifying \( f \), then the above requires that:

\[
\forall x' : t'. \text{spec}'(x', f'(x')) \rightarrow \text{spec}(\rho(x'), f(\rho(x')))
\]

This can be ensured by letting \( \text{spec}'(x', y') \leftrightarrow \text{spec}(\rho(x'), \rho(y')) \) and proving (4.1).

Luo defines a variety of operations for decomposing and composing specifications in such a way that the components of a specification may be implemented separately. In common with other work on algebraic refinement, there are two notions of composition:

1. **Vertical composition** allows successive refinement steps to be composed:

\[
\exists x : A . B(x). \text{In ECC, not all types are propositions, so} \ B(x) \text{may not be a proposition and this intuitive notion does not hold.}
\]
2. **Horizontal composition** allows separate refinement of a specification \( SP \) and a specification \( SP' \) it takes as a parameter:

\[
\begin{align*}
SP \xrightarrow{\theta} SP' & \quad SP' \xrightarrow{\theta'} SP'' \\
\hline
SP \xrightarrow{\theta \theta'} SP''
\end{align*}
\]

The definition of \( \tau \) is complex and omitted here.

These operations determine the result of composing two specifications, and the corresponding composition of their refinement maps. Refinement maps have computational content: conversion of data between representations.

Programs which are developed by specification refinement operations like those described in this section may be inefficient for several reasons:

1. Refinement maps introduce computational content (conversion functions) at the point the refinement is made. I deal with this problem using *propagation* (§4.8.2), which may move the conversion functions in order to improve the resulting program.

2. The implementation of a composition of functions \( f \circ g \) is typically the composition of the implementations, \( f' \circ g' \). This means that opportunities for sharing computation between \( f \) and \( g \) are lost. In particular, a recursive function is typically implemented by defining implementations for the constructor/destructor functions of the abstract type. This would lead, for example, to an inefficient binary addition function defined solely in terms of increment/decrement functions on binary numbers. I discuss this further in §9.2.2.
4.5 An example transformation

I now present a simple example which demonstrates how a type change is realised as a modified synthesis proof. The first attempt at a synthesis results in an inefficient extracted program because the type change takes place within the body of a recursive function. I subsequently show that the provision of transformed versions of the primitives which make up this function can allow the type change to be performed outside the recursion and thus yields a more efficient extracted program.

4.5.1 Definitions

Define a function $f$ which returns the sum of the odd elements in a list by equation (4.15) and wave rule (4.16):

\[
\begin{align*}
f(nil) &= 0 & (4.15) \\
f(h::t') &= \text{plus(times(mod2(h), h), f(t))} & (4.16)
\end{align*}
\]

nat is specified by an abstract data type:

\begin{verbatim}
DEFINE ADT NAT
SORTS: nat
FUNCTIONS: 0: nat
           succ: nat -> nat
           plus: nat -> nat -> nat
           times: nat -> nat -> nat
           mod2: nat -> nat
EQUATIONS:
            \forall x:nat \rightarrow 0 = succ(x)
            succ(x) = succ(y) \leftrightarrow x=y
\end{verbatim}
\begin{align*}
  &\text{plus}(0,y) = y \\
  &\text{plus}(\text{succ}(x),y) = \text{succ}(\text{plus}(x,y)) \\
  &\text{times}(0,y) = 0 \\
  &\text{times}(\text{succ}(x),y) = \text{plus}(y,\text{times}(x,y)) \\
  &\text{mod2}(0) = 0 \\
  &\text{mod2}(\text{succ}(0)) = \text{succ}(0) \\
  &\text{mod2}(\text{succ}(\text{succ}(x))) = \text{mod2}(x)
\end{align*}

The "mod 2" within the definition of \( f \) leads us to propose an enrichment of the natural numbers:

\begin{verbatim}
DEFINE ADT NAT2
ENRICH NAT BY
  SORTS:   bool, nat2
  FUNCTIONS:
    true, false: bool
    val: bool \rightarrow nat
    mod2': nat2 \rightarrow nat2
    first: nat2 \rightarrow nat
    second: nat2 \rightarrow bool
    \rho: nat2 \rightarrow nat
    \rho^{-1}: nat \rightarrow nat2
  EQUATIONS:
    val(false) = 0
    val(true) = \text{succ}(0)
    \rho(\text{mod2'}(x)) = \text{mod2}(\rho(x))
    first(n) = \rho(n)
    val(second(n)) = \text{mod2}(\rho(n))
    \rho(\rho^{-1}(n)) = n
ENDDEF
\end{verbatim}

The conversion functions, \( \rho: \text{nat2} \rightarrow \text{nat} \) and \( \rho_1: \text{nat2 list} \rightarrow \text{nat list} \) are defined by (4.17, 4.18, 4.19). There is also a cancellation rule (4.20), which is valid for any function \( \rho \), not just conversion functions.
\begin{align*}
\rho((n, b)) &= n \quad (4.17) \\
\rho(\text{nil}) &= \text{nil} \quad (4.18) \\
\rho((h :: t)) &= \rho(h) :: \rho(t) \quad (4.19) \\
\forall x = y \rightarrow \rho(x) &= \rho(y) \quad (4.20)
\end{align*}

The ADT NAT2 can be implemented using a subset of a product type:

\[ \text{nat2} = \{ (n, m) : (\mathbb{N} \times \text{bool}) \mid n \equiv \text{val}(m) \pmod{2} \} \]

### 4.5.2 Transformation

The original synthesis theorem can be written as:

\[ \forall l : \text{nat list} \exists s : \text{nat}. s = f(l) \]

We can choose to transform just the input to \( f \), or the output too. Let us transform both. As in figure 4-2, the proof divides into two branches.

**Justification that the transformation is sound**

As illustrated in figure 4-2, this entails proving that the conversion function, \( \rho_t \), represents an implementation:

\[ \forall x : \text{nat list}. \exists x' : \text{nat list}. \rho_t(x') = x \]

A proof of such a specification effectively synthesises an inverse for the conversion function (\( \rho_t \) here). I do not attempt to automate this, but rely on the user providing a suitable lemma.

In this case, an inductive proof plan can be constructed with the aid of speculative rippling. The proof is by induction on \( x : \text{nat list} \). The base case is trivial. The step case is:
\(x: \text{nat list}\)
\(h: \text{nat}\)
\(\exists x': \text{nat list} \cdot \rho_1(x') = x \vdash \exists x': \text{nat list} \cdot \rho_1(x') = h :: x\)

A speculative ripple of \(x''\) using \((4.19)\) gives a goal:

\(\vdash \exists h': \text{nat list} \exists x'': \text{nat list} \cdot \left[ \rho(h') :: \rho_1(x'') \right] = h :: x\)

This can be fully rippled out using the following wave rule:

\[a :: b = c :: d \Rightarrow a = c \land b = d\]

This gives:

\(\vdash \exists h': \text{nat list} \exists x'': \text{nat list} \cdot \left[ \rho(h') = h \land \rho_1(x'') = x \right]\)

Proof that \(\rho\) represents an implementation solves the left hand conjunct, and weak fertilisation solves the right, completing the proof that \(\rho_1\) represents an implementation.

The final part of the justification step of the proof (the rightmost branch of figure 4–2) is:

\(H1: l: \text{nat list}\)
\(H2: \exists l2: \text{nat list} \cdot \rho_1(l2) = l\)
\(H3: \forall l': \text{nat list} \exists s2: \text{nat2} \cdot \rho(s2) = f(\rho_1(l'))\)
\(\vdash \exists s: \text{nat list} \cdot s = f(l)\)

This is simply proved by letting \(s = \rho(s2)\) and plugging the \(l2\) obtained from \(H2\) into \(H3\), as explained in section 4.2.

**Synthesis of a new function**

This is presented below.

\(\forall l: \text{nat2 list} \cdot \exists s: \text{nat2} \cdot \rho(s) = f(\rho_1(l))\)

99
We perform induction on \( l : \text{nat2 list} \):

\[
\exists s : \text{nat2} . \rho(s) = f(\rho_1(l))
\]

\( n2 : \text{nat2} \)

\[\vdash \exists s' : \text{nat2} . \rho(s') = f(\rho_1([n2 :: l]))\]

Ripple the right hand side using wave rules (4.19) and (4.16), and replace \( \rho(n2) \) by \( n \).

\[\vdash \exists s' : \text{nat2} . \rho(s') = \text{plus}(\text{times}(\text{mod2}(\rho(n2)), \rho(n2)), f(\rho_1(l)))\]

Now we can rewrite using \( \text{mod2}(\rho(n2)) = \rho(\text{mod2'}(n2)) \) and since we are fully rippled on the right hand side, we can weak fertilise:

\[\vdash \exists s' : \text{nat2} . \rho(s') = \text{plus}(\text{times}(\rho(\text{mod2'}(n2)), \rho(n2)), \rho(s))\]  

(4.21)

Now, we can use the implementation lemma (section 4.2.2), which effectively gives us an inverse to \( \rho \):

\[\vdash \forall x : \text{nat} . \exists y : \text{nat2} . \rho(y) = x\]

This allows us to give a witness, \( s' = \rho^{-1}(\text{plus}(\text{times}(\rho(\text{mod2'}(n2)), \rho(n2)), \rho(s)))\). The program extracted from the proof of the entire revised synthesis theorem is:

\[
f'(l) = \rho(\lambda t . \text{list.ind}(t, \rho^{-1}(0),
\qquad [n2, l, s2, \rho^{-1}(\text{plus}(\text{times}(\rho(\text{mod2'}(n2)), \rho(n2)), \rho(s2))))(\rho_1^{-1}(l))))
\]

Equationally this can be represented as:

\[
f'(l) = \rho(f_\rho(\rho_1^{-1}(l)))
\]

(4.22)

\[f_\rho(\text{nil}) = \rho^{-1}(0)\]

\[f_\rho(h :: t) = \rho^{-1}(\text{plus}(\text{times}(\rho(\text{mod2'}(h))), \rho(h)), \rho(f_\rho(t)))\]
4.5.3 Avoiding conversion functions within the recursive loop

The recursive program synthesised above, (4.22) is very inefficient because at each step through the recursion, there is an application of both $\rho$ and $\rho^{-1}$.

We can avoid having to convert back from nat to nat2 if we also have nat2 functions $\times_\rho : \text{nat2} \rightarrow \text{nat2} \rightarrow \text{nat}$ and $+_\rho : \text{nat2} \rightarrow \text{nat2} \rightarrow \text{nat2}$ satisfying the following equations:

\[
\begin{align*}
\times_\rho(\rho(x), \rho(y)) &= \rho(\times_\rho(x, y)) \\
\rho(x) + \rho(y) &= \rho(x +_\rho y)
\end{align*}
\]

These functions may already be defined. If not, it will be necessary to synthesise them. I discuss such synthesis proofs in chapter 5.

Using these functions, we can successively rewrite (4.21):

\[
\begin{align*}
\vdash \exists s' : \text{nat2} . \rho(s') &= \text{plus}(\times_\rho(\rho(\text{mod2'}(\text{n2})), \rho(\text{n2})), \rho(s)) \\
\vdash \exists s' : \text{nat2} . \rho(s') &= \text{plus}(\rho(\times_\rho(\text{mod2'}(\text{n2}), \text{n2})), \rho(s)) \\
\vdash \exists s' : \text{nat2} . \rho(s') &= \rho(\text{plus}_\rho(\times_\rho(\text{mod2'}(\text{n2}), \text{n2}), \rho s))
\end{align*}
\]

The rewriting above can be seen as a process of rippling $\rho$ outwards in the right hand side. I make use of this idea in chapter 6.

Now both sides of the equality are dominated by $\rho$, and we can apply the cancellation rule (4.20) to obtain:

\[
\vdash \exists s' : \text{nat2} . s' = \text{plus}_\rho(\times_\rho(\text{mod2'}(\text{n2}), \text{n2}), s)
\]

The existential witness is $s' = \text{plus}_\rho(\times_\rho(\text{mod2'}(\text{n2}), \text{n2}), s)$, and the program extracted from the modified synthesis theorem is:
\[ f'(l) = \rho((\lambda t. \text{list}\_\text{ind}(t, \rho^{-1}(0)),
[n_2, l, s_2, \text{plus}_\rho(\text{times}_\rho(\text{mod2}'(n_2), n_2), s_2)))(\rho^{-1}(1))) \]

The equational representation of this is:

\[
\begin{align*}
  f'(l) &= \rho(f'_\rho(\rho^{-1}(l))) \\
  f'_\rho(\text{nil}) &= \rho^{-1}(0) \\
  f'_\rho(h :: t) &= \text{plus}_\rho(\text{times}_\rho(\text{mod2}'(h), h), f'_\rho(t))
\end{align*}
\]

This does not contain conversion functions in the body of the recursion and so is much more efficient than the previous version. We obtained this efficiency by using a cancellation rule (4.20) to remove all references to the conversion function and its inverse from the existential witness. In chapter 6, I show how this can be formed into a proof strategy which uses rippling to guide the rewriting and ensure an extract term free of conversion functions.

4.5.4 Efficiency

In isolation, the function we have synthesised is only as efficient as the original. If, however, this function is only part of a larger program in which \(\lambda x. x \mod 2\) is applied widely, the conversion from nat to nat2 and back again can be made early in the program and efficiencies result (see section 4.8.2). More complex type transformations may result in efficiency improvements even when the conversion process is taken into account. For example compiling a hash table from a large list can greatly speed up membership tests.
4.6 Making use of the source proof

4.6.1 Why use the source proof?

When carrying out data type transformations on a large program, we may expect large parts of the program to remain unaltered because no transformation has taken place in this branch of the program. These parts of the source proof may be copied across to the target. In addition, some type transformations, for example forming a tuple, result in similar proofs in the source and target.

Since program synthesis is a difficult task, a synthesis proof may contain steps which required human intervention to prove. We would like to reuse these steps, and as much of the rest of the proof as possible.

4.6.2 Proof by analogy

An analogy between two formulae is a mapping of the symbols in the formulae which makes the formulae (almost) identical. Given a proof of a specification (source) and a new specification to be proved, we can try to construct an analogy between the source and target specification, and apply this analogy (as a mapping of symbols) to the inferences in the source proof to derive corresponding inferences which we hope will prove the target. [Owen 90] is a good reference on analogy theorem proving and matching algorithms. I argue in the next section that proof planning strategies and methods capture much of the similarity between proofs which is captured by analogy.
4.7 Proof planning instead of program transformation

We have already seen in §2.6 that there are some benefits to transforming program synthesis proofs rather than the proofs themselves. In the previous section, mention was made of proof by analogy. Here we consider how relevant reasoning by analogy is in the proof planning framework.

The aim of reasoning by analogy is to capture a similarity between source and target specifications and apply it to the source proof to generate a target proof. The aim of proof planning is to capture the general pattern of a family of proofs as a proof plan. We can see that the aims of the two techniques are very similar.

In his proof transformation work (§2.6), Madden transforms an abstraction of the proof, rather than the object-level proof itself. This abstraction is very similar to a proof plan. Proof plans contain the information we need at the right level of abstraction. For example, the occurrence of induction(variable,scheme) in a Whelk synthesis proof (see §2.6.2) identifies where to put DELAY declarations in the corresponding program.

In this thesis, I do not explicitly make use of the “source” synthesis proof. Instead, I assume that the proof plan which generated the source proof will also be applicable in parts of the target proof. The wave rules and lemmas which were necessary to construct the source proof will be of use in the target proof.

Program transformation is viewed as the construction of a modified proof plan for the program synthesis theorem given that a synthesis proof for the source program has already been constructed. This subject is discussed further in §9.4.3. Instead of trying to adapt planning to the direct manipulation of programs, we can make use of existing proof planning technology. Not only can we use the strong heuristics available in proof planning to reduce search problems, but we also have powerful tools with which to tackle correctness of transform-
We do not need to have access to a complete proof plan because the information present in the proof which is useful for carrying out transformations is typically available at the point in the proof plan where it is needed, either in the hypotheses and goal, or as meta-level information recorded when theorems are loaded (e.g. function definition records and wave rule records). For example, the information needed to make a DELAY declaration is present where it is needed: immediately after induction in the proof. Some lookahead may sometimes be necessary.

CLAM is only an interactive theorem prover in a fairly weak sense: when it fails to construct a proof plan, the user examines the failed proof and decides how to modify the database of loaded methods and lemmas in order to achieve a successful proof. In this sense, the user interactions will also be of use in constructing a target proof.

### 4.8 Heuristics for guiding the transformation

In Pitches, I rely on heuristics to produce a more efficient program. It is the responsibility of the user to ensure that the motivating transformations result in efficiency improvements. The type change method incorporates a number of heuristics to try to reduce application of conversion functions in the program to a minimum, and these are outlined in the next section.

#### 4.8.1 Subexpressions motivate transformations

The SETL compiler chooses a representation for a set based on the operations which are applied to it (§2.4.8). We generalise this motivation by associating representation changes with particular functions. These representation changes are stored in a library, indexed by the expressions which motivate them. For example, one such transformation is:
<table>
<thead>
<tr>
<th>Function</th>
<th>Argument type changes</th>
<th>Output type change</th>
</tr>
</thead>
<tbody>
<tr>
<td>append(x, y)</td>
<td>$x: (T \text{ list}) \xrightarrow{\rho} x': (T' \text{ dlist})$, $y: (T \text{ list}) \xrightarrow{\rho} y': (T' \text{ dlist})$</td>
<td>$(T \text{ list}) \xrightarrow{\rho} (T' \text{ dlist})$</td>
</tr>
</tbody>
</table>

This determines how to change the types of certain subexpressions when they are encountered during synthesis. In the next section we show how the propagation of type changes can improve efficiency by minimising the need for conversion functions.

### 4.8.2 Propagation minimises conversions

Suppose we have a function:

$$f(x) = g(h_1(x), h_2(x))$$

If we transform $t$ to $t'$ in $h_1$ with conversion function $\rho$, then $h_1$ will be defined by:

$$h_1(x) = \rho(h'_1(\rho^{-1}(x)))$$

If we can also transform $h_2$, so $h_2(x) = \rho(h'_2(\rho^{-1}(x)))$, then we can compute $\rho^{-1}(x)$ in the body of $f$ before calling $h_1$ and $h_2$, saving one application of $\rho^{-1}$. Thus, we see that if a transformation step is made in a subcall within a function, it is advantageous to also make it in other subcalls of the function. If we propagate transformations up the program tree, we increase the scope for this kind of optimisation.

Propagation is even more effective when we are transforming recursive programs, indicated by the presence of induction in the synthesis proof. This is discussed in the next chapter, when the rules for propagation of type changes are presented.
4.9 Conclusion

In this chapter, the correctness of the type change transformation was proved in the type theory (theorem 4.1). Each change of type must be proved to be an implementation. The technique was illustrated with a simple example.

Abstract data types are encoded as existential types in Martin-Löf's Type Theory, allowing the complex types used in type changes to be expressed at an appropriate level of abstraction.

Instead of making explicit reference in the transformation to a source program, I assume that a proof plan for the source program has already been constructed. The methods and lemmas which were used in its construction will also be of use in the construction of a proof plan for the target. The information which we require from the proof is generally available at the point it is needed.

I demonstrate the technique with an example proof. Making a type change at the wrong point in a proof can result in an inefficient synthesised function. I introduce heuristics for choosing which type change to make and where in a proof to make it. These heuristics are formalised in the next chapter.
Chapter 5

The Type Change Method

5.1 Introduction

This chapter describes a type change method for information preserving transformations. Extension of the method to type abstraction is described in chapter 7. The specification of the basic data type changes, which are motivated by the presence of certain inefficient operations in the program, is described.

In order to avoid inefficiency caused by the conversion of data from one type to another, propagation rules are applied to move the type change so that it covers as much as possible of the proof/program. The most important rules are for propagation past induction.

When the type change method is applied to a goal, it looks ahead in the proof process until a transformation is triggered by one of the motivation rules. Propagation rules then propagate the type change back through the proof and so determine the type change to be made in the goal.

The method is illustrated with an analysis of the proof plan produced for a list to difference list transformation.
5.2 Notation

In order to describe type changes succinctly and accurately it is necessary to introduce some notation.

A type change is determined at a point in a synthesis proof by analysing the specification at that point. Currently, however, type changes are only made in synthesis goals which correspond to object-level sequents of a restricted form:

$$\text{Hyps} \vdash \forall \bar{x} : t_{\bar{x}} . \exists z : t_z . z = f(\bar{x}) \text{ in } t_z$$

Hyps is a list of hypotheses. $\bar{x} : t_{\bar{x}}$ is a set of variables with their respective types, and $f$ is a term which contains no logical connectives. The equality can also be the other way round ($f(\bar{x}) = z$). The names of variables, types and functions may be different to those above.

To simplify the notation in this chapter, I write the function expression (the $f(\bar{x})$ above) as a shorthand for the entire specification.

A term:

$$f(\bar{x} : t) \sim f'(\bar{x}' : t') : t'_0$$

denotes a type change in the argument of $f$, from type $t$ to type $t'$, and a change in the type of the output of $f(\bar{x})$ from type $t_0$ to $t'_0$.

If the conversion functions are $\rho : t' \rightarrow t$ and $\rho_0 : t'_0 \rightarrow t_0$, then the type change term above gives rise to the following specification of $f'$:

$$\forall \bar{x}' : t'. \rho_0(f'(\bar{x}')) = f(\rho(\bar{x}'))$$

$\sim$ is not a connective in the object-level logic. Propagation rules (5.3) are not part of the object-level logic, but are abbreviations of expressions in the meta-language. As well as $\sim$, the following terms are used in the presentation of the rules:
1. \( f(x) := \) expression. In the CLAM meta-language this translates to the existence of an appropriate func_defeqn record.

2. Step case \( f([c(x)]) \Rightarrow [g(f(x))] \). This translates to an application of the step_case/1 method, with input \( f([c(x)]) \) and output \( [g(f(x))] \).1

5.3 Propagation of type changes

5.3.1 Introduction

We saw in \( \S 4.8.2 \) that making a type change early in the proof can eliminate some inefficiency caused by the introduction of conversion functions.

Definition 5.1 A propagation rule specifies how the scope of a type change can be increased by replacing type changes in the subgoals of a proof step with a single type change before the proof step is applied.

I define propagation rules for function composition, function evaluation, and induction on parameterised and unparameterised types.

The ultimate goal of such a propagation process is to spread a type change, which was initiated by the need to make more efficient a certain operation used deep within a function, through the entire function. Sometimes the functions on the new type which correspond to those on the old type already exist. Other times they must be synthesised. When the type change has been propagated up throughout the whole function, conversion functions are only necessary at the interface between the new function and the rest of the program.

The most significant savings are to be made when a type change is propagated past induction. The two other propagation rules result in slight savings, but

---

1 As before, these expressions are abbreviations for meta-level sequents.
more importantly they may move the type change to a point where one of the induction propagation rules can apply.

Propagation past induction (recursive loops in programming terms) involves moving the conversion of a data structure from inside a loop to outside, and can result in major efficiency improvements.

5.3.2 Propagation rule syntax

In order to formalise the conditions and effects of a type change, propagation rules are written in a fairly abstract syntax.

A propagation rule describes how type changes in the subgoals which result after application of a proof rule can be replaced by a single type change. For example, (5.1) specifies how type changes in two functions which are composed can be combined into a single type change on the composition. Formulae above the line are conditions/premises. Formulae below the line are effects/conclusions. The name of each propagation rule is written to the right of the line. The rules can be used in a forward direction, in which case whenever the premises of a rule are satisfied, the rule is applied and leads to satisfaction of its conclusions. Alternatively, the rule can be used backwards in a goal-directed fashion, in which case an attempt to satisfy the conclusion(s) of the goal leads to an attempt to satisfy its premises.

In the typechange/2 method, the propagation rules are used in a goal-directed, bottom-to-top direction, just as inference rules are used in goal-directed theorem proving. This entails looking ahead in the synthesis proof, for example applying induction/2 then step_case/1.

This notation allows us to specify precisely the location and scope of a type change. Primed versions of functions and variables generally denote analogues of those functions and variables for the new types. Conversion functions are not written in the body of the abstract propagation rule, but given the conversion functions associated with a type change, the specifications of the primed functions can be deduced.
5.4 Initial choice of transformation

Certain specifications are used to motivate transformations during the planning of a synthesis proof. When such a specification is encountered, it triggers the application of a motivation rule, which determines how to change the types of the inputs and outputs in the specification to improve the efficiency of the motivating specification.

In a formal derivation of the type change, this is written:

\[ f(x : t) : t_1 \sim f'(x' : t') : t'_1 \text{motiv}(f) \]

For example, the nat to nat2 transformation of §4.5 is triggered by a rule:

\[ \text{mod2}(h : \text{nat}) : \text{nat} \sim \text{mod2}(h' : \text{nat2}) : \text{nat2} \text{motiv(mod2)} \]

5.5 Propagation past function composition

The propagation rule which was discussed in §4.8.2 was propagation past function composition.

In propagation past function composition it is necessary to ensure that the transformed types match.

\[ f(x) := g(h(x)) \quad (5.1) \]
\[ h(x : t_x) : t_h \sim h'(x' : t'_x) : t'_h \]
\[ g(x : t_h) : t_g \sim g'(x' : t'_h) : t'_g \]
\[ f(x : t_x) : t_g \sim f'(x' : t'_x) : t'_g \text{Fcomp} \]

Where
\begin{align*}
\rho : t'_x \rightarrow t_x, \ \rho_h : t'_h \rightarrow t_h, \ \rho_g : t'_g \rightarrow t_g \\
\forall x' : t'_x . \rho_h(h'(x')) &= h(\rho(x')) \\
\forall x' : t'_h . \rho_g(g'(x')) &= g(\rho_h(x')) \\
\forall x' : t'_x . \rho_g(f'(x')) &= \rho_g(h'(x')) = g(\rho_h(h'(x'))) \\
&= g(h(\rho(x'))) = f(\rho(x'))
\end{align*}

When this rule is used to determine a transformation, we can set up the following synthesis goal for the function \( f' \):

\[ \forall x' : t'_x . \exists z' : t'_g . \rho_g(z') = f(\rho(x')) \]

Rewriting with the definition of \( f \), we obtain:

\[ \forall x' : t'_x . \exists z' : t'_g . \rho_g(z') = g(h(\rho(x'))) \]

Note that there is no type change between \( g \) and \( h \). The information contained within the propagation rule specifying the output transformation on \( h \) and the input transformation on \( g \) has been lost.

The preferred alternative is to split the synthesis goal into two by synthesising \( g' \), \( h' \) separately:

\[ \vdash \forall x' : t'. \exists z'_0 : t'_0 . \rho_0(z'_0) = h(\rho(x')) \]
\[ \vdash \forall x'_0 : t'_0 . \exists z'_1 : t'_1 . \rho_1(z'_1) = g(\rho_0(x'_0)) \]

This makes the synthesis goal above trivial to prove by composing the implementations \( g' \), \( h' \) of \( g \), \( h \).
5.6 Propagation past function evaluation

This rule propagates type changes past simple function definitions.

\[ f(x) := h(x) \]  \hspace{1cm} (5.2)  
\[ h(x : t) : t_0 \sim h'(x' : t') : t'_0 \]  
\[ \text{Feval} \]

Where

\[ \rho : t' \rightarrow t, \rho_0 : t'_0 \rightarrow t_0 \]
\[ \forall x' : t', \rho_0(h'(x')) = h(\rho(x')) \]
\[ \forall x' : t', \rho_0(f'(x')) = f(\rho(x')) \]

This rule is a special case of (5.1) with \( g \) and \( g' \) the identity functions on \( t_0 \) and \( t'_0 \) respectively.

5.7 Propagating past induction

5.7.1 Introduction

For the purposes of description, it is convenient to divide propagation past induction into two cases:

1. Where the argument of a parameter type is being changed. For example, changing list(nat) to list(binary).

2. Where an entire type is being changed, for example nat to nat2 or list(nat) to difference list.
In the nat to nat2 example of §4.5, the expression which motivated the type change occurred within the body of a recursive definition. When a type change was made inside the recursive loop (after the induction in the corresponding synthesis proof), the synthesised program was inefficient because it contained repeated applications of conversion functions and their inverses.

Consider a recursively defined function, with recursive case:

\[ f(c(x)) := g(f(x)) \]

Here, \( c \) is a constructor function, for example adding an element to a list, or forming the successor of a natural number. In the synthesised program, a corresponding destructor function will appear.

If we make a type change of \( x \) in \( g(x) \), then this recursive definition will be replaced by:

\[ f(c(x)) = \rho(g'(\rho^{-1}(f(x)))) \]  \hspace{1cm} (5.3)

where \( g'(x) \) is the version of \( g(x) \) on the new type.

Now, we can see what happens when this recursive function is called by symbolically iterating the application of \( f \):

\[ f(c(c(x))) = \rho(g'(\rho^{-1}(\rho(g'(\rho^{-1}(f(x))))))) \]

This is clearly unacceptable.

5.7.2 Propagation past induction on a parameterised type

It is useful to identify several different ways in which a parameterised type can be changed:

1. The entire type can be changed. The new type need not be parameterised. \( t(t_1) \xrightarrow{\rho} t' \).

2. The inner type can be changed, \( t(t_1) \xrightarrow{\sim} t(t'_1) \).
3. The outer type can be changed, \( t(t_1) \rightarrow t'(t_1) \).

Although the last two are instances of the first, we can define propagation rules which exploit special properties of each of the three categories above. The first case is handled by the rule for simple inductions (5.9). The second case is considered in this section. Consideration of the third case is left as further work (§10.4.2).

In order to determine the form of the propagation rule, I show how a type change \( t(t_1) \rightarrow t'(t'_1) \) affects the step case of a subsequent induction.

Consider a synthesis from the following specification:

\[
\forall x : t(t_1). \exists z : t_0 . z = f(x)
\]

First of all I present a "source" synthesis proof, that is, the proof before a type change, and then I show that steps in this proof are useful for the transformed proof.

The source proof

The source proof proceeds by induction on \( x : t(t_1) \).

The step case of the induction is:

\[
\begin{align*}
& \quad h : t_1 \\
& x : t(t_1) \\
& \exists z_{t_0} : t_0 . z_{t_0} = f(x) \\
& \quad \vdash \exists z' : t_0 . z' = f(\langle c(h,x) \rangle)
\end{align*}
\]

The constructor term is \( \langle c(h,x) \rangle \). For example, if \( t(t_1) \) is list(pnat),\(^2\) then the constructor term is \( h::x \).

\(^2\)pnat list is the usual Oyster syntax.
For (weak) fertilisation to apply, the constructor term must be rippled out. This can be achieved by rippling with several wave rules in turn. The compound ripple can be written:

\[ f(c(h, x)) \Rightarrow g(h, f(x)) \]  

(5.4)

\( g \) may be some compound function.

Applying (5.4) to the step case conclusion and weak fertilising gives:

\[ \exists z' : t_0 . z' = g(h, z_{\text{in}}) \]

Introducing the term \( g(h, z_{\text{in}}) \) for \( z' \) completes the step case of the proof and yields a recursive extracted program.

**The target proof**

In order to derive the form of the propagation rule, and see how the source proof can be used, I will first make the type change and give the modified synthesis proof, and then demonstrate how this type change can be determined by looking at the source proof.

Perform type changes \( t(t_1) \xrightarrow{\bar{\rho}} t(t'_1) \) and \( t_0 \xrightarrow{\rho_0} t'_0 \). The conversion function \( \bar{\rho} : t(t'_1) \rightarrow t(t_1) \) is defined by wave rule (5.6) and equation (5.5). This structural definition of \( \bar{\rho} \) restricts \( t \) to constructor-based data types.

\[ \bar{\rho}(\text{empty}) = \text{empty} \]  

(5.5)

\[ \bar{\rho}(c(h', x)) \Rightarrow c(\rho(h'), \bar{\rho}(x)) \]  

(5.6)

The goal after the type change is:

\[ \vdash \forall x' : t(t'_1) . \exists z' : t'_0 . \rho_0(z') = f(\bar{\rho}(x')) \]
Perform induction on \( x' \). The step case is:

\[
\begin{align*}
    h' &: t' \\
    x' &: t(t' \\
    \exists z_{in} &: t_0 \cdot \rho_0(z_{in}) = f(\overline{\rho(x')}) \\
    \vdash \exists z &: t_0 \cdot \rho_0(z) = f(\overline{c(h',x')})
\end{align*}
\]

Ripple with (5.6):

\[
\vdash \exists z &: t_0 \cdot \rho_0(z) = f(\overline{c(\rho(h'),\overline{\rho(x')})})
\]

Now ripple with (5.4):

\[
\vdash \exists z &: t_0 \cdot \rho_0(z) = g(\rho(h'), f(\overline{\rho(x')})
\]

Again, we can weak fertilise to get:

\[
\vdash \exists z &: t_0 \cdot \rho_0(z) = g(\rho(h'), \rho_0(z_{in}))
\] (5.7)

As we have seen earlier (§5.7.1), introducing the term \( \rho_0^{-1}(g(\rho(h'), \rho_0(z_{in})) \) for the existential witness yields an inefficient extracted program. The goal, however, is exactly what we would expect if we make a type change on \( g \),

\[
g(h : t_1, x : t_0) : t_0 \leadsto g'(h' : t'_1, x' : t'_0) : t'_0
\]

Given such a type change, the right hand side of (5.7) can be rewritten to \( \rho_0(g'(h', z'_{in})) \). The \( \rho_0 \) can be cancelled from each side of the equation, giving the witness \( g'(h', z'_{in}) \) for \( z \) and yielding an efficient recursive synthesised function as required.

Note the following points:

1. Except for the wave rule for \( \overline{\rho} \) (5.6), the same wave rules are used in the source and target to allow weak fertilisation. Therefore, if the source proof plan has been constructed, and a wave rule such as (5.6) exists, the target proof is guaranteed to proceed as far as weak fertilisation.
2. The type change on \( f \) is determined by the type change on \( g \) (and vice versa):

\[
g(h : t_1, x : t_0) : t_0 \leadsto g'(h' : t'_1, x' : t'_0) : t'_0 \text{ if } f(x : t(t_1)) : t_0 \leadsto f'(x' : t(t'_1)) : t'_0
\]

These points are summarised in the following rule:

\[
\text{Step case } f(c(h, x)) \Rightarrow g(h, f(x)) \quad (5.8)
\]

\[
g(h : t_1, x : t_0) : t_0 \leadsto g'(h' : t'_1, x' : t'_0) : t'_0 \quad \text{ParmInd}
\]

\[
f(x : t(t_1)) : t_0 \leadsto f'(x' : t(t'_1)) : t'_0
\]

Where:

\[
\rho_0 : t'_0 \rightarrow t_0, \quad \rho_1 : t'_1 \rightarrow t_1, \quad \bar{\rho}_1 : t(t'_1) \rightarrow t(t_1)
\]

\[
\forall h' : t'_1 \forall x' : t'_0 \rho_0(g'(h', x')) = g(\rho_1(h'), \rho_0(x'))
\]

\[
\forall x' : t(t'_1) \rho_0(f'(x')) = f(\bar{\rho}_1(x'))
\]

\( \bar{\rho}_1 : t(t'_1) \rightarrow t(t_1) \) is defined recursively by wave rule \((5.6)\) and equation \((5.5)\).

This type change necessitates an initial conversion of the data to the new data type outside the loop. As long as the recursion accesses each element of the collection at least once, the transformed function will be no less efficient than the original. In fact, this is the condition used by the SETL compiler \((\S 2.4.8)\) when deciding whether to move a transformation outside of a loop.

### 5.7.3 Propagation past induction on an unparameterised type

The rule for propagation past induction on an unparameterised type can be derived from that for induction on a parameterised type as a degenerate case.

Let \( t : \text{type} \rightarrow \text{type} \) be the identity, \( \lambda x.x \). Then \( t(t_1) = t_1, t(t'_1) = t'_1 \), and \( h \) is deleted from the rules. The full rule is:
The output type is governed by the type change in the step case function, \( g \), with the restriction that the input and output transformations on \( g \) must be identical. The input type of \( f \) must remain unchanged, otherwise the induction used to determine the type change would change. As for the ParmInd rule, the target proof uses the same wave rules to reach weak fertilisation as those used in the source proof.

It may be profitable to loosen the restriction on the input of \( f \). Application of this rule would then not guarantee that the type change produced a more efficient function, because the proof/program expression which motivated it may no longer occur in the synthesis due to the change of induction schema. It may still be worth investigating whether such transformations are effective in practice.

5.8 An example: difference lists

5.8.1 Introduction

A well known example of program improvement by type transformation is the use of difference lists instead of lists in a Prolog program to improve the execution of append, which becomes simply unification instead of a recursively defined function.

In this section the technique is illustrated by transforming a function which flattens trees, changing the output type from lists to difference lists. The example
chosen is adapted from the *flatten* example in [Sterling & Shapiro 86, pp. 239-247].

5.8.2 Difference Lists

The standard Prolog formulation of difference lists, making use of free variables and unification, is not amenable to a simple axiomatisation.

For our purposes, we need only specify that difference lists are like lists, and define an extra induction scheme. The fact that the difference list equivalent of the append function is more efficient than list append is a piece of meta-level information.

In [Marriott & Sondergaard 88], several problems are identified with the naive translation of Prolog programs to use difference lists. These problems cannot occur in our formulation, primarily because we have a proper equality theory. The function dlist_unwrap ensures that every difference list corresponds to a real list, whereas the standard formulation allows difference lists such as (a :: nil, b :: nil) which do not represent any list.

5.8.3 The flatten synthesis theorem

The statement of the flatten synthesis theorem is of the following form:

\[ \exists \text{tree} : u(1). \quad \exists \text{bool} : u(1). \quad \exists \text{dlist} : u(1). \quad \forall t : \text{tree} \exists l : \text{pnat list}. \quad l = \text{flattent}(t) \text{ in } \text{pnat list} \]

The three ADTs are specified follows:

1. An abstract data type for binary trees, with leaves labelled with elements of \text{pnat}. \text{flattent : tree } \rightarrow \text{ pnat list} is declared and implicitly defined in the equation part of the ADT.
The standard induction schema over trees is defined.

2. An abstract data type for booleans.

3. An abstract data type for dlist. Functions dlist\_list and list\_dlist are defined to convert between lists and difference lists. A function app\_dl is defined which is the equivalent on difference lists of append on lists.\(^3\) A split predicate is defined but will not be used in the proofs in this thesis. Two induction schemata are declared, as in [Clark & Tärnlund 77].

dlist\_ind1 corresponds to structural induction on lists. dlist\_ind2:

\[
p(\text{empty\_dl}), \forall x: \text{pnat}. p(\text{build}(x)), \forall l, r: \text{dlist}. p(l) \land p(r) \rightarrow p(\text{app\_dl}(l, r))
\]

\[
\forall l: \text{dlist}. p(l)
\]

allows proof of a predicate over difference lists by proving it true for the empty difference list, all singleton difference lists, and all difference lists constructed by appending two smaller difference lists.

The proof plan is constructed automatically by CI\textsc{AM} using the standard proof strategy for induction supplemented by the type change method.

### 5.8.4 Wave rules and equations

The following wave rules and equations are used in this proof. The equations for flattent are from the tree ADT, and (5.13) is from the dlist ADT.

\[
\text{flattent(}\text{empty\_tree}) = \text{nil} \tag{5.10}
\]

\[
\text{flattent(leaf}(x)) = x :: \text{nil} \tag{5.11}
\]

\[
\text{flattent(node}(l, r)) \Rightarrow \text{app(flattent}(l), \text{flattent}(r)) \tag{5.12}
\]

\[
\text{app(dlist\_list}(x), \text{dlist\_list}(y)) = \text{dlist\_list(app\_dl}(x, y)) \tag{5.13}
\]

\(^3\)In the appendices app\_dl is written as append\_dl.
Figure 5–1 shows the equations from which the source program is synthesised.

\[\text{flatten(node(l,r))} := \text{app(flatten(l),flatten(r))}\]

\[\text{flatten(leaf(n))} := n::\text{nil}\]

**Figure 5–1:** The original *flatten* function.

### 5.8.5 The proof plan

#### Parsing the ADTs

During the first part of the proof the ADTs are parsed in turn. Each equation in the equation part of the ADT is given a name and run through the wave rule parser. The induction schemata are parsed and records made in a database so that they can subsequently be used by the new clause of the scheme/5 predicate.

#### The type change

The most important part of the proof is the determination of the type change. This is carried out by the typechange/2 method, which automatically chooses a transformation and decides where in the proof to apply it.

After parsing the ADTs, the typechange/2 method analyses the goal:

\[\forall t:\text{tree} \exists l:\text{pnat list} . l = \text{flattent(t)} \text{ in pnat list}\]

Since flattent is defined recursively, it looks ahead in the proof by performing induction on \( t : \text{tree} \) to get the step case:

\[\exists l:\text{pnat list} . l = \text{flattent(node(left,right))} \text{ in pnat list}\]

After rippling with (5.12) this becomes:
\[ \exists l: \text{pnat list}, l = \text{app(flattent(left), flattent(right))} \] in \text{pnat list} \]

Now the type change is motivated by the presence of \text{app(\_, \_)}. This induces the synthesis of a new flatten function of type \text{tree \rightarrow difference list}.

Formally, the type change is derived by the following application of propagation rules:

\[
\text{app(fl: list, fr: list): list} \leadsto \text{app\_dl (fl': dlist, fr': dlist): dlist} \quad \text{motiv(app)}
\]

\[
\text{Step case: flatten\([\text{node(l, r)}]\)} \Rightarrow \text{app(fl, fr)}
\]

\[
\text{flatten(t: tree): list} \leadsto \text{flatten'(t: tree): dlist} \quad \text{Ind}
\]

Once the type change has been determined it is translated into the following revised synthesis goal:

\[
\forall t: \text{tree}. \exists d\_\text{list}, \text{dlist\_list(dl)} = \text{flattent(t)} \in \text{dlist}
\]

Proof is by induction on \text{t: tree}. The full proof plan is listed in appendix D.2.

Figure 5-2 shows the transformed program in an equational form.

\[
\text{flatten(t)} := \text{dlist\_list(flatten\_dl(t))}
\]

\[
\text{flatten\_dl(node(l, r))} := \text{app\_dl(flatten\_dl(l), flatten\_dl(r))}
\]

\[
\text{flatten\_dl(leaf(n))} := \text{list\_dlist(n::nil)}
\]

**Figure 5-2**: The transformed \text{flatten} function. \text{list\_dlist} and \text{dlist\_list} convert between lists and difference lists.
5.9 Other ways of motivating type changes

One example of a simple induction type change is $\text{+ : nat} \rightarrow \text{nat} \rightarrow \text{nat}$ to $\text{+bin : bin} \rightarrow \text{bin} \rightarrow \text{bin}$. This cannot be motivated by the expressions encountered during the synthesis proof because the function $\lambda x.s(x)$, is less efficient in bin than in nat. It would be possible to motivate this transformation by reasoning that it splits the problem into more equal subproblems. This is like the choice of algorithm design method in KIDS. Alternatively, we could allow the presence of any arithmetic function to motivate a nat to bin transformation.

5.10 Direction of transformations

In view of the desirability of extending a type change to as much of the program as possible, there are two possible control regimes:

1. Perform type changes on program fragments at the leaves of the program tree and extend up towards the root.

2. Perform type changes close to the root and extend down to the leaves.

The direction of program synthesis is top-down, breaking a large specification into several smaller specifications, until simple specifications are reached at the leaves of the proof. The transformation process works in the opposite direction, bottom-up, motivating a transformation by expressions in the leaves of the synthesis, and extending them gradually upwards through the proof. There are two ways in which this conflict can be resolved:

1. The transformation can be chosen by analysing a source proof/program using the propagation rules. This is what happens in dataflow analysis (§9.2.1).
2. The transformation method can look ahead through the proof process. This is how the typechange/2 method (§5.13) works. When the method is applied to a goal, it applies induction and other program derivation steps which have associated propagation rules in an attempt to find a motivating program fragment. The propagation rules then determine the transformation induced on the goal.

5.11 Flexibility in type changes

A motiv rule specifies that a particular program expression can be made more efficient by changing the types of its inputs and or output. Sometimes, it is desirable to make a type change in an expression as long as it does not make that expression less efficient. This is a weaker criterion than that above.

There are two circumstances when this weaker criterion is enabled:

1. After application of a rule for propagation past induction (5.8, 5.9). This increases the chance that the conditions imposed by these rules, e.g. that the input and output transformations be equal, can be satisfied.

2. After application of a rule for propagation past function composition (5.1), for example in a composition \( f(g_1, \ldots, g_n) \). This has two beneficial effects:

   (a) It increases the chances that the output transformations of the arguments \( g_i \) will match the input transformations of \( f \). When these do match, the type changes in the \( g_i \) can be propagated up past \( f \), and there is the possibility that a synthesis which fuses the \( g_i \) with \( f \) can be performed.

   (b) It increases the chances that a function \( f: t \to t \to \ldots \to t \to t_0 \) can be transformed to a function \( f': t' \to t' \to \ldots \to t' \to t'_0 \). As I showed in §4.8.2, this may allow variables which appear several times in the compound expression to be converted to the new type once
but used several times. It also increases the possibility of reuse of the transformed function. A \( +' \) function of type \( \text{nat2} \to \text{nat2} \to \text{nat2} \) is likely to have more possibility for reuse than one of type \( \text{nat2} \to \text{nat} \to \text{nat} \). This is particularly the case if there is a commutativity law for \( +' \): if \( x + y = y + x \), then we would like \( x +' y = y +' x \), which requires the types of \( x \) and \( y \) to be the same.

These weaker transformations are specified by \textit{allow} rules. The most frequently used of these is a rule which allows a variable to have its type changed arbitrarily. The result is an anonymous (free) new variable and an anonymous (free) new type. These anonymous variables become instantiated by the heuristic mentioned above.

\[
\overline{v : t \mapsto -} 
\text{allow(var)}
\]

### 5.12 An analysis of the \texttt{nat} to \texttt{nat2} example

I illustrated chapter 4 with the \texttt{nat} to \texttt{nat2} example of §4.5. Figure 5-3 shows how this type change is derived using propagation and motivation rules. The \textit{allow(var)} rule is used to produce versions of \( + \) and \( \times \) which are of type \( \text{nat2} \to \text{nat2} \to \text{nat2} \) in line with the heuristic of 2b in §5.11.

### 5.13 The \texttt{typechange/2} method

The \texttt{typechange/2} method applies motivation and propagation rules to a synthesis goal, as described in this chapter. If it succeeds, it returns the list of synthesis subgoals resulting from the type change. If it fails, then other methods may be able to break the synthesis goal into simpler goals in which type changes can be made. This enables different type changes to be made in different parts of the synthesis proof (related by conversion functions), as suggested by [Low 78] (§2.4.8).
The implementation of the typechange/2 method is discussed further in §8.2.

5.14 Conclusion

This chapter has explained how changes of type are chosen. Initially, a transformation is motivated by the presence in the proof of an expression which can be made more efficient by a known type change. In order to avoid inefficiency caused by the conversion of data from one type to another, propagation rules are applied to move the transformation as high as possible in the proof/program. The most important rules are for propagation past induction, which not only improve the efficiency of the target program, but guarantee that part of the target proof plan can be constructed in the same way as the corresponding part of the source proof plan.

Some extra flexibility is permitted in the type changes which are allowable after application of a propagation rule.

The chapter was illustrated with two examples. The first example showed the derivation of the transformation and subsequent synthesis goal for a list to difference list transformation. The second (§5.12) illustrated how the nat to nat2 type change in (§4.5) is derived using propagation rules.
Figure 5-3: Derivation of the nat list to nat2 list transformation.
6.1 Introduction

In this chapter I demonstrate a way in which proving the synthesis goals which arise after a type change can require search.

In program synthesis, the problem of search control is much more serious than in program verification, since the need to explicitly introduce existential witnesses during the proof can result in infinite branching points.

I propose a proof strategy which exploits the common pattern of the synthesis goals which arise after an implementation type change. It consists of rewriting followed by cancellation of the conversion functions. This yields an explicit witness for the existential quantifier, and produces extracted programs of a certain desirable form.

The final proof step, cancellation, is similar to the fertilisation step in inductive proofs. Using difference matching (§6.4.1) we can introduce annotations which allow rippling to control the rewriting, greatly reducing the search space. The fact that the proof strategy which I have identified can be described so well in terms of rippling demonstrates the flexibility of rippling as a proof method.
The difference matching proof strategy is most valuable in difficult synthesis examples. It is illustrated with the synthesis of a function for adding binary numbers.

6.2 Branching during synthesis

6.2.1 Introduction

For the moment I consider only type changes from type $t$ to type $t'$ in a function $f: t \rightarrow t$. I discuss the proof strategy for more general type changes in §10.5.2.

After the type change, I synthesise a new function $f': t' \rightarrow t'$ from a specification:

$$\forall x': t'. \exists z': t'. p(z') = f(p(x')) \quad (6.1)$$

In this section I identify a source of search in such a synthesis proof, and show how it can be eliminated. The section is illustrated by returning to the list to difference list example of §5.8.

6.2.2 A simple example

Let us return to the difference list example of §5.8. Refer back for details of that proof.\footnote{For typesetting reasons, in this chapter I write infix $<>$ and $<>_{\text{at}}$ in place of \texttt{app} and \texttt{app}_{\text{at}} respectively.}

After the type change and weak fertilisation, the goal is:

$$x, y : \text{dlist} \vdash \exists z : \text{dlist}. \text{dlist} \cdot \text{dlist} \cdot \text{dlist} (z) = \text{dlist} \cdot \text{dlist} (x) <> \text{dlist} \cdot \text{dlist} (y) \quad (6.2)$$
In the proof, the base_case/1 method proved this subgoal, rewriting using equation (6.3) and applying the existential method to determine the existential witness.

\[
\forall d1, d2 : \text{dlist_dlist_list}(d1) <> \text{dlist_list}(d2) = \text{dlist_list}(d1 <>\alpha_d d2) (6.3)
\]

\[
\forall l : \text{list}, d1 : \text{dlist_dlist_list}(d1) <> l = d1 <>\alpha_d l (6.4)
\]

\[
\forall x, y : \text{dlist_dlist_list}(x) = \text{dlist_list}(y) \leftarrow x = y (6.5)
\]

If we also have equation (6.4) available, then we are faced with a choice point in the proof. Application of (6.4) to (6.2) gives:

\[
\exists z : \text{dlist_dlist_list}(z) = x <>\alpha_d \text{dlist_list}(y)
\]

This can be solved if we have available a function \( \text{dlist_list}^{-1} \) such that

\[
\forall x : \text{dlist_dlist_list}^{-1}(\text{dlist_list}(x)) = x (6.6)
\]

In fact, such a function does not exist. All of the many different representations for a list must be mapped to the same list by \( \text{dlist_list} \), e.g.

\[
\text{dlist_list}([(1, 2, 3), [3]]) = \text{dlist_list}([(1, 2, 3, 4), [3, 4]]) = [1, 2]
\]

Therefore:

\[
\text{dlist_list}^{-1}((\text{dlist_list}([(1, 2, 3), [3]]))) = \text{dlist_list}^{-1}((\text{dlist_list}([(1, 2, 3, 4), [3, 4]]))
\]

Clearly (6.6) does not hold.
Even if the \textit{list} to \textit{difference list} transformation were an isomorphism, giving us a true inverse \texttt{dlist\_list}^{-1}, the program fragment which would be extracted, \texttt{dlist\_list}^{-1}(\texttt{x <\_d} \texttt{dlist\_list(y))} contains applications of the conversion function and its inverse. As I showed in §5.7.1, this yields an inefficient synthesised program. I would like to avoid such extract terms. In the next section I describe the form of proof I wish to obtain, and introduce a proof strategy which attains this form.

6.3 Developing a proof strategy

6.3.1 Restrict search

Search control is a serious problem in program synthesis. It is sometimes necessary to choose explicit witnesses for existential quantifiers, creating an infinite branching point in the search space. In addition, the possibility of speculating lemmas (e.g. wave rules) must be taken into account. It is essential to prune unpromising branches of the search space as early as possible. This is especially true when user interaction takes place because it is important to avoid swamping the user with choices (§2.2.2).

In this section, I demonstrate that by using rippling to guide the proof, some search in type change synthesis proofs can be eliminated, and efficient programs synthesised.

6.3.2 Avoid conversion functions

I showed in §5.7.1 that when conversion functions are used in the body of a recursive function, the continual need to convert from the old to the new type and back again causes inefficiency. Ideally, I would like to produce extract terms which do not contain applications of the conversion functions.
In order to achieve this aim, I exploit the common form of the synthesis
conjectures which arise after a type change.

The final step of the proof in §4.5.3 was the application of rewrite rule (6.5).
This yielded an explicit witness which was free of conversion functions.

As a heuristic, I insist that the final step in the proof be cancellation of
conversion functions, using a rule like (6.5). As long as there are no conversion
functions in the goal apart from those directly involved in the match with (6.5),
this will yield an extracted program of the desired form.

6.4 Using rippling to reduce search

6.4.1 Difference matching

Rippling was originally conceived as a technique for mechanising the rewriting
process in inductive proofs. In the step case of an inductive proof, annotations
are introduced to mark the differences between the induction conclusion and the
induction hypothesis. Difference unification [Basin & Walsh 93] extends the
applicability of rippling by determining the differences between two formulae
and annotating them appropriately.

Here we restrict ourselves to difference matching [Basin & Walsh 92]. The
difference matcher takes as input two terms, a source, \( s \), and a target, \( t \), and
returns an annotated term \( s' \) together with an unannotated substitution \( \sigma \) such
that:

1. \( \text{erasure}(s') = s \)
2. \( \sigma(\text{skeleton}(s')) = t \)

The result of difference matching may not be unique, so the difference matcher
returns alternative matches on backtracking. When there is no \( s', \sigma \), such that
(1) and (2) above hold, difference matching fails.
6.4.2 Difference matching with the cancellation rule

The standard form method was developed for summing series [Walsh et al 92]. It has been modified for proofs using conversion functions such as nat in §6.8.

Since I have determined that the final step of the synthesis proof should be application of a cancellation rule such as (6.5), I difference match the synthesis goal with the left hand side of the cancellation rule. If the cancellation rule is of the form (6.7), then the target will be (6.8), built from the conversion functions specified in the type change.

\[ \forall x, y: t \cdot \rho(x) = \rho(y) \leftarrow x = y \quad (6.7) \]
\[ \forall x, y: t \cdot \rho(x) = \rho(y) \quad (6.8) \]

Subsequent rippling reduces the differences between the goal and the target, just as it does in an inductive proof, where the target is the induction hypothesis. If the conversion functions can be fully rippled out, leaving a subterm of the goal which matches the target, then the appropriate cancellation rule can be applied to yield the explicit extract term which completes the proof. Application of the cancellation rule is analogous to fertilisation in an inductive proof.

Cancellation removes conversion functions, so they should not appear in the extract term. Such a property could also be obtained in principle by checking that the extracted program has the correct form at the end of the proof, and forcing the proof planner to backtrack and produce alternative proofs. The strategy I propose greatly reduces search in addition to yielding an acceptable extract term.

6.4.3 The standard form/2 method

The synthesis goals to which we apply difference matching are of the form:

\[ \forall x: \tau \exists y: \tau_y. \text{lhs}[y] = \text{rhs}[x] \]
The existentially and the universally quantified variables appear on different sides of the goal. For this reason, it is possible to simplify the target against which we match so that it contains the same variables on each side; the unifier which is returned by the difference matching procedure takes account of the difference.

If $\rho$ is the conversion function in question, then in the simplest case the method difference matches the current sequent against a goal of the form:

$$\forall x : t . \rho(x) = \rho(x)$$  \hspace{1cm} (6.9)

There are two enhancements intended to get the closest match between the current sequent and the goal we are trying to ripple towards:

1. The method can also difference match against a variant of (6.9) in which both sides have been expanded using a wave rule for $\rho$. For example, expanding (6.27) using wave rule (6.17) gives the standard form (6.28). This results in more of the goal being in the skeleton after difference matching.

During rippling, unblocking steps (3.5.2) may apply rewrite rules (not just wave rules) to parts of the goal which are not in the skeleton. By ensuring that as much of the goal as possible is part of the skeleton, the applicability of unblocking is reduced with a consequent reduction in proof search. This may also lead to the speculation of more specific wave rules.

2. Using one-step ripples, the method can partially instantiate existentially quantified variables in the goal.

All possible applications of these two enhancements are made, and the resulting difference matches are ranked according to a measure $\mu$ on annotated terms, where $|T|$ is defined to be the number of nodes in the term tree of $T$:

$$\mu(AnnTerm) = \frac{|\text{skeleton}(AnnTerm)|}{|\text{erasure}(AnnTerm)|}$$

The match with the highest measure is selected, and all the others are discarded. To reduce search, there is no backtracking to alternative matches.
6.4.4 The wave rules

To ripple the annotated goal we must annotate our rewrite rules as wave rules.

In the current example, only the first of the two rewrites can be annotated as a wave rule (6.10).

\[
d\text{list\_list}(d1) \leftrightarrow d\text{list\_list}(d2) \Rightarrow d\text{list\_list}(d1 \leftrightarrow d\text{list\_list}(d2)) \quad (6.10)
\]

The wave rules are directed inwards because the conversion function which we wish to move out through the term structure is part of the skeleton. The movement of the outer term structure inwards effectively moves the conversion function outwards.

Equation (6.4) cannot be annotated as a wave rule because the conversion function \(d\text{list\_list}\) is not present on the right hand side. The choice of which rewrite rule to apply (§6.2.2) is eliminated, because only one of these equations is a wave rule. In fact, wave rules are exactly the kind of rewrites which we wish to apply in this situation.

1. Wave rules are skeleton preserving. Since \(p\) is part of the skeleton, this guarantees that we keep a copy of \(p\) on both sides of the equality, which will hopefully be cancelled later in the proof.

2. Wave rules are measure decreasing. This guarantees that \(p\) is moved out through the expression, thus getting closer to cancellation.

6.4.5 The benefits of rippling

The use of difference matching and rippling during the synthesis proof carries several advantages:

1. It typically reduces search to an almost linear path. In §9.3.1, I compare my strategy with a simple fold/unfold strategy in the example of §2.4.7.
2. In the absence of meta-variables, termination of rewriting is guaranteed.

3. Successful proof yields an extracted program of a desirable form.

4. The type change technique may benefit from extensions to rippling, e.g. transverse rippling, wave rule speculation/calculation [Ireland & Bundy 95].

5. Rippling may benefit from extensions needed to automate type change proofs, for example the ideas discussed in §10.5.2.

6.5 Difference matching in function composition

Suppose we are making a type change $x : T \xrightarrow{\rho} x' : T'$ within a function composition, with conversion function $\rho : T' \rightarrow T$, for example from type $\text{nat}$ to type $\text{bin}$, with conversion function $\text{nat} : \text{bin} \rightarrow \text{nat}$.

Consider the synthesis goal:

$$\vdash \forall x : T . \exists z : T . z = f(g(h(x)))$$

After the type change, the synthesis goal becomes:

$$\vdash \forall x' : T' . \exists z' : T' . \rho(z') = f(g(h(\rho(x'))))$$

Difference match with the standard form $\forall x : t . \rho(x) = \rho(x)$.

Here is one possible annotation:

$$\rho(z') = f(g(h(\rho(x'))))$$

The unifier is $z' = x'$. Wave fronts are oriented inwards because the functor, $\rho$, we wish to move out is not a wave functor but is part of the skeleton.
The wave rules we need to deal with this annotation of the goal are oriented inwards:

\[
\begin{align*}
  h(p(x'))^l & \Rightarrow p(h'(x')^l) \quad (6.12) \\
  g(p(x'))^l & \Rightarrow p(g'(x')^l) \quad (6.13) \\
  f(p(x'))^l & \Rightarrow p(f'(x')^l) \quad (6.14)
\end{align*}
\]

Apply these in turn to (6.11):

\[
\begin{align*}
  \rho(z') &= f\left( g\left( h(p(x'))^l \right) \right) \\
           &= f\left( g\left( h'(x')^l \right) \right) \\
           &= f\left( g\left( h'(x')^l \right) \right) \\
           &= f\left( g\left( h'(x')^l \right) \right) \\
           &= f\left( g\left( h'(x')^l \right) \right)
\end{align*}
\]

Now we can cancel \( \rho \) and get the witness \( z' = f'(g'(h'(x'))) \) as desired.

### 6.6 Induction vs. standard form

I am now using rippling in two different ways. The first way is after an induction, to guide the rewriting towards fertilisation. The second way is after application of the standard form method to guide rewriting towards cancellation of conversion functions. We cannot use one set of annotations to guide both kinds of rippling, since they have different aims and are frequently at odds with one another.

We separate the two kinds of rippling by restricting the standard form method so that it only applies to unannotated terms, i.e. either before application of induction, or after fertilisation. This neatly divides the proof into...
two phases: an inductive proof phase followed by difference matching and further rippling, terminating with cancellation.

These two phases of rippling cannot be carried out simultaneously because while wave rule application will ripple closer to one target (the induction hypothesis or the cancellation rule), it will disturb the skeleton of the other. Therefore, the technology of *coloured rippling* [Yoshida et al 94], which allows a goal to be annotated in several different ways simultaneously, cannot be used.

### 6.7 A summary of the proof strategy

1. Possibly apply induction, ripple and weak fertilise, then

2. Difference match with the standard form $\forall x : t. \rho(x) = \rho(x)$ or a variant as described in §6.4.3.

3. Ripple.

4. When a match with the standard form is achieved, apply a cancellation rule $\rho(x) = \rho(y) \leftrightarrow x = y$.

### 6.8 A more complicated example: binary addition

The difference matching proof strategy is most useful in difficult synthesis proofs. In this section we apply it to a hard example, the synthesis of binary addition.

#### 6.8.1 Theorem and definitions

The original synthesis theorem for addition on natural numbers is:

$$\forall x, y : \text{nat}. \exists z : \text{nat}. z = x + y$$
This allows us to synthesise an addition function from the recursion equations (6.20), (6.21) below.

After a type change from nat to bin, we get some justification goals, and a new synthesis goal:

\[ \forall x, y : \text{bin} \exists z : \text{bin}. \text{nat}(z) = \text{nat}(x) + \text{nat}(y) \]  \hspace{1cm} (6.15)

The type bin of binary numbers is defined as bool list. Binary numbers are represented in big-endian form, i.e. with the least significant digit at the head of the list.

In addition to the definition of + we have the following wave rules for nat, which converts a binary number to the corresponding natural number, and a function val which maps binary digits (false or true) to their equivalents in nat (0 or s(0)):

\[
\begin{align*}
\text{nat}(\text{nil}) &= 0 & \text{(6.16)} \\
\text{nat}(\text{d :: x}) &= \text{val}(\text{d}) + (\text{nat}(x) + \text{nat}(x))' & \text{(6.17)} \\
\text{nat}(\text{false :: x}) &= \text{nat}(x) + \text{nat}(x)' & \text{(6.18)} \\
\forall b : \text{bool} b = \text{false} \text{ in bool} & \forall b : \text{bool} b = \text{true} \text{ in bool} & \text{(6.19)} \\
\text{val}(\text{false}) &= 0 \\
\text{val}(\text{true}) &= s(0) \\
x + 0 &= 0 & \text{(6.20)} \\
\text{s(x) + y} &\Rightarrow \text{s(x + y)} & \text{(6.21)} \\
\text{s(x + y)} &\Rightarrow \text{s(x) + y} & \text{(6.22)} \\
\text{s(x) = s(y)} &\Rightarrow x = y & \text{(6.23)} \\
0 + x &= x \\
\text{s(s(x + y))} &\Rightarrow \text{s(x) + s(y)} & \text{(6.24)} \\
\text{a + b + b} + \text{c + d + d} &\Rightarrow \text{a + c + b + d + b + d} & \text{(6.25)} \\
\text{s(nat(x))} &\Rightarrow \text{nat(inc(x))} & \text{(6.26)} \\
\end{align*}
\]
\[
\text{nat}(w) = \text{nat}(w) \tag{6.27}
\]
\[
\text{val}(d) + (\text{nat}(w) + \text{nat}(w)) = \text{val}(d) + (\text{nat}(w) + \text{nat}(w)) \tag{6.28}
\]
\[
\text{nat}(w) + \text{nat}(w) = \text{nat}(w) + \text{nat}(w) \tag{6.29}
\]

Ideally, wave rules (6.24) and (6.25) would be generated automatically in a similar way to the generation of propositional wave rules [Kraan 94, pp.97–100]. This would, however, introduce additional search problems.

Wave rule (6.26) can be extracted from a definition of the increment function. It can also be used as a specification from which the increment function can be synthesised using difference matching to guide the proof.

Wave rule (6.18) is necessary as an alternative target for difference matching. The need for this extra wave rule can be eliminated by the use of difference unification, as suggested in §10.5.3.

Equations (6.27, 6.28, 6.29) are automatically generated when needed by the standard form method. The first is the basic standard form, and the other two are standard forms obtained by rewriting (6.27) using wave rules (6.17) and (6.18) respectively.

### 6.8.2 The synthesis proof

This section outlines the synthesis proof which is constructed automatically by ClAM. The proof plan is reproduced in appendix D.5. The new methods are described in full in appendix E.

The specification is (6.15). After performing \(\text{hd} :: \text{tl}\) induction on \(x\) and \(y\), we get:

\[
\begin{align*}
d_x &: \text{bool}, x : \text{bin} \\
d_y &: \text{bool}, y : \text{bin} \\
\exists z : \text{bin}. \text{nat}(z) &= \text{nat}(x) + \text{nat}(y) \\
\vdash \exists z' : \text{bin}. \text{nat}(z') &= \text{nat}(d_x :: x) + \text{nat}(d_y :: y)
\end{align*}
\]
The proof proceeds as follows.

Ripple using equation (6.17).

\[ \exists z : \text{bin. nat}(z) = \text{nat}(x) + \text{nat}(y) \]
\[ \vdash \exists z' : \text{bin. nat}(z') = \left( \text{val}(d_x) + (\text{nat}(x) + \text{nat}(x)) \right) + \left( \text{val}(d_y) + (\text{nat}(y) + \text{nat}(y)) \right) \]

Rearrange the right hand side of the equality by rippling with wave rule (6.25):

\[ \vdash \text{nat}(z') = \left( \text{val}(d_x) + \text{val}(d_y) + (\text{nat}(x) + \text{nat}(y) + \text{nat}(x) + \text{nat}(y)) \right) \] (6.30)

Weak fertilise (which removes the annotations):

\[ \vdash \text{nat}(z') = \text{val}(d_x) + \text{val}(d_y) + \text{nat}(z) + \text{nat}(z) \] (6.31)

Equation (6.19) specifies that bool is the disjoint union of a finite number of ground terms. Since there also exists an equation which can be used to evaluate \( \text{val}(d_x) \), a case split on \( d_x \) is performed by the \text{finitetypeelim/2} method.

- \( d_x = \text{false} \)

  This case is easily solved by the \text{base_case/1} method, giving \( z' = d_y :: z \).

- \( d_x = \text{true} \)

  The goal is simplified by \text{base_case/1}, and then difference matched by \text{standard_form/2} with variants of (6.27), as described in §6.4.3. The best match is achieved with (6.28), which gives the following annotated goal:
\[ \forall \exists v_11 : \text{bool}. \exists v_12 : \text{bin}. \, \text{val}(v_11) + (\text{nat}(v_12) + \text{nat}(v_12)) = \]
\[ s(\text{val}(d_y)) + (\text{nat}(z) + \text{nat}(z)) \]

This is rippled in with (6.22) to give:

\[ \forall \exists v_11 : \text{bool}. \exists v_12 : \text{bin}. \, \text{val}(v_11) + (\text{nat}(v_12) + \text{nat}(v_12)) = \]
\[ s(\text{val}(d_y)) + (\text{nat}(z) + \text{nat}(z)) \]

After rippling in, a further case split is performed, this time on \( d_y \).

- \( d_y = \text{false} \)

The \texttt{guess\_existential/3} method first tries \( v_11 = \text{false} \). Since application of the method may create a non-theorem, the planner only searches up to a predetermined fixed depth (which has been set to 7). In this case, no plan can be found. Next \( v_11 = \text{true} \) is tried. The goal is now:

\[ \forall \exists v_16 : \text{bool\ list}. \, (\text{val}(\text{true}) + (\text{nat}(v_16) + \text{nat}(v_16))) = \]
\[ (s(\text{val}(\text{false})) + (\text{nat}(z) + \text{nat}(z))) \text{ in } \text{pnat} \]

Application of the \texttt{base\_case/1} method evaluates the occurrences of \text{val}, partially evaluates + and applies (6.23) to give:

\[ \exists v_16 : \text{bool\ list}. \, (\text{nat}(v_16) + \text{nat}(v_16)) = (\text{nat}(z) + \text{nat}(z)) \text{ in } \text{pnat} \]

This is proved by \texttt{existential\_subterm/2}, giving a witness of \( z' = \text{true} :: z \) for this branch of the proof.
- dy = true

The guess_existential/3 method first tries v11 = false.

The goal is:

\[ \vdash \exists v16: \text{bool list}. (\text{val}(\text{false}) + (\text{nat}(v16) + \text{nat}(v16))) = (s(\text{val}(\text{true})) + (\text{nat}(z) + \text{nat}(z))) \text{ in pnat} \]

Again, base_case/1 carries out partial evaluation to give:

\[ \exists v16: \text{bool list}. (\text{nat}(v16) + \text{nat}(v16)) = s((\text{nat}(z) + s(\text{nat}(z)))) \text{ in pnat} \]

This is difference matched by standard_form/2 with variants of (6.27). The best match is achieved with (6.29), which gives the following annotated goal:

\[ \vdash \exists v16: \text{bool list}. (\text{nat}(v16) + \text{nat}(v16)) = s((\text{nat}(z) + s(\text{nat}(z)))) \text{ in pnat} \]

The step_case/2 method ripples in with (6.24) and (6.26), giving:

\[ \vdash \exists v16: \text{bin. nat}(v16) + \text{nat}(v16) = (\text{nat}(\text{inc}(z)) + \text{nat}(\text{inc}(z))) \]

The existential_subterm/2 method completes this branch of the proof, giving z' = false :: inc(z).

Search occurs during construction of the proof plan in the (eventually failing) branch of the proof plan in which an incorrect value has been guessed for the boolean digit. Other parts of the proof plan are constructed without search.
The synthesis presented above gives the following wave rules defining binary addition (written \( +_2 \)). There is one wave rule for each of the possible combinations of values of \( d_x \) and \( d_y \), but these can be simplified by defining a function \( \max \) on digits in the obvious way.

\[
\begin{align*}
\neg (d_x = d_y = \text{true}) & \rightarrow [d_x :: x] +_2 [d_y :: y] \Rightarrow \max(d_x, d_y) :: x +_2 y \quad (6.33) \\
(d_x = d_y = \text{true}) & \rightarrow [d_x :: x] +_2 [d_y :: y] \Rightarrow \text{false} :: \text{inc}(x +_2 y) \quad (6.34) \\
\text{inc}(\text{false} :: x) & = \text{true} :: x \quad (6.35) \\
\text{inc}(\text{true} :: x) & \Rightarrow \text{false} :: \text{inc}(x) \quad (6.36)
\end{align*}
\]

This is not the most efficient binary addition function. It is, however, much more efficient than unary addition. Synthesis of the standard definition of binary addition requires the introduction of a carry bit. The proof is more difficult and has not yet been automated.

### 6.9 Conclusion

In this chapter I have described a proof strategy which exploits the common form of the synthesis proofs which arise after a type change. The strategy uses difference matching and rippling to guide the rewriting process towards a goal from which a cancellation rule can be used to determine an explicit existential witness.

The use of rippling not only greatly reduces search in the synthesis proof, but also produces extracted programs which are free of conversion functions, and hence quite efficient. The fact that the proof strategy which I have identified can be described and implemented so well in terms of rippling demonstrates the flexibility of rippling as a proof method.

The chapter was illustrated with a difficult example: the synthesis of an addition function on binary numbers.
Chapter 7
Type Abstraction

7.1 Introduction

In this chapter I extend the mechanisms already developed for implementation type changes to allow the abstraction of types. Abstraction is an information losing transformation, and it is not always possible to synthesise a function from the synthesis goals which arise after an abstraction type change.

Whereas implementation type changes are characterised by conversion functions, abstraction type changes are characterised by conversion relations.

I show that the proof strategy given in chapter 5 for choosing implementation type changes using motivation and propagation rules is also applicable to abstraction type changes. Its effectiveness is illustrated by abstracting lists to pairs of their least and greatest elements.

The difference matching and rippling proof strategy of chapter 6 cannot readily be extended to abstraction type changes because the difference matching and rippling it uses are functional, whereas the specifications resulting from abstraction type changes are relational.

I define a proof strategy which proves the synthesis goals which result after an abstraction type change in simple cases. More sophisticated strategies are the subject of further work (§10.6.3).
7.2 What is abstraction?

In the enrich-forget-identify model of abstract data type development (§2.4.3), abstraction corresponds to a forgetting step, removing structure (functions, axioms etc.) from the ADT. This is abstraction from the point of view of ADT manipulation.

More generally, a type $t_{\text{obs}}$ is an abstraction of a type $t_{\text{conc}}$ if for every element of the concrete type there is an element of the abstract type, as in figure 7-1. We may wish to strengthen this requirement to insist that for each element of the concrete type there is exactly one element of the abstract type.

![Figure 7-1: An abstraction transformation is many-to-one.](image)

Implementation can improve the efficiency of a program by incorporating extra information into the input data type which helps perform certain calculations. When information is deleted from the type of the output of a function (i.e. the output type is abstracted) the function can be made more efficient because it is less constrained. When information is deleted from the type of its input, a function can be made more efficient because irrelevant information need no
longer be processed (e.g. abstracting a type of lists to a tuple consisting only of some pertinent properties of the list elements, such as maximum and minimum).

7.3 Examples of type abstraction

A simple example is an abstraction in which each list is represented by a pair \((\text{min}, \text{max})\) of its least and greatest elements. Clearly some functions on lists have no equivalent on \((\text{min}, \text{max})\) (e.g. there is no equivalent of length). This abstraction can be seen as a composite type change in which the type is first enriched with the functions \text{min} and \text{max}, and then the elements of the data type are forgotten. This forgetting process forces us to use the new information.

Other examples are:

1. \text{nat} \rightarrow \text{bool}. Represent each natural number by its parity — odd or even. This allows significant improvements in efficiency, as long as only a number’s parity is required. For example, addition is a simple non-recursive function.

2. \text{list} \rightarrow \text{set}. We can move from lists to sets by forgetting the \text{head} and \text{tail} operations, enriching by a \text{member} relation, and defining equality of sets in terms of the member relation rather than \text{head/tail}.

   This abstraction is very useful as there is a variety of interesting and efficient implementations for sets, e.g. hash tables, ordered lists or balanced trees.

3. The inverse of any implementation.
7.4 Generalising Conversion Functions to Conversion Relations

7.4.1 Introduction

I showed in chapter 4 that conversion functions can be used to characterise implementation type changes.

In this section I show that conversion functions are inadequate for abstraction type changes, and that we must generalise conversion functions to conversion relations. I then show how the conversion functions used in implementation type changes can be seen as a special case of conversion relations.

7.4.2 Conversion functions

For information-preserving transformations we specify the analogue \( f': t' \rightarrow t'_0 \) on new data types of a function \( f: t \rightarrow t_0 \) on the original data types using conversion functions (otherwise known as representation or retrieve functions) \( \rho, \rho_0 \) which map elements of the new input and output types to the old input and output types respectively:

\[
\forall x': t' \exists z': t'_0 . \rho(z') = f(\rho(x'))
\]  

(7.1)

Specifying the relationship between the two types in this way is possible because every member of the new type has a unique corresponding element in the old type.

When performing a type abstraction, however, elements of the new type generally have multiple equivalents in the old type. It is possible to maintain the one to one correspondence by choosing a unique canonical equivalent for each element of the abstract type. For example, the list to set transformation
be characterised

can

by

a

conversion function which

list in which every element of the set appears
this

exactly

If set_olist
then

:

set

(7.1) specifies

—►

an

list maps

a

ordered

The problem with

we can

canonical list

equivalent of <> (append)

on

set_olist(s)

=

an

synthesise

as we

function into canonical form.

each set to

Vs,t:set3u:set.set_olist(u)
This

new

each set to

once.

approach is that it greatly restricts the functions
compelled to put the results of the

are

maps

as

described above,

sets:

<>

set_olist(t)

(7.2)

specification cannot be satisfied because <> does not respect the

{2,3}, t

or¬

{1,2}, the right

dering of its argument lists. For example, with

s =

hand side of

[2,3,1,2], which is not ordered,

so

there is

7.4.3

If

relation instead of

types then

existence of
of

a

=

satisfying the left hand side.

Conversion relations

we use a

the two

(7.2) evaluates to [2,3] <> [1,2]

no u

=

we can

a

function to

specify the correspondence between

take advantage of the flexibility provided by the

multiple correspondents for abstract elements. For

type t to

a

type t' define

p(x,y)

a

an

abstraction

relation p(x: t, y : t') such that:

<-> x

is

a

concretisation of

y

Every element of the concrete type must have at least

one

corresponding

abstract element:

Vx:t.3y

I show in

:

t'.p(x,y)

§7.6 that conversion relations

plementations and abstractions.

For this

are

able to characterise both im¬

reason,

the implementation of the

typechange method uses conversion relations to characterise both abstractions
and

implementations.

151


7.5 The subgoals of a type change

After changing the types of its arguments, a specification:

\[ \forall x_1 : t_1, \ldots, x_n : t_n . \exists z : t_0 . \text{spec}(x_1, \ldots, x_n, z) \]

becomes:

\[ \forall x'_1 : t'_1, \ldots, x'_n : t'_n . \exists z : t_0 . \left( \forall x_1 : t_1, \ldots, x_n : t_n . \rho_1(x_1, x'_1) \land \ldots \land \rho_n(x_n, x'_n) \Rightarrow \right) \text{spec}(x_1, \ldots, x_n, z) \]  

(7.3)

The following side-condition is sufficient to ensure that the transformation is correctness-preserving:

\[ \forall x_i : t_i . \exists x'_i : t'_i . \rho_i(x_i, x'_i) \]  

(7.4)

The order of the quantifiers in (7.3) is crucial. Since the goal is of the form \( \forall v_1, \ldots, v_n \exists z . \text{specification} \), its proof will yield a function on the new type. Placement of the second set of universal quantifiers and the conversion relations (marked 'independence conditions') after the existential imposes a constraint that the value of the synthesised function should not depend on how members of the old type are converted to the new type. Given an input of the old type, (7.4) shows how to convert it to the new type, and (7.3) allows us to compute an output, ensuring that the information lost in the initial conversion has not affected the value of the function.

The inputs can be abstracted because the \( \rho_i \) conversion relations ensure that the information which is lost in the transformation does not affect the value of the synthesised function. If the output is abstracted, a condition must be imposed on the program context in which the transformed function is used. Currently, I
prefer to keep the transformation local and so disallow abstraction of the output, but in §10.6.1 I discuss the more general case. One particular situation in which an output transformation is allowed is the transformation of the argument \( g \) of a function \( f(g) \). Since the output transformation of the argument is an input transformation of the function \( f \), an output transformation on \( g \) is allowable. This is discussed further in §7.7.

As a particular case of (7.4), the analogue \( f' : t'_1 \to t'_2 \to \cdots \to t'_n \to t_0 \) of a function \( f : t_1 \to t_2 \to \cdots \to t_n \to t_0 \) can be specified by:

\[
\forall x'_1 : t'_1, \ldots, x'_n : t'_n \ . \exists z : t_0 . \forall x_1 : t_1, \ldots, x_n : t_n \ . \rho_1(x_1, x'_1) \land \cdots \land \rho_n(x_n, x'_n) \to \ z = f(x_1, \ldots, x_n)
\]  

(7.5)

While the input types may be abstracted, an implementation type change is also allowable on the output type. In order to simplify matters, I do not exploit this at the moment.

The proof methods which I have developed for proving the specifications which arise after an implementation type change cannot generally be applied to abstractions because they do not take account of the stronger proof obligations.

### 7.6 Conversion functions are a special case

Equation (7.5) can be reduced to the case for an information-preserving transformation by noting that a function \( k : t \to t' \) is equivalent to a relation \( r : t \times t' \) defined by:

\[
r(x, y) \leftrightarrow y = k(x)
\]

If each of the \( \rho_i(x_i : t_i, x'_i : t'_i) \) above describes an information-preserving transformation with conversion function \( \sigma_i : t'_i \to t_i \), then:
\[ \rho_1(x_i, x'_i) \leftrightarrow x_i = \sigma_1(x'_i) \]  

(7.6)

Given that:

\[ \forall x'_i : t'_i \exists z : t_0 . \forall x_1 : t_1 . x_1 = \sigma_1(x'_i) \rightarrow z = f(x_1) \]

is equivalent to:

\[ \forall x'_i : t'_i \exists z : t_0 . z = f(\sigma_1(x'_i)) \]

it is clear that the relational form above is equivalent to the form using conversion functions:

\[ \forall x'_1 : t'_1 ... \forall x'_n : t'_n \exists z : t_0 . z = f(\sigma_1(x'_1), ..., \sigma_n(x'_n)) \]

When an implementation type change is made on the output type, as mentioned in §7.5, we get exactly the synthesis goal produced by a general implementation type change.

The advantage of the functional form is that we can use functional rippling when proof planning. In the relational form necessary for type abstraction we may have to use relational rippling, although this too has some problems (§7.11.3).

### 7.7 Transformation of compound expressions

We must consider the impact of transformations on arguments nested within a function expression. In an implementation transformation, we can transform an argument in place by wrapping it in a conversion function. In the case of abstraction, we must transform the argument in a separate subgoal.

For example, consider an expression:

\[ \forall x : t_1, y : t_2 \exists z : t_0 . z = g(f(x), h(y)) \text{ in } t_0 \]
If we abstract $f : t_1 \rightarrow t_3$ to $f' : t'_1 \rightarrow t'_3$ then we have to prove the following subgoals:

1. The transformation on the first argument of $g$.
   \[
   \forall x' : t'_1 \forall y : t_2 \exists z : t_0. \forall x : t_3 \rho_3(x, x') \rightarrow z = g(x, h(y)) \text{ in } t_0
   \]

2. The transformation on $f$. The type change on the input type, $t_3$, of $g$ above allows us to make the same type change on the output of $f$ here without losing correctness.
   \[
   \forall x' : t'_1 \exists z' : t'_3. \forall x : t_1 \rho_1(x, x') \rightarrow \rho_3(f(x), z')
   \]

### 7.8 Extending the proof strategy to abstraction type changes

#### 7.8.1 Introduction

Abstraction type changes are motivated by motiv rules in the same way as implementation type changes (§5.4). I show in §7.8.2 that propagation rules can be used when performing abstraction type changes. Abstraction type changes are chosen in the same way that implementation type changes are. The typechange/2 method handles both.

The proof strategy developed in chapter 6 cannot be used for abstraction type changes. An alternative, but much weaker, proof strategy is defined in §7.9.

#### 7.8.2 Propagation and choice of type changes

The propagation rules presented in chapter 5 were developed in order to minimise overheads caused by the introduction of conversion functions into the synthesised
program. The same considerations apply in the case of abstraction type changes. In this section I demonstrate that the propagation rules I have defined are also applicable to type abstraction.

The propagation rules generally allow output type changes. Although currently my methods do not allow abstraction type changes on outputs, I retain the possibility in this presentation of the rules because abstraction of outputs can be allowed if the proof strategy is extended as described in §10.6.1.

The Feval propagation rule

The following rule was presented in §5.6:

\[
\begin{align*}
\text{f}(x) & := h(x) & (7.8) \\
\text{h}(x:t):t_0 & \leadsto h'(x':t'):t'_0 & (7.9) \\
\text{f}(x:t):t_0 & \leadsto f'(x':t'):t'_0 & \text{Feval} & (7.10)
\end{align*}
\]

When the type change is an abstraction, (7.9) produces specification (7.11):

\[
\begin{align*}
\forall x':t'. \exists z':t'_0. \forall x:t. \rho(x, x') & \rightarrow \rho_0(h(x), z') & (7.11) \\
\forall x':t'. \exists z':t'_0. \forall x:t. \rho(x, x') & \rightarrow \rho_0(f(x), z') & (7.12)
\end{align*}
\]

Applying (7.8) above as a rewrite rule gives (7.12), which is exactly the specification produced from (7.10). Therefore, this rule is also valid for an abstraction type change. An output transformation has been allowed, which makes the correctness of the transformation contingent on the context in which the functions \(f\) and \(h\) occur.

The Fcomp propagation rule

The following rule was presented in §5.5:
\[ f(x) := g(h(x)) \]
\[ h(x:tx):t_h \leadsto h'(x':t'_h):t'_h \]
\[ g(x:t_h):t_g \leadsto g'(x':t'_h):t'_g \]
\[ f(x:tx):t_g \leadsto f'(x':t'_x):t'_g \]

As above, applying the rewrite rule in the premises to the type changes in the premises gives a specification which implies that in the conclusions. An additional benefit is that an output transformation on \( h \) is guaranteed to be correct because it matches the input transformation on \( g \).

**The ParmInd propagation rule**

The following rule was presented in §5.7.2:

\[ \text{Step case } f(c(h, x)^l) \Rightarrow g(h, f(x)) \]  \( \text{ (7.13) } \)
\[ g(h: t_1, x: t_0): t_0 \leadsto g'(h': t'_1, x': t'_0): t'_0 \]  \( \text{ (7.14) } \)
\[ f(x: t(t_1)): t_0 \leadsto f'(x': t(t'_1)): t'_0 \]  \( \text{ParmInd } \)  \( \text{ (7.15) } \)

In order to simplify the proofs, I replace the existential quantifier with a skolem function \( f' \) or \( g' \) as appropriate. The proofs below are verification proofs that \( f' \) or \( g' \) is the analogue of \( f \) or \( g \). The corresponding synthesis conjectures can be proved only if the verification conjectures can.

The specification produced from (7.14) is:

\[ \forall h': t'_1 \forall x': t'_0 \forall h: t_1 \forall x: t_0 . \rho_1(h, h') \land \rho_0(x, x') \rightarrow \rho_0(g(h, x), g'(h', x')) \]  \( \text{ (7.16) } \)

The specification produced from (7.15) is:

\[ \forall y': t(t'_1) \forall y: t(t_1) . \bar{\rho}(y, y') \rightarrow \rho_0(f(y), f'(y')) \]  \( \text{ (7.17) } \)
After performing simultaneous induction on \( x : t(t_1) \) and \( x' : t(t'_1) \) in (7.17), we get the step case:

\[
\bar{p}(y, y') \rightarrow \rho_0(f(y), f'(y'))
\]

(7.18)

\[
\vdash \bar{p}(c(k, y), c(k', y')) \rightarrow \rho_0(f(c(k, y)), f'(c(k', y')))
\]

In order to proceed, we require a structural definition of \( \bar{p} \) analogous to that provided by equations (5.5, 5.6) of chapter 5. The ParmInd propagation rule is intended to apply to type changes \( t(t_1) \rightarrow t(t'_1) \) in which only the type \( t_1 \) is changed. One could also consider the possibility of type changes \( t(t_1) \rightarrow t(t'_1) \) in which the type change \( t \rightarrow t' \) is an isomorphism other than the identity, for example one which represents a list by its reversal. I do not consider such cases, so \( \bar{p} \) can be defined by (7.19):

\[
\bar{p}(x, x') \leftrightarrow \exists h : t_1, h' : t'_1, y :
\]

\[
x = c(h, y) \land x' = c(h', y') \land \rho(t_1)
\]

Applying (7.19) to the step case (7.18) and introducing the implication, thus making the antecedent into a hypothesis, gives:

\[
\bar{p}(y, y') \rightarrow \rho_0(f(y), f'(y'))
\]

\[
\rho_1(k, k')
\]

\[
\bar{p}(y, y')
\]

\[
\vdash \rho_0(f(c(k, y)), f'(c(k', y')))
\]

(7.20)

As long as \( f' \) has the same recursive structure as \( f \), we have \( f'(c(k', y')) \Rightarrow \]

\[
g'(k', f'(y'))
\]

, and we can ripple the conclusion to:

\[
\vdash \rho_0(g(k, f(y)), g'(k', f'(y')))
\]
This is proved by applying (7.14) as a lemma, with \( h = k, h' = k', x = y, x' = y' \).

Therefore we see that the ParmInd propagation rule is also applicable to abstraction type changes as long as the analogue \( f' \) has the same recursive structure as \( f \).

The Ind propagation rule

In §5.7.3, the Ind propagation rule is shown to be a degenerate case of the ParmInd rule. Its applicability to abstraction type changes follows from the applicability of the ParmInd rule established above.

7.8.3 Planning the synthesis proofs

The synthesis goals which arise after an implementation type change can be proved by the difference matching proof strategy which I described in chapter 6, which performs two phases of rippling:

1. The first phase of rippling is in the step case of an induction. The wave terms are initially nested inside conversion functions and are rippled progressively outwards.

2. The second phase of rippling is after difference matching, which annotates the goal so that the conversion functions are in the skeleton, and subsequent rippling moved wave fronts in past the conversion functions.

The synthesis goals which arise after an abstraction type change contain conversion relations instead of conversion functions. This moves the proof away from the domain of normal rippling and into the domain of relational rippling (§3.5.6). Relational rippling is not yet integrated into CLAM.¹ In addition, no

¹There is now an experimental implementation.
relational counterpart of difference matching has been developed, so the proof strategy developed for synthesis after an implementation type change cannot be used for synthesis after an abstraction type change. The use of relational rippling is a topic for further research (§10.5.4).

7.9 A strategy for abstraction synthesis proofs

The proof strategy for abstraction consists of the application of a succession of methods to the subgoals which are produced by the typechange method.

When a rewrite rule exists specifying a simple way of replacing functions on the old type with known functions on the abstract type, the rewrite_conversion submethod is enough to prove the main subgoal of the type change. When such a rewrite rule does not exist the other components of the simplify_typechange_goals iterator can still make some progress.

7.9.1 The typechange_strat/2 method

The typechange_strat/2 method applies the typechange/2 submethod and tries to simplify the resulting subgoals using the simplify_typechange_goals/1 submethod.

7.9.2 The typechange/2 submethod

This method insists that the output transformation be the identity.

The input must be of the form:

\[ \forall x : t. \exists z : t_0. z = f(x) \]

The main subgoal is:
\[ \forall x': \text{t}. \exists z_0. \forall x: \text{t}. \rho(x, x') \rightarrow z = f(x) \]  

(7.21)

For each argument, say \( g(x) \) of \( f \), if the transformation on \( g \) is the identity, then it appears unchanged in the expression above. If it is not the identity, then a subgoal similar to the one above must appear in order to transform it. These subgoals are of one of two forms:

1. If the argument is just a variable, then the subgoal is:

\[ \forall x: t. \exists x': t'. \rho(x, x') \]

2. If the argument is a function \( g \), then the subgoal is like (7.21) but with the possibility of an output transformation, as in (7.7).

7.9.3 The \texttt{simplify\_typechange\_goals/1} submethod

The \texttt{simplify\_typechange\_goals/1} submethod is described in detail in appendix E. It exhaustively performs the following:

1. Replace functions \( f(\rho(x : t)) \) with their equivalents \( f'(x' : t') \) (or the corresponding step involving conversion relations in the case of an abstraction).

2. Apply an independence result to justify an abstraction.

3. Rewrite the conversion relations using conversion functions.

4. Introduce a meta-variable for the most nested existential quantifier.
7.10 An example: lists to \( \langle \text{min}, \text{max} \rangle \)

7.10.1 Introduction

The example below illustrates how useful type abstraction can be. The intended application is a Prolog interpreter or theorem prover which needs to generate new variable names. The initial specification of the problem is to find a variable name (or an integer) which has not already been used. A naive implementation of this is to maintain a list of currently used variable names and generate new ones by generating names in turn until one is found which is not in the list, i.e. the Prolog:

\[
\text{genvar}(\text{Newvar}), \neg \text{member}(\text{Newvar}, \text{Varlist}).
\]

In fact this is how Oyster generates its variable names using hfree/2. A more sophisticated strategy is to record the highest and lowest variable names used, and when called upon to generate a new one, pick one more than the highest. This is how Prolog interpreters usually generate new anonymous variable names.

7.10.2 The ADT

The full ADT for the type \( \text{mminmax} \) of pairs \( \langle \text{min}, \text{max} \rangle \) is listed in appendix D.3. Figure 7-2 gives the signature of the ADT, and the two equations which are used in the proof.

7.10.3 Complications

A complication of this representation is that the \( \text{max} \) and \( \text{min} \) functions are not total on lists of integers; \text{nil} is not in the function domain.

Although it is possible to circumvent this problem by considering an abstraction in which each list is replaced by its maximal element, and \text{nil} replaced by 0, the \( \text{mminmax} \) data type is more general and so has more potential for reuse.
The signature of mminmax: Type

<table>
<thead>
<tr>
<th>Function</th>
<th>Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mmin</td>
<td>mminmax → (pnat\unary)</td>
<td>First element of pair</td>
</tr>
<tr>
<td>mmax</td>
<td>mminmax → (pnat\unary)</td>
<td>Second element of pair</td>
</tr>
<tr>
<td>mnil</td>
<td>mminmax</td>
<td>Equivalent to empty list</td>
</tr>
<tr>
<td>madd</td>
<td>pnat → mminmax → mminmax</td>
<td>Constructor, like cons</td>
</tr>
<tr>
<td>mminmaxtolist</td>
<td>mminmax → pnat list</td>
<td>Returns an ordered doubleton list.</td>
</tr>
<tr>
<td>listtomminmax</td>
<td>pnat list → mminmax</td>
<td>Makes a pair of min/max list elements</td>
</tr>
<tr>
<td>mminmaxlist</td>
<td>pnat list → mminmax → bool</td>
<td>Relational form of mminmaxtolist</td>
</tr>
<tr>
<td>mminmaxlistlist</td>
<td>(pnat list) list → mminmax list → bool</td>
<td>Extension of mminmaxlist to lists of lists</td>
</tr>
<tr>
<td>listtomminmaxlist</td>
<td>(pnat list) list → mminmax list</td>
<td>Functional version of mminmaxlistlist</td>
</tr>
</tbody>
</table>

\begin{align*}
mminmax.2 & \forall l : pnat list. mminmaxlist(l, listtomminmax(l)) \quad (7.22) \\
mminmax._\text{maxof} & \forall l : pnat list. \forall m : mminmax. mminmaxlist(l, m) \quad (7.23) \\
\text{maxof}(l) = \text{mmax}(m) \in (pnat\unary) \\
\end{align*}

Figure 7–2: The signature of the mminmax ADT and some of its equations.
The Oyster type theory does not have a special mechanism for dealing with partial functions or exceptions. If we wish to allow a function \( f : t \rightarrow t_0 \) to return an error, we replace its output type \( t_0 \) with a disjoint union type \( t_0 \cup \text{unary} \). If the function is applied successfully, it returns a left injection containing the result; otherwise it returns a right injection to indicate an error. We also have to modify functions which may take the output of \( f \) as an input so that they can handle errors which are passed to them. In this case, we define functions to return the least and the greatest elements of a list, or an error when applied to the empty list:

\[
\begin{align*}
\text{minof} : & \text{pnat list} \rightarrow (\text{pnat} \cup \text{unary}) \\
\text{maxof} : & \text{pnat list} \rightarrow (\text{pnat} \cup \text{unary})
\end{align*}
\]

The function \( \text{leftof} \) is defined to simplify expressions when we know that \( \text{minof} \) or \( \text{maxof} \) have been applied to a nonempty list. It is defined by:

\[
\begin{align*}
\text{leftof}(\text{inl}(n)) & = n \\
\text{leftof}(\text{inr}(\text{unit})) & = 0
\end{align*}
\]

### 7.10.4 Required lemmas

We require the following lemma, which states that for any list \( l \) of natural numbers, \( \text{max}(l) + 1 \) is not a member of that list:

\[
\text{maxmemsuc2}:
\]

\[
\forall l : \text{pnat list} \forall x : \text{pnat} . \text{member}(s(\text{leftof}(\text{maxof}(x :: l))), x :: l) \rightarrow \text{void}
\]

Discovery of this lemma is a eureka step and I would expect some user interaction to be required to produce it.
The proof plan

The theorem to be proved is:

$$\forall l:\text{pnat list} \exists x:\text{pnat} . (\text{member}(x, l) \rightarrow \text{void})$$  \hspace{1cm} (7.24)

The motivating expressions are:

<table>
<thead>
<tr>
<th>Function</th>
<th>Argument type changes</th>
<th>Output type change</th>
</tr>
</thead>
<tbody>
<tr>
<td>minof(x)</td>
<td>$x : (\text{pnat list}) \rightarrow x' : \text{mminmax}$</td>
<td>identity</td>
</tr>
<tr>
<td>maxof(x)</td>
<td>$x : (\text{pnat list}) \rightarrow x' : \text{mminmax}$</td>
<td>identity</td>
</tr>
</tbody>
</table>

The shape of the proof plan which is produced by CI\textsc{AM} is illustrated in figure 7–3. A more detailed proof plan is given in figures 7–6,7–7.

\[\text{induction}\]
\[\text{base_case} \quad \text{elim_existential}\]
\[\text{existential_split} \quad \text{existential_split}\]
\[\text{elementary} \quad \text{base_case} \quad \text{lemma} \quad \text{typechange_strat}\]
\[\text{elementary} \quad \text{lemma}\]

Figure 7–3: The shape of the mminmax proof.

The relatively late application of the typechange_strategy in the proof is due to the fact that the information on which it is based is only available after the first lemma application. Since the synthesised function is not recursive, this does not cause efficiency problems, but it would prevent the type change being propagated through this proof and up into a larger proof of which this was just
a part. This indicates that a new propagation rule is required which propagates a type change past application of lemmas.

The submethods which are applied by the type change strategy perform a large part of the proof. The structure of the typechange_strat subplan is depicted in figure 7–4. The right hand branch simplifies the subgoal which proves that the transformation is a valid abstraction. The left hand branch simplifies the synthesis goal after the type change in the following way.

First the type change method produces a revised synthesis goal, (7.25). This is rewritten by rewrite_conversion to (7.26), then by implies_independence to (7.27) and finally by embedded_existential to (7.28), replacing the existentially quantified variable with a meta-variable.

\[
\forall m: m\text{minmax}. \exists x: \text{pnat}. \forall v0: \text{pnat} \forall v1: \text{pnat list}. \quad (7.25)
\]

\[
m\text{minmaxlist}(m, v0 :: v1) \rightarrow x = s(\text{leftof}(\text{maxof}(v0 :: v1)))
\]

\[
\forall m: m\text{minmax}. \exists x: \text{pnat}. \forall v0: \text{pnat} \forall v1: \text{pnat list}. \quad (7.26)
\]

\[
m\text{minmaxlist}(m, v0 :: v1) \rightarrow x = s(\text{leftof}(m\text{max}(m)))
\]

\[
\forall m: m\text{minmax}. \exists x: \text{pnat}. x = s(\text{leftof}(m\text{max}(m))) \quad (7.27)
\]

\[
\forall m: m\text{minmax}. x = s(\text{leftof}(m\text{max}(m))) \quad (7.28)
\]

7.10.6 The applicability of the proof strategy

The main subgoal of the type change in §7.10.5 is proved without difficulty because equation (7.23) specifies how to translate \text{maxof} from the old to the new type. In general, the proof strategy which I have implemented for abstraction will succeed when such equations exist, giving abstract analogues of the basic functions in the concrete type, because it allows proof without induction and rippling (functional or otherwise).

When such equations do not exist, a more sophisticated strategy is needed. I discuss several possibilities in §10.6.3.
7.11 Planning more difficult synthesis proofs

7.11.1 Introduction

In this section I consider several possible strategies for proving the synthesis goals which arise after an abstraction type change.

The results are mostly negative, but raise topics for further research (§10.6.3).

7.11.2 Functional rippling with middle-out-reasoning

In this section I consider how the techniques of speculative rippling can be used to perform the synthesis of an analogue on sets of $append$ on lists. This has not yet been implemented.

The specification before the type change is:

$$\forall l, m : \text{list} \exists z : \text{list}. z = l <> m$$

After the type change on inputs from lists to sets we get:
The relation \( \text{listset} : \text{list} \times \text{set} \rightarrow \text{bool} \) is defined by equation (7.32). As in §7.6 we can rewrite it into a functional form, (7.33).

The definitions and wave rules needed for this proof are contained in figure 7–5.

\[
\forall s, t : \text{set} \exists z : \text{list} . \forall l, m : \text{list} . \text{listset}(l, s) \land \text{listset}(m, t) \rightarrow \ z = l \leftrightarrow m
\]

(7.29)

\[
\text{listset}(l, s) \leftrightarrow \text{empty}
\]

(7.34)

\[
\text{listset}(l : \text{list}, s : \text{set}) \leftarrow (\forall x \in \text{member}(x, l) \leftrightarrow x \in s)
\]

(7.32)

\[
\text{listset}(l, s) \leftrightarrow s = \text{listset}(l)
\]

(7.33)

\[
\text{listset}([]) = \text{empty}
\]

(7.30)

\[
\text{add}(h, \text{listset}(t))
\]

(7.35)

\[
\text{nil} \leftrightarrow x = x
\]

(7.31)

\[
\text{h} :: x \leftrightarrow y \Rightarrow \text{h} :: x \leftrightarrow y
\]

(7.36)

**Figure 7–5:** Equations and wave rules for the synthesis of a set analogue of append.

Using (7.33), and eliminating the second existential quantifier, we can rewrite (7.29) as:

\[
\forall s, t : \text{set} . \exists z : \text{list} . \forall l, m : \text{list} . s = \text{listset}(l) \land t = \text{listset}(m) \rightarrow \ z = \text{listset}(l \leftrightarrow m)
\]

(7.36)

Applying ripple analysis (§3.4.6) to (7.29) does not suggest \( s : \text{set} \) as there are no wave rules to carry out subsequent rippling. This results in the failure of
the proof strategy for induction. A solution to the problem is suggested by the work of [Kraan 94]. The approach taken there is to explicitly introduce a higher-order metavariable $F(s, t)$ as the existential witness. Proof planning proceeds as normal, but because $F$ is not yet defined, rippling will become blocked. A wave rule for $F$ will then be speculated to bridge the blockage. Once a successful inductive proof has been constructed, a function must be synthesised from the wave rules which have been discovered for $F$ in order to justify those wave rules and hence the rippling proof. This is suggested in §10.6.3 as a topic for future research.

7.11.3 Relational rippling

Rippling controls rewriting so that differences between the goal and some other formula, for example an induction hypothesis, move outwards through the term structure. In logic programming, instead of nesting functions, complex expressions are built by forming conjunctions of relational expressions, which are implicitly linked by existentially quantified variables. Relational rippling [Bundy & Lombart 95] is an adaptation of rippling to the relational case.

The use of relations to characterise abstraction type changes leads to the expectation that relational rippling will be necessary instead of functional rippling, and will solve the problems identified in §7.11.2.

Unfortunately, the solution is not so easy. The universally quantified variables on which we would like to perform induction ($s$ and $t$ in (7.29)) are linked by the conversion relation not to existentially quantified variables, as required for relational rippling, but to universally quantified variables. This blocks the rippling before it can even start.
Abstraction type changes are characterised by conversion relations.

The proof strategy given in chapter 5 for choosing implementation type changes using motivation and propagation rules is also applicable to abstraction type changes. The typechange/2 method has been extended to type abstraction.

The difference matching and rippling proof strategy of chapter 6 cannot readily be extended to abstraction type changes because the difference matching and rippling it uses are functional, whereas the specifications resulting from abstraction type changes are relational.

I define a proof strategy which proves the synthesis goals which result after an abstraction type change in simple cases. More sophisticated strategies are the subject of further work (§10.6.3).
The lemma instantiates the existential with \( X = \lambda \).

\[ \exists x : \text{pnat}. \ x = s(\text{leftOf} (\text{maxOf} (vO :: vl))) \]

We never fertilise, so this is really a case split.

This method is like the existential method, but has two

allow further planning with the witness.

determine the witness.

Subject.

The lemma instantiates the existential with \( X = \lambda \).

The lemma instantiates the existential with \( X = \lambda \).

\[ \exists x : \text{pnat}. \ \text{exists} (x : \text{pnat}, \ X) \]

\[ \text{apply lemma (max is max succ2).} \]

\[ \text{elim existential} (\_). \]

\[ \exists x : \text{pnat}. \ \text{member} (x :: vO :: vl) \]

\[ \text{base case} (\_). \]

\[ \text{determinacy} (\_). \]

Figure 7-6: The first half of the mminmax proof.
The type change. Most of the work is done by the type change proof strategy, with very little left to subsequent planning.

The type change proof. The second half of the minmax proof.

apply_lemma(mminmax-2)

\text{Figure 7-2: The second half of the minmax proof.}
Chapter 8

Results and Implementation

In this chapter I describe the implementation of PITCHES. Type changes are chosen by the typechange_strat/2 method using the motivation and propagation rules of chapter 5. This may involve some search.

The synthesis goals which arise after a type change are proved by the strategy of chapter 6, usually with very little search.

I briefly discuss the example proofs I have used to demonstrate the techniques described in this thesis. These involve a variety of different abstract data types.

I identify several areas which require further work.

8.1 Implementation

8.1.1 Introduction

PITCHES consists of a collection of CLAM\(^1\) methods. Figure 8–1 lists the methods which must be loaded in order to run PITCHES. They must be present in the method database in the order listed.

There are three main aspects to the implementation:

\(^1\)The version used was CLAM version 2.2.3.
1. the provision of methods for program synthesis,

2. the choice of a data type change and its location in the synthesis proof, and

3. the synthesis of efficient implementations of functions on the new data type.

In the following sections I briefly describe the implementation of these aspects. For more details, see either the relevant chapter or the appendices.

8.1.2 Synthesis methods

The technique I have developed depends on the construction of synthesis proofs. These are proofs of specifications of the form:

\[ \forall \text{args} : \overline{1}. \exists \text{output} : t'. \text{spec(args, output)} \]

I have extended the proof strategy for induction to allow the proof of theorems which contain existential quantifiers. The extensions are outlined in §3.6.
Rather than using middle-out reasoning and implementing a global strategy for controlling speculative steps, as suggested in §3.5.5, I have avoided the introduction of meta-variables during proof planning by insisting that methods may use meta-variables internally as long as they instantiate them with object-level terms before they succeed. These instantiations are also restricted. The restrictions are described in §3.6.

Except for the type change method and the methods which implement the difference matching strategy, guidance for program synthesis is quite weak, which means that care must be taken to provide appropriate lemmas for the proof. For example, I have not implemented a generalisation method which replaces a synthesis of a compound function $f(g(x), y)$ with syntheses of $g(x)$ and $f(x', y)$. This is because the focus of the thesis has been the development of a technique for changing types in synthesis proofs, not the development of a general program synthesis system. I have implemented a minimal set of program synthesis methods in order to allow the type change technique to be demonstrated.

In the construction of a source synthesis proof plan, rippling provides guidance only in the step cases of inductions. When a type change has been performed in the target proof plan, the proof strategy of chapter 6 also applies rippling after weak fertilisation in order to guide the proof towards cancellation. Frequently, this phase of rippling follows a linear path. Search control is in fact tighter in the target proof than in the source proof.

### 8.1.3 Adding new type changes

The following steps are required in order to add a new type change to PITCHES:

1. The abstract data types must be written.

2. For each motivating expression, Expr, a motiv(Expr) rule must be defined by adding a clause to the predicate candidate_typechange/5.
3. The types of any new functions which are declared using the CLAM library mechanism instead of in an ADT must be specified by adding clauses to the declared.type/2.

4. Conversion functions must be declared as such by an appropriate call to addconversion/1. This normally takes place in the needs file.

5. If the type, t, which is to be changed appears in a parameterised type, for example t list, then a clause should be added to extend_conversion/3.

6. Acceptable existential witnesses must be declared by a call to addcanonical/1. This normally takes place in the needs file.

8.2 Implementation of the typechange/2 method

8.2.1 The structure of the method

The method is split into two parts. The first part determines which type changes to make in a goal. The second part processes this specification of a type change and returns a list of the resulting subgoals.

A transformation is determined by applying the propagation and motivation rules described in chapter 5. Each rule is encoded as a clause of the predicate typechange/5, listed in appendix F.1. This allows the standard Prolog debugging tools to be used, and also allows an internal representation to be used for meta-level sequents and for the specification of type changes. Providing heuristic control over this search process is further work discussed in §10.4.3.

8.2.2 Controlling application of propagation, motivation and allow rules

It is important to ensure that type changes are carried out as deep as possible within an expression and that any necessary auxiliary synthesis proofs are per-
formed. For example, as they stand, the rules of chapter 5 allow a type change of \( \text{app}(l, m) \) from \( \text{pnat list} \) to \( \text{dlist} \) in an expression \( \text{app(app}(l, m), m) \) to be derived immediately by application of a \( \text{motiv(app)} \) rule. This would lead to a specification in which the output type of the inner \( \text{app} \) is changed from \( \text{pnat list} \) to \( \text{dlist} \) without changing the input type. In the example, this would fail to extend the type change on \( m \) which is motivated by the outer application of \( \text{app} \) into a type change on the input of the inner application of \( \text{app} \).

This potential problem is solved by insisting that the \( \text{motiv} \) rules apply only to functions whose arguments are not compound expressions. When the arguments are compound expressions, the \( \text{motiv} \) rule fails to apply, and the \( \text{Fcomp} \) propagation rule is applied instead, which attempts to make a type change in the arguments as required.

Some search is often necessary in order to derive a profitable type change using the available rules. This search is limited by imposing a maximum depth on the length of a derivation using the rules. Currently this is set to 9. In a larger system it would be necessary to improve this search control as described in §10.4.3, but the strategy employed here has allowed type changes to be decided declaratively without the imposition of an arbitrary control strategy.

8.2.3 Auxiliary syntheses

As indicated in §5.5, it may be necessary to split synthesis into two, after application of an \( \text{Fcomp} \) propagation rule, to avoid losing a type change on the output of the inner function and the input of the output function. This splitting is carried out when it is necessary (for example in the \( \text{revflatten} \) example (§D.2.4)).

8.2.4 Object-level proof

I showed in §4.3 that at the object-level, applying a type change requires only the application of the lemmas of figures 4–3, 4–4 (page 88) which justify a type
change on an input and the output of a function respectively. Currently, the
\texttt{typechange} tactic has not been written for the following pragmatic, rather than
theoretical, reasons:

1. The questions I address in this thesis concern the construction of proof
plans and the control of search at the meta-level. The final step of trans¬
lating a proof plan to an object-level proof is only of secondary interest.

2. Construction of object-level proofs is currently only possible from proof
plans which do not use the ADT mechanism. These proof plans are there¬
fore restricted to using the built-in types of Oyster, which do not in prac¬
tice result in examples which are interesting.

3. Related to the above, in order to carry out an object-level proof, it is
necessary to first justify all the lemmas used in its construction. This can
require a significant amount of work, which points to a deficiency in the
current level of program synthesis and theorem proving technology used,
rather than in the derivation of type changes.

Implementation of the \texttt{typechange_strat/2} tactic is left as further work
(§10.4.4).

8.3 Synthesis proofs after a type change

The synthesis goals which are produced by the \texttt{typechange_strat/2} method are
proved by a combination of the proof strategy for induction, and the difference
matching and rippling proof strategy. After weak fertilisation, the \texttt{standard_form}
method attempts to difference match the goal with a standard form built from
conversion functions. If this succeeds, then the \texttt{step_case/2} method will carry
out any subsequent rippling. This should eventually result in an equality which
is trivially proved by the \texttt{elementary/1} method by introducing the existential
witness.
This proof strategy is described in chapter 6, and the standard form method is listed in appendix F.2.2.

The proof strategy often reduces search to a linear path, yielding extracted programs of an efficient form. I compare the rippling search space with that for a fold/unfold derivation in §9.3.1.

8.4 Results

This thesis has developed a program transformation technique and realised it as an automated proof process. Information from the original program synthesis proof is indirectly used in the target proof. The rippling used in the source proof determines how transformations can be moved up from their motivating expressions in the target, and also guarantees that part of the target proof can be constructed. This is the basis of the automation of the typechange/2 method.

The resulting synthesis theorems are proved automatically with the aid of difference matching and rippling.

In the following sections I briefly describe the proofs which have been constructed by Pitches to illustrate the techniques of this thesis. Most of the examples were used during development. A few examples, marked by an asterisk, were used for testing, i.e. they were not tried until the Pitches implementation was in its final state. Types are sometimes omitted for brevity.

A comprehensive comparison with systems such as [Blaine & Goldberg 91] is difficult, because they concentrate on a different problem, i.e. on large-scale, user-assisted, software development in which it is the responsibility of the user to ensure that an adequate theory exists which provides efficient implementations of functions. More detailed comparison is possible with techniques which attempt to automate the derivation of efficient implementations of functions after a type change, such as those described in §2.4.7 and §2.4.9. See chapter 9 for a comparison with previous work.
The example proofs demonstrate the practical application of the techniques which I have developed in this thesis. The automation of these techniques is based on sound principles which are independent of the details of their implementation: motivating expressions coupled with propagation rules choose the transformation, and difference matching followed by rippling automate the resulting synthesis proofs.

Several practical difficulties were encountered during the construction of a corpus of example proofs:

1. The type change technique relies on the expression of complex data types. Formulation of these as abstract data types is difficult, although once an ADT has been formulated and proved correct it can be reused in other proofs. For this reason, most of the test examples illustrate the list to dlist type change.

2. As mentioned in §8.1.2, guidance for synthesis proofs is quite weak, except after a type change, when difference matching and rippling provide very strong guidance.

### 8.4.1 Synthesis of functions on binary numbers

<table>
<thead>
<tr>
<th>Name</th>
<th>Success</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>btopsynth</td>
<td>yes</td>
<td>( \forall x : \text{binary}. \exists z : \text{nat}. z = \text{nat}(x) )</td>
</tr>
<tr>
<td>binplus</td>
<td>yes</td>
<td>( \forall x : \text{binary}, y : \text{binary}. \exists z : \text{binary}. \text{nat}(z) = \text{nat}(x) + \text{nat}(y) )</td>
</tr>
</tbody>
</table>

In appendix B I present a synthesis proof of the nat function. The proof does not involve a type change or difference matching, but demonstrates the methods I have developed for program synthesis.

The difference matching proof strategy is illustrated by the synthesis of an addition function on binary numbers in the binplus theorem. The specification is what would be expected after a type change from unary numbers to binary numbers, specifying binary addition in terms of unary addition and a conversion
function which maps each binary number to its unary equivalent. The proof involves some search. This is discussed further in §6.8.

### 8.4.2 Difference list examples

These examples illustrate the application of the type change method to a variety of synthesis proofs. In all the dlist examples, the Ind propagation rule (§5.9) is applied to move a type change from a motivating occurrence of app up past an induction.

All of these example proofs illustrate the use of the difference matching proof strategy. Sometimes induction is not used.

The source specifications (with types omitted for brevity) are:

<table>
<thead>
<tr>
<th>Name</th>
<th>Success</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>revdl²</td>
<td>yes</td>
<td>∀l. ∃m. m = rev(l)</td>
</tr>
<tr>
<td>flattendl²</td>
<td>yes</td>
<td>∀t. ∃l. l = flattent(t)</td>
</tr>
<tr>
<td>revflatten</td>
<td>yes</td>
<td>∀t. ∃l. l = rev(flattent(t))</td>
</tr>
<tr>
<td>apprevdl</td>
<td>yes</td>
<td>∀l. ∃m. m = app(rev(l), l)</td>
</tr>
<tr>
<td>revappdl</td>
<td>yes</td>
<td>∀l1, l2. ∃m. m = rev(app(l1, l2))</td>
</tr>
<tr>
<td>app4*</td>
<td>yes</td>
<td>∀l1, l2, l3, l4. ∃m. m = app(app(l1, l2), app(l3, l4))</td>
</tr>
<tr>
<td>revappdl2*</td>
<td>no</td>
<td>∀l1. ∃m. m = rev(app(l1, l1))</td>
</tr>
<tr>
<td>appprev2*</td>
<td>yes</td>
<td>∀l, m. ∃z. z = app(app(l, m), rev(l))</td>
</tr>
</tbody>
</table>

Two of the examples (marked with a ‖) are from [Sterling & Shapiro 86]. Those marked with a * were used for final testing after the code had stabilised. The transformed programs extracted from these examples could also be derived by the technique of [Zhang & Grant 88], but without the guarantee of correctness provided by Pitches.

The revappdl and apprevdl examples illustrate the combination of the Ind and the Fcomp propagation rules. Since induction/2 is lower in the method database than standard_form/2, it is not applied in a synthesis proof if difference
matching and rippling together can achieve a direct translation of functions from the old to the new data type, in this case from app to appenddl.

The revflatten example illustrates the splitting of a synthesis proof into two to fully exploit a type change in function composition (§8.2.3). In order to match the type changes in the component functions, a change of the input type of rev is permitted, resulting in the application of a different induction schema and hence producing a different synthesis of rev to that of revdl.

Example app4 was used as a test, and was successful. The synthesis proof is split into three, one for each occurrence of app, with list to dlist type changes in the inputs and outputs of each of them. The proof does not require induction.

A second test, revappdl2, fails because the type change method attempts to make two separate type changes on l1, causing the simplify.implementation/1 submethod to fail.

A third test, apprev2 was also successful.

### 8.4.3 nat to nat2 examples

<table>
<thead>
<tr>
<th>Name</th>
<th>Success</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>mod2sumlist</td>
<td>yes</td>
<td>( \forall l: \text{nat list}. s: \text{nat}. s = \text{summod2list}(l) \text{ in nat} )</td>
</tr>
<tr>
<td>mod2comp*</td>
<td>partial</td>
<td>( \forall x, y: \text{nat}. s: \text{nat}. s = \text{mod2}(\text{natplus}(\text{mod2}(x), \text{mod2}(y))) )</td>
</tr>
</tbody>
</table>

The proof of mod2sumlist (§D.4.2) is an enrichment of the natural numbers which demonstrates the rule (§5.8) for propagation past induction on a parameterised type (\( p\text{nat list} \), in this instance). This proof is very like the nat to nat2 example of §4.5.

In the test example, mod2comp, the type change method succeeded in finding the expected type change and in deriving the correct synthesis goals. One of these is an auxiliary synthesis of a nat2 version of natplus:

\[ \forall v7: \text{nat2}. \forall v8: \text{nat2}. \exists v3: \text{nat2}. p1(v3) = \text{natplus}(p1(v8), p1(v7)) \text{ in nat} \]
Initially this auxiliary synthesis failed because ripple analysis incorrectly suggested induction on \(v_7\), on the basis of a wave rule for \(p_1\). The subsequent inductive proof attempt diverged.

Examination of the failure of this auxiliary synthesis suggests the addition of equation (8.1) to the ADT.

\[
\forall x, y : \text{nat}2., p_1(\text{natplus2}(x, y)) = \text{natplus}(p_1(x), p_1(y)) \text{ in nat} \quad (8.1)
\]

After adding this equation and specifying that the \(\text{natplus2}\) function is an allowable existential witness, the synthesis succeeded as expected. This illustrates that although a type change may fail because the synthesis methods cannot prove the resulting synthesis subgoals, the descriptive clarity of proof plans enables a successful course of action to be identified. The resulting proof plan is in §D.4.3.

### 8.4.4 Queue examples

<table>
<thead>
<tr>
<th>Name</th>
<th>Success</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>queue.clist</td>
<td>yes</td>
<td>(\forall c : \text{circ}, i : \text{pnat}. \exists z : \text{circ}. \text{rep}(z) = \text{addq} (\text{rep}(c), i))</td>
</tr>
<tr>
<td>listlastq</td>
<td>yes</td>
<td>(\forall l : (\text{pnat list}) \text{ list}. \exists m : \text{pnatlist}. m = \text{listlastq}(l))</td>
</tr>
</tbody>
</table>

In queue.clist, search is reduced to a linear path. In §9.3.1, I compare search in the proof of queue.clist with that in a fold/unfold derivation of the same target function described in [Darlington 80].

The listlastq example illustrates the use of the ParmInd propagation rule.

### 8.4.5 An abstraction type change

In §7.10 I outline a simple abstraction example, replacing lists of natural numbers with pairs of their least and greatest elements.
Extension of the difference matching proof strategy to type abstraction is further work (§10.5.4), so the synthesis after the type change is performed almost entirely by the simplify_abstraction/1 submethod.

The minmax proof also illustrates the interleaving of a type change with other methods. In this case, a type change can only be made after a lemma application.

8.5 Conclusion

In this chapter I have briefly discussed the implementation of Pitches, which consists of a collection of ClAM methods. The addition of a few new methods has enabled ClAM to construct program synthesis proofs. The motivation and propagation rules of chapter 5 are implemented as clauses of a Prolog predicate. The choice of type change may require search, so derivations are limited to a fixed depth.

Once a type change has been chosen, the typechange_strat/2 method automatically generates new synthesis goals. The proof strategy described in chapter 6 has been implemented using difference matching and rippling. This proof strategy can prove without search many of the synthesis goals which arise after a type change. When search is required, as in the synthesis of an addition function for binary numbers, the proof strategy provides sufficient control to make the proof practicable.

A number of example proofs have been constructed to demonstrate the effectiveness of the methods I have developed. These use ADTs to express a variety of data types: difference lists, queues, circular lists and a simple enrichment of the natural numbers. They demonstrate the choice of type changes using motivation and propagation rules, and the subsequent synthesis proofs. Several of these examples are from [Sterling & Shapiro 86], and are also automatically done by [Zhang & Grant 88], but without the correctness guarantees provided by Pitches.
Four test proofs were constructed after development of the implementation was complete. Two were successful, and one failed. One was a partial success, but the initially failed proof attempt suggested the addition of an equation to the ADT, which enabled a second proof attempt to succeed.

Further work (chapter 10) should address the implementation of a tactic for the typechange_strat/2 method, and tactics for the abstract data type mechanism. This will allow object-level proofs to be constructed from the example proof plans.
Chapter 9

A Comparison With Previous Work

9.1 Introduction

In this chapter I compare my work with some previous approaches. I consider two aspects of previous techniques for data type reformulation:

1. Determination of a refinement or data type change. In work on algebraic program development, it is usually the responsibility of the user, aided by a strong theory, to choose a refinement and apply it at an appropriate point in the development process. In [Blaine & Goldberg 91, Schonberg et al 81], dataflow analysis is applied to the source program in order to choose where to carry out a refinement. In my work, propagation rules perform an analogous role in a synthesis proof.

2. Derivation of a program on the new data type. In most work, functions on the data type in the source program are translated to their equivalents on the new data type in the target program. This may not make the best use of the new data type. In [Darlington 80], a fold/unfold system is used to derive efficient implementations of functions from the source program. In
§9.3.1 I demonstrate the reduction in the search space achieved by rippling when compared with a simple fold/unfold strategy.

In §9.4, I compare my use of a source synthesis proof with that made by [Madden 91, Goad 80]. I do not use the proof directly, but assume that it has been constructed from a proof plan, and that the proof plan could be reconstructed if necessary. Therefore, I do not directly analyse the source proof, but rebuild parts of it when required. This increases the flexibility of Pitches because the source proof is not required, although its existence ensures that parts of the target proof can be constructed.

9.2 ADT refinement and translation

9.2.1 Choice of refinement

ADT refinement benefits from a strong theoretical basis. Composition of refinements allows complex changes of representation to be achieved by composing a number of simpler changes. This approach is advocated by Korf (§2.3.2), and theories of data type reformulation are used by [Lowry 89] (§2.4.5), [Blaine & Goldberg 91] (§2.4.5), [Sannella & Tarlecki 92] (§2.5.5) and [Luo 91] (§4.4.3).

The emphasis in ADT refinement work such as [Sannella & Tarlecki 92, Luo 91] is on the use of a strong theory to aid the user in applying refinements and proving their correctness. Generally, no significant automated assistance is provided to choose which refinements to make and where to apply them. The development of theorem proving assistance is mentioned in [Sannella & Tarlecki 92] as a highly desirable aim, although their intention is probably that it be used to partially automate proof that a refinement is correct.

In common with program transformation, efficiency of the target program is my main criterion, and I emphasise the use of heuristics which result in effi-
cient programs. Propagation and motivation rules are used to choose appropriate type changes and decide where in the proof/program development to apply them. D'TRE (§2.4.6) does exploit a theory of data type refinement and applies dataflow analysis in conjunction with user input to choose appropriate refinements and where they are made in a program. Dataflow analysis is also used in SETL (§2.4.8) to choose how to implement sets in a program, but only a limited repertoire of implementations of sets are considered.

The propagation rules I defined in chapter 5 also perform a kind of dataflow analysis: (5.2, 5.1) determine dataflow through function evaluation and composition, and (5.9, 5.8) determine dataflow through recursion in the program.

Dataflow analysis is a static process, whereas the type change method performs a dynamic analysis because:

1. This allows transformations where no source program is available.

2. There is the possibility of performing more complex analysis.

There is a static element to my analysis. When definitions are loaded into the CLAM theorem prover, they are parsed and various kinds of records are created which are accessible through the method meta-language:

1. Wave rule records: these are used by the induction propagation rules.

2. Function definition records: these are used by the propagation rules for function evaluation and composition.

In §2.4.8 I list some interesting points raised by [Low 78], which I address individually below:

1. When choosing a representation for an abstract data type, it is necessary to know how the data structure is used within the program — which primitives are used, how often etc..
My choice of representation is based on which functions are applied to a data type in a program, i.e. how it is used, but I am not able to make any use of frequency information.

2. An abstract data structure may be \textit{implicitly} represented. For example, the set of all odd integers may be best represented by a predicate.

I do not exploit this possibility.

3. Several different representations for a data structure may be used in different parts of the program.

Propagation rules maximise the scope of data type changes, but it is quite possible for a data type to be represented in several different ways within a program. Conversion functions ensure that a consistent interface is maintained between the different representations.

4. A structure can be represented redundantly by storing the same data in more than one representation.

Redundant representation of data can be achieved by combining the separate representations into a tuple, e.g. defining \( t' = t_1 \times t_2 \). Conversion functions will then ensure that the two different representations are always consistent.

\subsection{9.2.2 Translation rules}

In DTRE, translation rules map functions on the data type in a source program to their analogues on the new data type in the target program. It is the responsibility of the user to ensure that there are translation rules which specify efficient analogues of functions in the source program. Where a translation does not exist, a \textit{backstop rule} is created which combines existing translation rules to define a new translation rule. Generally, the translation of a compound expression (e.g. \( f(g(x)) \)) in the source program will be the composition of the
translations \(f'(g'(x)))\) in the target. This may not make the best use of the new data type.

I aim to exploit the change of data type to the full, and so instead of applying translation rules to map the source to the target program, I set up a specification of the target program. The difference matching and rippling proof strategy of chapter 6 can apply translation rules if they exist, since a rule specifying that a function \(f\) can be replaced in the target by a function \(f'\) on the new data type will be a wave rule of the form:

\[
\left[ f(p(x)) \right] = \rho\left( f'(x) \right)
\]  

(9.1)

Application of difference matching followed by rippling with (9.1), achieves the same effect as a translation from \(f\) to \(f'\).

We also have the option of automatically deriving an efficient translation. In a restricted system based on algebraic implementation, it may suffice to provide a small library of ADT implementations together with efficient implementations of many functions on the original data type.

Such an approach may be inadequate when many ADT implementations of many functions are considered. For example, the construction of complex implementations by the composition of a number of more primitive implementations can result in a large number of possible transformations. It may be unreasonable to create by hand definitions for all the important functions in an original data type. It may also be the case that efficient implementations are required for a very large number of functions. For example, we may wish to derive an efficient implementation for a specialisation of a function, or we may wish to be able to derive efficient implementations for any of an infinite class of \(n\)-ary functions.

\[\text{As long as the input and output type changes are equal. Relaxation of this constraint is discussed in §10.5.2.}\]
functions (e.g. for any of the infinite class of addition functions defined by 
\[ \text{plus}_n = \lambda x_1 \lambda x_2 \ldots \lambda x_n \cdot x_1 + x_2 + \ldots + x_n \].

These efficient implementations can be automatically derived by the tech-niques I have described and implemented.

9.3 Synthesis of efficient implementations

9.3.1 A comparison of rippling with fold/unfold

In (§2.4.7), I reproduced from [Darlington 80] the synthesis of an efficient imple-mentation for circular lists of the addq function on queues. Development of the recursive case involved unfolding once, then folding several times. Figure 9-3 depicts the search space to depth 4. This derivation tree was produced using a simple fold/unfold system, with several restrictions:

1. Rewriting an expression to itself by folding then immediately unfolding (and vice versa) with an equation are disallowed.

2. Certain equations can only be used for unfolding because when used for folding they introduce free variables. These equations are marked with a "—" in the figure. This restriction significantly reduces the size of the search space because subsequent to any such folding, any folding or unfolding step would be applicable to the free variables which were introduced. A consequence of this restriction is that some expressions (which do not contain free variables) can no longer be derived by folding and unfolding — essentially we are outlawing derivations which involve the introduction of intermediate variables.

The derivation tree contains 24 nodes (including repetitions), and 10 nodes are visited before the answer (node 10) is found by a depth-first left to right search limited to depth 4. If left to right iterative deepening is used instead as the search strategy, then the search visits 14 nodes before the answer is found.
Using the proof strategy for induction supplemented by the proof strategy described in chapter 6, CLAM automatically constructs a synthesis proof for addctrc which involves no search. The proof plan is reproduced in figure 9-1. Figure 9-2 lists the wave rules CLAM derives when the equations in the ADT are parsed and which are thus available during planning. Wave rules which do not contain functors which occur during the synthesis proof will not be applicable, which makes the rippling proof quite robust when a large number of equations are loaded. The ADTs are copied from [Darlington 80], with a few small changes to make the functions which can return an error element total. One point to note is that equation (9.9) is necessary for CLAM to prove the base case; the equations in the ADT which contain create and no variables are not wave rules.

In a large problem, search for a fold/unfold derivation becomes more and more difficult, especially when folding with the equations marked "−" in this problem is allowed. In contrast, when rippling, only equations relevant to the problem in hand are applied because it is only those equations which are applicable wave rules.

Clearly, some derivations which can be performed using fold/unfold are impossible to perform using rippling because they use rewrite rules which are not wave rules. In practice, the explosive search of fold/unfold may make such derivations impracticable. It is frequently the case that a proof attempt will fail because of the absence of a key rewrite rule. When this happens, the heuristic nature of rippling can allow the required rewrite rule to be determined, either by hand or automatically by a critic (§3.5.3). Failed fold/unfold proofs are by contrast difficult to analyse.

9.3.2 Lowry

In [Lowry 89], Lowry briefly mentions the derivation of efficient concrete implementations of functions from the abstract type. His strategy consists of pushing the retrieve (conversion) function through the equations specifying the relation-
split_implementation(bool:u(1)) then
split_implementation(queue:u(l)) then
split_implementation(clist:u(l)) then
induction([insert(x, i)],[c:circ]) then
[standard_form(standard_form,
   \forall v_0: \text{circ} \cdot \text{rep}(v_0) = \text{rep}(v_0) \text{ in queue}) then
   step_case(ripple(wave([2, 1],[clist.addqeql1, equ(left)],[])) then
   existential_subterm(z:circ, \text{insert(create, i)}) then
elementary(... ),
elim_existential(ih,[elim(v2, z,v3, v4)]) then
step_case(ripple(wave([1, 2, 1,2,2],[clist.repthml, equ(left)],[] then
   [wave([2,1,2,2],[queue.appendqeql, equ(right)],[])) then
   [fertilize(weak, [...]])
) then
elim_existential(ih,[elim(v2, z,v6, v7)]) then
standard_form(standard_form,
   \forall v_8: \text{circ} \cdot \text{rep}(v_8) = \text{rep}(v_8) \text{ in queue}) then
step_case(ripple(wave([2, 1],[clist.repthml, equ(right)],[])) then
existential_subterm(z:circ, \text{insert(v3,v0)}) then
elementary(... )
]

Figure 9-1: The proof plan automatically generated by Cl\textsuperscript{AM} for the synthesis of addcirc.
\begin{align}
\text{right}(\text{insert}(\text{insert}(z, x), y)) & \Rightarrow \text{insert}(\text{right}(\text{insert}(z, y)), x) \tag{9.2} \\
\text{insert}(\text{right}(\text{insert}(z, x)), y) & \Rightarrow \text{right}(\text{insert}(\text{insert}(z, y), x)) \tag{9.3} \\
\text{insert}(\text{right}(\text{insert}(z, x), y)) & \Rightarrow \text{right}(\text{insert}(\text{insert}(z, y), x)) \tag{9.4} \\
\text{right}(\text{insert}(\text{insert}(z, x), y)) & \Rightarrow \text{insert}(\text{right}(\text{insert}(z, y)), x) \tag{9.5} \\
\text{append}(\text{add}(\text{empty}\text{q}, x), \text{rep}(y)) & \Rightarrow \text{rep}(\text{insert}(\text{empty}\text{q} \text{, } x)) \tag{9.6} \\
\text{rep}(\text{insert}(x, y)) & \Rightarrow \text{append}(\text{add}(\text{empty}\text{q}, y), \text{rep}(x)) \tag{9.7} \\
\text{add}(\text{rep}(\text{create}), x) & \Rightarrow \text{rep}(\text{insert}(\text{create}, x)) \tag{9.8} \\
\text{rep}(\text{insert}(\text{create}, x)) & \Rightarrow \text{add}(\text{rep}(\text{create}), x) \tag{9.9} \\
\text{isempty}(x) & = \text{false} \text{ in bool} \Rightarrow \\
\text{front}(\text{add}(x, y)) & \Rightarrow \text{front}(x) \tag{9.10} \\
\text{isempty}(y) & = \text{false} \text{ in bool} \Rightarrow \\
\text{remove}(\text{add}(y, x)) & \Rightarrow \text{add}(\text{remove}(y), x) \tag{9.11} \\
\text{isempty}(y) & = \text{false} \text{ in bool} \Rightarrow \\
\text{add}(\text{remove}(y), x) & \Rightarrow \text{remove}(\text{add}(y, x)) \tag{9.12} \\
\text{isempty}(y) & = \text{false} \text{ in bool} \Rightarrow \\
\text{add}(\text{remove}(y), x) & \Rightarrow \text{remove}(\text{add}(y, x)) \tag{9.13} \\
\text{isempty}(y) & = \text{false} \text{ in bool} \Rightarrow \\
\text{remove}(\text{add}(y, x)) & \Rightarrow \text{add}(\text{remove}(y), x) \tag{9.14} \\
\text{append}(z, \text{add}(y, x)) & \Rightarrow \text{add}(\text{append}(z, y), x) \tag{9.15} \\
\text{add}(\text{append}(z, x), y) & \Rightarrow \text{append}(x, \text{add}(z, y)) \tag{9.16} \\
\text{add}(\text{append}(z, x), y) & \Rightarrow \text{append}(x, \text{add}(z, y)) \tag{9.17} \\
\text{append}(z, \text{add}(y, x)) & \Rightarrow \text{add}(\text{append}(z, y), x) \tag{9.18}
\end{align}

**Figure 9–2:** The wave rules which are available during the CLAM synthesis of addcirc.
ship between the abstract and concrete functions. For example, he starts with
an equation specifying addition \( r\text{add}_n(x, y) \) in some radix, \( n \):

\[
\rho(r\text{add}_n(x, y)) = \text{add}(\rho(x), \rho(y))
\]

The recursive case of the concrete definition is derived by applying an equa-
tion which splits numbers into quotient and remainder and then rewriting. This
strategy lacks the control we gain from the use of induction: a guarantee that
a terminating total function is synthesised, and the use of rippling to guide
towards fertilisation.

The rewriting which is performed in my proof by wave rules (6.17,6.25) is per-
formed by a single complex lemma. Also used are the distributivity of multiplica-
tion over addition and some other equations. In particular, one step consists of
replacing a subterm \( x \) of an equation with \( \text{combine}(\text{quotient}(x), \text{remainder}(x)) \).
This is a potentially explosive step. The final term must then be partially eval-
uated to obtain an efficient implementation.

The difference matching strategy I have presented represents an improvement
in several respects.

1. The explicit use of induction allows rippling to guide rewriting towards
fertilisation and guarantees the synthesised function is total and terminat-
ing.

2. No artificial rewrite rules are needed.

3. Difference matching and rippling accurately characterise the strategy of
pushing out the conversion function, tightly controlling search during the
rewriting process.

4. Difference matching and rippling give us the expectation (which can be
enforced) that the synthesised implementation will contain no conversion
functions.
5. The integration in Pitches of the type change methods with other program synthesis methods allows partial evaluation and other proof steps to take place in parallel with the type changes and subsequent synthesis, guided by the annotations produced by difference matching.

9.4 Proof transformation

9.4.1 Advantages of proof transformation

In §2.6 I described the work of Goad, and later Madden, which exploited proof transformation to transform programs. The use of proof transformation has the following advantages:

1. Transformations are possible which are impossible using only correctness-preserving program transformations.

2. Information which is present in the proof but not the program can be exploited in the transformation process.

In the following sections I examine how my work makes use of these.

9.4.2 Behaviour-changing transformations

The pruning example from [Goad 80] which I described in §2.6 is a good example of the kind of transformation which is possible using proof transformation. I do allow this kind of behaviour-changing transformation. For example, in the abstraction example of §7.10, application of a lemma during the proof allowed the development of a non-recursive (constant-time) function for generating new variable names, instead of the recursive source program. It is important to note without the data type change, there would be no efficiency improvement because it would still be necessary to recurse over the entire list of variable names when a new one was required.
This demonstrates not only that I exploit the behaviour-changing properties of proof transformation, but also suggests that program synthesis is necessary in order to make best use of a data type change; application of correctness-preserving program transformations could not have produced the new algorithm.

9.4.3 Use of information from proof

Both Goad and Madden explicitly map the source proof to the target proof. Proof analogy is another technique mapping source to target proofs. Goad's work predates the development of proof planning, and Madden does not exploit it, although he does construct an abstract representation of the source proof which resembles a proof plan. As I argued in §4.7, proof planning strategies and methods capture much of the similarity between proofs which is captured by analogy.

I have set out to make the best use possible of proof planning. One of my assumptions is that the source proof has been constructed from a proof plan. I do not explicitly refer to the source proof plan, but the fact that it has been constructed has several consequences which I do exploit:

1. A source proof plan exists and information can be extracted from it.

2. The theory (wave rules, lemmas etc.) which was used during the construction of the source proof is available during construction of the target proof.

3. The methods from which the source proof plan was constructed are available during construction of the target proof.

4. As I showed in §5.7.2, when a ParmInd propagation rule (5.8) has been applied in a proof, the step case of the target proof is guaranteed to proceed at least as far as weak fertilisation because it uses the same wave rules as were applied to reach weak fertilisation in the source proof.
When (2) and (3) are taken together, it suggests that the source proof, or any part of it, can be reconstructed if required. Therefore, it is not necessary to provide mechanisms for mapping parts of the source proof across to the target proof, although the source proof may be analysed to extract information useful in the construction of an improved target proof. The difference between my approach and that of Goad is illustrated in figures 9-5 and 9-6.

The rules governing propagation past induction could directly access the source proof plan to exploit the rippling in the step case of its induction. In practice, the analysis proceeds by looking ahead in the target proof. In effect, the parts of the source proof which are analysed are reconstructed as they are needed. This has advantages (+) and disadvantages (−):

1. (+) No special-purpose algorithm is required to analyse the source proof.

2. (+) Analysis is still possible when the source and target proofs diverge.

3. (−) The computational complexity of the transformation process is increased.

4. (−) If construction of the source proof required user interaction, and this interaction has not been stored in a form which allows its reuse in proof planning, then construction of the target proof can fail.

So, instead of extracting information from the source proof, I reconstruct and analyse parts of the source proof when they are required. I make use of the following information which is available during the course of the proof:

1. Propagation rules analyse the form of the source proof induction step case. It could be argued that the analysis of the step case is quite shallow, revealing only the recursive structure of the resulting program. This may be the case, although it may not necessarily be the case that the extracted program exactly mirrors the rippling which took place during its synthesis. The source proof offers an accurate characterisation of the recursive struc-
ture of the resulting program, which is available without additional analysis such as dataflow analysis.

2. Rewriting in the synthesis of new functions is guided by wave annotations.

The difference matching and rippling proof strategy of chapter 6 can result in proofs/programs which differ significantly from the source. Construction of these proofs/programs would generally require access to the theory used to construct the source, not the source program alone.

9.4.4 A summary of my use of proof transformation

I do not directly transform the source proof, but the analysis performed by propagation rules relies indirectly on the source proof, either the rippling part of the source proof plan itself, or the theory used in its construction.

I do exploit the opportunities afforded by proof transformation for the derivation of target programs which have different computational behaviour from the source programs. The availability of the theory used in the construction of the source program is crucial in the derivation of efficient target programs using the difference matching and rippling strategy.

9.5 Conclusion

In this chapter I have compared the work in this thesis with previous work on data type reformulation.

In some previous work, dataflow analysis is used to determine where in a program to carry out a data type change. The propagation rules I have defined perform an analogous role in program synthesis proofs.

Instead of translating the functions in the source program to their equivalents on the new data type in the target program, I set up a specification of the target
program. The difference matching and rippling proof strategy can prove such specifications with very little search.

I indirectly exploit the source synthesis proof.
Figure 9-3: The fold/unfold search space to depth 4 for the derivation of addcirc. Folds and unfolds are indicated with $f$ (equation) and $u$ (equation) respectively. Node 10 is the desired answer.
\[ \text{rep}(\text{addcirc}(\text{C, I})) = \text{addq}(\text{rep}(\text{C}), \text{I}) \] (star)

\[ \text{isemptycirc}(\text{create}) = \text{true} \] (isemptycirc1)

\[ \text{isemptycirc}(\text{insert}(\text{C, I})) = \text{false} \] (isemptycirc2)

\[ \text{value}(\text{create}) = \text{error} \] (value_err)

\[ \text{value}(\text{insert}(\text{C, I})) = \text{I} \] (value1)

\[ \text{removecirc}(\text{create}) = \text{error} \] (removecirc_err)

\[ \text{removecirc}(\text{insert}(\text{C, I})) = \text{C} \] (removecirc1)

\[ \text{right}(\text{create}) = \text{create} \] (right1)

\[ \text{right}(\text{insert}(\text{create, I})) = \text{insert}(\text{create, I}) \] (right2)

\[ \text{right}(\text{insert}(\text{insert}(\text{C, I}), \text{I})) = \text{insert}(\text{right}(\text{insert}(\text{C, I}))) \] (right3)

\[ \text{rep}(\text{create}) = \text{emptyq} \] (rep1)

\[ \text{rep}(\text{insert}(\text{C}, \text{I})) = \text{appendq}(\text{addq}(\text{emptyq}, \text{I}), \text{rep}(\text{C})) \] (rep2)

\[ \text{appendq}(\text{Q}, \text{emptyq}) = \text{Q} \] (<> q1)

\[ \text{appendq}(\text{Q}, \text{addq}(\text{R}, \text{I})) = \text{addq}(\text{appendq}(\text{Q}, \text{R}), \text{I}) \] (<> q2)

\[ \text{isemptyq}(\text{emptyq}) = \text{true} \] (q1)

\[ \text{isemptyq}(\text{addq}(\text{Q}, \text{I})) = \text{false} \] (q2)

\[ \text{frontq}(\text{emptyq}) = \text{error} \] (q3)

\[ \text{frontq}(\text{addq}(\text{Q}, \text{I})) = \text{if}(\text{isemptyq}(\text{Q}), \text{I}, \text{frontq}(\text{Q})) \] (q4)

\[ \text{removeq}(\text{emptyq}) = \text{error} \] (q5)

\[ \text{removeq}(\text{addq}(\text{Q}, \text{I})) = \text{if}(\text{isemptyq}(\text{Q}), \text{emptyq}, \text{addq}(\text{removeq}(\text{Q}, \text{I}))) \] (q6)

**Figure 9-4:** Equations for the fold/unfold synthesis of addcirc. Those marked with a - are only used for unfolding.
Figure 9-5: Information flow for direct program transformation, e.g. in Goad’s and Madden’s systems.

Figure 9-6: Information flow in PITCHES.
Chapter 10

Further Work

10.1 Introduction

In this chapter I briefly outline further avenues of research. The difference matching proof strategy of chapter 6 leads to a number of interesting lines of enquiry, which would also benefit other uses of rippling such as inductive proofs.

A comparison of propagation rules with the dataflow analysis and an analysis of their effect on the computational complexity of the extracted programs and the complexity of the proof planning process would be interesting.

The use of more algebraic techniques for data type refinement would increase the range of transformations PITCHES could carry out and bring my own work closer to that of [Lowry 89], for example.
10.2 Use of algebraic techniques

Several previous works have made use of algebraic techniques for data type refinement. In [Lowry 89, Blaine & Goldberg 91, Sannella & Tarlecki 88], refinements can be composed, and [Lowry 89] builds new refinements as compositions of an abstraction followed by an implementation.

As in §4.4.3, the retrieve function describing a composition of refinements can be formed from the retrieve functions describing the component refinements. In PITCHES, data type changes are characterised by retrieve functions or relations as appropriate, and the retrieve functions or relations for a composition could be derived from those for the components as above. This would allow a much greater range of type changes. Since the retrieve functions have computational content, there would be extra efficiency considerations.

10.3 ADT mechanism

As mentioned in §4.4.2, the split_implementation tactic has not yet been written. Application of the split_implementation method leaves only one subgoal, in which it is assumed that an implementation (model) of the ADT has been given and proven correct. The tactic would be expected to apply a lemma which supplies an implementation of the ADT and proves it correct, and extracts from this lemma proofs of the equations which have been created by the split_implementation/1 method. This is not a trivial task.

Once a lemma of this type has been written, it would be useful to develop tools (further methods and tactics) to aid the process of supplying an implementation for an ADT and proving it correct.
10.4 Propagation and program efficiency

10.4.1 Measuring program efficiency

As mentioned in §4.8, I rely on the heuristics to ensure that the modified program will be more efficient than the original. In this section I discuss a more rigorous approach which could be taken to ensuring that the transformation produces real efficiency improvements.

Clearly, we do not want to make the transformed programs less efficient than the originals. The justification step of the transformation proof (equation 4.5) involves inserting conversion functions, and these incur a computational overhead in the extracted program.

Complexity measures can be used to compare functions and determine time-efficiency. This is the approach I would like to take. In [Sands 89], complexity rules are attached to the proof rules of a logic in the same way that program construction rules are.

Although we are synthesising programs in Martin-Löf's Type Theory, we may intend to translate them into another programming language, for example pure Prolog or LISP. The implementations of data types in those languages may be much more efficient than in Martin-Löf's Type Theory. For example, an array can be represented in Martin-Löf's Type Theory as a function on the natural numbers, with an \( n + 1 \) element array \( x_0, ..., x_n \) represented as a lambda term:

\[
\lambda(x, p\text{nat.eq}(x, 0, x_0, p\text{nat.eq}(x, s(0), x_1, ..., p\text{nat.eq}(x, s^n(0), x_n, 0)...)))
\]

\( n + 1 \) \text{pnat.eq}'s

This is inefficient because it takes \( n + 1 \) execution steps to access the \( n^{th} \) element of the array, whereas in LISP array elements can be accessed directly after a little multiplication. Also, elementary arithmetic functions are generally implemented directly in hardware and so are cheap.
For these reasons it is desirable to attach complexity functions to data types directly, rather than calculating the complexity of the access functions as implemented in the type theory.

10.4.2 Extension of propagation rules

In §5.7.2 I identified three categories of rule for propagation through inductions on types of different structures. One of these has not been implemented: a type change $t(t_1) \rightsquigarrow t'(t_1)$. An example of such a transformation is provided by the motivation rule of (§4.8.1), in which an occurrence of `append` motivates a type change from `t list` to `t dlist`. A type change on `t` can be determined separately, either by expressions nested in the arguments of the motivating occurrence of `append`, or by other parts of the program.

In §7.10.5, I also proposed that a rule for propagation past lemma application should be developed. It seems likely that a propagation rule could be associated with each proof rule or synthesis method, just as computation rules are associated with proof rules in Martin-Löf's Type Theory.

Propagation rules perform the role in synthesis proofs performed by dataflow analysis in programs. I would like to investigate the relationship between propagation and dataflow analysis.

10.4.3 Controlling search for a type change

In §8.2 I suggested that instead of encoding propagation and motivation rules as clauses of a Prolog predicate, they be encoded as separate submethods. A type change would then be chosen by constructing a list of the possible type changes which result from a fixed-depth proof search with the propagation and motivation rules, and picking the best according to some heuristics relating to the expected efficiency of the synthesised programs.
In a large system which determines type changes in the way I have described in this thesis, further research may be necessary to control the search for a type change and choose between competing possibilities.

10.4.4 Implementation of the typechange_strat/2 tactic

I have not yet implemented the typechange_strat/2 tactic for the reasons outlined in §8.2.4. This should be implemented after implementation of tactics for the abstract data type mechanism (§10.3).

10.4.5 Use of proof critics

I briefly mentioned proof critics in §3.5.3. Proof critics analyse failed proofs, and have been used to speculate lemmas and to patch failed inductive proofs [Ireland & Bundy 95]. If the typechange/2 method applied a complexity criterion to decide whether a type change was profitable, then a critic could be defined which would be triggered by the failure of a type change due to the complexity criterion, and which would apply propagation rules to correct the situation. This would allow application of propagation rules during the course of a synthesis proof without the need for lookahead.

10.5 Extension of the difference matching proof strategy

10.5.1 Refinement of the strategy

The difference match and ripple strategy described in chapter 6 does not necessarily produce an extract term free of conversion functions. There are two reasons for this:
1. The annotated goal after difference matching may contain occurrences of the conversion function, \( p \), which are not in the skeleton, for example an annotated term such as \( \rho(f(x, \rho(y))) \). These extra occurrences of \( p \) may remain in the term after cancellation. To partially alleviate this problem, we can insist that the difference matcher returns multi-hole matches in which all occurrences of \( p \) are in the skeleton.

2. Some wave rules may move a copy of \( p \) out of the skeleton while keeping another copy in it, for example:

\[
\begin{align*}
\lfloor f(p(x)) \rceil & \Rightarrow \rho(\lfloor f'(p(x)) \rceil)
\end{align*}
\]

Both problems can be solved by checking that every annotated term produced by difference matching or by subsequent rippling contains no occurrences of the conversion function which are not in the skeleton. Neither problem has yet been seen in examples.

### 10.5.2 Different input and output transformations

At the moment the difference matching strategy can only be used when the input and output transformations are the same, which allows both sides of the equality to contain identical conversion functions.

If the input and output transformations differ, then the standard form (10.1) will no longer difference match with the goal (10.2).

\[
\begin{align*}
\rho_0(w) &= \rho_0(w) \\
\vdash \exists z. \rho_0(z) &= f(\rho_1(x)) \\
f(\rho_1(x')) &= \rho_0(f'(x'))
\end{align*}
\]

In addition, the rewrites (for example (10.3)) which are used in the proof will no longer be wave rules.
A related problem occurs when applying rewrite rules derived from mutually recursive functions. For example, the following are not wave rules:

\[
\begin{align*}
\text{even}(s(x)) & \leftrightarrow \text{odd}(x) \\
\text{odd}(s(x)) & \leftrightarrow \text{even}(x)
\end{align*}
\]

A possible solution is to define classes of functions which are considered equivalent for the purposes of skeleton preservation. For example, \{even, odd\} could be one such class. Such a solution is discussed in [Basin & Walsh 94b]. For my purposes, all conversion functions would be in a single class, so (10.3) could be annotated \( f(\rho_1(x')) = \rho_0(\rho'(x')) \).

A refinement of this strategy is to introduce a distance function such that the distance between two conversion functions \( \rho_1 \) and \( \rho_2 \) is defined by:

1. \( |\rho_1 - \rho_2| = |\rho_2 - \rho_1| \).
2. \( |\rho_1 - \rho_1| = 0 \).
3. \( |\rho_1 - \rho_2| = 1 \) if there is a rewrite rule \( F(\rho_1(x)) = \rho_2(F'(x)) \).
4. \( |\rho_1 - \rho_2| = \min_{\rho_3}(|\rho_1 - \rho_3| + |\rho_3 - \rho_2|) \)

Wave rules may displace the skeleton as indicated above, but the measure on this displaced skeleton must be decreased. Such a modification to rippling allows the rewrites we need to be seen as wave rules, and still guides rewriting towards cancellation. It is not certain, however, how the complexity of wave rule parsing and the number of wave rules generated would be affected. Such a modification of rippling would also be useful in handling inductive proofs involving mutually recursive functions, for example \( \text{even}(s(x)) \leftrightarrow \text{odd}(x) \). Any such weakening of rippling increases its applicability, but increases the size of the search space of possible ripples, and it is for this latter reason that the extension proposed here has not been implemented.
10.5.3 Using difference unification

Currently difference matching is used to produce the annotations which guide rippling towards application of a cancellation rule. Difference unification (§6.4.1) allows annotation of both the source (the goal after a type change) and target (equality over conversion functions). This can simplify some proofs.

In the synthesis of binary addition, for example, a standard form obtained by a one step ripple of the basic standard form with (6.18) was used. If instead we construct the standard form by a one step ripple with (6.17), the source and target no longer difference match. Difference unification succeeds in annotating both the source and target. The annotated source (10.4) and target (10.5) are as follows:

\[
\begin{align*}
\text{val}(d) + \text{nat}(w) + \text{nat}(w) & = \text{val}(d) + \text{nat}(w) + \text{nat}(w) \\
\forall \exists \text{v12: bin, v13: digit} \cdot \text{val}(\text{v13}) + \text{nat}(\text{v12}) + \text{nat}(\text{v12}) & = \text{s(nat(z) + s(nat(z)))}
\end{align*}
\]

Instantiation of the existentially quantified variables d and v13 to false followed by wave front normalisation yields exactly the annotated terms which are used in the proof presented in (§6.8.2).

The use of difference unification allows us to do without one wave rule (6.18), but introduces the complication of rippling on hypotheses as well as conclusions.

10.5.4 Difference matching after type abstraction

In chapter 7, the type change method was extended to abstraction type changes. This requires conversion functions to be generalised to conversion relations. The difference matching proof strategy can not be applied because it marks differences caused by functions present in the source term but not in the target, not relations.
Extension of the difference matching proof strategy to abstraction type changes would therefore involve generalising difference matching to mark \textit{relational} differences. This would be allow relational rippling (§3.5.6) to be applied to non-inductive proofs in the same way that functional difference matching can extend the applicability of functional rippling to non-inductive proofs.

It would also be necessary to verify that relational rippling is suitable for the subsequent synthesis proofs.

\textbf{10.5.5 Comparison of rippling with fold/unfold}

I have argued that the use of rippling significantly restricts search rewriting proofs compared to fold/unfold. Some empirical evidence which supports this claim is presented in §9.3.1. A thorough theoretical comparison of rippling with fold/unfold would be valuable in the context of my own work, and of rippling in inductive proofs.

\textbf{10.6 Abstraction}

\textbf{10.6.1 Allowing output transformations}

In chapter 7, I insisted that any output type change be an implementation; abstraction type changes on the output are not permitted. This ensures that a concrete function in the source proof/program can be replaced by its analogue on the abstract function while retaining correctness. Unfortunately, this strategy obstructs the smooth propagation of an abstraction type change throughout a function. Both the ParmInd (§7.8.2) and Ind rules, for example, require that the same type change be made on an input and the output of the step case function \(g\).

We cannot, for example, synthesise an analogue of append mapping sets to sets, only mapping sets to lists.
As noted in §7.7, an output transformation in a composition of functions is possible while still maintaining correctness. This means that we can allow an abstraction of the output type of a function \( f \) as long as we ensure that it always occurs as an argument of other functions \( g, h, \ldots \), and an input transformation can be made on each of these other functions.

A strategy based on this principle is suggested in [Blaine & Goldberg 91]. The quantifiers and conversion relations marked ‘independence condition’ in (7.3) enforce the condition that all concrete elements which correspond to the same abstract element should yield the same value when the concrete function is applied to them. This can be verified by showing that the equality axioms of the abstract type are respected by the concrete function when they are translated into the concrete type. For example, to show that a list to set abstraction type change is correct in a context \( g \), it is sufficient to show that:

\[
\forall a, b: \text{obj}. \forall l: \text{list}. g(a :: b :: l) = g(b :: a :: l) \land \\
\forall a: \text{obj}. \forall l: \text{list}. g(a :: a :: l) = g(a :: l)
\]

If this cannot be verified, then a wider context, \( f(g) \) can be considered, and so on, considering successively wider contexts.

It may be possible to carry out this strategy in parallel with the application of propagation rules.

### 10.6.2 Abstraction tactic in Oyster

The tactic for abstraction type changes has not yet been implemented, and the lemmas which justify an abstraction type change have not yet been proved.
10.6.3 Strategies for abstraction synthesis proofs

I discussed in §7.11 the difficulty of proving the synthesis goals which are produced after an abstraction type change. These problems are due to the relational nature of the synthesis goals, and the fact that they are stronger specifications than their counterparts for implementation type changes; analogues do not always exist after an abstraction, whereas they do after an implementation.

One possibility is to apply the techniques of [Kraan 94] which use middle-out reasoning to derive wave rules for the synthesised function.

10.7 Scaling up the proof strategy

The techniques I have developed have so far only been applied to small problems. Application to larger problems, or integration into a more mature program synthesis/development system would inevitably raise interesting questions.

10.8 Use of information in synthesis proofs

As I have already shown, there is information available in a synthesis proof which is either unavailable in the corresponding program, or which can only be derived from it by extensive analysis. Propagation rules indirectly use information contained in the rippling part of a synthesis proof to create a synthesis proof leading to a more efficient program. It seems important to use this information while it is available during program synthesis, rather than trying to manipulate the extracted program on its own, when it is too late. A possible line of future research is to identify other kinds of information which are available from synthesis proofs but not from programs alone.
10.9 Conclusion

In this chapter I have presented a number of possible avenues for further work. The principal areas of interest are in the areas of the difference matching and rippling proof strategy, and proof methods for the synthesis goals which arise after abstraction type changes.
Chapter 11

Conclusions

11.1 Contributions

I set out in this thesis to show that proof planning could be used to automate a particular kind of program transformation in which data types in a program are changed in order to make it more efficient.

The thesis makes the following contributions:

1. I have shown how type changes can be performed and proved to be correct for programs synthesised in Martin-Löf's Type Theory.

2. I have demonstrated that an ADT mechanism can be integrated into the CLAM proof planner.

3. I have developed heuristics for choosing and placing type changes in a program synthesis proof. In order to synthesise efficient programs, I have defined propagation rules which perform a similar kind of analysis in a synthesis proof to that performed by dataflow analysis in programs.

4. I have developed a proof strategy which proves automatically the synthesis goals which arise after a type change. This is valuable in itself, but also demonstrates the utility of proof planning: I identified the shape of the synthesis proofs I wanted to construct, and effectively encoded this as a proof strategy.
5. I have demonstrated the generality of existing proof methods, in particular rippling, by using them to automate new types of proof.

6. I have identified information available in the construction of a synthesis proof plan but not in a source program alone, which is useful in the development of an efficient target program (via its synthesis proof). Some of this information is used by propagation rules. Other information guides the synthesis proofs. I have indicated how this results in a great reduction in the size of the search space when compared to the fold/unfold strategy.

In the following sections I discuss each of these contributions in turn.

11.2 ADTs and type changes in CL\textsc{A}M

The use of ADTs was primarily made necessary by the need to express complex data types. The use of ADTs in proofs also has similar advantages to the use of ADTs in programs: they hide unnecessary details and allow the programmer (or proof planner) to reason at an appropriate level of abstraction. This makes ADTs particularly well suited to proof planning.

The use of ADTs allows more complex data types to be expressed in Martin-Löf's Type Theory than for example [Hamilton 93], which is limited to recursive data types. [Hamilton 93] adds inference rules to the type theory itself, an approach which could easily lead, if were extended to more complex data types, to the introduction of unsoundness into the type theory. By contrast, the equivalents of these inference rules are available as equations in the ADT, and part of the object-level proof involves supplying and proving correct an implementation of the ADT, thus guaranteeing the continued soundness of the logic.

ADTs fit in very well with the planning level, but further research is required to enable the corresponding object-level proofs to be constructed.

Once a type change has been chosen in a synthesis proof, the necessary subgoals are automatically constructed. When these subgoals are proved, a
lemma application combines them to form a new proof of the original specification. Since the end result is a proof of the original specification, the program which is synthesised is automatically correct, although it may compute a different function to the original program. Thus, I avoid potentially incorrect type changes such as the naïve difference list transformation whose unsoundness was identified in [Marriott & Sondergaard 88]. In addition, proof transformation permits transformations which cannot be achieved solely by the application of correctness-preserving program transformations.

The type change proof method can be extended to allow abstraction type changes, but the subsequent synthesis proofs are more difficult.

### 11.3 Heuristics for choosing type changes

The basic motivation for carrying out a type change, namely that it allows certain functions to be computed more efficiently, results in inefficient programs unless steps are taken to minimise the overheads caused from the introduction of conversion functions. The heuristics I use are similar to those used in other systems (§2.4.6,§2.4.8). In [Schonberg et al 81], representations of sets are chosen in order to ensure the efficient execution of parts of the program. I generalise this reasoning to allow representations to be chosen for many different data types.

Both [Blaine & Goldberg 91] and [Schonberg et al 81] use dataflow analysis to maximise the scope of a representation choice. Instead of applying dataflow analysis to a source program, I apply propagation rules to a synthesis proof as it is being constructed. These are effective in producing efficient programs, and can exploit some information which is available during proof planning. The application of a rule for propagation past induction guarantees that part of the target proof will succeed.

The choice and execution of type changes is carried out by a proof method, which can be interleaved in the synthesis with other proof methods.
11.4 Synthesis after a type change

In work based on implementing data types [Lowry 89, Blaine & Goldberg 91] or refining a theory [Luo 91], operations on the original data type are translated to operations on the new data type. This may not make the best possible use of the data type change. Darlington has used a fold/unfold system to derive efficient functions after a type change, but the proofs may involve a lot of search (§9.3.1).

In chapter 6 I showed how synthesis proofs of a particular form could synthesise efficient functions from the goals which arise after a type change. I then showed how proofs of this form can be constructed by a proof strategy which consists of difference matching with a standard form constructed from conversion functions, followed by rippling. This models the intended proof shape very well, and controls search in the proof very effectively when compared with the fold/unfold strategy.

The proof strategy is not only of value as a means of tightly controlling search in some synthesis proofs, but also suggests that proof planning may be able to overcome some of the search problems associated with program synthesis, and with transformation techniques such as fold/unfold.

11.5 Information in synthesis proofs

[Goad 80] demonstrated that useful information is contained in synthesis proofs which can be used to guide transformation (§2.6.2). Much of this information is also available in a program synthesis proof at the point it is required, i.e. it can be extracted where it is needed from the current synthesis goal and hypotheses. Therefore, we can exploit many of the advantages of proof transformation without reference to a source proof. Additional information can be obtained by performing some lookahead in the proof construction process.
Proof plans are an abstraction of object-level proofs which contain much of the useful information we require, e.g. propagation rules can exploit information extracted directly from the rippling part of a proof plan, or from the meta-level information which is accumulated when equations are loaded into the proof planner.

11.6 Summary

I have developed heuristics for changing the types in a program in order to make it more efficient.

These heuristics are implemented by proof methods which automatically decide which type changes to make and where in a program to make them. Instead of deriving a target program by applying correctness-preserving transformations to a source program, I construct programs indirectly via their synthesis proofs. The synthesised program must satisfy the original program specification, so it is necessarily correct. Information available during the construction of the synthesis proof plan can be used to determine an appropriate type change.

After a type change, the synthesis subgoals take a certain form. I have developed a proof strategy which uses rippling to synthesise efficient programs from goals of this form while controlling proof search very tightly. This not only produces a very-well controlled algorithm for program synthesis, but also demonstrates the flexibility of rippling.
Appendix A

Glossary

This appendix contains a glossary of technical terms used in this thesis.

Abstract Data Type (§2.4.2): A specification of a data type and the functions it requires. The structures which satisfy the specification are its models. The abstract data type contains axioms which must be satisfied by any model. An abstract data type can be expressed in type theory as an existential type, asserting the existence of the relevant type, functions and proofs that the axioms are satisfied.

Abstraction (§7): When information or structure is removed from a type. A function on the original type can only have an equivalent on the new type if the value of that function is the same for every concrete element which corresponds to a single abstract element.

CIAM (§3.3): The system which contains both an object-level logic, Oyster, and a set of proof planners.

Constructive Logic (§2.5.3): A restriction of classical logic. The law of the excluded middle is not valid, and from any proof of the form $\exists x. \text{spec}(x)$ it is possible to extract a program which computes an $x$ such that $\text{spec}(x)$ holds. The constructive logic used in this thesis is Martin-Löf's Type Theory.

Conversion Function (§4.2.2): A synonym for retrieve function.
Conversion Relation (§7.4.3): A synonym for retrieve relation.

Difference Matching (§6.4.1): The annotation of a source term $T$ such that its skeleton is equal to a target term $S$. Free variables in $T$ may be instantiated to unannotated terms.

Difference Unification (§6.4.1): An extension of difference matching in which both the source and target terms may be annotated in order to make their skeletons equal, and variables in both terms may be instantiated to unannotated terms.

Erasure (§3.7): The erasure of a wave term is the wave term with all the annotations removed.

Existential Witness (§2.5.3): The term introduced for an existential quantifier in a proof.

Extract Term (§2.5.3): A function which is constructed from a proof by replacing the inference rules in the proof by their corresponding program construction rules.

Fertilisation (§3.4.5): There are two types of fertilisation. In weak fertilisation, the induction hypothesis is an equality (or, more generally, a formula in which the main connective is transitive) and the induction conclusion contains a subterm which matches with one side the induction hypothesis, which is then applied as a rewrite rule to the induction conclusion. In strong fertilisation, the induction hypothesis and the induction conclusion are identical, and the former can be used directly to prove the latter.

Fold/Unfold (§2.2.2): A program transformation developed independently by Darlington and Manna/Waldinger. A single fold step rewrites a term in an expression by matching it with the left hand side of an equation (or function definition) and replacing the term with the instantiated right hand side of the equation. A single unfold step is the inverse of the above, matching a term with the right hand side of an equation and replacing it with the instantiated left hand side.
Implementation (§4.2.2): When information or structure is added to a type.

Induction (§3.4): A logical inference rule. Application of induction usually produces one or more base cases together with one or more step cases. In the proof plan for induction, the former subgoals are proved by a strategy which involves brute force rewriting. The latter subgoals contain wave fronts which mark the differences between the induction hypothesis and the induction conclusion, and they are proved by rippling these differences away until fertilisation can be performed.

Martin-Löf's Type Theory (§2.5.3): A higher-order constructive logic, implemented in the Oyster proof development system.

Meta-Level (§3.3): Meta-level terms are an abstraction of object-level terms in which hypotheses and type information may be missing. Certain information may be added in the form of annotations, either annotating object-level terms to give wave terms or marking certain hypotheses as induction hypotheses, amongst other things.

Method (§3.3.2): A method is a partial specification of a tactic. It is composed of several slots. When a method applies to a goal, it generates a list of subgoals to be proved. Associated with each method is the tactic which constructs the part of the object-level proof which corresponds to the part of the meta-level proof constructed by the method.

Motivating Expression (§5.4): An expression in a program or in the corresponding synthesis proof which can be made more efficient by a certain type change. Motivating expressions and the type changes which make them more efficient are stored in a library.

Object-Level (§3.2.2): Well-formed terms, proofs and inference steps of the logic in use, Martin-Löf's Type Theory in this thesis.

Oyster (§3.2.2): A Prolog implementation of a variant of Martin-Löf's Type Theory. Originally Oyster was based on NuPRL. Both Oyster and NuPRL are tactic-based, goal-directed theorem provers.
Pitches (§8.1.1): This is the name of the system I have developed to demonstrate the research described in this thesis. It stands for "PlannIng Type CHanges and Efficient Synthesis".

Program Synthesis (§2.5.3): The formal construction of a program. In this thesis, programs are synthesised by constructing proofs of their specifications and extracting programs from these proofs.

Program Transformation (§2.2): Manipulation of a program to produce an equivalent program which is better according to some criteria, typically that the transformation produce a program which executes in less time than the original. In the fold/unfold paradigm, a program is transformed by applying a sequence of equivalence-preserving fold/unfold transformations.

Proof Plan (§3.3): A proof plan can be viewed as a tree in which the nodes are instantiated methods. It describes how a theorem can be proved. A tactic can be constructed to prove a theorem by composing the tactics corresponding to the methods in a proof plan of the theorem.

Proof Planner (§3.3.3): A mechanism which creates proof plans. Initially the input to the proof planner will be the initial statement of the theorem to be proved. The proof planner tries to apply methods from the method database in some order determined by the order of methods in the method database and the particular search strategy employed by the proof planner. When a method applies, it generates a list of subgoals to be proved. This list may be empty in which case the method has completed a branch of the proof. The proof planner is then recursively called on all of these subgoals until either no subgoals remain and a complete proof plan has been constructed, or proof planning fails. The proof planner used in this thesis is called ClAM.

Proof Strategy (§3.4): A sequence of methods and submethods in the method database which generates proof plans of a particular form.

Proof Transformation (§2.6): Manipulation of a source proof to produce a target proof. This can be used to transform programs by transforming their synthesis proofs. The transformations which can be performed in this way are
more powerful than transformations on the program alone because after a pro-
gram transformation, the target program must have the same behaviour as the
source program, whereas after a proof transformation, the target program need
only satisfy the original specification.

Propagation (§5.1): The process of moving type changes which can be
performed in the subgoals of a proof step in a synthesis proof to as single type
change before application of the proof step. Propagation rules specify how type
changes can be propagated past various proof steps.

Retrieve Function: When a data type $t_1$ is implemented by a data type
$t_2$, we define a retrieve function $\rho : t_2 \rightarrow t_1$ such that $\forall x : t_1 \exists x' : t_2 . \rho(x') = x$. See references for conversion function.

Retrieve Relation: A binary relation on two types, $t_1 \times t_2$ which is a
generalisation of retrieve functions required for type abstraction. See references for conversion relation.

Rippling (§3.4.5): The successive application of wave rules.

Skeleton (§3.6): In the case of a single wave hole, the skeleton of a wave
term is the part of the wave term which is either completely outside the wave
fronts, or is inside a wave front and underlined. When there is more than one
wave hole the skeleton is a set. When a wave rule is applied to a wave term, the
skeleton of the result is a subset of the skeleton before wave rule application.

Submethod (§3.3.3): Submethods are exactly like methods, except that
they are stored in a different internal database and cannot be applied directly
by the proof planner but must be explicitly invoked by other methods or sub-
methods.

Type Change (§4): Replacing one type in a program and the operations
on that type with a new type and corresponding new operations. Given a type
change, a specification can be derived automatically of the operations on the
new type in terms of the operations on the original type and the retrieve func-
tion/relation which relates the old and new types.
Wave Front (§3.4): A wave front is represented by a box surrounding a term, some of whose subterms are underlined. The underlined subterms are the wave holes. A wave front has a direction which is indicated by an arrow attached to the box.

Wave hole (§3.4): See wave front.

Wave Measure (§3.4.4): A measure on annotated terms, i.e. a mapping from annotated terms to the natural numbers with certain properties.

Wave Rule (§3.4.4): A rewrite rule which has been annotated such that the skeleton of the left hand side is a superset of the skeleton of the right hand side, and the measure of the right hand side is strictly less than the wave measure of the left hand side. The skeleton preserving and measure-decreasing properties of wave rules allow a proof of the termination of rippling.

Wave Term (§3.5): A term which contains wave fronts. Every wave term has an erasure and a skeleton.
Appendix B

An example synthesis: nat

B.1 Introduction

This appendix details the synthesis of the nat function which converts binary numbers to natural numbers. This is partly for its own sake, partly as an example of program synthesis, and partly to illustrate the difference between proof at the planning level and at the object level.

B.2 The synthesis of nat

Binary numbers are represented by lists of booleans, with the least significant digit at the head of the list ("big-endian" representation).

The type bool has two elements, false and true, and obeys the following theorem, boolxor:

\[ \forall b : \text{bool}. b = \text{false} \; \text{in} \; \text{bool} \lor b = \text{true} \; \text{in} \; \text{bool} \]  
(B.1)

This theorem can be used to prove goals \( \vdash \forall x : \text{bool}. \text{spec}(x) \) by proving two subgoals, \( \vdash \text{spec}(\text{false}) \) and \( \vdash \text{spec}(\text{true}) \). Its proof provides the following extract term:
\[
\lambda(b, \text{decide}(b, [\sim, \text{inl(axiom)}], [\sim, \text{inr(axiom)}]))
\]

This extract term returns \text{inl(axiom)} or \text{inr(axiom)} depending on whether its input is a left injection (like \text{false}) or right injection (like \text{true}). It is used in the definition of \text{val}, (B.2).

The following definitions are required:

\[
\begin{align*}
\text{val}(x) & \overset{\text{def}}{=} \lambda x. \text{decide} \left( \text{term_of} (\text{boolxor}) \text{ of } x, [\sim, 0], [\sim, s(0)] \right) \quad \text{(B.2)} \\
\text{false} & \overset{\text{def}}{=} \text{inl(unit)} \quad \text{(B.3)} \\
\text{true} & \overset{\text{def}}{=} \text{inr(unit)} \quad \text{(B.4)} \\
\text{bool} & \overset{\text{def}}{=} \text{unary} \backslash \text{unary} \quad \text{(B.5)}
\end{align*}
\]

The following equations and wave rules are required for the proof.

\[
\begin{align*}
\text{val}(\text{false}) & = 0 \quad \text{(B.6)} \\
\text{val}(\text{true}) & = s(0) \quad \text{(B.7)} \\
0 + x & = x \quad \text{(B.8)} \\
[s(x)] + y & \Rightarrow [s(x) + y] \quad \text{(B.9)} \\
\text{nat}(\text{nil}) & = 0 \quad \text{(B.10)} \\
\text{nat}(\text{b :: } x) & \Rightarrow \text{val}(b) + (\text{nat}(x) + \text{nat}(x)) \quad \text{(B.11)}
\end{align*}
\]

The method database contains the following:

- split_implementation/1
- elim_existential/2
- elementary/1
- step_case/1
- standard_form/2

228
typechange_strat/2
existential_subterm/2
induction/2
base_case/1
guess_existential/3
sym_eval/1

CL\textsc{AM} automatically produces the proof plan which is represented in figure B-1. The corresponding object-level proof is illustrated in figure B-2. \texttt{wfftacs} denotes the application of tactics which prove certain well-formedness goals but make no contribution to the extract term. Currently, there are some well-formedness goals which \texttt{wfftacs} is not able to prove. In order for this object-level proof to run, \texttt{wfftacs} was turned off; well-formedness goals are not proved but are nevertheless marked as completed.

The final object-level term which is produced is:

\[
\text{nat}(x) = \lambda x. \text{list.ind}(x, \text{nat}(\text{nil}) \& \text{axiom}, [v_0, v_1, v_2, \\
\quad \text{spread}(v_2, [v_3, \sim, \text{spread}(v_2, [\sim, \sim, (\text{val}(v_0) + (v_3 + v_3)) \& \text{axiom}])])])
\]

The synthesis theorem is of the form $\forall x \exists z. \text{spec}(x, z)$, so the extracted program is of the form $\lambda x. f(x) \& p(x, f(x))$. When this is applied to a term $x$, it yields a pair of a $z$ such that $\text{spec}(x, z)$ holds, and a proof that $\text{spec}(x, z)$ is true. The proof is reduced by evaluation to \text{axiom}.

The following is an example run:

?– extract(X),
  eval(X of (\text{inl}(\text{unit})::\text{inr}(\text{unit})::\text{inl}(\text{unit})::\text{inr}(\text{unit})::\text{nil}), Y).
X = ...
Y = s(s(s(s(s(s(s(s(s(s(s(0)))))])).)))\&\text{axiom}

The input above is $[\text{false}, \text{true}, \text{false}, \text{true}] = 1010_2$, and the output is $10$ (ten) as expected.
\begin{figure}
\centering
\includegraphics[width=\textwidth]{proof_plan}
\caption{The proof plan for the synthesis of \texttt{nat}.}
\end{figure}

\begin{figure}
\centering
\includegraphics[width=\textwidth]{proof_tree}
\caption{The object-level synthesis of \texttt{nat}. A complete proof tree has a specification at its root, which is gradually broken down into simpler specifications at the intermediate nodes. The leaves of the proof tree are empty goals which are trivially true.}
\end{figure}
Appendix C

A session with Pitches

In this chapter I show an example session, in which CLAM derives a proof plan for the synthesis of flattendl.

The session runs as follows:

1. The theorem is loaded. It loads the necessary lemmas and methods that are needed for the proof.

2. The proof plan is constructed. Indentation is used to indicate the current depth in the proof plan.

3. The completed proof plan is displayed.

```prolog
?- lib_load(thm(flattendl)).
loading smthd(weak_fertilize/4) . done
loading smthd(fertilize_left_or_right/2) . done
loading smthd(wave/3) . . . . done
loading smthd(unblock/3) . . . . done
loading smthd(existential/2) . . done
loading smthd(wave/3) . . . . done
loading smthd(ripple/1) . done
loading mthd(induction/2) . done
loading mthd(split_existential/14222) . done
loading mthd(split_implementation/13677) . done
loading smthd(typechange5/479263) . done
loading smthd(rewrite_conversion/2) . done
```
loading smthd(implies_independence/480136)  .  done
loading smthd(relation_to_function/480143)  .  done
loading smthd(embedded_existential/480150)  .  done
loading smthd(simplify_abstraction/479595)  .  done
loading smthd(substitute_using_cfaq/480183)  .  done
loading smthd(prove_output_surjectivity/480190)  .  done
loading smthd(simplify_implementation/479702)  .  done
loading smthd(simplify_typechange_goals/479260)  .  done
loading smthd(typechange_strat/13692)  .  done
loading smthd(dm/13713)  .  done
deleting mthd(elementary/1)  .  done
loading mthd(existential_subterm/13744)  .  done
loading mthd(elim_existential/13759)  .  done
loading mthd(weak_fertilize/4)  .  done
loading mthd(guess_existential/13702)  .  done
loading mthd(eval_def/2)  .  done
loading mthd(wave/3)  .  .  .  .  .  done
loading mthd(ripple/1)  .  done
loading mthd(existential_wave/13810)  .  done
loading mthd(existential/2)  .  done
loading mthd(step_case/1)  .  done
loading mthd(weak_fertilize/4)  .  done
loading mthd(fertilize_left_or_right/2)  .  done
loading mthd(wave/3)  .  .  .  .  .  done
loading mthd(unblock/3)  .  .  .  .  done
loading mthd(existential/2)  .  done
loading mthd(wave/3)  .  .  .  .  .  done
loading mthd(ripple/1)  .  done
loading mthd(induction/13839)  .  done
loading mthd(base_case/1)  .  done
loading mthd(standard_form/13880)  .  done
deleting mthd(generalise/2)  .  done
CLnM WARNING: mthd(existential_split/13919)  not present,  so cannot be deleted
loading mthd(elementary/13931)  .  done
loading def(head)  .  .  .  .  .  done
loading def(tail)  .  .  .  .  .  done
loading eqn(hd)  .  .  .  .  .  done
loading eqn(hd1)  .  .  .  .  .  done
loading eqn(hd2)  .  .  .  .  .  done
  adding defeqn-record for hd1  .  .  .  .  .  .  done
  adding defeqn-record for hd2  .  .  .  .  .  .  done
loading def(t1)  .  .  .  .  .  done
loading eqn(t1)  .  .  .  .  .  done
loading eqn(t12)  .  .  .  .  .  done
  adding defeqn-record for t1  .  .  .  .  .  .  done

loading def(h1)  .  .  .  .  .  done
loading def(h2)  .  .  .  .  .  done
loading eqn(hd1)  .  .  .  .  .  done
loading eqn(hd2)  .  .  .  .  .  done
  adding defeqn-record for hd1  .  .  .  .  .  .  done
  adding defeqn-record for hd2  .  .  .  .  .  .  done
loading def(t1)  .  .  .  .  .  done
loading eqn(t1)  .  .  .  .  .  done
loading eqn(t12)  .  .  .  .  .  done
  adding defeqn-record for t1  .  .  .  .  .  .  done


232
adding defeqn-record for tl2...done
loading def(build)...done
loading def(appendd1)...done
loading def(app)...done
loading eqn(app1)...done
loading eqn(app2)...done
    adding wave-record for app2...done
    adding wave-record (backwards) for app2...done
    adding wave-record for app2...done
    adding wave-record (backwards) for app2...done
    adding defeqn-record for app1...done
    adding defeqn-record for app2...done
    adding recursive-record for app...done
loading def(split)...done
loading def(dlisttolist)...done
loading synth(cons)...done
loading def(cons)...done
loading def(body)...done
loading synth(node)...done
loading def(node)...done
loading def(leaf)...done
loading def(flattent)...done
loading eqn(aux_eqn1)...done
    adding defeqn-record for aux_eqn1...done
loading def(list_dlist)...done
loading def(dlist_list)...done
loading def(listdlist)...done
loading def(consd1)...done
loading thm(flattendl)...done

yes
| ?- select(flattendl).

yes
| ?- dplan.

DEPTH: 0
SELECTED at depth 0: split_implementation(tree: u(1))
    |DEPTH: 1
    |SELECTED at depth 1: split_implementation(bool: u(1))
    | |DEPTH: 2
    | |SELECTED at depth 2: split_implementation(dlist: u(1))
    | | |DEPTH: 3
    | | | ?- \forall t: tree. \exists ! pnat list. l = flattent(t) in pnat list
    | | |SELECTED at depth 3: typechange_strat( 
\[\text{depth} \leq 4\]

\[\text{depth} \leq 4\]

\[\text{depth} \leq 5\]

\[\text{depth} \leq 5\]

\[\text{depth} \leq 6\]

\[\text{depth} \leq 6\]

\[\text{depth} \leq 6\]

\[\text{depth} \leq 6\]

\[\text{depth} \leq 6\]
```plaintext
11111v1 : tree
11111v0 : tree
111111 : tree
11111l : tree
1111111 + \exists v3 : dlist. flattent(node(v1, v0)) = dlist. list(v3) in pnat list
11111111 SELECTED at depth 6: step.case(ripple(wave([1, 1, 2, 2], [tree. flattent v1, equ(left)]), ())) then
    [fertilize(weak, fertilize(weak-fertilize(left. in, [1, 1, 2, 2], v8), weak.fertilize(left. in, [2, 1, 1, 2, 2], v6)))]
1111111111 DEPTH: 7
11111111111111ih : [ USED,
    v6 : flattent(v0) = dlist. list(v5) in pnat list,
    v2 : \exists v3 : dlist. flattent(v0) = dlist. list(v3) in pnat list,
    v8 : flattent(v1) = dlist. list(v7) in pnat list,
    v4 : \exists v3 : dlist. flattent(v1) = dlist. list(v3) in pnat list]
1111111111111111 v5 : dlist
1111111111111111 v7 : dlist
1111111111111111 v1 : tree
1111111111111111 v0 : tree
1111111111111111 l : tree
1111111111111111 l : tree
1111111111111111 + \exists v3 : dlist. app(dlist. list(v7), dlist. list(v5)) = dlist. list(v3) in pnat list
11111111111111111 SELECTED at depth 7: elim.existential(ih, [elim(v2, v3, v9, v10), elim(v4, v3, v11, v12)])
11111111111111111111 DEPTH: 8
1111111111111111111111v9 : dlist
111111111111111111111111v11 : dlist
11111111111111111111111111ih : [ USED,
    ih : [ USED, elim.existential, v10 : flattent(v0) = dlist. list(v9) in pnat list,
    v12 : flattent(v1) = dlist. list(v11) in pnat list,
    v6 : flattent(v0) = dlist. list(v5) in pnat list,
    v2 : \exists v3 : dlist. flattent(v0) = dlist. list(v3) in pnat list,
    v8 : flattent(v1) = dlist. list(v7) in pnat list,
    v4 : \exists v3 : dlist. flattent(v1) = dlist. list(v3) in pnat list]
111111111111111111111111111111111111v5 : dlist
111111111111111111111111111111111111v7 : dlist
111111111111111111111111111111111111v1 : tree
111111111111111111111111111111111111v0 : tree
111111111111111111111111111111111111l : tree
111111111111111111111111111111111111l : tree
111111111111111111111111111111111111 + \exists v3 : dlist. app(dlist. list(v7), dlist. list(v5)) = dlist. list(v3) in pnat list
111111111111111111111111111111111111111 SELECTED at depth 8: standard.form(standard.form, \forall v3 : dlist. dlist. list(v3) = dlist. list(v3) in pnat list)
111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
vl2: flattent(vl) = dlist_list(v11) in pnat list,
v6: flattent(v0) = dlist_list(v5) in pnat list,
v2: 3v3: dlist.flattent(v0) = dlist_list(v3) in pnat list,
v8: flattent(v1) = dlist_list(v7) in pnat list,
v4: 3v3: dlist.flattent(v1) = dlist_list(v3) in pnat list]

|vl2: | dlist |
|vl7: | dlist |
|vl1: | tree |
|vl0: | tree |
|lt: | tree |

3v3: dlist_app(dlist_list(v7), dlist_list(v5)) = dlist_list(v3) in pnat list

SELECTED at depth 9: step_case(ripple(wave([1, 1], [dllst_appml, equ(left)], []))

DEPTH: 10

|standard_form: | [standard_form, 3v3: dlist, dlist_list(v3) = dlist_list(v3) in pnat list] |
|vl9: | dlist |
|vl11: | dlist |

th: [ USED, 

th: [ USED, ellm.existential, v10: flattent(v0) = dlist_list(v9) in pnat list, 
v12: flattent(v1) = dlist_list(v11) in pnat list, 
v6: flattent(v0) = dlist_list(v5) in pnat list, 
v2: 3v3: dlist, flattent(v0) = dlist_list(v3) in pnat list, 
v8: flattent(v1) = dlist_list(v7) in pnat list, 
v4: 3v3: dlist, flattent(v1) = dlist_list(v3) in pnat list] 

|vl5: | dlist |
|vl7: | dlist |
|vl1: | tree |
|vl0: | tree |
|lt: | tree |

3v3: dlist.dlist_list([appenddl(v7, v5)]) = dlist_list([v3]) in pnat list

SELECTED at depth 10: existential_subterm([v3]: dlist, [appenddl(v7, v5)])

DEPTH: 11

|standard_form: | [standard_form, 3v3: dlist, dlist_list(v3) = dlist_list(v3) in pnat list] |
|vl9: | dlist |
|vl11: | dlist |

th: [ USED, 

th: [ USED, ellm.existential, v10: flattent(v0) = dlist_list(v9) in pnat list, 
v12: flattent(v1) = dlist_list(v11) in pnat list, 
v6: flattent(v0) = dlist_list(v5) in pnat list, 
v2: 3v3: dlist, flattent(v0) = dlist_list(v3) in pnat list, 
v8: flattent(v1) = dlist_list(v7) in pnat list, 
v4: 3v3: dlist, flattent(v1) = dlist_list(v3) in pnat list] 

236
The following proof plan is constructed:

\[
\text{split\_implementation(tree} : \text{u(1)}) \text{ then}
\]
\[
\text{split\_implementation(bool} : \text{u(1)}) \text{ then}
\]
\[
\text{split\_implementation(dlist} : \text{u(1)}) \text{ then}
\]
\[
\text{typechange\_strat}[\text{[]} = \text{=} \text{tree} \rightarrow \text{tree} \rightarrow \text{\lambda x y. listdlist(x, y) + \text{[]}}. \\
[\text{relation\_to\_function}([\text{listdlist(v4, v3)}, \text{dist\_dlist\_listthm3}]), \\
\text{substitute\_using\_cfeq(v4 : pnat\_list), prove\_output\_surjectivity([])]. \\
[\text{relation\_to\_function}([\text{listdlist(flattent(v3, v3)}, \text{dist\_dlist\_listthm3}])]
\]
\]
\[
[\text{elementary\_intro(new[v3])\_then[identity, \text{wftacs}],} \\
\text{induction((node(left, right)), [t : tree]) then}
\]
\[
[\text{base\_case([sym\_eval([eval\_def([1, 1, 2, 2], [tree.flattenteqn1, equ(left)]), existential(v3 : dlist, build(nil)], \\
\text{eval\_def([2, 1], [dlist.dlist\_listthm1, equ(left)])), elementary(identity)).} \\
\text{base\_case([sym\_eval([eval\_def([1, 1, 2, 2], [tree.flattentthm1, equ(left)]), \\
\text{existential(v3 : dlist, build(v0 :: nil)], eval\_def([2, 1], [dlist.dlist\_listthm1, equ(left)])),} \\
\text{elementary\_intro(new[v0])\_then[identity, \text{wftacs}],} \\
\text{elim\_existential(ih, elim(v2, v3, v5, v6), elim(v4, v3, v7, v8)) then}
\]
\[
[\text{step\_case(ripple(wave([1, 1, 2, 2], [tree.flattenteqnl, equ(left)]), [])) then}
\]
\[
[\text{fertilize(weak, fertilize(weak_fertilize(left, in, [1, 1, 1, 1, 2, 2], v8),} \\
\text{weak_fertilize(left, in, [2, 1, 1, 2, 2], v6]))]}
\]
\[
\text{elim\_existential(ih, elim(v2, v3, v5, v6), elim(v4, v3, v7, v8)) then}
\]
\[
[\text{standard\_form(standard\_form, v3 : dlist, listdlist(v3) = listdlist(v3) in pnat\_list) then}
\]
\[
[\text{step\_case(ripple(wave([1, 1], [dlist.appthml, equ(left)]), [])]) then}
\]
\[
[\text{existential\_subterm(v3) : dlist, listdlist([[appenddl(v7, v5)^5]) in pnat\_list} \\
\text{elementary(identity)}
\]
\]
Appendix D

Example proofs and their ADTs

D.1 Introduction

This appendix lists the ADTs and proof plans for a range of example proofs.

The proof plans are essentially as output from CLAM. For the sake of clarity, in some places I have indicated the sequent to which a method applies. I have also suppressed the second argument of typechange_strat/2 which specifies the subplan which is used to convert the synthesis subgoals from a relational to a functional form.

D.2 The list to difference list transformation

This section lists the ADT for the list difference list transformation. This is followed by the synthesis conjectures for a number of programs, and the proof plans which are automatically generated for them by CLAM.
D.2.1 The difference list ADT and definitions

ADT tree:Type

FUNCTIONS

emptytree : tree
leaf : pnat → tree
node : tree → tree → tree
flattent : tree → pnat list

EQUATIONS

Vt : tree. t = emptytree in tree
   → 3n : pnat. t = leaf(n) in tree
   → 3left : tree. 3right : tree. t = node(left, right) in tree
flattent(emptytree) = nil in pnat list
Vn : pnat. flattent(leaf(n)) = n :: nil in pnat list
Vt : tree. Vr : tree. flattent(node(l, r)) = app(flattent(l), flattent(r)) in pnat list

INDUCTION treeind

∀p : (tree → u(l)). ∀t : tree.
p(emptytree) →
   (Vn : pnat. p(leaf(n))) →
   (Vleft : tree. Vright : tree. p(left) → p(right) → p(node(left, right))) →
   p(t)

ADT bool:Type

FUNCTIONS

true : bool
false : bool

EQUATIONS

true ≠ false in bool
∀q : bool → u(l). q(false) → q(true) → ∀b : bool. q(b)

ADT dlist:Type

FUNCTIONS

emptydl : dlist
head : dlist → pnat
tail : dlist → dlist
cons : pnat → dlist → dlist
build : pnat list → dlist
appenddl : dlist → dlist → dlist
split : dlist → dlist → dlist → bool
dlist.list : dlist → pnat list
dlist.dlist : dlist → dlist
list.dlist : pnat list → dlist
listdlist : pnat list → dlist → bool

EQUATIONS

239
The following definitions are loaded in addition to the ADTs above:

\[ \text{app}(\text{nil}, x) = x \]  \hspace{1cm} (D.1)
\[ \text{app}(\text{h} :: x, y) = h :: \text{app}(x, y) \]  \hspace{1cm} (D.2)
\[ \text{rev}(	ext{nil}) = \text{nil} \]  \hspace{1cm} (D.3)
\[ \text{rev}(\text{h} :: t) = \text{app}(\text{rev}(t), \text{h} :: \text{nil}) \]  \hspace{1cm} (D.4)

The "needs" file specifies that the following are conversion functions: \text{dllist_list} and \text{list_dllist}. The following are allowed in existential witnesses: emptytree,
leaf(\_), node(\_, \_), true, false, emptydl, head(\_), tail(\_), cons(\_, \_), build(\_), body(\_), appenddl(\_, \_), \_ :: \_.

All the type changes in this appendix are initiated by the following motivating expression:

<table>
<thead>
<tr>
<th>Function</th>
<th>Argument type changes</th>
<th>Output type change</th>
</tr>
</thead>
<tbody>
<tr>
<td>app(x, y)</td>
<td>( x : \text{pnat list} \rightarrow x' : \text{dlist}, )( y : \text{pnat list} \rightarrow y' : \text{pnat dlist} )</td>
<td>( \text{pnat list} \rightarrow \text{dlist} )</td>
</tr>
</tbody>
</table>

D.2.2 The transformation of flatten

The body of the synthesis conjecture, thm(flattendl) is:

\[ \forall t : \text{tree} \exists l : \text{pnat list} . l = \text{flattent}(t) \text{ in } \text{pnat list} \]

The type change is on the output type of the flatten function. The proof plan which ClAM constructs without search is listed in appendix C.

D.2.3 The transformation of rev

The body of the synthesis conjecture thm(revdl) is:

\[ \forall l : \text{pnat list} \exists z : \text{pnat list} . z = \text{rev}(l) \text{ in } \text{pnat list} \]

The transformation is again a list to difference list type change on the output type, leaving the input type unchanged.

The following proof plan is automatically generated by ClAM without search:

split_implementation(bool:u(1)) then
  split_implementation(dlist:u(1)) then
  typechange_strat(
    [] => \text{pnat list} \rightarrow \text{pnat list} - \lambda x.\lambda y.x = y \text{ in } \text{pnat list} + [] \Rightarrow
    \text{pnat list} \rightarrow \text{dlist} - \lambda x.\lambda y.\text{list}dlist(x,y) + [],

241
D.2.4 The transformation of revflatten

The synthesis conjecture is:

\[ \forall t : \text{tree} \exists l : \text{pnat list}. l = \text{rev}(\text{flattent}(t)) \text{ in } \text{pnat list} \]

The transformation is on the output type of flattent, changing it from type pnat list to type dlist. This induces a corresponding type change from pnat list to dlist on the input and the output of rev. Since there is a type change on both of these two functions, the proof is split into a synthesis of the modified flattent': tree \rightarrow dlist, and a synthesis of the modified rev': dlist \rightarrow dlist. This second synthesis is different to the synthesis of §D.2.3 because both the input and output types have been changed.

The proof plan which is automatically generated by CL\(^{AM}\) without search is:

\[
\text{split.implementation(tree:u(1)) then split.implementation(bool:u(1)) then split.implementation(dlist:u(1)) then typechange.strat(}
\]
(\emptyset \Rightarrow \text{tree} \rightarrow \text{tree} \rightarrow \lambda \; \mathcal{L} \; \text{y}. \; \mathcal{L} = \mathcal{L} \; \text{y} \; \mathcal{L} \; \text{in} \; \text{tree} \; + \; [\mathcal{L}]) \Rightarrow \text{pnat list} \rightarrow \text{dlist} \rightarrow \lambda \; \mathcal{L} \; \text{y}. \; \text{list} \; \text{dlist} \; (\mathcal{L}, \mathcal{L}) \Rightarrow \text{dlist} \rightarrow \lambda \; \mathcal{L} \; \text{y}. \; \text{list} \; \text{dlist} \; (\mathcal{L}, \mathcal{L}) \Rightarrow [\text{elementary}([\text{intro}(\text{new} \mathcal{V})] \text{then} [\text{identity}, \text{wfftacs}]),

\text{\forall} \mathcal{V} : \text{dlist}. \; \exists \mathcal{V} : \text{dlist}. \; \text{dlist list} \; (\mathcal{V}) = \text{rev} (\text{dlist list} \; (\mathcal{V})) \; \text{in} \; \text{pnat list}

\text{induction} ([\text{cons} \; (x, \text{dlist})], \; \mathcal{V} : \text{dlist}) \text{ then}

[\text{base case} ([\text{sym eval} ([\text{eval def} ([1, 2, 1, 2], \text{dlist list eqn1, eqv (left)])),

\text{eval def} ([2, 1, 2], \text{rev 1, eqv (left)]), \text{existential} ([\mathcal{V} : \text{dlist}, \text{build} (\text{nil})]),

\text{eval def} ([1, 1], \text{dlist list thm1, eqv (left)])), \text{elementary (identity)]}],

\text{elim existential} ([\text{ih}, \text{elim} (\mathcal{V} : 2, \mathcal{V} : 4, \mathcal{V} : 5, \mathcal{V} : 6)]) \text{ then}

\text{step case} (\text{ripple} ([\text{wave} ([1, 2, 1, 2], \text{dlist list thm3, eqv (left)]), []]) \text{ then}

[\text{wave} ([2, 1, 2, 2], \text{rev 2, eqv (left)]), []]) \text{ then}

[\text{fertilize} (\text{weak}, \text{fertilize} (\text{weak}, \text{fertilize} (\text{right}, \text{in}, [1, 2, 1, 2, 2], \mathcal{V})))]) \text{ then}

\text{elim existential} ([\text{ih}, \text{elim} (\mathcal{V} : 2, \mathcal{V} : 4, \mathcal{V} : 7, \mathcal{V} : 8)]) \text{ then}

\text{standard form} ([\text{standard form}, \mathcal{V} : 4 : \text{dlist list} \; (\mathcal{V}) = \text{dlist list} \; (\mathcal{V}) \text{ in} \; \text{pnat list}] \text{ then}

\text{step case} (\text{ripple} ([\text{wave} ([1, 2, 1, 2], \text{dlist app thm2, eqv (left)]), []]) \text{ then}

[\text{existential subterm} ([\mathcal{V} : 4 : \text{dlist, append list} \; (\mathcal{V} : 5, \text{list list} \; (\text{v0 :: nil}))]) \text{ then}

\text{elementary (identity)}]

].

\text{\forall} \mathcal{V} : \text{tree}, \; \exists \mathcal{V} : \text{dlist}. \; \text{flattent} \; (\mathcal{V}) = \text{dlist list} \; (\mathcal{V}) \; \text{in} \; \text{pnat list}

\text{induction} ([\text{node} \; (\text{left}, \text{right})], \; \mathcal{V} : \text{tree}) \text{ then}

[\text{base case} ([\text{sym eval} ([\text{eval def} ([1, 2, 1, 2], \text{tree flattent eqn2, eqv (left)])),

\text{existential} ([\mathcal{V} : 3 : \text{dlist}, \text{build} (\text{nil})]),

\text{eval def} ([2, 1], \text{dlist list thm1, eqv (left)])), \text{elementary (identity)]}],

\text{base case} ([\text{sym eval} ([\text{eval def} ([1, 2, 2], \text{tree flattent thm1, eqv (left)])),

\text{existential} ([\mathcal{V} : 3 : \text{dlist}, \text{build} (\text{v0 :: nil})]),

\text{eval def} ([2, 1], \text{dlist list thm1, eqv (left)])), \text{elementary (intro (new \mathcal{V})] then [identity, wfftacs])}],

\text{elim existential} ([\text{ih}, \text{elim} (\mathcal{V} : 2, \mathcal{V} : 3, \mathcal{V} : 5, \mathcal{V} : 6), \text{elim} (\mathcal{V} : 4, \mathcal{V} : 3, \mathcal{V} : 7, \mathcal{V} : 8)] \text{ then}

\text{step case} (\text{ripple} ([\text{wave} ([1, 1, 2, 2], \text{tree app thm1, eqv (left)]), []]) \text{ then}

[\text{fertilize} (\text{weak}, \text{fertilize} (\text{weak}, \text{fertilize} (\text{left}, \text{in}, [1, 1, 1, 2, 2], \mathcal{V}))), \text{weak fertilize} (\text{left}, \text{in}, [2, 1, 1, 2, 2], \mathcal{V}))]) \text{ then}

\text{elim existential} ([\text{ih}, \text{elim} (\mathcal{V} : 2, \mathcal{V} : 3, \mathcal{V} : 9, \mathcal{V} : 10), \text{elim} (\mathcal{V} : 4, \mathcal{V} : 3, \mathcal{V} : 11, \mathcal{V} : 12)] \text{ then}

\text{standard form} ([\text{standard form, \mathcal{V} : 3 : \text{dlist list} \; (\mathcal{V}) = \text{dlist list} \; (\mathcal{V}) \text{ in} \; \text{pnat list}] \text{ then}

\text{step case} (\text{ripple} ([\text{wave} ([1, 1], \text{dlist app thm1, eqv (left)]), []]) \text{ then}

[\text{existential subterm} ([\mathcal{V} : 3 : \text{dlist, append list} \; (\mathcal{V} : 5, \text{list list} \; (\text{v0 :: nil}))]) \text{ then}

\text{elementary (identity)}]

].
D.2.5 The transformation of apprevdl

The synthesis conjecture is:

\[ \forall l: \text{pnat list}. \exists m: \text{pnat list}. m = \text{app}(\text{rev}(l), l) \in \text{pnat list} \]

The following proof plan is generated automatically by CLAM.

```
split_implementation(tree: u(1)) then
  split_implementation(bool: u(1)) then
  split_implementation(dlist: u(1)) then
  typechange_strat([], => \text{pnat list} -> ... + []) => \text{pnat list} -> \text{dlist} - \lambda x. \lambda y. \text{listdlist}(x, y) + [],
    \[\lambda l: \text{pnat list} -> \text{dlist} - \lambda x. \lambda y. \text{listdlist}(x, y) + [] =: \]
    \[\text{pnat list} -> \text{dlist} - \lambda x. \lambda y. \text{listdlist}(x, y) + [],
      \]
  
  ...

  [elementary(intro(new[v5]) then[identity, wff tacs]),
   standard_form(standard_form, \forall v0: \text{dlist}. \text{dlist_list}(v0) = \text{dlist}. \text{list}(v0) in \text{pnat list} then
   \forall \forall: \text{dlist}. \forall \forall: \text{dlist}. \text{dlist_list}(\forall \forall) = \text{app}(\text{dlist_list}(\forall), \text{dlist_list}(\forall)) \text{ in } \text{pnat list}
   \text{step_case}(\text{ripple}(\text{wave}([2, 1], \text{dlist}. \text{appthm1}, \text{equ(left)}), [])) then
   \text{existential_subterm}(\forall \forall: \text{dlist}. \text{append}(\forall, \forall)) then
     \text{elementary(intro(new[v3]) then[identity, wff tacs], wff tacs]),
   
   \forall \forall: \text{pnat list}. \forall \forall: \text{dlist}. \text{rev}(l) \text{ in } \text{pnat list}
   \text{induction}([v0 :: v1], (l: \text{pnat list}]) then
   \text{base_case}(\text{sym_eval(\text{eval_def}([1, 1, 2, 2], \text{rev}, \text{equ(left)})), \text{existential}(\forall \forall: \text{dlist}. \text{build}(\text{nil})),
     \text{eval_def}([2, 1], \text{dlist}. \text{dlist_listthm1}, \text{equ(left)}), \text{elementary}(\text{identity})]),
   \text{elim_existential}(\text{th, \text{elim}([v2, v4, v6, v7]) then
     \text{step_case}(\text{ripple}(\text{wave}([1, 1, 2, 2], \text{rev}, \text{equ(left))}, [])) then
       \text{existential_subterm}(\forall \forall: \text{dlist}. \text{append}(\forall, \text{list}(\forall :: \text{nil})) then
         \text{elementary}(\text{identity})
   )]
  ]

```

244
D.2.6 The transformation of revappdl

The synthesis conjecture is:

\[ \forall l: \text{pnat list} \forall l2: \text{pnat list} \exists m: \text{pnat list} \cdot m = \text{rev(app(l1, l2))} \text{ in pnat list} \]

The proof plan which is automatically generated by CLAM is:

```
split_implementation (tree: u(1)) then
  split_implementation (bool: u(1)) then
    split_implementation (dlist: u(1)) then
      typechange_strat (\[
        \begin{align*}
          \forall l: \text{pnat list} & \rightarrow \text{dlist} - \lambda x. \lambda y. \text{listdlist}(x, y) + [], \\
          \forall l2: \text{pnat list} & \rightarrow \text{dlist} - \lambda x. \lambda y. \text{listdlist}(x, y) + [], \\
          \forall l: \text{pnat list} & \rightarrow \text{dlist} - \lambda x. \lambda y. \text{listdlist}(x, y) + [], \\
        \end{align*}
      \])
      pnat list \rightarrow \text{dlist} - \lambda x. \lambda y. \text{listdlist}(x, y) + [], \\
      pnat list \rightarrow \text{dlist} - \lambda x. \lambda y. \text{listdlist}(x, y) + [], .
      ...)
    elementary (intro(new[l4]) then [identity, wftacs]),
  \]
  \[ \forall l2: \text{dlist} \exists v4: \text{dlist} \text{. listdlist}(v4) = \text{rev} (\text{dlistdlist}(v3)) \text{ in pnat list} \]  
  induction ([cons(x, dll)]) then
    base.case (sym.eval ([eval.def([1, 2, 1, 2, 2], [dlist.dlist.dlist1, equ(left)]), \\
      eval.def([2, 1, 2, 2], [rev1, equ(left)]), \\
      existential(v4: dlist, build(nil)), eval_def([1, 1], [dlist.dlist.dlist3, equ(left)])),
      elementary (identity)),
    elim_existential (ih, [elim(v2, v4, v5, v6)]) then
      step_case (ripple (wave([1, 2, 1, 2, 2], [dlist.dlist.dlist4, equ(left)]))) then
        wave([2, 1, 2, 2], [rev2, equ(left)])) then
          [fertilize (weak, fertilize (weak, fertilize (right, in, [1, 2, 1, 2, 2], v6)))]) then
        elim_existential (ih, [elim(v2, v4, v7, v8)]) then
          standard_form (standard_form, Vv4: dlist . dlist.dlist(v4) = dlist.dlist(v4) in pnat list) then
            \[ \exists v4: \text{dlist} . \text{dlist.dlist}(v4) = \text{app} (\text{dlist.dlist}(v5), v0 :: nil) \]  
            in pnat list
          step_case (ripple (wave([2, 1], [dlist.appthm2, equ(left)]))) then
            existential_subterm (v4: dlist . \[appenddl(v5, dlist.dlist(v0 :: nil))\]) then
              elementary (identity)
    elementary (intro(new[l1]) then [identity, wftacs]),
  elementary (intro(new[l2]) then [identity, wftacs]),
  elementary (intro(new[l3]) then [identity, wftacs]) \]
```

245
D.2.7 The transformation of app4

This is one of the test examples. The synthesis conjecture before the type change is:

$$\forall 1, 2, 3, 4 : \text{pnat list } \exists m : \text{pnat list} . = \text{app(app}(1, 2), \text{app}(3, 4))$$

The proof plan which is produced without search (and also without induction) is:

```plaintext
split_implementation(tree: u(1)) then
split_implementation(bool: u(1)) then
split_implementation(dlist: u(1)) then
  typechange_strategy([1: \text{pnat list } \rightarrow : \text{dlist } - \lambda x. \lambda y. \text{listdlist}(x, y) + []],
  12: \text{pnat list } \rightarrow : \text{dlist } - \lambda x. \lambda y. \text{listdlist}(x, y) + [] =: >
  \text{pnat list } \rightarrow : \text{dlist } - \lambda x. \lambda y. \text{listdlist}(x, y) + []],
  13: \text{pnat list } \rightarrow : \text{dlist } - \lambda x. \lambda y. \text{listdlist}(x, y) + [] =: >
  \text{pnat list } \rightarrow : \text{dlist } - \lambda x. \lambda y. \text{listdlist}(x, y) + [] =: >

\text{pnat list } \rightarrow : \text{dlist } - \lambda x. \lambda y. \text{listdlist}(x, y) + [] =: >
...
[elementary(intro(new[v5]) then [identity, wfftacs]),
standard_form(standard_form, \forall v0 : \text{dlist. dlist. list}(v0) = \text{dlist. list}(v0) in \text{pnat list}) then
  step_case(ripple(wave([2, 1], [\text{dlist. appthm1}, \text{equ}(left)], [])]) then
    existential_subterm([v5: \text{dlist. appenddil}(v4, v3)] then
      elementary(intro(new[v3]) then [intro(new[v4]) then [identity, wfftacs, wfftacs]],
    standard_form(standard_form, \forall v0 : \text{dlist. dlist. list}(v0) = \text{dlist. list}(v0) in \text{pnat list}) then
      step_case(ripple(wave([2, 1], [\text{dlist. appthm1}, \text{equ}(left)], [])]) then
        existential_subterm([v4: \text{dlist. appenddil}(v10, v9)] then
          elementary(intro(new[v9]) then [intro(new[v10]) then [identity, wfftacs, wfftacs]],
          elementary(intro(new[l1]) then [identity, wfftacs]),
        elementary(intro(new[l2]) then [identity, wfftacs]),
      standard_form(standard_form, \forall v0 : \text{dlist. dlist. list}(v0) = \text{dlist. list}(v0) in \text{pnat list}) then
        step_case(ripple(wave([2, 1], [\text{dlist. appthm1}, \text{equ}(left)], [])]) then
          existential_subterm([v3: \text{dlist. appenddil}(v12, v11)] then
            elementary(intro(new[v11]) then [intro(new[v12]) then [identity, wfftacs, wfftacs]],
            elementary(intro(new[l3]) then [identity, wfftacs]),
          elementary(intro(new[l4]) then [identity, wfftacs]))
]
```
D.2.8 The transformation of apprev2

This is another test example. The synthesis conjecture is:

∀l, m : pnat list ∃z : pnat list. z = app(app(l, m), rev(l)) in pnat list

The proof plan which is automatically generated by CLAM without search is reproduced below. The arguments of the typechange_strat/2 method have been abbreviated to save space.

split_implementation(tree : u(1)) then
split_implementation(boot : u(1)) then
split_implementation(dlist : u(1)) then
typechange_strat(... ; pnat list — difference list ...)
[elementary(intro(new[v5]) then [identity, wffacs]),
standard_form(standard_form, v5 : dlist.dlist.list(v0) = dlist.list(v0) in ; pnat list) then
step_case(ripple(wave([2, 1], [dlist_appthm1, equ(left)]), [], [])) then
existential_subterm(v5 : dlist, [appendd(v4, v3)]) then
elementary(intro(new[v3]) then [intro(new[v4]) then [identity, wffacs]], wffacs),
standard_form(standard_form, v5 : dlist.dlist.list(v0) = dlist.list(v0) in ; pnat list) then
step_case(ripple(wave([2, 1], [dlist_appthm1, equ(left)]), []) then
existential_subterm(v5 : dlist, [appendd(v10, v9)]) then
elementary(intro(new[v9]) then [intro(new[v10]) then [identity, wffacs]], wffacs),
elementary(intro(new[l]) then [identity, wffacs]),
elementary(intro(new[m]) then [identity, wffacs]),
induction([v0 :: v1], [1 : pnat list]) then
[base_case([sym_eval([eval.def([1, 1, 2, 2], [rev1, equ(left)]),
existential(v3 : dlist, build(nil)], eval.def([2, 1], [dlist.dlist.listthm1, equ(left)])),
elementary(identity)])],
elim.existential(ih, [elim(v2, v3, v4, v5)]) then
step_case(ripple(wave([1, 1, 2, 2], [rev2, equ(left)]), [])) then
[fertilize(weak, fertilize([weak.fertilize(left.in., [1, 1, 1, 2, 2, v5])])]) then
elim.existential(ih, [elim(v2, v3, v6, v7)]) then
standard_form(standard_form, v5 : dlist.dlist.list(v3) = dlist.list(v3) in ; pnat list) then
step_case(ripple(wave([1, 1], [dlist.appthm2, equ(left)]), []) then
existential_subterm(v5 : dlist, [appendd(v4, list.dlist(v0 :: nil)])]) then
elementary(identity)]}
D.3 The list to \((\min, \max)\) abstraction example

This appendix lists the ADT and proof plan for a list to \((\min, \max)\) abstraction.

**ADT** \texttt{mminmax}:Type

**FUNCTIONS**

\[
\begin{align*}
\texttt{mmin} &: \texttt{mminmax} \to \texttt{pnat} \texttt{\unary} \\
\texttt{mmax} &: \texttt{mminmax} \to \texttt{pnat} \texttt{\unary} \\
\texttt{mnil} &: \texttt{mminmax} \\
\texttt{madd} &: \texttt{pnat} \to \texttt{mminmax} \\
\texttt{mminmaxtolist} &: \texttt{mminmax} \to \texttt{pnat list} \\
\texttt{listtomminmax} &: \texttt{pnat list} \to \texttt{mminmax} \\
\texttt{mminmaxlist} &: \texttt{pnat list} \to \texttt{mminmax} \\
\texttt{mminmaxlistlist} &: \texttt{pnat list list} \to \texttt{mminmax list} \\
\texttt{listtomminmaxlist} &: \texttt{pnat list list} \to \texttt{mminmax list}
\end{align*}
\]

**EQUATIONS**

\[
\begin{align*}
\forall \texttt{n:pnat}.\forall \texttt{m:mninmax}.\exists \texttt{a:pnat}.\ \texttt{mmin}(\texttt{m}) &= \texttt{inl}(\texttt{a}) \texttt{in \ unary} \to \\
& (\texttt{n} < \texttt{a}) \to \texttt{mmin}(	exttt{madd}(\texttt{n}, \texttt{m})) = \texttt{inl}(\texttt{n}) \texttt{in \ unary} \\
\forall \texttt{n:pnat}.\forall \texttt{m:mninmax}.\exists \texttt{a:pnat}.\ \texttt{mmin}(\texttt{m}) &= \texttt{inl}(\texttt{a}) \texttt{in \ unary} \to \\
& (\texttt{n} < \texttt{a}) \to \texttt{mmin}(	exttt{madd}(\texttt{n}, \texttt{m})) = \texttt{inl}(\texttt{a}) \texttt{in \ unary} \\
\forall \texttt{n:pnat}.\forall \texttt{m:mninmax}.\exists \texttt{a:pnat}.\ \texttt{mmax}(\texttt{m}) &= \texttt{inl}(\texttt{a}) \texttt{in \ unary} \to \\
& (\texttt{n} < \texttt{a}) \to \texttt{mmax}(	exttt{madd}(\texttt{n}, \texttt{m})) = \texttt{inl}(\texttt{n}) \texttt{in \ unary} \\
\forall \texttt{n:pnat}.\forall \texttt{m:mninmax}.\exists \texttt{a:pnat}.\ \texttt{mmax}(\texttt{m}) &= \texttt{inl}(\texttt{a}) \texttt{in \ unary} \to \\
& (\texttt{n} > \texttt{a}) \to \texttt{mmax}(	exttt{madd}(\texttt{n}, \texttt{m})) = \texttt{inl}(\texttt{n}) \texttt{in \ unary} \\
\texttt{mmin}(\texttt{mnil}) &= \texttt{inr}(\texttt{unit}) \texttt{in \ unary} \\
\texttt{mmax}(\texttt{mnil}) &= \texttt{inr}(\texttt{unit}) \texttt{in \ unary} \\
\texttt{mminmaxtolist}(\texttt{mnil}) &= \texttt{nil}
\end{align*}
\]
\[\forall m:\mminmax. \exists \text{amin}: \text{pnat}. \text{mmin}(m) = \text{inl}(\text{amin}) \text{ in } \text{pnat}\text{\_unary} \rightarrow\]
\[\exists \text{amax}: \text{pnat}. \text{mmax}(m) = \text{inl}(\text{amax}) \text{ in } \text{pnat}\text{\_unary} \rightarrow\]
\[\text{mminmaxtolist}(m) = \text{amin} : \text{amax} : \text{nil} \text{ in } \text{pnat}\text{\_list}\]
\[\forall l:\text{pnat}\text{\_list}. l \neq \text{nil} \text{ in } \text{pnat}\text{\_list} \rightarrow\]
\[\text{listtomminmax}(l) = \text{listtomminmaxlist}(l) \text{ in } \text{mminmax}\text{\_list}\]
\[\forall m:\mminmax. \text{maxof}(\text{mminmaxtolist}(m)) = \text{mmax}(m) \text{ in } \text{pnat}\text{\_unary}\]
\[\forall l:\text{pnat}\text{\_list}. \forall m:\mminmax. \text{mminmaxlist}(l, m) \rightarrow \text{maxof}(l) = \text{mmax}(m) \text{ in } \text{pnat}\text{\_unary}\]

\[\text{D.3.1 The genvar synthesis proof}\]

The synthesis conjecture is:

\[\forall l:\text{pnat}\text{\_list}. \exists x: \text{pnat}. \text{\_member}(x, l)\]

Further details of this proof, and the proof plan generated by Cl\text{\textsc{am}} are presented in §7.10.
D.4 The nat to nat2 transformation

This chapter lists the ADT for the nat to nat2 type change, and the proof plan which was generated for an example transformation.

D.4.1 The nat and nat2 ADTs

**ADT bool:u(1)**

**FUNCTIONS**

- false : bool
- true : bool
- eq : (nat — nat — bool)
- or : (bool — bool — bool)
- and : (bool — bool — bool)

**ADT nat:u(1)**

**FUNCTIONS**

- zero : nat
- succ : (nat — nat)
- natplus : (nat — nat — nat)

**EQUATIONS**

\[ \forall x : \text{nat}. \text{natplus}(\text{zero}, x) = x \text{ in } \text{nat} \]

\[ \forall y : \text{nat}. \text{natplus}(\text{succ}(x), y) = \text{succ}(\text{natplus}(x, y)) \text{ in } \text{nat} \]

**INDUCTION natind**

\[ \forall p : (\text{nat} — u(1)). \forall n : \text{nat}. \]

\[ p(\text{zero}) —\]

\[ (\forall m : \text{nat}. p(m) — p(\text{succ}(m))) —\]

\[ p(n) \]

**INDUCTION natind2**

\[ \forall p : (\text{nat} — u(1)). \forall n : \text{nat}. \]

\[ p(\text{zero}) —\]

\[ p(\text{succ}(\text{zero})) —\]

\[ (\forall m : \text{nat}. p(m) — p(\text{succ}(\text{succ}(m)))) —\]

\[ p(n) \]

**ADT nat2:u(1)**

**FUNCTIONS**

250
boolnot : (bool -> bool)
val : (bool -> nat)
pair : (nat -> nat2)
p1 : (nat2 -> nat)
p2 : (nat2 -> bool)
natpl : (nat -> nat2 -> boolean)
natpllist : (nat list -> nat2 list -> bool)
lp1 : (nat2 list -> nat list)
mod2 : (nat -> nat)
succ2 : (nat2 -> nat2)
natplus2 : (nat2 -> nat2 -> nat2)

EQUATIONS

\forall n : nat. \forall n2 : nat2. natpl(n, n2) \rightarrow n = p1(n2) in nat
\forall l : nat list. \forall l2 : nat2 list. natpllist(l, l2) \rightarrow 1 = lp1(l2) in nat list
n = p1(n2) in nat \rightarrow natpl(n, n2)
\forall x : nat2. \forall y : nat2. natplus2(x, y) = pair(natplus(p1(x), p1(y))) in nat2
\forall x : nat2. \forall y : nat2. p1(natplus2(x, y)) = natplus(p1(x), p1(y)) in nat
\forall x : nat2. mod2(p1(x)) = val(p2(x)) in nat
val(false) = 0 in nat
val(true) = succ(0) in nat
p1(pair(n)) = n in nat
val(p2(pair(n))) = mod2(n) in nat
p1(succ2(n2)) = succ(p1(n2)) in nat
p2(succ2(succ2(n2))) = p2(n2) in nat
p2(succ2(n2)) = boolnot(p2(n2)) in bool
mod2(zero) = zero in nat
mod2(succ(zero)) = succ(zero) in nat
\forall x : nat. mod2(succ(succ(x))) = mod2(x) in nat
\exists x : nat. natplus(mod2(n), natplus(x, x)) = n in nat
boolnot(false) = true in bool
boolnot(true) = false in nat
\forall l : nat list. \exists l2 : nat2 list. l = lp1(l2) in nat list
lp1(nil) = nil in nat list
\forall n : nat2. \forall t : nat2 list. lpl(h :: t) = p1(h :: p(succ2(m))) in nat list

INDUCTION nat2ind:

\forall p : (nat2 -> u(1)). \forall n : nat2.
p(pair(zero)) \rightarrow
(p(pair(zero)) \rightarrow
(p(pair(zero)) \rightarrow
(p(pair(zero)) \rightarrow
\ldots
p(n))
INDUCTION nat2ind2:

\[ \forall p: (\text{nat2} \rightarrow \text{u}(1)). \forall n: \text{nat2}.
\]
\[ p(\text{pair}(\text{zero})) \rightarrow p(\text{pair}(\text{succ}(\text{zero}))) \rightarrow
\]
\[ (\forall m: \text{nat2}. p(m) \rightarrow p(\text{succ}(\text{succ}(m)))) \rightarrow p(n) \]

The "needs" file specifies that Ipl is a conversion function, and that the following functions are allowed to appear in existential witnesses: \text{succ}(-), \text{zero}, \text{succ2}(-), \text{pair}(-), \text{p1}(-), \text{p2}(-), \text{val}(-).

D.4.2 The synthesis of \text{mod2sumlist}

The synthesis conjecture used to illustrate this type change is:

\[ \forall l: \text{nat list} \exists s: \text{nat}. s = \text{summod2list}(l) \text{ in nat} \]

\text{summod2list} is defined by the following equations:

\[ \text{summod2list}(\text{nil}) = 0 \]
\[ \text{summod2list}([h::t]) = \text{mod2}(h) + \text{summod2list}(t) \]

The type change is initiated by the following motivating expression:

<table>
<thead>
<tr>
<th>Function</th>
<th>Argument type changes</th>
<th>Output type change</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{mod2}(x)</td>
<td>( x: \text{nat} \rightarrow x': \text{nat2}, )</td>
<td>( \text{nat} \rightarrow \text{nat2} )</td>
</tr>
</tbody>
</table>

The proof plan which is generated automatically by CLAM without search is as follows:

\[ \text{split implementation}(\text{nat}: \text{u}(1)) \text{ then} \]
\[ \text{split implementation}(\text{nat2}: \text{u}(1)) \text{ then} \]
\[ \text{typechange strat}(\)
\[ [] \Rightarrow \text{nat: list} \rightarrow \text{nat2: list} - \lambda x. \lambda y. \text{natp1list}(x, y) + [] \Rightarrow \]

252
D.4.3 The transformation of \texttt{mod2comp}

The specification before a type change is:

\[
\forall x, y : \text{nat} \exists s : \text{nat}. s = \text{mod2}(\text{natplus}(\text{mod2}(x), \text{mod2}(y))) \text{ in nat}
\]

The proof plan which is produced without search by \texttt{ClAM} is:

\begin{verbatim}
split.implementation(nat:u(1)) then
  split.implementation(nat2:u(1)) then
  typechange.strategy(
    \[
    x : \text{nat} \rightarrow .14504 : \text{nat2} - \lambda x. \text{lambday. natpl}(x,y) + [] =:>
    \text{nat} \rightarrow \text{nat2} - \lambda x. \text{natpl}(x,y) + []
    \]
    \[
    y : \text{nat} \rightarrow .14779 : \text{nat2} - \lambda x. \text{natpl}(x,y) + [] =:>
    \text{nat} \rightarrow \text{nat2} - \lambda x, y. \text{natpl}(x,y) + [] =:>
    \]
\end{verbatim}

253
\[
\text{nat} \rightarrow \text{nat2} \rightarrow \lambda x. \lambda y. \text{natpl}(x, y) + [1] =:>
\text{nat} \rightarrow \text{nat2} \rightarrow \lambda x. \lambda y. \text{natpl}(x, y) + [1].
\]

\[
...)
\[
\text{[elementary(intro(new[v4])then[identity,wfftacs]),}
\text{base.case([sym_eval([eval.def([2,1,2,2],[nat2.mod2thm2,eq(left)])),
\text{existential(v4:nat2,pair(val(n2(v3))]],eval.def([1,1],[nat2.p1eqn2,eq(left)])),
\text{elementary(intro(new[v3])then[identity,wfftacs])]),
\text{standard.form(standard.form,vw0:nat2.pl(w0) = pl(v0); in : nat) then
\text{step.case(ripple[wv0:nat2.pl(w0) = pl(v0); in : nat) then
\text{existential.subterm(v3:nat2,\text{natplus2(v8,v7)}) then
\text{elementary(intro(new[v3])then[identity,wfftacs])]),
\text{base.case([sym_eval([eval.def([2,1,2,2],[nat2.mod2thm2,eq(left)])),
\text{existential(v8:nat2,pair(val(n2(v1))))]],eval.def([1,1],[nat2.p1eqn2,eq(left)])),elementary(intro(new[v11]) then [identity,wfftacs])],
\text{elementary(intro(new[x])then[identity,wfftacs]),
\text{base.case([sym_eval([eval.def([2,1,2,2],[nat2.mod2thm2,eq(left)])),
\text{existential(v7:nat2,pair(val(n2(v2))))]],eval.def([1,1],[nat2.p1eqn2,eq(left)])),elementary(intro(new[v12]) then [identity,wfftacs])],
\text{elementary(intro(new[y])then[identity,wfftacs])})]
\]
\]
\]

D.5 The synthesis of a binary addition function

Here I list the complete proof plan for the binary addition synthesis.

The plan was produced using the following top-level query:

```
?- lib_load(mthd(induction/_)),goal(G),
applicable([],G,induction(Sx,[x:Tx]),[B1,S1]),
dplan(B1,BP,[]),
applicable(S1,induction(Sy,[y:Ty]),[B2,S2]),
dplan(B2,BP2,[]),
lib_delete(mthd(induction/_)),
dplan(S2,P,[]),
print_plan(induction(Sx,[x:Tx]) then [BP,induction(Sy,[y:Ty]) then [BP2,P]])
```

Note that the induction method is deleted from the method database after it has been used. This reduces search problems considerably.

Here is the proof plan:

254
induction([v0 :: v1], [x : bool list]) then
  base_case(sym.eval([eval_def([1, 2, 1, 2, 2], [btop1, equ(left)]), eval_def([2, 1, 2, 2], [plus1, equ(left)])])) then
  existential_subterm(z : bool list, y) then
    elementary(intro(new[y])); then[identity, wffacte]),
  induction([v3 :: v4], [y : bool list]) then
    existential_subterm(v2, z, v3, v4)) then
      base_case(sym.eval([eval_def([1, 2, 1, 2, 2], [btop1, equ(left)]),
        eval_def([2, 1, 2, 2], [plus3, equ(left)]), eval_def([2, 1, 2, 2], [btop2, equ(left)]),
        existential(z : bool list, v0 :: v1), eval_def([1, 1], [btop2, equ(left)])), elementary(identity))).
    elim_existential(ih, [elim(v2, z, v3, v4)]) then
    step_case(ripple(wave([1, 2, 1, 2, 2], [plus3, equ(left)])), [elim_existential(ih, [elim(v2, z, v8, v9)]) then
      [elim_existential(ih, [elim(v5, z, v6, v7)]) then
        standard_form(standard_form, Vv15: bool.
          Vv16: bool list.
          (val(v15) + (btop(v16) + btop(v16))) = (val(v15) + (btop(v16) + btop(v16))) in : pnat) then
        step_case(ripple(wave([2, 1], [plus2, equ(right)])), []]) then
        finitetypeelim(v8 : bool, boolxor) then
          [guess_existential(v15 : bool, true, Vv16: bool list.
            [guess_existential(v15 : bool, false, Vv16: bool list.
              [base_case(sym.eval([eval_def([1, 1, 2, 1, 2, 2], [btop, eval_def([1, 1, 2, 1, 2, 2], [val1, equ(left)]), eval_def([1, 2, 1, 2, 2], [plus2, equ(left)]), reduction([2, 2], cnc.s), eval_def([1, 1, 2, 2], [plus2, equ(left)]),
                existential_subterm(v16 : bool list, v10) then
                elementary(identity))),])])
        base_case(sym.eval([eval_def([1, 1, 2, 1, 2, 2], [val1, equ(left)]), eval_def([1, 2, 1, 2, 2], [plus2, equ(left)]), eval_def([1, 1, 2, 2], [plus2, equ(left)]), eval_def([1, 1, 2, 2], [plus2, equ(left)]),
          eval_def([2, 1, 2, 2], [plus1, equ(left)]), eval_def([1, 1, 2, 2], [plus1, equ(left)]))]) then
          standard_form(standard_form, Vv16: bool list.
            (val(v15) + (btop(v16) + btop(v16))) = (val(v15) + (btop(v16) + btop(v16))) in : pnat) then
            step_case(ripple(wave([2, 1], [plus2, equ(right)])), []) then
            finitetypeelim(v8 : bool, boolxor) then
              [guess_existential(v15 : bool, true, Vv16: bool list.
                [guess_existential(v15 : bool, false, Vv16: bool list.
                  [base_case(sym.eval([eval_def([1, 1, 2, 1, 2, 2], [btop, eval_def([1, 1, 2, 1, 2, 2], [val1, equ(left)]), eval_def([1, 2, 1, 2, 2], [plus2, equ(left)]), reduction([2, 2], cnc.s), eval_def([1, 1, 2, 2], [plus2, equ(left)]),
                    existential_subterm(v16 : bool list, v10) then
                    elementary(identity))),])])
        base_case(sym.eval([eval_def([1, 2, 1, 2, 2], [plus2, equ(left)]), eval_def([1, 1, 2, 2], [plus2, equ(left)]), eval_def([2, 1, 2, 2], [plus4, equ(left)]), eval_def([2, 1, 2, 2], [plus2, equ(left)]),
          eval_def([1, 1, 2, 2], [val1, equ(left)]), eval_def([1, 2, 1, 2, 2], [plus2, equ(left)]),
          eval_def([2, 1, 2, 2], [plus4, equ(left)]), eval_def([1, 1, 2, 2], [val1, equ(left)]), eval_def([1, 2, 1, 2, 2], [plus1, equ(left)]))]) the
          standard_form(standard_form, Vv16: bool list.
            (btop(v16) + btop(v16)) = (btop(v16) + btop(v16)) in : pnat) then
            step_case(ripple(wave([2, 1], [plus2, equ(right)]), [])) then
            255
D.6 Queue examples

In this section I list the queue examples I performed.

D.6.1 The queue ADT

The queue and clist abstract data types are as specified in [Darlington 80], except that all functions have been made total. Some new functions have been added to the queue abstract data types, which is therefore listed below.

ADT bool:Type

FUNCTIONS

true : bool
false : bool

EQUATIONS

true ≠ false in bool
∀ q : bool → u(1). q(false) → q(true) → ∀ b : bool. q(b)

ADT queue:Type

FUNCTIONS
emptyq : queue
addq : (queue —► pnat —► queue)
removeq : (queue —► queue)
frontq : (queue —► pnat)
isemptyq : (queue —► bool)
appendq : (queue —► queue —► queue)
listq : (pnatlist —► queue —► bool)
q.list : (queue —► pnatlist)
llistq : ((pnatlist)list —► (queue list) —► bool)
lq.llist : ((queue list) —► (pnatlist)list)

EQUATIONS
isemptyq(emptyq) = true in bool
Vq : queue Vi : pnat isemptyq(addq(q,i)) = false in bool
 VI : pnatlist Vq : queue lista(1,q) → l = q.list(q) in pnatlist
Vq : queue lasta(q.list(q)) = frontq(q) in pnat
q.list(emptyq) = nil in pnatlist
Vn : pnat Vq : queue alist(addq(q,n)) = append(q.list(q),n :: nil) in pnatlist
 VI : (pnatlist)list Vq : (queue list) llistq(l,q) → l = lq.llist(q) in (pnatlist)list
lq.llist(nil) = nil in (pnatlist)list
Vq : queue Vl : (queue list) alist(q :: l) = q.list(q) :: lq.list(l) in (pnatlist)list
frontq(emptyq) = 0 in pnat
Vq : queue Vi : pnat isemptyq(q) = true in bool → frontq(addq(q,i)) = i in pnat
Vq : queue Vi : pnat isemptyq(q) = false in bool → frontq(addq(q,i)) = frontq(q) in pnat
removeq(emptyq) = emptyq in queue
Vq : queue Vi : pnat isemptyq(q) = true in bool → removeq(addq(q,i)) = emptyq in queue
Vq : queue Vi : pnat isemptyq(q) = false in bool → removeq(addq(q,i)) = addq(removeq(q),i) in queue
Vq : queue appenda(q,emptyq) = q in queue
Vq : queue Vr : queue Vi : pnat appenda(q,addq(r,i)) = addq(appenda(q,r),i) in queue

D.6.2 The transformation of listlastq

The synthesis conjecture before the type change is:

\[ \forall l : (\text{pnat list}) \exists l : \text{pnat list}. l = \text{listlast}(l) \] in pnat list

The definition of listlastq is provided by the following equations:

\[
\text{listlast}(\text{nil}) = \text{nil}
\]
\[
\text{listlast}(h :: t) \Rightarrow \text{last}(h) :: \text{listlast}(t)
\]
The expression which motivates a pnat list to queue type change is:

<table>
<thead>
<tr>
<th>Function</th>
<th>Argument type changes</th>
<th>Output type change</th>
</tr>
</thead>
<tbody>
<tr>
<td>last(l)</td>
<td>( 1 : (\text{pnat list}) \text{list} \xrightarrow{\text{pq-llist}} \text{queue list} )</td>
<td>( \text{(pnat list)} \xrightarrow{\text{pq-llist}} \text{(pnat list)} )</td>
</tr>
</tbody>
</table>

The proof plan which is automatically produced by CLAM is:

```plaintext
split_implementation(bool : u(1)) then
split_implementation(queue : u(1)) then
typechange_strat(\[] => \text{pnat list list} \rightarrow \text{queue list} - \lambda x . y . llist(x, y) + \[] = =>
  \text{pnat list} \rightarrow \text{pnat list} - \lambda x . y . x = y \text{ in } \text{pnat list} + \[],
...)
[induction([x0 :: v1], [v3 : \text{queue list}]) then
  [base_case([sym_eval([eval_def([1, 2, 1, 2, 2], [queue, \text{llisteqn1, equ(left)}], eq(last1), eq(left)]),
    eval_def([2, 1, 2, 2], [listlast1, equ(left)]),
    existential([v2 : \text{pnat list}, nil]), elementary(identity))],
    elim_existential(ih, [elim(v4, v2, v5, v6)]) then
      step_case(ripple(wave([1, 2, 1, 2, 2], [queue, \text{llisteqn1, equ(left)]], [])) then
        [wave([2, 1, 2, 2], [listlast2, equ(left)], [])] then
      [fertilize(weak, fertilize([weak, fertilize(right, in, [2, 2, 1, 2], \text{v6}]])) then
        ripple_and_cancel([... ]))]) then
    elim_existential(ih, [elim(v4, v2, v7, v8)]) then
    base_case(sym_eval([existential([v2 : \text{pnat list}, frontq(x0 :: v5)]), elementary(identity)])
  ],
  elementary(intro(new[ll]) then [identity, wftacs])
].
```

258
Appendix E

Descriptions of the new methods

E.1 Introduction

Here I describe the new methods necessary to plan the proofs in this thesis.

E.2 standard_form(standard_form,Term)

This method is designed specifically for proofs using conversion functions. A function $f : t' \rightarrow t$ is noted as a conversion function by applying the predicate to $\text{add} \_ \text{conversion}(f(\_):t' \rightarrow t)$. If $p$ is the conversion function in question (for example $\text{nat}$), then in the simplest case the method difference matches the current sequent against a goal of the form:

$$\forall x \rho(x) = \rho(x) \quad (E.1)$$

There are two enhancements intended to get the closest match between the current sequent and the goal we are trying to ripple towards:

1. It can also difference match against a variant of (E.1) in which both sides have been expanded using a wave rule for $\rho$, e.g. (6.17).
2. Using one-step ripples, it will expand potential wave fronts in the goal.

All possible applications of these two enhancements are made, and the resulting difference matches are ranked according to a simple measure, where $|T|$ is defined to be the number of nodes in the term tree of $T$:

$$\mu(AnnTerm) = \frac{|\text{skeleton}(AnnTerm)|}{|\text{erasure}(AnnTerm)|}$$

The match with the highest measure is selected, and all the others are discarded. There is no backtracking to alternative matches.

This method is used in the proof after a case split to move the resulting $s(\cdot)$ wave front inwards, and also to expand the potential wave front on the LHS.

$$\exists z : \text{bool list.} \text{nat}(z) = s((\text{val}(v3) + (\text{nat}(v8) + \text{nat}(v8)))) \text{ in } \text{pnat}$$

**SELECTED METHOD at depth 6:**

$$\forall v11 : \text{bool.} \forall v12 : \text{bool list.} \text{val}(v11) + (\text{nat}(v12) + \text{nat}(v12)) = \text{val}(v11) + (\text{nat}(v12) + \text{nat}(v12)) \text{ in } \text{pnat}$$

$$\exists v11 : \text{bool.} \exists v12 : \text{bool list.} \text{val}(v11) + (\text{nat}(v12) + \text{nat}(v12)) = s((\text{val}(v3) + (\text{nat}(v8) + \text{nat}(v8)))) \text{ in } \text{pnat}$$

**E.3 guess_existential/3**

This method tries to prove a sequent which contains an existentially quantified variable by trying all possible substitutions of values for this variable. This only works, of course, for variables which are of "small" types, where by "small" I mean that the elements of the type could be enumerated on one hand. Currently this is only used for the bool type.

We are quite likely to guess a value for the existential variable which produces a non-theorem, which often leads to a nonterminating search for a proof plan, so
the search for a subplan only proceeds to a certain (6 at the moment) maximum depth.

This is used in the proof to find the correct values for the least significant digit when adding two binary numbers.

\[ \exists \{v\} : \text{bool}. \exists v2 : \text{bool list} \]

\[ \text{val}(v) + (\text{nat}(v2) + \text{nat}(v2)) = (\text{val}(\text{false}) + s((\text{nat}(v3) + \text{nat}(v8)))) \text{ in pnat} \]

Two subproofs: one \( v1 = \text{false} \) fails, the other \( v1 = \text{true} \) succeeds.

\[ \text{ITERMINATING METHOD at depth 9: guess_existential(v1 : \{bool\}, \{true\}, base_case[[ ... ]]) then [existential_subterm(v12 : \{bool\} list, v8) then [elementary(...)]]] \]

E.4 finitetypeelim/1

This method performs a case split on the value of a universally quantified variable of a "small" type (see above). If the variable which is the subject of the case split occurs in a subexpression \( f(\text{variable}) \) in the goal, then there must exist an equation to symbolically evaluate this subexpression.

This is used in the proof to case split on the values of binary digits.

\[ \exists z : \text{bool list}. \text{nat}(z) = ((\text{val}(v0) + \text{val}(v3)) + (\text{nat}(v3) + \text{nat}(v8))) \text{ in pnat} \]

\[ \text{SELECTED METHOD at depth 4: finitetypeelim(v0 : bool)} \]

\[ \exists z : \text{bool list}. \text{nat}(z) = ((\text{val}(\text{false}) + \text{val}(v3)) + (\text{nat}(v3) + \text{nat}(v8))) \text{ in pnat} \]
E.5 elim_existential/2

elim_existential/2 is used in conjunction with enhanced rippling and fertilisation submethods to allow rippling and fertilisation under existential quantifiers.

The method applies when there is an induction hypothesis which contains an existentially quantified proposition:

$$\text{IH} : \exists \text{Var} : \text{EType}. \text{Prop}$$

If the method succeeds, then the goal is unchanged, and two new hypotheses are added:

1. NWit : Prop[NVar/EVar]
2. NVar : EType

Application of this methods allows induction and fertilisation on existentially quantified goals.

E.6 split_implementation/1

Preconditions:

The goal G in the input sequent H ==> G is quantified with an ADT. An ADT typename with n_f functions f_i : t_i, n_e equations eq_i and n_i induction schemas ind_k is represented as is (4.12) as an existential type:

typename : u(1)#
    f_1 : t_1#
    ...
    f_n : t_n, #
Each of the $t_i$ can be any valid type, e.g. $\text{pnat} \rightarrow \text{pnat} \rightarrow \text{pnat}$. The $eq_i$ are generally universally quantified. Each induction schema must be of the following form:

\[
\text{indname} : (p : (\text{adtname} \Rightarrow u(1)) \Rightarrow x : \text{adtname} \Rightarrow \\
\quad \text{base}_1 \Rightarrow \ldots \Rightarrow \text{base}_{n_p} \Rightarrow \\
\quad \text{step}_1 \Rightarrow \ldots \Rightarrow \text{step}_{n_s} \Rightarrow \\
\quad p \ \text{of} \ x)
\]

Each of the $n_b$ base cases is of the form:

\[
x_1 : t_1^b \Rightarrow \ldots \Rightarrow x_{n_q} : t_{n_q}^b \Rightarrow \\
(p \ \text{of} \ x_1 \ \text{of} \ \ldots \ \text{of} \ x_n \ \text{of} \ c_1 \ \text{of} \ \ldots \ \text{of} \ c_{n_c})
\]

There may be zero or more universally quantified variables. None of the $t_{n_a}^b$ types may be $\text{adtname}$. The $c_i$ are constants.

Each of the $n_s$ step cases is of the form:

\[
y_1 : t_1^s \Rightarrow \ldots \Rightarrow y_{n_s} : t_{n_s}^s \Rightarrow \\
\quad p \ \text{of} \ y_1 \Rightarrow \ldots \Rightarrow p \ \text{of} \ y_{n_s} \Rightarrow \\
\quad p \ \text{of} \ c\text{term}
\]
c-term is a compound term which may contain any of the $y_i$.

**Postconditions:**

The types of the functions which have been declared in the function part of the ADT are recorded. The equation part of the ADT is broken into separate equations, each of which is stored internally as an (unproved) theorem in the Oyster database. The equations are parsed as though they have been loaded into CLAM from an external file, adding the following records:

1. wave rules,

2. complementary rewrite sets,

3. complementary wave rule sets,

4. equations defining a function (func_defeqn records).

The induction schemas are parsed, and records added to the internal database:

```
induction_schema( Indname, Type, Bases, Steps )
```

Indname is the name of the schema used in the ADT declaration; it must contain the substring "ind". Type is the name of the ADT. Bases is a list of the base cases, and Steps a list of the step cases. These are in the same form as they are in the ADT declaration, except individual base and step cases are collected into lists instead of being separated by $=>$.

The induction schema records are used by a new clause of the schema/5 predicate.

**Outputs:**

The single output is $H \Longrightarrow \text{body_of_theorem}$.

**Tactic:**
The tactic must supply a concrete implementation for the ADT: concrete types for the carrier type, and concrete functions for each of the functions declared in the ADT. It must prove that the implementation is correct. This entails proving each of the equations, and proving the induction schemas. These proofs must be applied both to the theorem being proved, and to each of the equations which has been created during the parsing process.

E.7 existential_subterm/2, existential/2

existential_subterm/2 tries to prove a goal of the form \( \exists x . L = R \) by substituting subterms of \( L \) or subterms of \( R \) for \( x \) in \( L = R \). This solves quite a lot of goals without recourse to the existential method. Without any further restrictions, a goal of the form \( \exists z . z = \text{term} \) could be trivially solved by supplying the witness term for \( z \). Generally, we would like to apply further methods to break this goal down into simpler subgoals and so synthesise a compound expression for \( \text{term} \), so only certain terms are allowed to be used as existential witnesses in this method. These terms are specified by executing the predicate addcanonical/1. The argument is a list of functions which are allowable as existential witnesses. Such a declaration is part of the meta-theory, and appears in the "needs" file.

The same restriction has been made in the existential/2 submethod.

An extra clause has been added to wave/3 to allow rippling after difference matching. The terms which are returned after application of this clause of wave/3 do not now have their annotations stripped. Ripples consisting entirely of applications of the unblock submethods are now disallowed, because difference matching produces maximally annotated terms which can be rewritten by unblocking to terms with the same erasure, which can subsequently be re-difference matched and so on.
E.8 The typechange_strat/2 method

This method is described in §8.2. After a successful application of method typechange_strat(Transformation, SubPlan), Transformation is instantiated to a specification of the type change which has been chosen, in the form [...argument transformations...] =:> output transformation. The argument transformations may contain nested type changes. SubPlan is instantiated to the list of submethod applications which simplify_typechange.goals/2 used to the simplify the initial subgoals of the type change.

E.9 The simplify_typechange.goals/2 submethod

The simplify_typechange.goals/1 submethod applies one of two iterators, whose submethods are described below. If the type change is an abstraction, then it applies:

iterator(submethod, simplify_abstraction, submethods,
            [rewrite_conversion(_,_),
             implies_independence(_),
             relation_to_function(_,_),
             embedded_existential(_)]).

The role of these is outlined in §7.9.3.

If the type change is an implementation, then it applies:

iterator(submethod, simplify_implementation, submethods,
            [prove_output_surjectivity(_),
             implies_independence(_),
             relation_to_function(_,_)].

266
The role of these is to convert the conversion relations in the subgoals into conversion functions. For example, (E.2) is transformed in the following steps:

1. `relation_to_function/2` rewrites it to (E.3).
2. This is rewritten by `relation_to_function/2` to (E.3).
3. This is rewritten by `substitute_using_cfeq/1` to (E.5).
4. This is rewritten by `substitute_using_cfeq/1` to (E.6), which completes the translation to functional form.

\[
\forall v3 : \text{dlist}. \exists v4 : \text{dlist}. \forall v6 : \text{pnat list}. \text{listdlist}(v6, v3) \rightarrow (E.2) \\
\exists v5 : \text{pnat list}. \text{listdlist}(v5, v4) \land v5 = \text{rev}(v6) \text{ in } \text{pnat list} \\
\forall v3 : \text{dlist}. \exists v4 : \text{dlist}. \forall v6 : \text{pnat list}. v6 = \text{dlist_list}(v3) \text{ in } \text{pnat list} \rightarrow (E.3) \\
\exists v5 : \text{pnat list}. \text{listdlist}(v5, v4) \land v5 = \text{rev}(v6) \text{ in } \text{pnat list} \\
\forall v3 : \text{dlist}. \exists v4 : \text{dlist}. \forall v6 : \text{pnat list}. v6 = \text{dlist_list}(v3) \text{ in } \text{pnat list} \rightarrow (E.4) \\
\exists v5 : \text{pnat list}. v5 = \text{dlist_list}(v4) \text{ in } \text{pnat list} \land \\
v5 = \text{rev}(v6) \text{ in } \text{pnat list} \\
\forall v3 : \text{dlist}. \exists v4 : \text{dlist}. \exists v5 : \text{pnat list}. v5 = \text{dlist_list}(v4) \text{ in } \text{pnat list} \land (E.5) \\
v5 = \text{rev}(\text{dlist_list}(v3)) \text{ in } \text{pnat list} \\
\forall v3 : \text{dlist}. \exists v4 : \text{dlist}. \text{dlist_list}(v4) = \text{rev}(\text{dlist_list}(v3)) \text{ in } \text{pnat list} \quad (E.6)
\]

E.9.1 The `implies_independence/1` submethod

`implies_independence/1` implements a rule:
\[ \vdash q \land \text{x not free in } q \]
\[ \vdash \forall x \cdot p(x) \rightarrow q \]

When applied to (E.7) it gives (E.8).

\[
\forall x': t' \cdot \exists z': t'_0 . \forall x: t. (p(x, x') \rightarrow \rho_0(f(x'), z'))
\]

(E.7)

\[
\forall x': t' \cdot \exists z': t'_0 . \rho_0(f(x'), z')
\]

(E.8)

If \( f' \) is already defined, and \( \rho_0 \) is the identity, then this subgoal is trivial. Otherwise some more proof planning may be required.

**E.9.2 The relation_to_function/2 submethod**

The relation_to_function/2 submethod attempts to rewrite the conversion relations in the goal using equations:

\[
\rho(x, y) \rightarrow y = \sigma(x)
\]

(E.9)

This may allow us to later perform induction and functional rippling.

**E.9.3 The embedded_existential/1 submethod**

The embedded_existential/1 submethod replaces any existential variables in the goal with a metavariable. Its preconditions are much weaker than those of the existential/2 submethod.

**E.9.4 prove_output_surjectivity/1**

This method proves goals of the form:
\[ \vdash \forall x : t. \exists x' : t'. x = \rho(x') \in t \]

Such goals must be proved to ensure that a type change is an implementation.

**E.9.5 substitute using cfeq/1**

The synthesis subgoals which are produced by the typechange/2 method are initially contain conversion relations, not conversion functions. Replacing conversion relations with conversion functions produces goals of the form:

\[ \vdash x = p(y) \rightarrow p(x) \]

This is rewritten to:

\[ \vdash p(\rho(y)) \]

This is done by applying the following rule of inference:

\[
\begin{align*}
\forall b. p(t[b])
\end{align*}
\]

\[
\forall a, b. a = t[b] \rightarrow p(a)
\]
Appendix F

Code for the main methods and predicates

F.1 The typechange code

This section lists the main code for the derivation of type changes. Some of the subsidiary predicates are omitted.

/*
   typechange predicate

   Julian Richardson, 96/09/18 typechange4.pl

   Perform an abstraction or implementation transformation.

*/

% 
% Add a maximum search depth. Currently 9.
% 
% typechange(Kind, Force, Spec, Trn) :- 
%     typechange(Kind, Force, Spec, Trn, 9).

not_too_deep(D, D1) :- 
    D1 is D-1, 
    D1>0.
% TypeChange(AType, ?Force, +H==>|G, -Input_xforms => Output_xform)
% Determine a type change in a goal +H==>|G, Force in a boolean which
% if true will permit the application of -allow- rules.
%
% Typechange in motivating and allowing expressions.
% typechange(AType, Force, +H==>|Z:TO#SpecZX, Input_xforms => Output_xform, D) :-
% not_too_deep(D, D1),
% fn_equality(2:TO#SpecZX, Z = FX in TO),
% type_of(H, FX, _F: Inputs => _Output:OType),
% (motiv(ATYPE, +H==>|Z:OType#Z=FX in OT, Input_xforms, Output_xform) ;
% Force = true,
% allow(ATYPE, +H==>|Z:OType#Z=FX in OT, Input_xforms, Output_xform)).
%
% Typechange in simple rewrites.
%
% typechange(ATYPE, Force, +H==>|Z:TO#SpecZX, Input_xforms => Output_xform, D) :-
% not_too_deep(D, D1),
% fn_equality(2:TO#SpecZX, Z = FX in TO),
% func_defeqn(FX, equ(left), EqName: FX := GX),
% typechange(ATYPE, Force, +H==>|Z:TO#Z = GX in OT,
% Input_xforms => Output_xform + Operators, D1).
%
% Typechange in function composition: ensure that the input and output
% transformations match. Since a typechange of the argument may be only
% partially determined (i.e. contain type metavariables, we ensure that
% the output transformation must be well-typed, i.e. be of type TO =>
% something.
%
% typechange(ATYPE, Force, +H==>|Z:TO#SpecZX, Input_xforms => (TO => TI - ORho + Operators), D) :-
% not_too_deep(D, D1),
% fn_equality(2:TO#SpecZX, Z = FX in TO),
% type_of(H, FX, F:Inputs => _Output:TO),
% generalise_args(N=>Inputs, H=>Vars, [], []),
% NFGX =.. [F | Vars],
% NFGX \= FGI,
Could do these two goals either way round.

typechanges(ATYPE, true, Z, \(\text{H} = \text{Inputs}, \text{Arg}_xforms, \text{Input}_xforms, \ldots, \text{Dl}\)),
(typechange(ATYPE, true, \(\text{H} = \text{Z} \Rightarrow \text{TO} \Rightarrow \text{ZFOI in TO}\)),
Arg_xforms := (TO \Rightarrow T1 - ORho + Ops), Dl);
identity_typechange(ATYPE, \(\text{H} = \text{Z} \Rightarrow \text{TO} \Rightarrow \text{ZFOI in TO}\)),
Arg_xforms := (TO \Rightarrow T1 - ORho + Ops)).

Typechange through induction on a parameterised type, e.g. \text{pnat list}.
Currently the induction variable's transformation is marked in the output transformation specification, but no other transformations are allowed.
Possibly should allow type changes on the noninductive arguments of the function too, i.e. more than one input transformation.

typechange(ATYPE, Force, \(\text{H} = \text{Z} \Rightarrow \text{OType} \Rightarrow \text{SpecZX, Input}_\text{trns} := \Rightarrow \text{OType} \Rightarrow \text{OTypel - ORho + [\ldots \ldots, \text{Dl}\) not_too_deep(D, Dl),
fn_equality(\(\text{Z} \Rightarrow \text{OType} \Rightarrow \text{SpecZX, Z = FX in OTyoes}\)),
type_of(H, FX, \_F; \_Inputs \Rightarrow \_Output:OType),
applicable(H \Rightarrow \text{Z} \Rightarrow \text{OType} \Rightarrow \text{FX in OTyoes},
induction([\text{HeadTail}], [\text{VarTypes}]), \ldots, \text{G1}),
VarTypes = (\text{V:TT1}),
TT1 = [T, T1], \_ TT1 = T(T1), e.g. (\text{pnat list}) = \text{list(pnat)}
HeadTail = \_ [\text{Constr, HT1, \_TTT1}], e.g. (h::t) = \_ [h::t, t:TT1],
induction_stepcase(01, GStep),
applicable(GStep, \text{glim_existential(\ldots), \ldots, [00]}),
applicable(G0, step_case(SubPlan), \ldots, \text{[H2 \Rightarrow Z2:T2\#Z2=0G3 in T2]}),
strip_meta_annotations(G3, G4),
determine_fertilisation_varname(H2 \Rightarrow Z2:T2\#Z2=0G3 in T2, SubPlan,
FertVarName),
typechange(ATYPE, true, H2 \Rightarrow Z2:T2\#Z2=0G4 in T2, Xforms, Dl),
find_component_xform((\text{RT1:T1} \Rightarrow \_ : T12 - KRho + [\ldots], Xforms),
find_component_xform((\text{FertVarName:OType} \Rightarrow \_ : \text{OType1 - ORho +[\ldots]_Xforms),
TT12 = [T, T12],
extend_conversion(KRho: [T1, T12], [T, T12], KRhoBar:[TT1, TT12]),
recreate_trn_tree(H, FX, \_:TT12 \Rightarrow \_ : KRhoBar + [\ldots],
Input_trns := \_ \Rightarrow \_ + [\ldots]).
Typechange through induction on an unparameterised type, e.g. \( \text{pnat} \).

The input transformation is not constrained, and so is left as a meta-variable. May be instantiated later.

Step case wave rule \( f(c(x)) \Rightarrow g(f(x)) \).

Typechange \( g : t \rightarrow t' \Rightarrow g' : t' \rightarrow t' \).

Typechange \( f : t \rightarrow t \Rightarrow f' : t' \rightarrow t' \).

Typechange \((\text{AType}, \text{Force}, H = \Rightarrow Z : \text{OType} \# \text{Spec} Z X, \text{Input_trns} \Rightarrow) \):

\[
\begin{align*}
\text{OType} &\Rightarrow \text{OType}_1 - \text{ORho} + [], D) :- \\
\text{not_too_deep}(D, D1), \\
\text{fn.equality}(Z : \text{OType} \# \text{Spec} Z X, Z \Rightarrow FX \text{ in OType}), \\
\text{type.of}(N, FX, ...F; \text{Inputs} \Rightarrow \text{Output} : \text{OType}), \\
\text{applicable}(N = \Rightarrow Z : \text{OType} \# Z = FX \text{ in OType}, \\
\text{induction}([\text{ConstrTerm}], [\text{VarTypes}], _, G1), \\
\text{VarTypes} = (V : \text{TT} 1), \\
\text{induction_stepcase}(G1, GStep), \\
\text{applicable}(GStep, \text{elim.existential}(...), [GG]), \\
\text{applicable}(GG, \text{step.case}(	ext{SubPlan}),..., [H2 = \Rightarrow Z2 : T2 \# Z2 = G3 \text{ in T2}], \\
\text{strip.meta.annotations}(G3, G4), \\
\text{determine.fertilisation.varname}(H2 = \Rightarrow Z2 : T2 \# Z2 = G3 \text{ in T2}, \text{SubPlan}, \\
\text{FertVarName}), \\
\text{typechange}(\text{AType}, \text{true}, H2 = \Rightarrow Z2 : T2 \# Z2 = G4 \text{ in T2}, \text{Xforms}, D1), \\
\text{find.component.xform}((\text{FertVarName} = \text{OType} \Rightarrow \text{OType}1 - \text{ORho} + []), \text{Xforms}),
\end{align*}
\]

% Only allow the identity transformation if we are forcing.

\[
(\text{Force} = \text{true} ; \\
\text{ORho} \setminus \lambda(X, \lambda(Y, X=Y \text{ in OType})), \\
\text{ORho} \setminus \lambda(X, \lambda(Y, Y=X \text{ in OType}) \\
) ,
\]

The output type of the composite function \( g \) must be changed in exactly the same way as its input type.

\[
\text{Xforms} = (_ = \Rightarrow \text{OType} \Rightarrow \text{OType1} - _ + _), \\
\text{recreate_trn_tree}(N, FX, V : \text{TT} 1 = \Rightarrow _ . \text{TT1} 2 = _ \text{Rho} + []), \\
\text{Input_trns} = (_ = \Rightarrow _ = _ + []). 
\]
typechanges(+AType, Force, +Z, +H->GoalList, +Arg.xforms, -Xforms)

Apply a typechange to a list of goals, (which are actually the arguments of a function f in an expression exists x. x=f(arguments)). Z is simply the name of the existentially quantified variable, and so can safely be used as the output for each of the argument transformations.

Arg.xforms is a list of the transformations which were made when the elements of GoalList were generalised to variables and type changed. These are then used as the output transformations for the expressions in GoalList.


typechanges(AType, true, Z, H->[Var:Type | GoalList],
[Var:Type->_.T2 - Rho + [] | Arg.xforms],
[Var:Type->_.T2 - Rho + [] | Xforms],
Type->T2 - Rho + [], D):-
not_too_deep(D, D1),
\+compound(Var),
hyp(Var:Type, H),
typechanges(AType, true, Z, H->GoalList, Arg.xforms, Xforms,
Type->T2 - Rho + [], D1).

typechanges(AType, Force, Z, H->[Goal:Type | GoalList],
[_.:T1 ->_.T2 - Rho + [] | Arg.xforms],
[Xform ->(T1 -> T2 - Rho + Opers) | Xforms],
T1->T2 - Rho + [], D):-
not_too_deep(D, D1),
typechange(AType, true, H->Z:Type|Z=Goal in Type,
Xform ->(T1 -> T2 - Rho + Opers),D1),
typechanges(AType, Force, Z, H->GoalList, Arg.xforms, Xforms,
T1->T2 - Rho + [], D1).

typechanges(AType, Force, Z, H->[Goal:Type | GoalList],
[_.:T1 ->_.T2 - Rho + [] | Arg.xforms],
[Xform ->(T1 -> T2 - Rho + Opers) | Xforms],
_.:T, D):-
not_too_deep(D, D1),
typechange(AType, true, H->Z:Type|Z=Goal in Type,
% Motivate a list -> queue transformation by the appearance of last(L) in % the program.
% motiv(implementation, H===>Z:pnat#Z=last(L) in pnat,
%       [L:pnat list => _Var:queue - lambda(x,lambda(y,listq(x,y)))+[]],
%       pnat => pnat - lambda(x,lambda(y,x=y in pnat)) + []):-
%       decl(L:pnat list, H).

% Motivate a nat -> nat2 transformation by the appearance of mod2(x) in % the program.
% motiv(implementation, H===>Z:nat#Z=mod2(Expr) in nat,
%       [Expr:nat => _Var:nat2 - lambda(x,lambda(y,natpl(x,y)))+[]],
%       nat===>nat2-lambda(x,lambda(y,natpl(x,y)))+[]):-
%       decl(Expr:nat, H).

% Motivate a pnat -> binary transformation by the appearance of parity(x) in % the program.
% motiv(implementation, H===>Z:pnat#Z=parity(Expr) in pnat,
%       [Expr:pnat => _Var:{bool} list - lambda(x,lambda(y,natbin(x,y)))+[]],
%       pnat==>{bool} list - lambda(x,lambda(y,natbin(x,y)))+[]):-
%       decl(Expr:pnat, H).

% Motivate a list -> difference list transformation by the appearance of % -append- in the program.
% motiv(implementation, H===>Z:pnat list#Z = app(El, E2) in pnat list,
%       [El:pnat list => _Var: dlist - lambda(x,lambda(y,listdlist(x,y)))+[]],
%       E2:pnat list => _Var: dlist - lambda(x,lambda(y,listdlist(x,y)))+[]],
%       (pnat list=> dlist - lambda(x,lambda(y,listdlist(x,y)))+[])):-
%       decl(El:pnat list, H),
%       decl(E2:pnat list, H).

% A list -> list+length transformation, motivated by the appearance of
% -length- in the program.

% motiv(implementation, 
  H ==> _Z:pnat\[Z=length(L)\] in pnat,
  [ L:pnat list ==> _lenlist - lambda(x, lambda(y, llist(x,y))) + []],
  pnat ==> pnat = lambda(x, lambda(y, x in pnat)) + []):-
  decl(L:pnat list, H).

% % First abstraction: lists -> min/max
% % Need to modify code to determine and propagate correctness conditions.
% motiv(abstraction,
  _N ==> _Z:(pnat\[unary\]\[Z - minof(Expr)\] in (pnat\[unary\]),
  [Expr:pnat list ==> _Var: minmax - lambda(x, lambda(y, mminmaxlist(x,y))) + []],
  (pnat\[unary\]) ==> (pnat\[unary\]) - lambda(x, lambda(y, x = y in (pnat\[unary\]))) + []).

% motiv(abstraction,
  _N ==> _Z:(pnat\[unary\]\[Z - maxof(Expr)\] in (pnat\[unary\]),
  [Expr:pnat list ==> _Var: minmax - lambda(x, lambda(y, mminmaxlist(x,y))) + []],
  (pnat\[unary\]) ==> (pnat\[unary\]) - lambda(x, lambda(y, x = y in (pnat\[unary\]))) + []).

% % Allow a succ -> succ2 translation in the nat -> nat2 transformation. This
% % is necessary to allow the trn of natplus, which boils down to trn of
% % succ(z:nat):nat -> succ2(x':nat2):nat2
% % allow(implementation, _N ==> _Z:nat\[Z = succ(Expr)\] in nat,
%  [Expr: nat => _Var:nat2 - lambda(x, lambda(y, natpl(x,y))) + []],
  nat => nat2= lambda(x, lambda(y, natpl(x,y))) + []).

% % Allow [h] to be translated from list to dlist.
% motiv(implementation, H ==> _Z:pnat list\[Z = X::nil\] in pnat list,
  [(X::nil):pnat list => _dlist - lambda(x, lambda(y, listdlist(x,y))) + []],
  (pnat list => dlist = lambda(x, lambda(y, listdlist(x,y))) + []):-
  decl(X:pnat, H).

% % Allow the null type change on variable under some circumstances.
allow(implementation, H«=>Z:T= V in T, 
[V:T => V:T - lambda(x,lambda(y,x=y in T)) + []],
T => T - lambda(x,lambda(y,x=y in T)) + []):=
decl(V:T, H).

allow(abstraction,
H«=>Z:T= V in T, [V:T => V:T - lambda(x,lambda(y,x=y in T)) + []],
T => T - lambda(x,lambda(y,x=y in T)) + []):= decl(V:T, H).

% Allow identity transformations in the outer function of a Fcomp rule
% identity_typechange(AType, H«=>Z:TO«=FVs in TO, Input_identity_xforms :=
   (TO => TO - Identity + [])) :=
type_of(H, FVs, _F:VTs => _:TO),
map_list(VTs, (V:T) := (V:T => V:T - Identity + []),
   identity(AType, T, Identity),
   Input_identity_xforms).

F.2 The main methods

This section lists the code for the main methods, that is for the typechange_strat/2 method and for the standard_form/2 method.

F.2.1 typechange_strat/2

% Package a change of type with some rewriting:
% 1) Make a type change, then
% 2) Try simplify the resulting goals.
% method(typechange_strat(Ifoms,TheSubPlan),
H«=> G_ann,
[
   strip_meta_annualions(G_ann,G),
   applicable_submethod(H«=>G, typechange(Kind, Ifoms), _, SubGoals),
   map_list(SubGoals,
(H1 => G1) => (SubPlan - $SubGoal),
applicable_submethod(H1 => G1,
simplify_typechange_goals(Kind, SubPlan), _, [$SubGoal]),
PlansGoals),
zip(PlansGoals, TheSubPlan, $SubGoals)
),
];
$SubGoals,
typechange_strat($forms, TheSubPlan)).

F.2.2 standard_form/2

/*
Julian Richardson, 14 September 1995

standard_form/2

Difference match the goal with a standard form which is an equality built from conversion functions which are present in the goal.

Try one-step existential ripples and matching with expanded versions of the standard form in order to get the best possible match.

*/

method(standard_form(standard_form, Sform2),
  H => G_ann,
  [
  ]
  /*
  Only apply to unannotated terms.
  This stops ripples after induction
  and difference unification from interfering with each other.
  */
  strip_meta_annotations(G_ann, Gq),
  Gq = G_ann,
  matrix(Gq/FFs, G, Gq),
  /*
Now try all possible one-step existential ripples and matching with expanded versions of the standard form. Score them and pick the best.

```haskell
/*
 bagof(TM, Pos'GPon'H1'H2'Sform2'MetaG1'DmOutput'Measure'Lemma1'
    Arg1'Arg2'Arg3'Arg4'Arg5'Arg6'Arg7'HV'GR'(';
  /
  Try existential ripples on this standard form. This allows us to obtain a fuller match with the goal, i.e. more of the goal is in the skeleton. This helps with wave rule speculation, amongst other things.

  /*
  (HW = H, GR = G :
    applicable_submethod(H => G,
      existential_wave(GPos, Arg4, Arg5), Arg6, [HW => GR])),
  */

  Find a standard form to ripple towards. This is a rippled version of 
  \forall x: t. \rho(x) = \rho(x) in t

  /*
  standard_form_target(H => G, H2 => Sform2),
  */

  Now difference match the LHS and the RHS

  /*
  applicable_submethod(HW => GR, dm(standard_form, H2 => Sform2),
    Arg7, [DmOutput]),
  */

  Measure the difference

  /*
  measure_annseq(DmOutput, Measure),
  TM = (Measure - (H2 => Sform2) - DmOutput)),
  Measured_Seq),
  */

  Pick the best match.

  /*
  sort(Measured_Seq, SList),
  forall [(S \ SList): (WS => SM1 - (===> SSL1) - (===> SSG1))
    tracing(standard_form: NSM1-SSL1-SSG1)),
  reverse(SList, RList),
  member(_, Score - (===> Sform2) - BestSeq, RList),
  */

  Select best match to which method step_case/1 is applicable.

  /*
  BestSeq = (BH => BG),
  applicable([standard_form: [standard_form,Sform2] | BH] => BG),
  */
```
step_case(SHE), \[ \text{unblock}(\_,\_,\_)) \],

\+/+(SHE = unblock(\_,\_,\_))},

/*
The cut means we do not backtrack to alternative difference matches.
*/

matrix(GqUVs, B0, B0q), !.

/*
Add a marker to the hypotheses.
*/
[A|B] = \{[standard_form:[standard_form,Sform2] | SH] = \> B0q]

[]

[\[A|B\]]

standard_form(standard_form,Sform2)).
Appendix G

Bibliography


Satoru Kawai. A proof method with program and data structure transformation. Report TR 81-03, Department of Information Science, Faculty of Science, University of Tokyo, 1981.


