Automatic Inductive Theorem Proving and Program Construction Methods Using Program Transformation

Md. Humayun Kabir, M.Sc. in Computer Science

A thesis presented in fulfillment of the requirements
for the degree of Doctor of Philosophy (Ph.D.)
to the

Dublin City University
Faculty of Engineering and Computing, School of Computing

Supervisor: Dr. Geoff Hamilton

September 2007

© Md. Humayun Kabir 2007

Declaration

I hereby certify that this material, which I now submit for assessment on the programme of study leading to the award of the degree of Doctor of Philosophy (Ph.D.) is entirely my own work and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.

Signed: H.Kabir

(Md. Humayun Kabir)

Student ID No. 52171221

Date: 20/09/2007

Acknowledgements

I would like to express my thank to several people and organisations without whom this thesis would not exist.

First, and foremost, I would like to extend my sincerest gratitude and respect to my supervisor Dr. Geoff Hamilton for his supervision and encouragement during the period of my doctoral research without whom this work would never have seen the light of day. I would also like to extend my sincerest gratitude and respect to Dr. David Gray for many valuable discussions that I had with him during my second year of study. He was my acting supervisor during my second year of study when Geoff was on sabbatical. I would like to thank all of my colleagues who helped me directly or indirectly during the course of this work. I would also like to thank my employer for granting me the necessary higher study leave to complete my doctoral research. Finally, the School of Computing must also be thanked for funding the work of this thesis and providing me with a nice working atmosphere.

Md. Humayun Kabir

Contents

Declaration ii

Acknowledgements iii

Abstract viii

1 Introduction 1

1.1 Motivation 1

1.2 Aims of Thesis 3

1.3 Program Transformation 5

1.4 Inductive Theorem Proving 6

1.4.1 Mathematical Induction 6

1.4.2 Limitations of Inductive Inference 10

1.5 Program Synthesis 11

1.6 Program Transformation and Inductive Theorem Proving 12

1.7 Thesis Contributions 14

1.8 An Overview of Poitín 15

1.9 Structure of Thesis 16

2 Background 18

2.1 Introduction 18

2.2 Program Transformation 18

2.2.1 Language 18

2.2.2 Unfold/Fold Methodology (Burstall and Darlington) 20

2.2.3 Partial Evaluation 22

2.2.4 Supercompilation 23

2.3 Inductive Theorem Proving 38

2.3.1 Recursion Analysis 39

2.3.2 Rippling 41
2.3.3 Proof Planning 44
2.3.4 Proof Critics 47
2.4 Program Synthesis 50
2.4.1 Constructive Synthesis 50
2.4.2 Deductive Synthesis 52
2.4.3 Middle-Out Synthesis 53
2.4.4 Inductive Synthesis 54
2.5 Inductive Theorem Proving Using Program Transformation 55
2.5.1 Metacomputation 55
2.5.2 Partial Evaluation 56
2.5.3 Supercompilation 56
2.6 Use of Lemmas and Generalization Techniques 58
2.6.1 Cut Elimination 59
2.6.2 Use of Lemmas and Generalization in Induction 60
2.7 Conclusion 63

3 Distillation **64**
3.1 Introduction 64
3.2 Program Transformation Using Distillation 65
3.2.1 Folding and Generalization 65
3.2.2 Construction of Partial Process Trees 66
3.2.3 Rules for Residual Program Construction 68
3.3 Examples 70
3.4 Termination of the Distillation Algorithm 74
3.5 Correctness of Distillation Algorithm 80
3.6 Distilled Form 82
3.7 Conclusion 82

4 Theorem Proving in Poitín **84**
4.1 Introduction 84
4.2 Pre-Processing Phase 84
4.3 Explicit Quantification in Poitín 85
4.4 Inductive Theorem Proving in Poitín 88
4.4.1 Proving Universally Quantified Conjectures 88
4.4.2 Proving Existentially Quantified Conjectures 96

4.5 Soundness of Proof Techniques 102
4.6 Completeness 109
4.7 Conclusion 110

5 Program Construction in Poitín **111**
5.1 Introduction 111
5.2 Form of Input Specification 111
5.3 Construction of Program in Poitín 112
5.3.1 Distillation Rule for Program Construction 112
5.3.2 Precondition and Postcondition Analysis 113
5.3.3 Construction Process 114
5.3.4 Program Construction Rules $\mathcal{C}$ 114
5.4 Examples 118
5.5 Proof of Correctness 126
5.6 Conclusion 131

6 Implementation and Results **132**
6.1 Introduction 132
6.2 Poitín: a Prototype Version 132
6.2.1 Module Toplevel 133
6.2.2 Module ATP 133
6.2.3 Module Distill 135
6.2.4 Implementation of the Proof Rules $\mathcal{A}$ and $\mathcal{E}$ 137
6.2.5 Implementation of the Program Construction Rules $\mathcal{C}$ 138
6.3 Results 138
6.4 Conclusion 144

7 Conclusion and Future Work **145**
7.1 Summary of Thesis 146
7.1.1 Background 146
7.1.2 Distillation 146
7.1.3 Theorem Proving in Poitín 147
7.1.4 Program Construction in Poitín 147
7.1.5 Implementation and Results 147
7.2 Research Contributions 148
7.3 Future Work 149

7.3.1 Distillation 149
7.3.2 Inductive Theorem Proving 150
7.3.3 Program Verification 151
7.3.4 Program Construction 151
7.3.5 Implementation 152

Bibliography **153**

A Distillation **163**
A.1 Examples 163
A.1.1 Accumulating Patterns 163
A.1.2 Accumulating Parameters 174
A.1.3 Obstructing Function Calls 185
A.1.4 Examples for Theorem Proving 192

B Theorem Proving in Poitín **207**
B.1 Example 207

Abstract

We present new approaches to prove universally and existentially quantified conjectures and to construct programs from the resulting proofs. These theorem proving and program construction techniques make use of the *distillation* algorithm to transform input conjectures into a normalised form which we call *distilled form*. The proof rules are applied to the resulting distilled program. Our theorem proving and program construction techniques have been implemented in a theorem prover which we call Poitín. We give an overview of the distillation algorithm, and then present the proof and program construction techniques implemented in Poitín. Our implementation of the proof and program construction techniques used in Poitín is then presented. The soundness of the proof technique is shown with respect to a logical proof system using sequent calculus. We show that the constructed programs are correct with respect to their specification.

The main contributions of this thesis can be summarised as follows. First, we present fully automatic, and efficient inductive theorem proving techniques. Second, we present a novel program construction technique to construct correct programs. Third, we have shown how automatic program transformation can be used in a novel way in an inductive theorem prover. Finally, the use of distillation to obtain a normal form of the input program reduces over-generalization and generation of non-theorems. We have implemented the theorem prover and demonstrated it on some examples. The use of distillation in the framework of Poitín has eased the automation of the proof and program construction techniques in a reduced search space to make it fully automatic and efficient.

List of Figures

1.1 Theorem proving in Poitín 15

1.2 Program construction in Poitín 16

2.1 Higher order functional language 19

2.2 Example of a program 20

2.3 Grammar of redexes, contexts and observables 24

2.4 Normal order reduction rules 25

2.5 Partial process tree for $\mathcal{T}[\![append\ (append\ xs\ ys)\ zs]\!]$ 27

2.6 Rules for residual program construction in supercompilation 28

2.7 Constructed residual program from the partial process tree in Fig. 2.5 30

2.8 Examples and non-examples of homeomorphic embedding 33

2.9 Examples of most specific generalization 35

2.10 Rules for **let** expression 36

2.11 Algorithm of supercompilation 37

2.12 Partial process tree for $\mathcal{T}[\![reva\ xs\ Nil]\!]$ 38

2.13 Slots of a method 46

2.14 Induction method 47

2.15 Cut rule ($Cut(1)$) and the cut construct ($Cut(2)$) 59

3.1 Distillation algorithm 67

3.2 Function definitions 71

3.3 Partial process tree (1) for $\mathcal{T}[\![leq\ x\ (plus\ x\ y)]\!]$ 72

3.4 Partial process tree (2) for $\mathcal{T}[\![leq\ x\ (plus\ x\ y)]\!]$ 73

3.5 Partial process tree (3) for $\mathcal{T}[\![leq\ x\ (plus\ x\ y)]\!]$ 74

3.6 Residual program for $\mathcal{T}[\![leq\ x\ (plus\ x\ y)]\!]$ 74

3.7 Distilled form 82

4.1 Distillation pre-processing rules 86

4.2 Form of input conjecture 86
4.3 Form of proof expressions 87
4.4 Distillation rules for quantifiers 88
4.5 Proof rules for universal quantification 90
4.6 Proof rules for existential quantification 97
4.7 Sequent calculus rules for language 103

5.1 Form of input specification for program construction 111
5.2 Distillation rule for program construction 113
5.3 Program construction steps 115
5.4 Program construction rules $\mathcal{C}$ 116
5.5 Program construction rules $\mathcal{C}$ (Continued) 117

6.1 Data type of ATP module 134
6.2 Functions of ATP module 134
6.3 Some function definitions for failed proofs 141
6.4 Constructed programs for specifications of Table 6.4 143

A.1 Partial process tree (1) for $\mathcal{T}[\![even\ (plus\ x\ x)]\!]$ 165
A.2 Partial process tree (2) for $\mathcal{T}[\![even\ (plus\ x\ x)]\!]$ 166
A.3 Partial process tree (3) for $\mathcal{T}[\![even\ (plus\ x\ x)]\!]$ 166
A.4 Partial process tree (4) for $\mathcal{T}[\![even\ (plus\ x\ x)]\!]$ 168
A.5 Partial process tree (5) for $\mathcal{T}[\![even\ (plus\ x\ x)]\!]$ 171
A.6 Partial process tree (6) for $\mathcal{T}[\![even\ (plus\ x\ x)]\!]$ 174
A.7 Residual program for $\mathcal{T}[\![even\ (plus\ x\ x)]\!]$ 174
A.8 Partial process tree (1) for $\mathcal{T}[\![even\ (doublea\ x\ Zero)]\!]$ 176
A.9 Partial process tree (2) for $\mathcal{T}[\![even\ (doublea\ x\ Zero)]\!]$ 177
A.10 Partial process tree (3) for $\mathcal{T}[\![even\ (doublea\ x\ Zero)]\!]$ 179
A.11 Partial process tree (4) for $\mathcal{T}[\![even\ (doublea\ x\ Zero)]\!]$ 181
A.12 Partial process tree (5) for $\mathcal{T}[\![even\ (doublea\ x\ Zero)]\!]$ 182
A.13 Partial process tree (6) for $\mathcal{T}[\![even\ (doublea\ x\ Zero)]\!]$ 185
A.14 Residual program for $\mathcal{T}[\![even\ (doublea\ x\ Zero)]\!]$ 185
A.15 Partial process tree (1) for $\mathcal{T}[\![append\ (reverse\ xs)\ ys]\!]$ 187
A.16 Partial process tree (2) for $\mathcal{T}[\![append\ (reverse\ xs)\ ys]\!]$ 188
A.17 Partial process tree (3) for $\mathcal{T}[\![append\ (reverse\ xs)\ ys]\!]$ 191
A.18 Residual program for $\mathcal{T}[\![append\ (reverse\ xs)\ ys]\!]$ 192
A.19 Function definitions 192

A.20 Residual program for $\mathcal{T}[\![eqnum\ (plus\ x\ y)\ (plus\ y\ x)]\!]$ 194
A.21 Partial process tree (1) for $\mathcal{T}[\![iff\ (even\ x)\ (eqnum\ (double\ y)\ x)]\!]$ . . 196
A.22 Partial process tree (2) for $\mathcal{T}[\![iff\ (even\ x)\ (eqnum\ (double\ y)\ x)]\!]$ . . 197
A.23 Partial process tree (3) for $\mathcal{T}[\![iff\ (even\ x)\ (eqnum\ (double\ y)\ x)]\!]$ . . 198
A.24 Partial process tree (4) for $\mathcal{T}[\![iff\ (even\ x)\ (eqnum\ (double\ y)\ x)]\!]$ (Cont. of Fig. A.21) 200
A.25 Partial process tree (5) for $\mathcal{T}[\![iff\ (even\ x)\ (eqnum\ (double\ y)\ x)]\!]$ . . 201
A.26 Partial process tree (6) for $\mathcal{T}[\![iff\ (even\ x)\ (eqnum\ (double\ y)\ x)]\!]$ . . 203
A.27 Partial process tree (7) for $\mathcal{T}[\![iff\ (even\ x)\ (eqnum\ (double\ y)\ x)]\!]$ . . 206
A.28 Residual program for $\mathcal{T}[\![iff\ (even\ x)\ (eqnum\ (double\ y)\ x)]\!]$ 206

List of Tables

2.1 Induction suggestions by recursion analysis 40

6.1 Some conjectures proved in Poitín 139
6.2 Some of Poitín's failures 141
6.3 Conjectures of Table 6.2 proved by SPIKE using divergence critic and rippling 142
6.4 Some specifications for program construction 142

Chapter 1

Introduction

1.1 Motivation

The realization that a powerful theorem prover can provide a key component of many "intelligent machines" and the objective and creative attributes of mathematical reasoning made mathematicians and computer scientists interested in automated theorem proving. The dream of mechanizing mathematical reasoning with computer programs has been around since the early stages of electronic computers.

Mathematical induction is a method to reason about mathematical and computational objects containing repetition. Augustus De Morgan defined and introduced the term *Mathematical Induction* in his article *Induction* (*Mathematics*) which was published in the Penny Cyclopaedia [80]. Though the method of mathematical induction has been exploited in proofs for several centuries, this was the first formal definition.

Using mathematical induction, by generating a finite number of cases from an input formula, the formula can be proved for an infinite number of cases. To prove a universally quantified conjecture, the premises of an induction consist of one or more base cases, and one or more step cases. For example, to prove a conjecture about natural numbers, in the base case, the conclusion of the induction rule will be proved for some initial value 0. In the step case, the conclusion is proved for $Succ(n)$ assuming that the conjecture is true for a generic natural number n. In this way, the theorem is proved for an infinite number of successive values.

To reason about mathematical objects like natural numbers, data structures like lists and trees, recursively defined functions, hardware verification and many more aspects in mathematics, properties to be proved are specified as universally and/or

existentially quantified conjectures. In order to prove that a statement is *true* from some assumptions using the rules of inference, the proof system is provided with a rich knowledge of the domain, which constructs the relevant underlying theory. An automatic inductive theorem prover is used to prove these conjectures for an infinite number of successive values by performing base and step case proofs.

Some examples of inductive provers using explicit induction rules are the Boyer-Moore theorem prover (NQTHM) [8, 9], ACL2 [62], INKA, AF2, QuodLibet, Oyster/CLAM [21] and IsaPlanner [37, 38]. ACL2 is the re-implementation of NQTHM. We henceforth refer to the Boyer-Moore Theorem Prover as BMTP. RRL and SPIKE are implicit inductive theorem provers. Two examples of inductive theorem provers using program transformation techniques are Poitín [45] and Turchin's theorem prover [102]. Alan Bundy and his research group have developed knowledge-based theorem proving techniques *rippling* [13, 23, 15, 18] using explicit induction, and *proof planning* [13, 14, 52] for the automation of inductive reasoning. The details of the early and recent developments in automatic theorem proving can be found in the Bundy's survey paper [16].

Despite significant improvement in this area, automatic inductive reasoning creates challenging problems in the search for inductive proofs of some conjectures. In explicit induction, an infinite number of induction rules can exist in inductive proofs, which cannot be pre-stored. An inductive proof may require an arbitrary lemma to be conjectured and proved to complete the proof, or may require generalization to be performed. The search for appropriate lemmas and performing appropriate generalizations may cause infinite branching points in the search space. The usual approach to proving existential theorems is also more problematic.

Metacomputation [102, 103, 42] is an alternative to formal logic to prove the truth or falsity of logical formulae. *Metacomputation* is the task of simulating, analyzing or transforming programs by means of other programs. In recent work [45], Hamilton has presented a novel theorem proving technique using the distillation program transformation technique [45] to prove inductive theorems fully automatically. This technique is similar to that of Turchin's theorem proving technique [103, 102] in conjunction with the supercompiler [101]. We consider a proof system where a computer program is a model, and we conjecture that a model-based approach reduces the search space usually associated with the axiomatic approach, thus making it easier to automate. By specifying an input conjecture as a program, it is possible to transform this to a more efficient equivalent program using automatic

program transformation techniques, and the proof can be completed on this transformed program without using any intermediate lemmas. Thus, automatic program transformation can be used to aid inductive theorem proving.

We also propose the use of metacomputation-based formal methods, which lead to the development of computer programs from program specifications derived from existential theorems. The purpose of program transformation is to develop an equivalent but correct and efficient program by a sequence of manipulations using a set of transformation rules from a possibly inefficient input program. The method of program synthesis develops a correct and efficient executable program from an unexecutable specification describing the behaviour of the expected program, and ensures that the developed program meets its specification by verification. Hence, program synthesis can be seen as an extreme form of program transformation [85].

Program transformation is closely related to inductive theorem proving and program construction. This thesis sets out to use the distillation program transformation technique to develop metacomputation-based inductive theorem proving and program construction methods.

1.2 Aims of Thesis

The aim of this thesis is to design and implement a fully automatic metacomputation-based inductive theorem prover which can be used to prove universally and existentially quantified conjectures, and to construct programs from input program specifications. Poitín is written in Standard ML. To make the user of the theorem prover free from the burden of supplying explicit type annotations, we plan to implement the current version without any explicit use of type systems. We consider a higher order functional language with first order quantifiers. The language is typed using the Hindley-Milner polymorphic typing system [49, 79, 34]. We assume programs in the language are well-typed. Though we do not include the Hindley-Milner polymorphic type checker within our current version, it is possible to include this type checker within our system. As we have explained in the previous section, the limitations of inductive inference are major obstacles in the automation of proof, which limit the power of a theorem prover. This is also problematic for automatic program construction. In this thesis, we tackle these problems by incorporating the distillation program transformation algorithm within the inductive theorem proving and program construction framework of Poitín. We construct a hierarchy of source

to source transformations of the input conjectures and program specifications to facilitate metacomputation using additional rules for handling quantification.

We extend the theorem proving technique of Poitín [45] to handle explicit universal and existential quantifications to prove explicitly quantified inductive conjectures fully automatically. In [45], all free variables of the input conjectures are considered implicitly universally quantified, and therefore does not deal with explicit quantification. We have defined distillation rules for quantifiers and the proof rules for universal and existential quantifications. We have developed a program construction method to construct correct, efficient and executable functional programs from the proofs of non-executable input specifications using program construction rules in Poitín.

We present the distillation program transformation algorithm, which is used to transform the programs associated with the input conjectures and program specifications to obtain output programs which are in normal form. Distillation has the effect of removing intermediate data structures from programs, which could otherwise cause proof failures. This makes the proof and program construction techniques free from the problem of conjecturing intermediate lemmas and reduces over-generalization.

We then present proof techniques to prove inductive conjectures. We define distillation rules to deal with quantifiers at the meta-level, and proof rules for universal and existential quantification. To prove an inductive conjecture, distillation is first applied to the input conjecture. The distilled program is then pre-processed to obtain a proof expression to which the proof rules are then applied.

Finally, we present a constructive approach to constructing programs from input program specifications. The construction method performs a verification proof of the input specification to reject unsatisfiable specifications (i.e., specifications derived from non-theorems) to ensure that programs are constructed only from correct specifications. A distillation rule is defined to handle input specifications, and program construction rules are defined to construct programs from the resulting proof expression obtained.

We have implemented our proposed methods for inductive theorem proving and program construction, and added them to the theorem prover Poitín. We demonstrate the theorem prover on a number of inductive conjectures, and program specifications.

1.3 Program Transformation

Program transformation deals with the development of techniques and strategies which can be used to transform an inefficient program using a set of meaning preserving rules guided by the application of strategies to obtain a more efficient equivalent program (faster execution and less storage requirements).

There are two different approaches to program transformation: the algebraic approach and the operational approach. The algebraic approach uses axioms and theorems to rewrite expressions to obtain more efficient equivalent expressions. In this approach, a new theorem has to be invented to perform a new class of transformations.

The operational approach to program transformation uses a set of meaning preserving rules to obtain a more efficient equivalent program by generating new recursion equations. An example of this approach is the unfold/simplify/fold methodology of Burstall and Darlington [24]. Unfolding replaces a function call with the function body and folding replaces an expression which matches the function body with the corresponding function call. New recursive equations are generated by simplifying the old ones through the application of a set of meaning preserving rules.

The use of intermediate data structures in functional programming makes programs more readable, but this makes programs inefficient. Burstall and Darlington's transformation technique [24] has been extended to more powerful automatic transformation techniques such as deforestation [104, 105], supercompilation [101] and distillation [45, 46] to remove intermediate data structures.

Distillation is more powerful than deforestation and supercompilation; some useful transformations cannot be performed by these techniques which can be performed by distillation [46]. Distillation can produce superlinear improvement in the runtime of programs, whereas other techniques can produce only linear improvement. In deforestation and supercompilation, matching is performed on flat terms; functions are considered to match if they have the same names. Distillation allows matching of recursive terms where different recursive terms are considered to match if they have the same recursive structure even though they contain different function names.

The operational approach of program transformation using distillation is used in this thesis to develop our metacomputation-based inductive theorem proving and program construction framework.

1.4 Inductive Theorem Proving

1.4.1 Mathematical Induction

Mathematical induction uses induction rules to infer universal statements incrementally. To evaluate the various capabilities of different inductive theorem proving systems, two categories of problems are identified [51]:

- $\forall$-*quantified:* The category which only uses the universal quantifier and do not include any synthesis problems.
- $\forall\exists$-*quantified:* The category which includes program synthesis problems that require proving existentially quantified formulas, and construction of existential witnesses.

Different types of induction can be used to deal with inductive proofs. Two approaches for constructing inductive proofs are explicit and implicit induction. Explicit induction techniques depend on a semantic ordering while implicit induction techniques rely on a syntactic ordering (the one which shows the termination of the definitions).

Explicit Induction

In explicit induction, induction rules are explicitly incorporated into proofs. One such rule is Peano induction for natural numbers ($\mathbb{N}$) of the following form [17]:

$$\frac{P(0), \quad \forall n : nat.(P(n) \rightarrow P(Succ(n)))}{\forall n : nat.P(n)} \tag{1.1}$$

In the application of the above induction rule in the proof of the conjecture $\forall n : nat.P(n)$ where P is the property to be proved, $P(0)$ is called the base premise, $\forall n : nat.(P(n) \rightarrow P(Succ(n)))$ is the step premise, $P(n)$ is called the *induction hypothesis*, $P(Succ(n))$ is the *induction conclusion*, n is the *induction variable*, and $Succ(n)$ is the *induction term.*

The *one*-step induction rule for lists is of the following form:

$$\frac{P(Nil), \quad \forall h : \tau.\forall t : list(\tau).(P(t) \rightarrow P(h :: t))}{\forall l : list(\tau).P(l)} \tag{1.2}$$

A two-step induction rule for natural numbers is given by the following form:

$$\frac{P(0), \quad P(Succ(0)), \quad \forall n : nat.(P(n) \rightarrow P(Succ(Succ(n))))}{\forall n : nat.P(n)} \tag{1.3}$$

The *two*-step induction rule (1.3) is structurally similar to the recursive definition of the *even* predicate which is given below.

$$\begin{array}{rcl} even(0) & \leftrightarrow & true \\ even(Succ(0)) & \leftrightarrow & false \\ even(Succ(Succ(n))) & \leftrightarrow & even(n) \end{array}$$

This shows the duality relationship between recursive definitions and the form of induction rules. This duality relationship allows us to construct new induction rules and also to select the proper induction rule to prove the properties of recursive functions. The success of an inductive proof mainly depends on the selection of the induction rule and the induction variable. Most of the inductive theorem proving techniques generate customised induction rules from the recursive definitions appearing in the conjecture to be proved. The *recursion schema* contributes to the corresponding *induction schema*; e.g., a 2 step recursion schema constructs a 2 step induction schema, the schema $Succ(Succ(n))$ of the *even* predicate builds the induction schema $P(Succ(Succ(n)))$. The patterns in the left hand side of the base and recursive equations of the function are used to build the required induction schema. According to *recursion analysis* [8, 9, 15, 19, 97] (§2.3.1), the variable in the recursive argument position of a function appearing in a conjecture is selected as a potential induction variable. Induction is performed on the structural form of the finally selected induction variable using the induction rule suggested by recursion analysis.

Example 1

Consider the proof of the *associativity of addition* theorem about natural numbers given by conjecture (1) using the recursive definition of the + function using the standard rewriting technique [17].

$$\forall x : nat.\forall y : nat.\forall z : nat.x + (y + z) = (x + y) + z \quad (1)$$

$$\begin{array}{l} 0 + y = y \\ Succ(x) + y = Succ(x + y) \\ Succ(x) = Succ(y) \leftrightarrow x = y \end{array}$$

The following rewrite rules (i) and (ii) derived from the recursive definition of + function, and (iii) derived from the replacement rule for *Succ* are used in the proof of conjecture (1) for the free data type *nat* [17].

$$
\begin{array}{ll}
0 + y \Rightarrow y & \text{(i)} \\
Succ(x) + y \Rightarrow Succ(x + y) & \text{(ii)} \\
Succ(x) = Succ(y) \Rightarrow x = y & \text{(iii)}
\end{array}
$$

We use the 1-step induction rule (1.1) for *nat* on x. In the base case, the conclusion is proved for $x = 0$ by the base case premise of the conclusion of rule (1.1). The application of the rewrite rule (i) to the base case results in the following proof step.

$$
\begin{array}{lll}
\vdash & 0 + (y + z) = (0 + y) + z & \text{(by base case premise of induction rule (1.1))} \\
\vdash & y + z = y + z & \text{(by (i))}
\end{array}
$$

This can be proved by symbolic evaluation.

In the step case, the conclusion is proved for $x = Succ(x)$ by the step case premise of the conclusion of rule (1.1) assuming the conjecture (1) is *true* for some generic natural number x. Thus, $x + (y + z) = (x + y) + z$ is the *induction hypothesis.* The application of the rewrite rules (ii) and (iii) to the induction conclusion results in the following proof steps.

$$
\begin{array}{lll}
x + (y + z) = (x + y) + z & \vdash Succ(x) + (y + z) = (Succ(x) + y) + z & \\
& \quad \text{(by step case premise of induction rule (1.1))} & \\
& \vdash Succ(x + (y + z)) = (Succ(x + y)) + z & \text{(by (ii))} \\
& \vdash Succ(x + (y + z)) = Succ((x + y) + z) & \text{(by (ii))} \\
& \vdash x + (y + z) = (x + y) + z & \text{(by (iii))}
\end{array}
$$

In this state of the proof, a complete copy of the induction hypothesis is found in the simplified induction conclusion, and the proof can be easily completed.

Strong Fertilization

Strong fertilization is a technique that uses the induction hypothesis to prove the induction conclusion. In the step case, a copy of the induction hypothesis is found embedded in the induction conclusion, which then can be replaced by the value *true* ($\top$). Let $\mathcal{R}$ be the set of rewrite rules, *Ind* and *IH* are the suggested induction rule and induction hypothesis respectively. Then, the application of strong fertilization in the step case proof can be represented as follows.

$$
\begin{array}{lll}
\text{Induction Hypothesis} & \vdash_{Ind} & \text{Induction Conclusion} \\
& \vdash_{\mathcal{R}} & E[\text{Induction Hypothesis}] \\
& \vdash_{IH} & E[\top]
\end{array}
$$

E is the context which may be empty or a subterm which is a part of the simplified induction conclusion.

We apply strong fertilization to the simplified induction conclusion in Example 1 to prove it to be *true* using the induction hypothesis $x + (y + z) = (x + y) + z$. This completes the step case proof successfully, which demonstrates that conjecture (1) is an inductive theorem.

Weak Fertilization

Sometimes the proof attempt gets stuck before obtaining a complete copy of the induction hypothesis embedded within the simplified induction conclusion, but a part of the induction hypothesis is found embedded within the simplified term. By replacing this part of the simplified induction conclusion with the opposite side of the induction hypothesis, a simplified goal can be obtained which can be proved easily in some cases. This fertilization technique is called *weak fertilization* [17]. This technique is applicable only when the conjecture and the hypothesis are expressed as equations, thus allowing the use of the hypothesis as a rewrite rule. In some cases, neither strong nor weak fertilization is applicable. In these cases, appropriate intermediate lemmas or generalization can help to complete the proof.

Implicit Induction

In the implicit induction approach, the induction scheme is not known beforehand. Examples of this approach include the *cover set* method [110, 111], *test set* method [6, 5], and *rewriting induction* method [84]. Each of these methods provides a set of terms or pairs(context,term) which is used to replace the induction variable depending on the context. This produces a set of new conjectures which can be further simplified by using smaller instances of the original conjecture called the *induction hypothesis.* The proof is completed when all newly generated conjectures are simplified into known inductive theorems.

Descente Infinie

Induction is a very commonly used technique for proving theorems, but it is less commonly used in the form of *descente infinie*, (re)discovered by Pierre de Fermat (1606-1665). In this method, for any proof of a conjecture, it is required to show for each assumed counterexample of the conjecture, the existence of another counterexample of the conjecture that is strictly smaller in some well-founded ordering.

The resulting infinite sequence of "smaller" counter examples contradicts the well-founded order requirement, hence original counterexample is invalid. First, the proof is started with the initial conjecture, and it is simplified using case analysis. In the step case, the induction conclusion is simplified, and, every time it is searched for a current goal which is a similar but a different instance of the original conjecture. The original conjecture is then applied as the induction hypothesis to prove the induction conclusion. Finally, it is needed to show the well-founded ordering in which all the instances of the original conjecture that have been applied as the induction hypotheses are smaller than the original conjecture.

For example, to prove a property P that is true of all natural numbers $\mathbb{N}$, one may demonstrate that if P is not true of an arbitrary natural number n, then it is not true for a smaller number $m < n$, which can be used to infer an infinite decreasing sequence of natural numbers. This can be explained by an inference rule of the following form [11]:

$$\frac{\forall x : nat.(\neg P(x) \rightarrow (\exists y : nat.y < x \wedge \neg P(y)))}{\forall x : nat.P(x)} \tag{1.4}$$

specific to *nat* where $<$ is the reducing well-founded ordering. A formal framework has been presented by integrating induction in the form of descente infinie with deductive theorem proving system in [109].

1.4.2 Limitations of Inductive Inference

Inductive theories are (i) usually incomplete [43], i.e., there exists true but unprovable formulae and (ii) they do not admit *cut elimination*, so, arbitrary intermediate formulas may need to be proved and then used to prove the current conjecture [69]. These two problems introduce infinite branching into the search space.

The cut rule is required to introduce intermediate lemmas and to perform generalizations. Gentzen's cut rule for sequent calculus is of the following form:

$$\frac{A, \Gamma \vdash \Delta \quad \Gamma \vdash A}{\Gamma \vdash \Delta} \tag{1.5}$$

The cut rule allows us to first prove Δ with the aid of A, and then eliminate A by proving it from Γ where A is the cut formula. In inductive proof, this cut formula is the generalized formula or lemma. The use of cut elimination in logical systems means that if a proposition has a proof which uses some intermediate proposition for that proof, then it has a direct proof with a series of proof rewriting which does

not require any intermediate proposition. This was shown to be true by Gentzen for first order theories [39], but Kreisel has shown that it is not true for inductive theories [69]. See [15] for details.

An unbounded number of induction rules are required to construct and apply dynamically, which cannot be pre-stored. Some common problems that arise in the search for an inductive proof are induction rule choice, speculating lemmas and identifying the need for and performing generalization.

1.5 Program Synthesis

Program synthesis deals with the systematic development of an executable program from an unexecutable specification describing the behaviour of the program to be constructed, and verifying that the constructed program satisfies the specification. Synthesis methods need to incorporate techniques which use a constructive approach to construct the unknown program/value. The main techniques for program synthesis are:

1. Constructive synthesis
2. Deductive synthesis
3. Middle-out synthesis

It is possible to construct recursively defined programs by proving a synthesis conjecture of the form $\forall x : \tau_1.\exists y : \tau_2.spec(x, y)$, where x and y are the input and output variables respectively and *spec* is the formal relationship between x and y usually expressed in terms of predicates, relations, and functions.

Constructive synthesis or proofs-as-programs in functional programming is based on the Curry-Howard isomorphism [50] in constructive type theory, e.g., Martin-Löf's constructive type theory [77]. The constructive type theories are logics for reasoning about functional programs. In the proofs-as-programs concept, the proof itself is considered as the program to be extracted. There is a one-to-one relationship between a constructive proof of an existence theorem and a program (a function) that computes witnesses of the existentially quantified variables. For example, a synthesis specification can be expressed in the following form in constructive type theory:

$$f(x) : \text{ALL } x : nat.\text{EX } y : nat.(even(x)) \leftrightarrow (double(y) = x)$$

The function $f(x)$ can be constructed from the proof of the above specification. The constructed function satisfies the specification. The function $f(x)$ will compute a witness for y for every x.

Deductive synthesis can derive executable programs from high level specifications by applying inference rules. This synthesis technique usually employs theorem proving to synthesise correct programs from specifications.

The third approach (middle-out reasoning) allows undefined functions in the synthesis conjecture [68]. In order to extract the definition of the undefined function from the synthesis proof of this conjecture, definition-like subgoals are identified during the synthesis proof, and these are converted to program definitions. These definitions are then used to complete the proof, and to define the synthesised program. Higher-order unification is used to instantiate the undefined function.

1.6 Program Transformation and Inductive Theorem Proving

There is a close correspondence between program transformation and inductive theorem proving.

Recursion and induction can be regarded as duals. The induction in proof corresponds to the recursion in the program. The unfold/fold transformation technique can be used for inductive proof that does not use any explicit induction schema. The schema is constructed implicitly by unfolding the recursive definitions of the functions. This idea is analogous to the recursion analysis technique employed in the Boyer-Moore theorem prover [8]. The given function definitions are utilised to prove theorems about them. The recursion present in the definitions corresponds to the required induction. Unfolding accomplishes the base case and the induction step, and folding roughly corresponds to the application of the induction hypothesis [26, 45].

The unfold/fold program transformation technique proposed by Burstall and Darlington [24] uses the *associativity* or *commutativity* properties of functions as *laws* only when they can make a fold possible. In a diverged inductive proof attempt, the information obtained from the divergence pattern can be used to suggest lemmas which are used as rewrite rules to overcome this divergence. Usually, the subterm(s) of the rewritten conclusion is(are) used to identify lemmas based on some heuristics. After conjecturing the lemmas, the initial conjecture can be proved by applying the

computing the task specified by that proposition. By controlling the proofs, we can improve the efficiency of programs extracted from the proof of existential theorems. "Proofs and programs are the same thing, and simplifying a proof corresponds to executing a program" [106].

1.7 Thesis Contributions

This thesis mainly contributes to the fields of metacomputation-based inductive theorem proving and program construction. The primary objective of this thesis is to show how automatic program transformation can be used in a novel way in an inductive theorem prover. This thesis undertakes the theoretical study as well as the practical implementation of the intended inductive theorem proving and program construction techniques to deal with explicit quantification using the distillation program transformation algorithm.

We define a higher order functional language with quantifiers to express input conjectures and program specifications. We explore the distillation program transformation algorithm, and apply this algorithm to some example programs. We show that the distillation algorithm terminates on all input programs, and that it is correct.

We present fully automatic, and efficient, inductive theorem proving techniques. We define distillation rules for universal and existential quantification to deal with quantifiers at the meta-level, and define proof rules for proof expressions. We show how the distilled form of the program associated with an input conjecture can be converted to proof expressions, and show how the proof rules can be used to prove them. One big advantage of these proof techniques is that no intermediate lemmas are required, which helps to avoid infinite branch points in the search space. The existential proof rules perform a pure existence proof of the existential conjecture without requiring the construction of any witness to obtain the truth value of the conjecture. The inclusion of the distillation algorithm within our proof techniques has reduced over-generalization and generation of non-theorems, and allows us to prove more theorems than Turchin's theorem prover. We show the soundness of our proof techniques with respect to a logical proof system using sequent calculus.

We present a novel program construction method to construct executable programs from input specifications derived from existential theorems. As far we know, this is the first time automatic program transformation is used in a program con-

struction method. We define distillation rules to handle input specifications at the meta-level, and program construction rules for proof expression obtained by distillation of the program associated with an input specification. We show that the constructed program is correct with respect to the input specification. We give a proof of correctness of the program construction method. We also argue that as programs are constructed using distillation, they are likely to be more efficient than programs constructed using other techniques.

We have implemented the distillation algorithm, the inductive theorem proving and program construction techniques and added them to the theorem prover Poitín. The use of distillation within the framework of Poitín has eased the automation of the proof and program construction techniques to make Poitín a fully automatic and efficient theorem prover. We have presented some results of the application of the Poitín theorem prover to inductive theorems and program specifications. The main outcome is that the proof techniques of Poitín can be used to prove inductive conjectures fully automatically without the need for conjecturing any intermediate lemmas. Our program construction techniques can be used to construct totally correct programs from input specifications.

Finally, we give some suggestions for future research.

1.8 An Overview of Poitín

A diagrammatic overview of the processes realized in the theorem prover Poitín is shown in Figs 1.1 and 1.2.

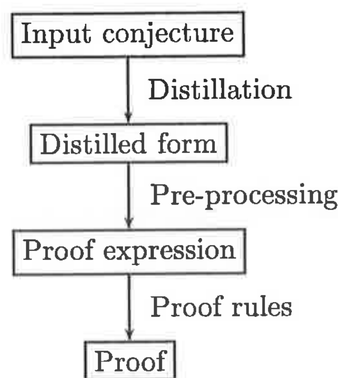

Figure 1.1: Theorem proving in Poitín

For universally and existentially quantified conjectures, distillation is applied to the input conjecture to obtain a distilled form as shown in Fig. 1.1. A proof expression is obtained by pre-processing this distilled expression. The proof rules for universal and existential quantifications are then applied to this proof expression to obtain the truth value of the conjecture.

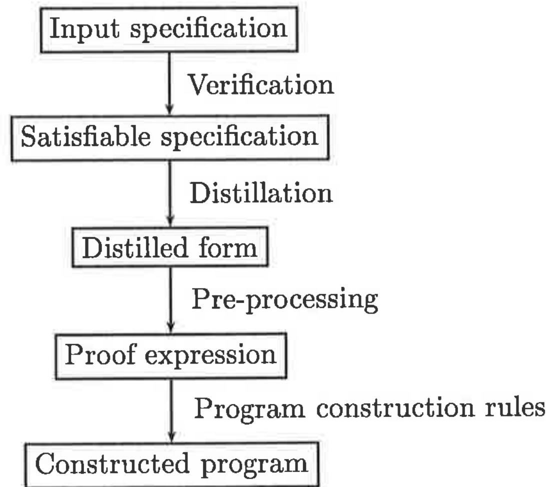

Figure 1.2: Program construction in Poitín

For program construction, the input specification is verified to check whether it is satisfiable or not as shown in Fig. 1.2. If the specification is satisfiable, distillation is applied to the input specification to obtain a distilled form. A proof expression is obtained by pre-processing this distilled expression. Finally, the program construction rules are applied to the proof expression to construct a program.

1.9 Structure of Thesis

The rest of this thesis is structured as follows:

- Chapter 2: We survey the research carried out in the fields of inductive theorem proving and program synthesis. Also, we give an overview of the metacomputation-based inductive theorem proving technique using supercompilation.
- Chapter 3: We give an overview of the novel program transformation technique distillation. We explore the details of the distillation algorithm and its

termination proof. We show how distillation can be used to transform more complex input programs which cannot be transformed using supercompilation.

- Chapter 4: We present proof techniques to prove universally and existentially quantified conjectures using the normalized program obtained with the distillation algorithm presented in Chapter 3. We also give a proof of the soundness of these rules.

- Chapter 5: We present program construction techniques for the construction of programs from input specifications using the normalized program obtained with the distillation algorithm presented in Chapter 3. We then show the correctness of the constructed programs with respect to their specification.

- Chapter 6: We give an overview of the prototype version of Poitín which we have implemented using the distillation algorithm, proof and program construction techniques as presented in the previous chapters, and give some results of applying this tool.

- Chapter 7: We conclude our thesis, and give a summary of the work presented. Finally, we give suggestions for future research which can be carried out based on the work done so far.

Chapter 2

Background

2.1 Introduction

This chapter gives an overview of the research carried out in the fields of automatic program transformation, inductive theorem proving using explicit induction, program synthesis techniques, and metacomputation-based inductive theorem proving using automatic program transformation. We also discuss the limitations of inductive inference, and existing metacomputation-based inductive theorem proving techniques.

2.2 Program Transformation

In this section, we give a brief overview of Burstall and Darlington's program transformation technique and partial evaluation. We also present the supercompilation program transformation algorithm based on the presentation in [46] and [95].

2.2.1 Language

In this section, we describe the syntax and semantics of the language which will be used throughout this thesis. The language is a simple higher order functional language as described in Fig. 2.1.

The syntax of the language covers all possible forms of expression of a higher order functional language using *variables, application*, and *abstraction*. A program in the language is defined by an expression to be evaluated and a set of definitions of the functions exploited in the expression. All of the user defined functions must have unique names, and all of the variables in the function body must be bound to

$prog$ ::=	e_0 **where** $f_1 = e_1; \ \ldots; \ f_n = e_n;$	program
e ::=	v	variable
|	$c \ e_1 \ldots e_n$	constructor application
|	$\lambda v.e$	lambda abstraction
|	f	function variable
|	$e_0 \ e_1$	application
|	**case** e_0 **of** $p_1 : e_1 \mid \ldots \mid p_k : e_k$	case expression
|	**let** $v_1 = e_1, \ \ldots, \ v_n = e_n$ **in** e_0	let expression
|	**letrec** $f = e_0$ **in** e_1	letrec expression
p ::=	$c \ v_1 \ldots v_n$	pattern

Figure 2.1: Higher order functional language

the formal arguments of the function. Recursion is introduced at the top level using the **where** construct. The **letrec** expression is used in the language to allow local function definitions which may contain non-local variables.

The language uses constructs to build and facilitate operations on *algebraic* data structures. An algebraic data type is constructed by combining other data types with the help of constructors. For example, the *List* type is a common example of algebraic data type with two constructors: *Nil* for empty list ([]) with no argument, and *Cons* (::) with two arguments for a non-empty list.

$$List \ \tau \ = \ Nil \mid Cons \ \tau \ (List \ \tau)$$

Cons constructs a non-empty list by combining the head element of type τ with the tail of the list.

Each constructor has a fixed arity: *Nil* and *Zero* both have arity 0, *Cons* has arity 2, *Succ* has arity 1. Each constructor application must be saturated in order to construct a data structure. The truth values *True* and *False* are defined as constructors.

Data structures are decomposed and operated on in the selector of a **case** expression by *pattern matching.* Within case expressions of the form:

$$\textbf{case } e_0 \textbf{ of } p_1 : e_1 \mid \ldots \mid p_k : e_k$$

e_0 is called the *selector*, $e_1 \ldots e_k$ are called the *branches*, and $p_1 \ldots p_k$ are the *patterns.* The pattern variables of a **case** expression and λ-abstraction argument variables are

locally bound. Variables with the same name in an outer scope will no longer be in scope inside this binding. Within a **case** expression, patterns are distinct and mutually exclusive. The selector expression e_0 is evaluated to head normal form to match with any of the patterns appearing in the alternative branches before selecting any of the branch expressions by pattern matching.

The conditional **if** e_0 **then** e_1 **else** e_2 is represented using a **case** expression of the form **case** e_0 **of** $True \; : \; e_1 \mid False \; : \; e_2$.

The **case** expression can also be used for decomposing compound data structures. For example, a **case** expression that decomposes a list data structure is of the form: **case** e_0 **of** $Nil \; : \; e_1 \mid Cons \; x \; xs \; : \; e_2$.

The language is typed using the Hindley-Milner polymorphic typing system [49, 79, 34], which prevents the forming of any type incorrect expressions. We assume programs in the language are well-typed, and the *recursive* data types are defined as algebraic types. The operational semantics of the language is *normal order reduction.*

An example of a program in the language is given below, which reverses the list xs. The program consists of the expression *reva xs Nil* and uses the accumulative recursive definition of the reverse function.

$$
\begin{array}{l}
reva \; xs \; Nil \\
\text{where} \\
reva = \lambda xs.\lambda ys.\, \textbf{case} \; xs \; \textbf{of} \\
\qquad\qquad\qquad\qquad Nil \qquad\quad : ys \\
\qquad\qquad\qquad\quad \mid Cons \; x \; xs' : reva \; xs' \; (Cons \; x \; ys)
\end{array}
$$

Figure 2.2: Example of a program

2.2.2 Unfold/Fold Methodology (Burstall and Darlington)

Burstall and Darlington's unfold/fold program transformation technique [24] is a semiautomatic transformation system, which requires user guidance for supplying *eureka steps* in transforming programs. In this transformation system, the following six transformation rules are used to transform an input program defined using first order recursion equations.

1. *Definition.* This rule introduces new definitions ensuring that the left hand side of each new definition is not an instance of the left hand side of an existing definition.

2. *Instantiation.* Introduce a substitution instance of an existing definition.

3. *Unfolding.* For any two definitions $f\ v_1 \ldots v_n = e$ and $f'\ v'_1 \ldots v'_n = e'$, if e' contains an occurrence of $f\ v_1 \ldots v_n$, then this occurrence is replaced by e in $f'\ v'_1 \ldots v'_n = e'\ [e/f\ v_1 \ldots v_n]$ producing a new definition.

4. *Folding.* For any two definitions $f\ v_1 \ldots v_n = e$ and $f'\ v'_1 \ldots v'_n = e'$, if e' contains an occurrence of an instance of e, then this occurrence is replaced by the corresponding instance of $f\ v_1 \ldots v_n$ in $f'\ v'_1 \ldots v'_n = e'\ [f\ v_1 \ldots v_n/e]$ producing a new definition.

5. *Abstraction.* A *where* clause may be introduced to create a new definition from an existing definition $f\ v_1 \ldots v_n = e$ by replacing sub-expressions with variables ensuring that the new variables do not exist in the source definition: $f\ v_1 \ldots v_n = e[x_1/e_1, \ldots, x_n/e_n]$ where $\langle x_1, \ldots, x_n \rangle = \langle e_1, \ldots, e_n \rangle$.

6. *Laws.* The associativity or commutativity properties of the primitives are used to rewrite the right hand side of a definition to obtain a new definition.

These rules ensure the partial correctness [65, 88, 90] of the derived program. Total correctness of the derived program is achieved by applying transformation strategies. Instantiation and unfolding do not alter efficiency in the transformed program. Folding at least preserves efficiency when the argument used in substitution is lower in some well-founded ordering than that used in the input equation being transformed. Improvements is introduced by rewriting lemmas and using abstraction. Burstall and Darlington have devised a simple strategy which leads to powerful transformations applicable to many example programs. This strategy includes the steps which are described by making any necessary definitions, instantiating, for each instantiation unfolding repeatedly, trying to apply *laws* and *abstraction*, then folding repeatedly.

Consider the unfold/fold transformation of the term *reverse xs* using the recursive definitions of *reverse* and *append* as given below [24].

1. $reverse\ []$	$=$	$[]$	given
2. $reverse\ (x :: xs')$	$=$	$append\ (reverse\ xs')\ (x :: [])$	given
3. $append\ []\ ys$	$=$	ys	given
4. $append\ (x :: xs')\ ys$	$=$	$x :: (append\ xs'\ ys)$	given

The associativity of append is given by the following equation:

$$append\ (append\ xs\ ys)\ zs \;=\; append\ xs\ (append\ ys\ zs)$$

A new function f is defined by generalizing $(x :: [])$ to ys.

5. $f\ xs\ ys$	$=$	$append\ (reverse\ xs)\ ys$	definition (eureka)
6. $f\ []\ ys$	$=$	ys	instantiate and unfold
7. $f\ (x :: xs')\ ys$	$=$	$append\ (append\ (reverse\ xs')\ (x :: []))\ ys$	
			instantiate and unfold
	$=$	$f\ xs'\ (append\ (x :: [])\ ys)$	associativity and fold
8. $reverse\ (x :: xs')$	$=$	$f\ xs'\ (x :: [])$	fold 2 with 5

The following program is obtained from the above transformation steps.

$$\begin{aligned} reverse\ [] \quad &= \quad [] \\ reverse\ (x :: xs') \quad &= \quad f\ xs'\ (x :: []) \end{aligned}$$

$$\begin{aligned} f\ []\ ys \quad &= \quad ys \\ f\ (x :: xs')\ ys \quad &= \quad f\ xs'\ (append\ (x :: [])\ ys) \end{aligned}$$

By using a *Redefinition* rule, we can obtain a more succinct program $reverse\ xs = f\ xs\ []$.

In their program improvement system [24], users were required to supply the new equations of any definitions, useful lemmas to use as rewrite rules, information about the associativity or commutativity of functions, and the properly instantiated definitions.

2.2.3 Partial Evaluation

Partial evaluation [56, 55] is a program optimization technique which transforms a given program distinguishing between *static* and *dynamic* data. It derives a residual program based on the static input data, and when the rest of the input, called dynamic data is provided, this residual program calculates the whole output as would the original program.

For example, consider a source program p which needs two inputs $in1$ (static) and $in2$ (dynamic) for its evaluation. If the input data $in1$ is supplied, the *partial evaluator* will construct a program p_{in1}. When the rest of the input $in2$ is supplied, it will produce the same result that would have been produced if $in1, in2$ were supplied

together [55]. Thus, $[\![p]\!]\ [in1, in2] = [\![p_{in1}]\!]\ in2$. A partial evaluator is a *program specializer*, where the specialization is done by performing those calculations that depend on static data, and generating code for those calculations that depend on dynamic data.

Using the definition of *append* (§2.2.2), the partial evaluation of the term *append* (*append* [] *xs*) *ys* results in the term *append xs ys* where [] is the static input. The partial evaluation of *append* ($x :: xs$) *ys* results in $x ::$ (*append xs ys*) where the partially static input $x :: xs$ consists of the dynamic input x and xs.

2.2.4 Supercompilation

Turchin's supercompiler [101, 103] is a semantics-based program transformation technique defined for the *Refal* language. The supercompiler is more powerful than partial evaluation and deforestation, and it can lead to a very deep structural transformation of an input program. It supervises the operation of the whole input program, compiles it, and produces a faster program with the same semantic value. The supercompiler uses a set of transformation rules which preserve the functional meaning of the program to perform a step-by-step transformation of the input program. The positive supercompiler [94, 95, 42] is a newer version of Turchin's supercompiler in a functional language setting.

Transformation Using Supercompilation

A residual program is constructed by transforming an input program defined with an expression containing free variables and definitions of functions used in the expression using supercompilation. The language for which supercompilation is to be performed is a simple higher-order functional language defined in §2.2.1.

The supercompilation rules are defined by identifying the next reducible expression (*redex*) within some evaluation context. An expression which cannot be broken down into a redex and a context is called an *observable.*

Definition 2.2.1 *Redexes, contexts and observables*

Redexes, contexts and observables are defined more formally by the grammar shown in Fig. 2.3, where *red* ranges over redexes, *con* ranges over contexts and *obs* ranges over observables. We use the notation $con\langle e\rangle$ to represent an expression which is broken down into an evaluation context *con* and the redex e.

$$
\begin{array}{lcl}
red & ::= & f \\
 & | & (\lambda v.e_0)\ e_1 \\
 & | & \text{case}\ (v\ e_1 \ldots e_n)\ \text{of}\ p_1\ :\ e'_1 \mid \ldots \mid p_k\ :\ e'_k \\
 & | & \text{case}\ (c\ e_1 \ldots e_n)\ \text{of}\ p_1\ :\ e'_1 \mid \ldots \mid p_k\ :\ e'_k \\
con & ::= & \langle\rangle \\
 & | & con\ e \\
 & | & \textbf{case}\ con\ \textbf{of}\ p_1 : e_1 \mid \cdots \mid p_k : e_k \\
obs & ::= & v\ e_1 \ldots e_n \\
 & | & c\ e_1 \ldots e_n \\
 & | & \lambda v.e
\end{array}
$$

Figure 2.3: Grammar of redexes, contexts and observables

Lemma 2.2.2 *Unique decomposition property*

For every expression e, either it must be an *observable* or it can be decomposed into a unique context con and a redex r such that $e = con\langle r\rangle$.

Example 2 (Context and Redex)

For a case expression **case** $(f\ e_1 \ldots e_n)$ **of** $p_1\ :\ e'_1 \mid \ldots \mid p_k\ :\ e'_k$, the redex is f, and it must be unfolded to select the appropriate branch of the **case** expression. The context is **case** $(\langle\rangle\ e_1 \ldots e_n)$ **of** $p_1\ :\ e'_1 \mid \ldots \mid p_k\ :\ e'_k$. In a function application, $f_1\ (f_2\ x)\ y$, the redex is f_1 and the context is $\langle\rangle\ (f_2\ x)\ y$. So the function f_1 must be unfolded.

The central operation in supercompilation is *driving*. Supercompilation applies *folding* and/or *generalization* techniques to ensure the termination of driving.

Driving

Driving is achieved by *normal order reduction* of an input program, which may construct a potentially infinite process tree to represent all of the possible computations of the given expression of the input program.

The rules for normal order reduction are defined by the map $\mathcal{N}$ from expressions to ordered sequences of expressions $[e_1 \ldots e_n]$ as shown in Fig. 2.4. Within

these rules, the notation $e\{v_1 := e_1, \ldots, v_n := e_n\}$ represents the simultaneous substitutions of the sub-expressions $e_1, \ldots, e_n$ for the free occurrences of $v_1, \ldots, v_n$, respectively, within e.

$$\mathcal{N}[\![v\ e_1 \ldots e_n]\!]\ \phi = [e_1, \ldots, e_n] \quad (\mathcal{N}1)$$
$$\mathcal{N}[\![c\ e_1 \ldots e_n]\!]\ \phi = [e_1, \ldots, e_n] \quad (\mathcal{N}2)$$
$$\mathcal{N}[\![\lambda v.e]\!]\ \phi = [e] \quad (\mathcal{N}3)$$
$$\mathcal{N}[\![con\langle f\rangle]\!]\ \phi = [unfold(con\langle f\rangle)\ \phi] \quad (\mathcal{N}4)$$
$$\mathcal{N}[\![con\langle(\lambda v.e_0)\ e_1\rangle]\!]\ \phi = [con\langle e_0[e_1/v]\rangle] \quad (\mathcal{N}5)$$
$$\mathcal{N}[\![con\langle \textbf{case}\ (v\ e_1 \ldots e_n)\ \textbf{of}\ p_1 : e'_1 | \ldots | p_k : e'_k\ \rangle]\!]\ \phi \quad (\mathcal{N}6)$$
$$= [v\ e_1 \ldots e_n, con\langle e'_1\rangle\{v\ e_1 \ldots e_n := p_1\}, \ldots, con\langle e'_k\rangle\{v\ e_1 \ldots e_n := p_k\}]$$
$$\mathcal{N}[\![con\langle \textbf{case}\ (c\ e_1 \ldots e_n)\ \textbf{of}\ p_1 : e'_1 | \ldots | p_k : e'_k\ \rangle]\!]\ \phi \quad (\mathcal{N}7)$$
$$= [con\langle e'_i\{v_1 := e_1, \ldots, v_n := e_n\}\rangle]$$
$$\text{where } p_i = c\ v_1 \ldots v_n$$

Figure 2.4: Normal order reduction rules

The function *unfold* unfolds the function f in its argument expression $con\langle f\rangle$ as follows:

$$unfold(con\langle f\rangle)\ \phi = con\langle e\rangle \text{ where } f \text{ is defined by } f = e, \text{ and } (f, e) \in \phi$$

The normal order reduction rules are mutually exclusive and together exhaustive by the unique decomposition property. In rule $(\mathcal{N}6)$, the pattern information in each of the **case** branches is propagated to the occurrences of the redex variable within the corresponding branch expression. This information propagation within the **case** expression is called *unification-based information propagation* [95].

Definition 2.2.3 (Process trees)

A *process tree* is a directed acyclic graph where each node is labelled with an expression, and all edges leaving a node are ordered. One node is chosen as the *root* of the tree, and the original expression is assigned to the root.

If a node N is labelled with an expression e and $\mathcal{N}[\![e]\!] = [e_1, \ldots, e_n]$, then N has n child nodes from left to right which are labelled with the expressions $e_1, \ldots, e_n$ respectively. A process tree $e \rightarrow t_1, \ldots, t_n$ is the tree with the root labelled e and n children which are the subtrees $t_1, \ldots, t_n$ respectively. Within a process tree t, for

any node α, $t(\alpha)$ denotes the label of α. The set of ancestors of α in t is denoted by $anc(t, \alpha)$, and $t\{\alpha := t'\}$ denotes the tree obtained by replacing the subtree with root α in t by the tree t'.

A process tree is constructed from the transformation of an expression e using the following rule:

$$\mathcal{T}[\![e]\!]\ \phi = \ e \rightarrow \mathcal{T}[\![e_1]\!]\ \phi, \ldots, \mathcal{T}[\![e_n]\!]\ \phi \text{ where } \mathcal{N}[\![e]\!]\ \phi = \ [e_1, \ldots, e_n] \quad (\mathcal{T}1)$$

Thus, driving is achieved by the application of rule ($\mathcal{T}1$), which constructs a process tree using normal order reduction and unification based information propagation. The continued application of rule ($\mathcal{T}1$) may construct an infinite process tree.

To construct a residual program, we need to construct a *partial process tree* during the application of rule ($\mathcal{T}1$) by creating *repeat nodes* within the process tree t at the occurrence of an *instance* β of an ancestor α where $t(\beta)$ contains a recursive call to a function f in the redex. A repeat node corresponds to the fold step during transformation. The motivation behind the creation of repeat nodes is that the continuous unfolding of a recursive function will lead to non-termination of the transformation process if *folding* is not performed.

Definition 2.2.4 (Partial process trees)

A *partial process tree* is a process tree which contains repeat node(s). A repeat node has a *dashed edge* to an ancestor within the process tree.

Definition 2.2.5 (Instance)

An expression e is an instance of another expression e', denoted by $e' \lessdot e$, if there is a substitution θ such that $e'\theta \equiv e$ where θ gives the values for the unique free variables of e', and $\equiv$ represents the semantic equivalence between two expressions.

During the construction of a process tree, if the current expression e is an instance of the label e' of an ancestor α, then a dashed edge $e \dashrightarrow \alpha$ is created within the process tree representing the occurrence of a repeat node. This process is repeated for every occurrence of an instance of the label of an ancestor node, which leads to the construction of a partial process tree. Within a *partial process tree* t', if β is the repeat node for a matching ancestor node α, then α is called the *function node.* The possibility of creating a repeat node is only checked when the redex of the current expression is a function.

Example 3

Consider the transformation of the following program using the driving rule ($\mathcal{T}1$) with repeat nodes.

$$
\begin{array}{l}
\mathit{append}\ (\mathit{append}\ xs\ ys)\ zs \\
\text{where} \\
\mathit{append} = \lambda xs.\lambda ys.\,\mathbf{case}\ xs\ \mathbf{of} \\
\qquad\qquad\qquad\qquad\quad \mathit{Nil} \qquad\quad : ys \\
\qquad\qquad\qquad\qquad\ \mid \mathit{Cons}\ x\ xs' : \mathit{Cons}\ x\ (\mathit{append}\ xs'\ ys)
\end{array}
$$

This constructs the partial process tree as shown in Fig. 2.5.

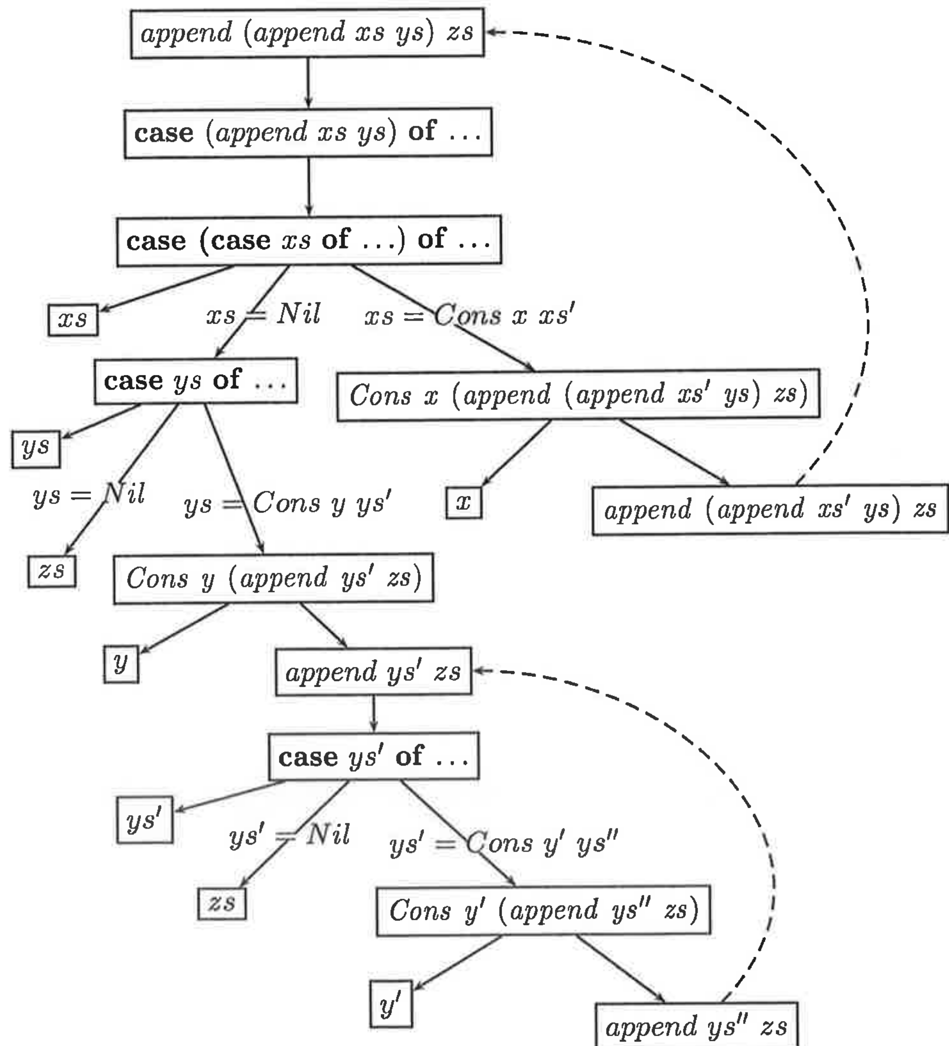

Figure 2.5: Partial process tree for $\mathcal{T}[\![\mathit{append}\ (\mathit{append}\ xs\ ys)\ zs]\!]$

The partial process tree in Fig. 2.5 has been simplified for ease of presentation. A residual program is extracted from a partial process tree using a set of rules $\mathcal{P}$ which are defined in Fig. 2.6. Each of the function nodes to which the dashed edge points, corresponds to the initial call of a function, and each of the repeat nodes from which the dashed edge originates, corresponds to the recursive call of that function in the residual program. The body of the function is constructed from the labels of the intermediate nodes.

$$\mathcal{P}[\![(v\ e_1 \ldots e_n) \rightarrow t_1, \ldots, t_n]\!] \;=\; v\ (\mathcal{P}[\![t_1]\!]) \ldots (\mathcal{P}[\![t_n]\!]) \qquad (\mathcal{P}1)$$

$$\mathcal{P}[\![(c\ e_1 \ldots e_n) \rightarrow t_1, \ldots, t_n]\!] \;=\; c\ (\mathcal{P}[\![t_1]\!]) \ldots (\mathcal{P}[\![t_n]\!]) \qquad (\mathcal{P}2)$$

$$\mathcal{P}[\![(\lambda v.e) \rightarrow t]\!] \;=\; \lambda v.(\mathcal{P}[\![t]\!]) \qquad (\mathcal{P}3)$$

$$\begin{aligned} &\mathcal{P}[\![\alpha = (con\langle f \rangle) \rightarrow t]\!] && (\mathcal{P}4)\\ &\quad = \textbf{letrec}\ f' = \lambda v_1 \ldots v_n.(\mathcal{P}[\![t]\!]) \\ &\qquad \textbf{in}\ f'\ v_1 \ldots v_n, \quad \text{if } \exists \beta \in t.\beta \dashrightarrow \alpha \text{ and } \beta \equiv \alpha\{v_1 := e_1, \ldots, v_n := e_n\} \\ &\quad = \mathcal{P}[\![t]\!], \quad \text{otherwise} \end{aligned}$$

$$\mathcal{P}[\![\beta = (con\langle f \rangle) \dashrightarrow \alpha]\!] \;=\; f'\ e_1 \ldots e_n \quad \text{if } \beta \equiv \alpha\{v_1 := e_1, \ldots, v_n := e_n\} \qquad (\mathcal{P}5)$$

$$\mathcal{P}[\![(con\langle(\lambda v.e_0)\ e_1\rangle) \rightarrow t]\!] \;=\; \mathcal{P}[\![t]\!] \qquad (\mathcal{P}6)$$

$$\begin{aligned} &\mathcal{P}[\![(con\langle \textbf{case}\ (v\ e_1 \ldots e_n)\ \textbf{of}\ p_1 : e'_1 | \ldots | p_n : e'_n \rangle) \rightarrow t_0, \ldots, t_n]\!] && (\mathcal{P}7)\\ &\quad = \textbf{case}\ (\mathcal{P}[\![t_0]\!])\ \textbf{of}\ p_1 : (\mathcal{P}[\![t_1]\!]) | \ldots | p_n : (\mathcal{P}[\![t_n]\!]) \end{aligned}$$

$$\begin{aligned} &\mathcal{P}[\![(con\langle \textbf{case}\ (c\ e_1 \ldots e_n)\ \textbf{of}\ p_1 : e'_1 | \ldots | p_n : e'_n \rangle) \rightarrow t]\!] && (\mathcal{P}8)\\ &\quad = \mathcal{P}[\![t]\!] \end{aligned}$$

Figure 2.6: Rules for residual program construction in supercompilation

---

Rule $(\mathcal{P}1)$ processes a tree with root labelled with a variable application and n subtrees $t_1, \ldots, t_n$. The result of this rule application is a variable application where the subtrees $t_1, \ldots, t_n$ are further processed to construct the arguments of the variable application. Similarly, rule $(\mathcal{P}2)$ processes a tree with root labelled with a constructor application with n subtrees $t_1, \ldots, t_n$ to construct the constructor application. Rule $(\mathcal{P}3)$ processes a tree with root labelled with a λ-expression with a single subtree t. The body of this λ-abstraction is obtained by processing the subtree t.

In rule ($\mathcal{P}4$), the root node α of the tree is labelled with an expression $con\langle f\rangle$ which contains the function f in the redex, and the root is connected with the subtree t. Two situations may arise in the application of rule ($\mathcal{P}4$) to the current tree. If there exists a node β in t such that $\beta \dashrightarrow \alpha$ and β is an instance of α, then the result of processing the current tree is to introduce a local function of the form of **letrec** $f = e_0$ **in** e_1 in the residual program. The constructed **letrec** expression contains the definition of a new function f' with an initial call $f'\ v_1 \ldots v_n$ where $v_1 \ldots v_n$ are the variables in α which are instantiated to give β. The new function f' is parameterised only by those free variables in the function node which are instantiated with different values in the repeat node(s) and hence change, so the defined local function may contain non-local variables. The definition of this new function is obtained by processing the subtree t. If there is no repeat node for α, the result of processing the tree is given by the result of processing the subtree t.

In rule ($\mathcal{P}5$), a repeat node β containing an expression $con\langle f\rangle$ is processed, which has a dashed edge to an ancestor node α. As the repeat node β is an instance of the function node α, an appropriate recursive call to the function f' (which was made at the occurrence of the function node α by rule ($\mathcal{P}4$)) is introduced in the residual program. The parameters $e_1 \ldots e_n$ in the recursive call to f' correspond to the variables $v_1 \ldots v_n$ of the original call to f'.

Rule ($\mathcal{P}6$) processes a tree with root labelled with a λ-application with a subtree t. An expression contributing to the residual program is constructed by processing the subtree t. Rule ($\mathcal{P}7$) processes a tree with root labelled with an expression containing a **case** expression in the redex where the selector of the **case** expression is a variable application $v\ e_1 \ldots e_n$. The root of the tree is connected with the subtrees $t_0, \ldots, t_n$ from left to right. A **case** expression is constructed to contribute to the residual program where the selector variable application is given by processing the first subtree t_0, and the branch terms are constructed by processing the remaining subtrees $t_1, \ldots, t_n$ respectively. Rule ($\mathcal{P}8$) processes a tree with root labelled with an expression containing a **case** expression in the redex where the selector of the **case** expression is a constructor application $c\ e_1 \ldots e_n$. An expression is constructed to contribute to the residual program by processing the single subtree t which is connected with the root.

According to the folding scheme as described above, during the supercompilation of the term *append* (*append xs ys*) *zs*, the application of the residual program construction rules P to the constructed partial process tree in Fig. 2.5 introduces

two new function definitions and their corresponding initial calls using two **letrec** expressions in the resulting residual program. The recursive function $f2$ is introduced within the residual program for the function node labelled with *append ys′ zs* (Fig. 2.5). The recursive function $f0$ is introduced within the residual program for the function node labelled with *append (append xs ys) zs* (Fig. 2.5). The residual program which is extracted from the partial process tree in Fig. 2.5 resulting from the supercompilation of the term *append (append xs ys) zs* is shown in Fig. 2.7.

```
letrec
f0 = λxs. case xs of
            Nil           : case ys of
                              Nil          : zs
                            | Cons y ys'   : Cons y ( letrec
                                                      f2 = λys'. case ys' of
                                                                   Nil            : zs
                                                                 | Cons y' ys''   : Cons y' (f2 ys'')
                                                      in f2 ys')
          | Cons x xs'    : Cons x (f0 xs')
in f0 xs
```

Figure 2.7: Constructed residual program from the partial process tree in Fig. 2.5

Termination of Supercompilation

It cannot be guaranteed that the transformation rules for supercompilation (so far as we have presented) will terminate even in the presence of folding for all input programs. The user defined functions exploited in the input program to be transformed may have divergent properties. These divergent properties will prevent the creation of repeat nodes within the process tree, so folding cannot be accomplished. In this section, we show how one can obtain termination of supercompilation by performing generalization.

To construct a partial process tree, generalization must be performed through the extraction of sub-expressions which cause successively larger expressions to be encountered during transformation. The divergence properties which are common in functional program transformations are identified in [25, 44, 94, 75].

Example 4 (Obstructing function call)

Consider the transformation of the expression *reverse xs* using the following naive definition of the list reversal function using the *append* function.

$$
\begin{aligned}
reverse = \lambda xs.\,\textbf{case}\ xs\ \textbf{of}\ & \\
& Nil \qquad\quad : Nil \\
& \mid Cons\ x\ xs' : append\ (reverse\ xs')\ (Cons\ x\ Nil)
\end{aligned}
$$

The transformation of the expression *reverse xs* encounters successively larger expressions

reverse xs, **case** (*reverse xs'*) **of**..., **case** (**case** (*reverse xs''*) **of**...) **of**..., ...

with an accumulating context. The call to *reverse* prevents the context from being reduced, so this function call is an obstructing function call. This will cause the non-termination of the transformation process. In supercompilation, the obstructing function call *reverse xs'* is extracted from its surrounding context and is transformed separately to avoid non-termination.

A recursive call to a function f is an obstructing function call if it causes a successively larger context in each unfolding.

Example 5 (Accumulating parameter)

Consider the transformation of the program given in Fig. 2.2. The transformation of the expression *reva xs Nil* encounters successively larger expressions *reva xs Nil*, *reva xs'* (*Cons x Nil*), *reva xs''* (*Cons x'* (*Cons x Nil*)) .... In each unfolding, the recursive call to the *reva* function accumulates new output in its second parameter generating a successively growing sub-expression in this parameter. So, the second parameter in the definition of *reva* is an accumulating parameter. In supercompilation, the accumulating parameter is therefore extracted automatically and is transformed separately to avoid non-termination.

A parameter in a recursively defined function f is an accumulating parameter if f accumulates output in this parameter resulting in progressively larger sub-expressions in this position on each unfolding of f.

Example 6 (Accumulating pattern)

Consider the transformation of the following program.

$$
\begin{aligned}
& plus\ x\ x \\
& \textbf{where} \\
& plus = \lambda x.\lambda y.\,\textbf{case}\ x\ \textbf{of} \\
& \qquad\qquad Zero \quad : y \\
& \qquad\quad \mid Succ\ x' : Succ\ (plus\ x'\ y)
\end{aligned}
$$

The transformation of this program encounters successively larger expressions *plus* x x, *plus* x' (*Succ* x'), *plus* x'' (*Succ* (*Succ* x'')), ... In each unfolding, the second parameter in the recursive call to the *plus* function accumulates a successively larger term by accumulating patterns. This pattern accumulation is caused by the *unification-based information propagation* used in supercompilation. This cannot be removed by rewriting as the *plus* function recurses over its first argument. In supercompilation, this problem is avoided by extracting the accumulating variable automatically and transforming it separately.

The transformation of an input program may cause an infinite sequence of transformation steps even in the presence of folding, if any of the above non-termination properties exists within the program. In all of the above cases, each of the expressions in the divergent sequence embeds a previous expression in the sequence. So, we need to detect when the current expression becomes an embedding of a previous expression, and generalize accordingly.

The form of embedding which is used to detect embedding and to perform generalization is based on the *homeomorphic embedding relation* ($\trianglelefteq$) derived from the results of Higman and Kruskal [70, 48]. The Higman and Kruskal theorem states that in an infinite sequence of terms t_0, t_1, ..., there definitely exists numbers i, j such that $i < j$ for which $t_i \trianglelefteq t_j$. If $t_i \trianglelefteq t_j$, then the term t_i is fully embedded in a larger term t_j.

Theorem 2.2.6 (Higman and Kruskal theorem) *If F is a finite set of function symbols, then any infinite sequence t_1, t_2, ... of terms in the set $T(F)$ of terms over F contains two terms t_i and t_j, where $i < j$, such that $t_i \trianglelefteq t_j$.*

We extend the notion of homeomorphic embedding relation to all of the expressions of the higher order functional language described in §2.2.1. Homeomorphic embedding is also used in term rewrite systems [35], supercompilation [103], positive supercompilation [95, 42, 46], and in distillation [45, 46] to obtain termination.

Definition 2.2.7 (Well-quasi order)

A *well-quasi order* on a set of elements S is a reflexive, transitive relation $\leq_s$ such that for any infinite sequence of elements s_1, s_2,... from S there are numbers i, j with $i < j$ and $s_i \leq_s s_j$.

Based on this relation, in an infinite sequence of expressions e_1, e_2,... which may be encountered during the transformation of an expression e, there definitely exists

some terms e_i, e_j with $i < j$ and $e_i \trianglelefteq_S e_j$. So, eventually, an embedding of e_i is found in e_j and unfolding will not be continued indefinitely. Then generalization(s) followed by folding will help to obtain termination.

A dynamic embedding detection algorithm based on the homeomorphic embedding relation is devised to monitor diverging sequences of terms during transformation of the input program, and generalization is performed to avoid possible non-termination. Homeomorphic embedding is defined more formally as follows.

Definition 2.2.8 (Homeomorphic embedding relation)

Variable	*Diving*	*Coupling*
$x \trianglelefteq y$	$\dfrac{\exists i \in \{1 \ldots n\}. e \trianglelefteq e_i}{e \trianglelefteq \sigma(e_1, \ldots, e_n)}$	$\dfrac{\forall i \in \{1 \ldots n\}. e_i \trianglelefteq e'_i}{\sigma\ (e_1, \ldots, e_n) \trianglelefteq \sigma\ (e'_1, \ldots, e'_n)}$

For variables, a variable x is embedded within another variable y. Diving detects embedding of a sub-expression within a larger expression and coupling detects embedding of all the sub-expressions of two expressions which have the same top-level functor. In diving, an expression e is embedded within any of the sub-expressions $e_1, \ldots, e_n$ of a larger expression. In coupling, all of the sub-expressions $e_1, \ldots, e_n$ are embedded within the corresponding sub-expressions $e'_1, \ldots, e'_n$ where σ is the common outermost functor of the two expressions. Some examples of homeomorphic embedding and non-embedding are shown in Fig. 2.8 where e' is the previous expression and e is the current expression.

e'	$\trianglelefteq$	e	e'	$\ntrianglelefteq$	e
$f_1\ x$	$\trianglelefteq$	$f_2\ (f_1\ x')$	$f_1\ (f_2\ x)$	$\ntrianglelefteq$	$f_1\ x$
$f_1\ x$	$\trianglelefteq$	$f_1\ (f_2\ x')$	$f_1\ (f_2\ x)$	$\ntrianglelefteq$	$f_2\ (f_1\ x)$
$f_1\ (f_2\ x)$	$\trianglelefteq$	$f_1\ (f_2\ (f_2\ x'))$			
$f_1\ y$	$\trianglelefteq$	$f_1\ (f_2\ (f_2\ y))$			
$f_1\ x\ x$	$\trianglelefteq$	$f_1\ (f_2\ x)\ (f_2\ x)$			

Figure 2.8: Examples and non-examples of homeomorphic embedding

The form of embedding as shown by the example $f_1\ x \trianglelefteq f_2\ (f_1\ x')$ in Fig. 2.8 is called *strict embedding* ($\triangleleft$) in [46]. This form of embedding is identified when a previous expression e' is found fully embedded in any of the argument e_i of an expression of the form $\sigma(e_1, \ldots, e_n)$. A real example of such an embedding is detected in the transformation of *reverse xs* in Example 4.

We extend the notion of strict embedding to deal with the embedding of **case** expressions, where the selectors of the **case** expressions are embedded within each other. Thus, $con\langle f_1\ x\rangle \lhd con\langle f_2\ (f_1\ x')\rangle$.

Generalization involves replacing sub-expressions with new variables. The expression resulting from the generalization of an expression e is expressed with a **let** expression of the form **let** $v_1 = e_1, \ldots, v_n = e_n$ **in** e_g to permanently extract the sub-expressions $e_1, \ldots, e_n$ from e, which will be transformed separately. The occurrences of the sub-expressions $e_1, \ldots, e_n$ within e are replaced with the variables $v_1, \ldots, v_n$ respectively.

Definition 2.2.9 (Generalization)

The generalization of two expressions e and e' is a triple $(e_g,\ \theta,\ \theta')$, where θ and θ' are the corresponding substitutions which are used to specialize e_g back to e and e', respectively, such that $e \equiv e_g\theta$ and $e' \equiv e_g\theta'$. The substitutions θ and θ' give the values of the sub-expressions extracted from the expressions e and e', respectively.

There is a loss of knowledge about an expression due to the extraction of sub-expressions in generalization. The *most specific generalization* causes the least possible loss of knowledge. Hence, the most specific generalization is performed in supercompilation. If there is more than one ancestor which is embedded within the current expression, then the closest ancestor is used to perform the most specific generalization.

Definition 2.2.10 (Most specific generalization (msg))

The generalization $(e_g,\ \theta,\ \theta')$ of two expressions e and e' is the most specific generalization if for every other generalizations $(e'_g,\ \theta'',\ \theta''')$, e_g is an instance of e'_g.

The most specific generalization of two expressions e and e', denoted by $e \sqcap e'$, is computed by exhaustively applying the following rewrite rules to the initial triple $(v, \{v := e\}, \{v := e'\})$ as given in [46].

$$(e,\ \{v := \phi(e_1, \ldots, e_n)\} \cup \theta,\ \{v := \phi(e'_1, \ldots, e'_n)\} \cup \theta')$$
$$\Downarrow$$
$$(e\{v := \phi(v_1, \ldots, v_n)\}, \{v_1 := e_1, \ldots, v_n := e_n\} \cup \theta, \{v_1 := e'_1, \ldots, v_n := e'_n\} \cup \theta')$$

$$(e, \{v_1 := e', v_2 := e'\} \cup \theta, \{v_1 := e'', v_2 := e''\} \cup \theta')$$
$$\Downarrow$$
$$(e\{v_1 := v_2\}, \{v_2 := e'\} \cup \theta, \{v_2 := e''\} \cup \theta')$$

The first of these rewrite rules is applied to expressions where both of the expressions have the same outermost functor. This functor is made the outermost functor of the generalized expression. The second rule looks for the common sub-expressions within an expression, and replaces them with the same variable. The rewrite rules are repeatedly applied to the arguments of the functors to obtain the generalized expression, and the sets of extracted sub-expressions. Any two expressions e_1 and e_2 are strictly embedded if and only if $e_1 \sqcap e_2 = con\langle v \rangle$.

Fig. 2.9 shows the most specific generalizations of the embedding examples of Fig. 2.8 which cover generalizations of *obstructing function calls, accumulating patterns* and *accumulating parameters.*

e'	e	e_g	θ	θ'
$f_1\ x$	$f_2\ (f_1\ x')$	v	$\{v := f_1\ x\}$	$\{v := f_2\ (f_1\ x')\}$
$f_1\ x$	$f_1\ (f_2\ x')$	$f_1\ v$	$\{v := x\}$	$\{v := f_2\ x'\}$
$f_1\ (f_2\ x)$	$f_1\ (f_2\ (f_2\ x'))$	$f_1\ (f_2\ v)$	$\{v := x\}$	$\{v := f_2\ x'\}$
$f_1\ y$	$f_1\ (f_2\ (f_2\ y))$	$f_1\ v$	$\{v := y\}$	$\{v := f_2\ (f_2\ y)\}$
$f_1\ x\ x$	$f_1\ (f_2\ x')\ (f_2\ x')$	$f_1\ v\ v$	$\{v := x\}$	$\{v := f_2\ x'\}$

Figure 2.9: Examples of most specific generalization

During supercompilation, if the current expression e is a strict embedding of a previously encountered expression e', then the current expression contains an obstructing function call. In this case, the subtree rooted at e is replaced with the result of transforming the generalized form of e in which the obstructing function call is extracted. If e is a non-strict embedding of e', then the subtree rooted at e' is replaced with the result of transforming the generalized form of e'.

Generalization introduces **let** expressions into the partial process tree. The normal order reduction rule and the residual program construction rule for the **let** expression are given by the rules ($\mathcal{N}8$) and ($\mathcal{P}9$) as shown in Fig. 2.10.

In rule ($\mathcal{P}9$), a tree rooted at a **let** expression with $n+1$ subtrees is processed. The expressions obtained by processing the subtrees $t_1, \ldots, t_n$ are substituted for the variables $v_1, \ldots, v_n$ within the expression obtained by processing the leftmost subtree t_0.

The *extract* and *abstract* operations deal with the generalization of strict and non-strict embeddings of expressions respectively, within the process tree.

$$\mathcal{N}[\![\textbf{let } v_1 = e_1, \ \ldots, \ v_n = e_n \textbf{ in } e_g]\!] \ \ \phi \ = [e_g, e_1, \ldots, e_n] \quad (\mathcal{N}8)$$

$$\mathcal{P}[\![(\textbf{let } v_1 = e_1, \ \ldots, \ v_n = e_n \textbf{ in } e_g) \rightarrow t_0, \ldots, t_n]\!] \quad (\mathcal{P}9)$$
$$= (\mathcal{P}[\![t_0]\!]) \ \{v_1 \ := \ \mathcal{P}[\![t_1]\!], \ldots, v_n \ := \ \mathcal{P}[\![t_n]\!]\}$$

Figure 2.10: Rules for **let** expression

In a strict embedding $e' \lhd e$, the generalized form of e is obtained by extracting the obstructing function call from e using the *extract* operation. The extraction of an obstructing function call e' from an expression e, denoted by $e' \leftarrow e$, is defined by applying the following rewrite rules to the initial triple $(\langle\rangle, e', e)$.

$$(con, f \ e_1 \ldots e_n, f' \ e'_1 \ldots e'_k)$$
$$\Downarrow$$
$$(con, f \ e_1 \ldots e_n, unfold(f' \ e'_1 \ldots e'_k))$$
$$\Downarrow$$
$$(con, f \ e_1 \ldots e_n, \textbf{case } e'_0 \textbf{ of } p_1 \ : \ e'_1 | \ldots | p_k \ : \ e'_k)$$
$$\Downarrow$$
$$(\textbf{case } con \textbf{ of } p_1 \ : \ e'_1 | \ldots | p_k \ : \ e'_k, f \ e_1 \ldots e_n, e'_0)$$

The generalized form of an expression e resulting from extracting an obstructing function call from it is constructed using the following *extract* operation:

Definition 2.2.11 (Extract operation)

$$\begin{aligned} extract(e', e) &= \textbf{let } v' = e'_2 \textbf{ in } (e_g\{v := con\langle v' \rangle\})\theta_e, && \text{if } match(e'_1, e'_2) \\ &= (e_g\{v := con\langle e'_2 \rangle\})\theta_e, && \text{otherwise} \\ \text{where } e' \sqcap e &= (e_g, \{v := e_1\} \cup \theta_{e'}, \{v := e_2\} \cup \theta_e) \\ e_1 \leftarrow e_2 &= (con, e'_1, e'_2) \end{aligned}$$

In the above *extract* function, $e_1 \lhd e_2$ and $\theta_{e'}, \theta_e$ give other substitutions for most specific generalization of the expressions e' and e respectively, which do not involve obstructing function calls.

If the current expression e is a non-strict embedding of a previously encountered expression e', then the generalized form of e' is obtained using the *abstract* operation.

Definition 2.2.12 (Abstract operation)

$$abstract(e', e) = \textbf{let } v_1 = e_1, \ \ldots, \ v_n = e_n \textbf{ in } e_g$$
$$\text{where } e' \sqcap e = (e_g, \{v_1 := e_1, \ \ldots, \ v_n := e_n\}, \theta)$$

If the partially constructed process tree t contains a node $\beta = con\langle f\rangle$ in which the redex is a function, supercompilation checks for the occurrence of a *repeat node*, or else applies *unfolding* or a *generalization* operation. The generalization operation introduces a *generalization node* within the process tree, which contains the generalized form of an expression. To construct a partial process tree which contains repeat nodes and generalization nodes, supercompilation is more formally defined as shown in Fig. 2.11.

$$
\begin{array}{l}
\mathcal{T}[\![\beta]\!] \ \phi = \textbf{if } t(\beta) = con\langle f\rangle \qquad\qquad (\mathcal{T}1)\\
\qquad\qquad \textbf{then if } \exists\alpha \in anc(t,\beta).t(\alpha) \lessapprox t(\beta)\\
\qquad\qquad\qquad \textbf{then } t\{\beta := t(\beta) \dashrightarrow \alpha\}\\
\qquad\qquad\qquad \textbf{else if } \exists\alpha \in anc(t,\beta).t(\alpha) \trianglelefteq t(\beta)\\
\qquad\qquad\qquad\quad \textbf{then if } t(\alpha) \lhd t(\beta)\\
\qquad\qquad\qquad\qquad \textbf{then } t\{\beta := \mathcal{T}[\![extract(t(\alpha), t(\beta))]\!] \ \phi\}\\
\qquad\qquad\qquad\qquad \textbf{else } t\{\alpha := \mathcal{T}[\![abstract(t(\alpha), t(\beta))]\!] \ \phi\}\\
\qquad\qquad\qquad\quad \textbf{else } t\{\beta := t(\beta) \rightarrow \mathcal{T}[\![unfold(t(\beta)) \ \phi]\!] \ \phi\}\\
\qquad\qquad \textbf{else } t\{\beta := t(\beta) \rightarrow \mathcal{T}[\![e_1]\!] \ \phi, \ldots, \mathcal{T}[\![e_n]\!] \ \phi\} \quad \text{where } \mathcal{N}[\![e]\!] \ \phi = \ [e_1, \ldots, e_n]
\end{array}
$$

Figure 2.11: Algorithm of supercompilation

Example 7 (Generalization of accumulating parameter)

The supercompilation of the expression $reva\ xs\ Nil$ for the program given in Fig. 2.2 encounters successively larger expressions $reva\ xs\ Nil$, $reva\ xs'\ (Cons\ x\ Nil)$, $reva\ xs''\ (Cons\ x'\ (Cons\ x\ Nil))$, ... This will cause non-termination of the transformation process if generalization is not performed.

We can see that $reva\ xs'\ (Cons\ x\ Nil)$ is a non-strict embedding of $reva\ xs\ Nil$. So, the most specific generalization of $reva\ xs\ Nil$ and $reva\ xs'\ (Cons\ x\ Nil)$ is performed to achieve termination of the supercompilation process, which results in the triple $(reva\ xs\ v, \{v := Nil\}, \{v := Cons\ x\ Nil\})$.

Now, the subtree rooted at $reva\ xs\ Nil$ is replaced with the result of transforming the generalized form **let** $v = Nil$ **in** $reva\ xs\ v$. This results in the partial process tree which is shown in Fig. 2.12.

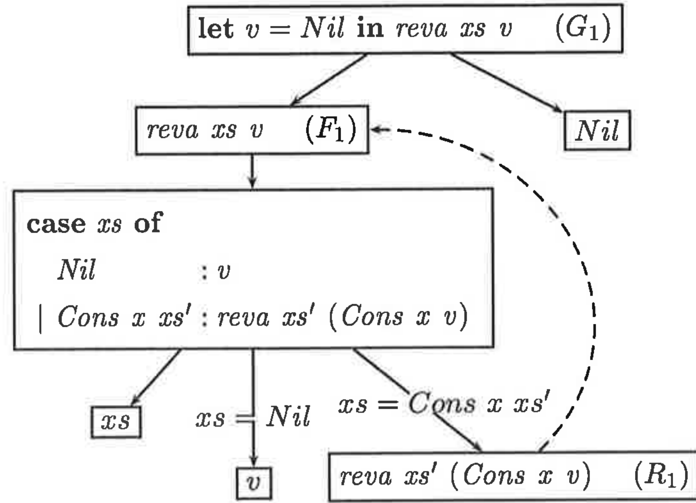

Figure 2.12: Partial process tree for $\mathcal{T}[\![reva\ xs\ Nil]\!]$

The partial process tree contains a generalization node (G_1), a function node (F_1) and a repeat node (R_1). During the transformation of the function node (F_1), the repeat node (R_1) is created at the occurrence of the instance $reva\ xs'\ (Cons\ x\ v)$ of the ancestor expression $reva\ xs\ v$.

The application of the residual program construction rules $\mathcal{P}$ to the partial process tree shown in Fig. 2.12 introduces a local function $f0$ defined with a **letrec** expression and the following residual program is constructed from this partial process tree.

$$\begin{aligned}
\textbf{letrec}\ f0 \ &= \ \lambda xs.\lambda v.\,\textbf{case}\ xs\ \textbf{of} \\
&\qquad\qquad\quad Nil \qquad\quad : v \\
&\qquad\qquad\quad \mid Cons\ x\ xs' : f0\ xs'\ (Cons\ x\ v) \\
\textbf{in}\ f0\ xs\ Nil &
\end{aligned}$$

2.3 Inductive Theorem Proving

The proofs of most inductive conjectures have a similar overall structure. To prove an inductive conjecture, one needs to select an induction schema, and an induction hypothesis for fertilization. The proofs of some conjectures may require intermediate lemmas, or a more generalized formula may need to be generated and proved to prove the original conjecture. Sometimes, the information obtained from failed proof attempts is used productively to give a successful proof of such conjectures. The selection of an induction variable, induction term and generation of the induction hypothesis are codified as a heuristic to incorporate an induction rule into a proof

system that has a greater chance of success in finding successful proofs for inductive conjectures. *Recursion analysis* [8] is one such heuristic which is used in explicit inductive theorem provers.

Logic provides a low-level explanation of a mathematical proof by showing the proof as a sequence of steps, but a high-level understanding is also needed to explain many common observations about the mathematical proofs to complement the low-level understanding. It is possible to capture the common structure in proofs by collecting similar proofs into families having similar structure [14]. For example, the common structures in theorems that require *induction* to prove them, are classified into one family. This common structure is then implemented as an *induction* heuristic, and is used to guide the proofs about such theorems. A *proof plan* represents the overall understanding of such a proof.

2.3.1 Recursion Analysis

Explicit inductive theorem provers use induction rules to prove inductive theorems. Boyer and Moore exploited the recursion-induction duality and implemented a theorem prover using structural induction to prove theorems about recursive LISP functions by evaluation and fertilization using the induction hypothesis [7]. BMTP [8, 9] uses explicit induction rules to prove inductive theorems. *Recursion analysis* is the process of constructing an appropriate induction rule and suggesting the induction variable(s) for a conjecture. Boyer and Moore invented recursion analysis [8] to use in BMTP. It was studied and extended by Stevens [97] and Walther [108] for generalizing, combining and refining induction suggestions. We describe recursion analysis for the following conjecture using the recursive definition of *even* (§1.4.1) which is adopted from [15]:

$$\forall x : nat. \forall y : nat. even(x) \wedge even(y) \rightarrow even(x + y)$$

- Recursion analysis works by recursion-induction duality. It analyses each of the recursive definitions of the conjecture to be proved to construct an induction rule.

- The preliminary analysis outputs n induction suggestions from a conjecture consisting of n recursive definitions. Each induction suggestion consists of an induction scheme corresponding to the recursion scheme of the function analysed and a potential induction variable. The base equations of each recursive definition generate the base premises of the induction rule, and the

recursive equations generate the step premises. Thus, the following induction scheme is suggested for the *even* predicate.

$$\frac{\Gamma \vdash P(0) \quad \Gamma \vdash P(Succ(0)) \quad \Gamma, n' : nat, P(n') \vdash P(Succ(Succ(n')))}{\Gamma, n : nat \vdash P(n)}$$

- BMTP [9] suggests induction based on a *measure* of the formal parameters of the recursive function appearing in the conjecture by inspecting which arguments are decreasing in the recursive call. The decreasing formal arguments of a function are called *recursive arguments*. This suggests all of the decreasing variables of the function definition as candidate induction variables. The non-variable arguments are excluded from the recursion analysis process to suggest induction even though the formal arguments are decreasing.

- Once the induction suggestions are obtained, the system tries to merge them to accommodate all of the functions. Finally, a suitable induction is chosen for the conjecture by using various heuristics.

The occurrence of a variable in the recursive argument position of a function is called *unflawed*, and the occurrence in the non-recursive argument position is called a *flawed* occurrence. Only the universally quantified variables which have no flawed occurrences are selected as candidate induction variables.

In the above conjecture, *even* recurses over its single argument in 2 steps, and $+$ is a single-stepping recursively defined function (§1.4.1) which recurses over its first argument x. The induction suggestions are summarised in Table 2.1.

Induction variables	**Functions**	**Schema**	**Recursion term**	**Status**
x	*even*	2 step	$Succ(Succ(x))$	unflawed
y	*even*	2 step	$Succ(Succ(y))$	flawed
x	$+$	1 step	$Succ(x)$	subsumed

Table 2.1: Induction suggestions by recursion analysis

In the raw induction suggestions, two different induction schemata: a 2-step and a 1-step schema are suggested for the variable x for the functions *even* and *plus* respectively. The 2-step schema subsumes the 1-step induction. As y has a flawed occurrence in the term $x + y$, the induction suggestion is cancelled (as unflawed induction is available). The problem with the flawed occurrences is that if

such occurrences are substituted in the induction term, these occurrences cannot be rewritten in the induction conclusion, which prevents the application of strong fertilization. So, recursion analysis suggests the 2-step induction rule and the induction variable x as the final induction suggestion for the above conjecture. Using the final induction suggestion, the base and step cases are formed from the input conjecture. The base and recursive equations of the functions are used as rewrite rules to rewrite these newly formed goals.

If all of the available induction variables are flawed, recursion analysis suggests all of them as induction variables. In the cases where the required induction cannot be suggested from the recursion of the functions appearing in the conjecture, an extension of recursion analysis is developed to suggest appropriate induction as presented in [19]. Recursion analysis also does not perform well in the presence of existential variables [68].

2.3.2 Rippling

Rippling [13, 15, 18, 23] is a powerful proof technique in guiding the search for an inductive proof using explicit induction. The rippling technique presented in [23] is static as the wave-rules are formed from the function definitions, lemmas, and equivalences. It uses meta-level annotations called wave-fronts to indicate the mismatch between the induction hypothesis and the induction conclusion in the step case. In the step case of an inductive proof, the induction conclusion differs from the induction hypothesis in the presence of constructors (i.e., induction terms) and wave-fronts. The wave-front represents the induction term, and the wave-hole represents the induction variable. For presentation purposes, we use boxes to represent wave-fronts, and underlines (_) to represent wave-holes. An annotated term is called a **wave term**. For example, $even(double(\boxed{Succ(\underline{n})}))$ is an annotated wave term, which is the annotated induction conclusion for the conjecture $even(double(n))$. Within the annotated term, $\boxed{Succ(\underline{\ldots})}$ is the wave-front and n is the wave-hole.

We use the following terminologies to describe the rippling process. More formal definitions of these terminologies are available in [23, 18, 15, 85].

Wave-fronts contain functions and arrows $\uparrow \downarrow$ to indicate the directions of movement. The $\uparrow$ indicates the outward directed wavefront and $\downarrow$ indicates the inward directed wavefront. Rewriting of the annotated induction conclusion using the wave-rules will move the wave-fronts in the indicated direction.

Definition 2.3.1 (Skeleton)

The parts of the annotated induction conclusion deleting the wave-fronts and their contents, but retaining the contents of the wave-holes are called the **skeleton**. This skeleton will be a renaming of the inductive hypothesis.

For example, the deletion of the wave-front $\boxed{Succ(\underline{\ldots})}$ from the annotated induction conclusion $even(double(\boxed{Succ(\underline{n})}))$ will construct the skeleton $even(double(n))$.

The **wave-rules** are annotated rewrite rules formed from the step cases of recursive function definitions, non-recursive definitions, lemmas and equivalences.

The annotated wave rules are applied successively to rewrite the annotated induction conclusion while preserving the skeleton, so that a complete copy of the induction hypothesis is found embedded within the induction conclusion, which enables strong fertilization to be performed. A dynamic version of rippling has also been developed [92]. Rippling can also apply any of the techniques, such as weak fertilization, generalization of the current goal, or discovering lemmas to unblock rippling. Rippling has also been extended to handle existential variables for program synthesis in the work on *middle-out reasoning* [68] from synthesis conjectures expressed as existential theorems.

Rippling is used to help select an appropriate form of induction for a conjecture to be proved. Various forms of rippling such as *rippling-out, rippling-in, rippling-sideways* and *rippling-across* have been embodied in the generalized ripple tactic as special cases to extend the range of theorems to be proved. All of these techniques are described in [23]. To give an account of how rippling works, we give an overview of the *rippling-out* technique.

Rippling Out

Rippling out is a heuristic control technique of step case proofs in mathematical induction. In the ripple-out technique, a tactic is used to manipulate the induction conclusion using wave-rules, so that a complete copy of the induction hypothesis is found embedded within the induction conclusion. Then, using the induction hypothesis, the induction conclusion is *fertilized*, which completes the step case proof.

Consider the proof of the *associativity of append* for lists of natural numbers, which we adopted from [15]. In demonstrating the example, we use the syntax as presented in [15].

$$\forall xs : list(nat).\forall ys : list(nat).\forall zs : list(nat).xs <> (ys <> zs) = (xs <> ys) <> zs$$

Ripple analysis suggests a 1-step list induction on xs. The induction hypothesis is: $t <> (ys <> zs) = (t <> ys) <> zs$ where t is any list of type $list(nat)$. We skip the base case proof here as it is trivial. In the step case, the induction conclusion is:

$$\vdash \quad \boxed{h :: \underline{t}}^{\uparrow} <> (ys <> zs) = (\boxed{h :: \underline{t}}^{\uparrow} <> ys) <> zs$$

The following wave-rules are used in the step case proof.

$$(\boxed{H :: \underline{T}})^{\uparrow} <> L \Rightarrow \boxed{H :: (\underline{T <> L})}^{\uparrow} \tag{2.1}$$

$$\boxed{X1 :: \underline{X2}}^{\uparrow} = \boxed{Y1 :: \underline{Y2}}^{\uparrow} \Rightarrow \boxed{X1 = Y1 \wedge \underline{X2 = Y2}}^{\uparrow} \tag{2.2}$$

These wave-rules are annotated versions of the recursive definition of <> and the replacement rule for :: [15]. Applying wave-rules (2.1) and (2.2) to the annotated induction conclusion, we obtain the following proof steps in the step case.

$$\vdash \quad \boxed{h :: \underline{t}}^{\uparrow} <> (ys <> zs) = (\boxed{h :: \underline{t}}^{\uparrow} <> ys) <> zs$$

$$\vdash \quad \boxed{h :: \underline{t <> (ys <> zs)}}^{\uparrow} = \boxed{(h :: \underline{t <> ys})}^{\uparrow} <> zs$$

$$\vdash \quad \boxed{h :: \underline{t <> (ys <> zs)}}^{\uparrow} = \boxed{h :: \underline{(t <> ys) <> zs}}^{\uparrow}$$

$$\vdash \quad \boxed{h = h \wedge \underline{t <> (ys <> zs) = (t <> ys) <> zs}}^{\uparrow}$$

Now, a complete copy of the induction hypothesis is embedded within the simplified induction conclusion, and strong fertilization is performed to obtain $h = h \wedge \top$. This is simplified to $\top$, which completes the step case proof.

Ripple Analysis

Ripple analysis is the rational reconstruction and extension of the heuristics used in recursion analysis as implemented in BMTP to select the induction variables and schemes [19]. Ripple analysis performs a one-level look-ahead into the rippling process to see what induction rules would permit the initial stage of rippling to take

place [15]. It is observed that the *flawed* induction variables (as defined by recursion analysis) in some conjectures give rise to successful proofs, whereas the *unflawed* ones cause proof failures.

Ripple analysis uses all of the available wave-rules to suggest induction variables and rules. The wave-fronts in these wave-rules suggest the form of induction. This allows the use of any function arguments as induction variables provided there is a wave-rule in which this argument contains a wave-front. Thus, ripple analysis can suggest an appropriate induction even if recursion analysis fails, which allows rippling to prove more theorems.

Termination of Rippling

The various forms of rippling direct and restrict the search for an inductive proof to avoid *combinatorial explosion.* Rippling moves the wave-front outwards until it is "beached". The proof of termination of rippling is based on a well-founded measure that decreases when outward directed wave-fronts move towards the root of a term, and inward directed wave-fronts move towards the leaves. A termination proof for rippling is given in [23, 3, 2]. In verification and synthesis problems using meta-variables and existential quantifiers, rippling may not terminate [23].

Advantages of Rippling

The main advantages of rippling compared to conventional rewritings are described as follows [15]:

1. The application of the wave-rules to an annotated term based on wave annotations match is skeleton preserving, and this always terminates.

2. Rippling provides a useful heuristic guidance in failed proof attempts by suggesting patches to a partial proof, which help in the selection of lemmas, induction rule choice and generalization.

2.3.3 Proof Planning

The rationale behind *proof planning* [13, 14, 52] is to guide an inductive theorem prover in the search for a proof leading to a probable success, and predicting probable failures by making use of explicit plans so that the proof process can be managed using different heuristics to obtain a successful proof. A proof plan is an outline or

the specification of the strategy for controlling a whole proof, or a large part of a proof.

Proof plans can provide a high-level understanding of a proof to ease the process of automatic reasoning. Proof plans are constructed to capture the common structure in proofs by collecting similar proofs into families having similar structure.

The search towards a successful inductive proof requires the application of various heuristics in an ordered sequence, for example, *induction, symbolic evaluation, unfolding, ripple-out, fertilization* etc. All of these heuristics can be implemented as programs called *tactics* which will control the application of the rewrite rules to control the search for a proof.

A **proof plan** is a tree consisting of customised tactics based on the current theorem to be proved to direct the search for a proof, which is used to reason about the current conjecture, available methods to prove it, and to facilitate the flexible application of the plan.

Components of a Proof Plan

A proof plan consists of two parts: *tactic* and *method.*

- *Tactic:* This is a procedure that applies a sequence of rules of inference at the object level. High-level tactics are defined by combining lower-level sub-tactics.
- *Method:* The specification of a tactic is called a method. It is a frame containing information about the *preconditions* for the attempted application of the tactic, and *effects* of the successful application of the tactic.

If the syntactic structure of an input formula matches the preconditions specified in the meta-logic of a method, the corresponding tactic will be applicable.

Tactic Specifications

A method expresses the specifications of a tactic with a list of slots in the frame. Fig. 2.13 lists the basic slots of a method specifying a tactic in general. Each of these slots contains a formula in the sorted meta-logic, and shows the syntactic properties of the input formula before and after the tactic application.

Each tactic has its own method to specify itself. The details of the specifications of the tactics are beyond the scope of this thesis, and are explained in [13].

Slot name	Description
Name	Specifies the name of the method and applicable argument lists
Declarations	A list of quantifiers, and sort declarations for meta-variables global to all the slots except the Tactic slot
Input	A schematic representation of the goal formula before tactic application
Preconditions	Conditions expressed in meta-logic, which must hold for the tactic to be applicable
Output	A schematic representation of the goal formula after tactic application
Effects	Properties written in meta-logic, which the output formula must satisfy after the application of the tactic
Tactic	A program which controls the application of the object-level rules of inference

Figure 2.13: Slots of a method

As an example, the specification for the induction tactic is given in Fig. 2.14 [13]. The tactic slot gives the definition of a program, *induction*, which takes the input formula, and returns the output formula. Here, *forms* is the set of all formulae, *vars* is the set of all variables over natural numbers, BFm and SFm denote formulae formed in the base case, and formulae formed in the step case with proper replacements of universal variables (with constructors) in the input formula Fm. *prim-rec-ind*$(\forall Y.Fml, Y)$ is the application of primitive recursive induction to Fml with respect to Y, and *replace-all*(S, T, Exp) constructs an expression by replacing all occurrences of S with T in Exp.

Use of Proof Plans

A method is constructed to formalise a proof plan for the current conjecture to be proved corresponding to one of the top-level tactics. In the domain of inductive theorem proving, proof planning has been implemented in the Oyster-CLAM system [20], λCLAM [87] and IsaPlanner [37, 38]. In the domain of program synthesis, some of the proof planning-based synthesis frameworks are syntheses of functional programs by Smaill and Green [91], syntheses of recursive programs by Armando, Smaill and Green [1], and middle-out synthesis by Kraan, Basin and Bundy [68].

Slot name	Specification
Name	*Induction*
Declarations	$\forall X \in vars$, $\forall Fm \in forms$, $\forall BFm \in forms$, $\forall SFm \in forms$
Input	$\forall X.Fm$
Preconditions	nil
Output	$BFm \wedge \forall X(Fm \longrightarrow SFm)$
Effects	$BFm = \mathit{replace\text{-}all}(X, 0, Fm) \wedge$ $SFm = \mathit{replace\text{-}all}(X, Succ(X), Fm)$
Tactic	$\mathit{induction}(\forall Y.Fml) = \mathit{prim\text{-}rec\text{-}ind}(\forall Y.Fml, Y)$

Figure 2.14: Induction method

Advantages

Proof planning extends the tactic-based theorem proving paradigm through the explicit representation of proof strategies which reduces the search control problem. Some key benefits of the proof planning approach to the development of proof strategies are clarity, modularity, reliability, flexibility, re-usability, and increased automation.

2.3.4 Proof Critics

An inductive proof often fails because of improper use of induction, lacking appropriate rewrite rules or failing to perform appropriate generalization. To learn from failed proof attempts, the standard patterns of proof failure and appropriate patches to the failed proof attempts are represented as *critics.* Several approaches to the productive use of proof failures in inductive proofs are described in [52, 53, 107, 54]. These techniques study the divergence pattern of a failed proof attempt, and automatically suggest ways to recover from the failure leading to a successful proof.

Use of Planning Critics

Ireland has proposed an extension of proof plans by using *proof critics* to exploit partial success or failure in the search for an inductive proof [52]. A critic is a small program which identifies problems as well as providing solutions to the problems in a failed proof plan. Proof critics are used to capture patchable exceptions to a tactic and hence to the basic proof plan. Each critic is associated with a method.

The failure or partial success of a method activates its associated critics, which is determined by the critic preconditions.

In exploiting partial success in a failed proof plan, Ireland [52] analyses the success and failures of the preconditions to identify the causes of failure, for example, *missing wave rule* or missing *sink* etc. He then uses this information to propose proof critics to initiate the search for a lemma corresponding to the missing wave-rule or perform online generalization through the introduction of a sink by providing appropriate patches.

The work of Ireland and Bundy [53] is based on the concept of proof critics in a proof planning framework. They have presented a novel architecture to automatically discover eureka steps like refining induction, missing lemmas, generalization etc. by systematic analysis of the failure of rippling by using corresponding critics. For example, the failure of the inductive proof of $\forall xs.reverse(xs) = reversea(xs, Nil)$ (i.e., *reversea* is the accumulative version of *reverse*) because of a lack of proper generalization during rippling can be captured as a critic. The solution provided by the *patch* associated with the critic is the generalization of the goal through the introduction of an accumulator variable into the original conjecture. Using the notion of proof critic, Ireland and Bundy have extended the proof critic mechanism for accumulator generalization involving multiple sinks [54]. This technique is built upon the technique of patching proofs used in [53], but greatly extends its power.

Divergence Critic

The divergence critic [107] is a computer program to monitor the construction of inductive proofs to identify diverging proof attempts. It identifies when and how the proof attempt is diverging by means of a difference matching procedure. The critic then proposes appropriate lemmas and generalizations that guide the proof successfully without divergence. In the SPIKE system [66], this critic has been implemented and a number of diverging theorems were proved successfully.

While the proof process is continued, an accumulating term structure may appear that causes the divergence. The difference matching technique identifies the accumulating term structure causing divergence. Difference matching and rippling are both used to propose lemmas that ripple out the accumulating term structure.

Consider the proof of the following conjecture as illustrated in [107].

$$\forall n : nat.double(n) = n + n$$

The following rewrite rules are used in the proof of the above conjecture.

$$X + 0 \Rightarrow X \qquad double(0) \Rightarrow 0$$
$$X + Succ(Y) \Rightarrow Succ(X + Y) \qquad double(Succ(X)) \Rightarrow Succ(Succ(double(X)))$$
$$Succ(X) = Succ(Y) \Rightarrow X = Y$$

A 1-step induction for *nat* is applied on x. The base case is trivial. In the step case, the induction hypothesis is, $double(x) = x+x$, and the induction conclusion is, $double(Succ(x)) = Succ(x) + Succ(x)$. Rewriting of the induction conclusion using the above rewrite rules, and then fertilizing the left hand side using the induction hypothesis results in the proof term $Succ(x + x) = Succ(x) + x$. This equation cannot be simplified any further. So, the prover tries to apply another induction on x, which generates the following diverging sequence:

$$\begin{array}{rcl} Succ(x+x) & = & Succ(x) + x \\ Succ(Succ(x+x)) & = & Succ(Succ(x)) + x \\ Succ(Succ(Succ(x+x))) & = & Succ(Succ(Succ(x))) + x \\ & \vdots & \end{array}$$

The cause of this divergence is the lack of the rewrite rule

$$Succ(X) + Y = Succ(X + Y)$$

which is needed to remove the *Succ* function accumulating in the first argument position of $+$ in the right hand side. The prover repeatedly performs an induction on x, but it is unable to simplify. This rewrite rule can be derived from the lemma $\forall x : nat.\forall y : nat.Succ(x) + y = Succ(x + y)$.

To recognize *when* a proof attempt is diverging, the critic looks for diverging patterns in the proof attempt. It first determines the sequence of equations which the prover tries to prove by repeated induction on x as shown above. The critic then tries to identify the accumulating and nested term structure which is causing divergence by *difference matching* [107] of the successive equations of the divergence sequence.

The divergence critic suggests the following wave rule:

$$\boxed{Succ(\underline{X})} + Y = \boxed{Succ(\underline{X + Y})}$$

The annotated induction conclusion $double(\boxed{Succ(\underline{x})}) = \boxed{Succ(\underline{x})} + \boxed{Succ(\underline{x})}$ can be proved without divergence by using the above wave rule along with the wave rules that are obtained from the rewrite rules.

There are some limitations to the divergence critic as pointed out in [107]. Identifying divergence is undecidable in general. This critic may identify a divergence even if there is none at all, or even if it identifies a divergence correctly, it may not be able to suggest appropriate lemmas.

2.4 Program Synthesis

A program is often written with respect to a specification, and it is assumed that this program will satisfy the specification. Only a formal proof of correctness can guarantee that a program meets its specification. A program should be developed in such a way that it must behave according to the specification [99].

In this section, we present an overview of the research that has been undertaken in various program synthesis methods in functional and logic programming as declarative programs are easier to analyze and reason about. A survey of proof planning based synthesis methods is presented in [86]. A survey of existing work on constructive, deductive and inductive synthesis of logic programs is presented in [36].

2.4.1 Constructive Synthesis

Classical logic, which is the standard foundation of mathematics, is based on *truth functional semantics*, and allows the *law of the excluded middle* ($A \vee \neg A = \top$) as an axiom. In this semantics, every proposition is either true or false. A proof of a proposition asserts the existence of an object without showing how it can be constructed.

In constructive logic, the proof of a proposition has a computational content using constructive derivation. The *law of the excluded middle* is not valid in this logic, so, for a general proposition A, $A \vee \neg A$ is not provable. The *pure existence proof* of $\exists y.spec(y)$ in classical logic is replaced with a constructive proof which involves the construction of the object y and showing that the specification *spec* holds for y; the result is a pair (y, p). The proof of $\forall x.spec(x)$ is a function taking any object x to a proof of $spec(x)$. The constructive proof of $\forall x.\exists y.spec(x, y)$ will construct a function $f(x)$ which will compute the *witness* y in terms of x so that $\forall x.spec(x, f(x))$. The function takes a as a value of x to compute a value $f(a)$ which satisfies $spec(a, f(a))$. So, $\forall x.\exists y.spec(x, y)$ is the specification to construct the program $f(x)$ which satisfies the specification.

Types originate from programming languages and propositions from logic. Constructive type theory, based primarily on the work of Martin-Löf [77], is simultaneously a logic and a programming language in which propositions and types are analogous based on certain assumptions. The logicians Curry and Howard observed the correspondence between proofs and programs: propositions and types are duals forming the notion of *propositions-as-types* which is known as the Curry-Howard isomorphism [50, 33]. The specification $p : P$ means both that p is of type P and p is the proof of the proposition P.

Curry-Howard isomorphism links typed lambda calculus and constructive logic. Using this isomorphism, proofs become terms of lambda calculus. Thus, each proof rule of constructive logic has a corresponding program formation rule. In addition to these rules, there are also induction rules for inductive data types. These rules are the basis of constructive type theory, i.e., a formal system where program development and verification are done hand-in-hand. The following are some rules of constructive type theory:

$$\frac{\vdash A \ is\ a\ type}{\vdash x\colon A} AS \qquad \frac{x\colon A \in \Gamma}{\Gamma \vdash x\colon A}(Hyp) \qquad \frac{\Gamma, x\colon A \vdash e\colon B}{\Gamma \vdash (\lambda x\colon A).e\colon (A \Rightarrow B)}(\Rightarrow I)$$

$$\frac{\Gamma \vdash e_1\colon A \Rightarrow B \quad \Gamma \vdash e_2\colon A}{\Gamma \vdash (e_1\ e_2)\colon B}(\Rightarrow E) \qquad \frac{\Gamma, x\colon A \vdash p\colon P}{\Gamma \vdash (\lambda x\colon A).e\colon (\forall x\colon A).P}(\forall I)$$

To synthesise a program p from the specification $A \Rightarrow (B \Rightarrow A)$ for types A and B such that $\vdash p : A \Rightarrow (B \Rightarrow A)$, the program is represented with a metavariable. The proof rules are applied to the specification from the conclusion to the premises of each rule [99, 4]. The application of the rule $\Rightarrow I$ twice to the specification results in the following steps:

$$\cfrac{\vdash M : A \Rightarrow (B \Rightarrow A)}{\cfrac{x : A \vdash M_1(x)\colon (B \Rightarrow A)}{x : A, y : B \vdash M2(x, y)\colon A}}$$

M is unified with $\lambda x.M_1(x)$, and $M_1(x)$ is unified with $\lambda y.M_2(x, y)$. The application of Hyp unifies $x\colon A$ with $M_2(x, y)\colon A$, instantiating $M_2(x, y)$ to x which completes the proof. Substitutions result in the synthesised program $\lambda x.\lambda y.x$.

2.4.2 Deductive Synthesis

The purpose of deductive program synthesis is to derive an executable program from a high level specification by applying correct inference rules.

Deductive Synthesis of Logic Programs

Deductive synthesis can deduce logic programs from a specification using some predefined deduction rules [36]. In order to synthesise a correct program from a specification, the deduction strategy involves theorem proving methods. The synthesis process starts with a pair ($\mathcal{M}$,$\mathcal{Q}$) where $\mathcal{M}$ is a set of axioms, containing the logic specification, and $\mathcal{Q}$ is the query for which a logic program will be deduced by correct inference rules from $\mathcal{M}$.

The work of Lau et al. for logic program synthesis in [71] is an unfold/fold based semi-automatic system. This approach provides a partially correct program. The specification is an *iff* formula, and $\mathcal{M}$ is a set of *iff* formulas. The *iff* formula is a definition. The method employs fold-unfold in a strictly top-down manner. To synthesise a recursive logic procedure from a given first-order logic specification (with definitions in normal form) with a given head of the procedure, head of the required implication and form of the required set of recursive calls, they start by defining an initial problem called *folding problem* involving $\mathcal{Q}$. This folding problem is decomposed into subproblems until the subproblems are easily solved. The subsolutions are then composed into a solution to the initial fold problem. Unfolding is performed only when it contributes to a fold. In this system, the initial fold problem corresponds to selecting the type of induction, initial unfolding corresponds to induction, and the final folding to fertilization.

Deductive Synthesis of Functional Programs

A deductive framework to synthesise functional programs using a derivation proof method from an input specification, describing the relation between the input and output of the desired program, is presented in [73, 74]. The framework incorporates ideas from resolution and inductive theorem proving for both interactive and automatic implementation. They adopt classical logic, but restrict it to be constructive whenever necessary in a deductive-tableau proof system to extract programs from proofs. In this synthesis system, for a given specification of the form $f(a) \Leftarrow$ *find* z *such that* $spec(a, z)$, a theorem of the form $(\forall a)(\exists z).spec(a, z)$ is proved to extract

a program of the form $f(a) \Leftarrow t[a]$ that meets the specification. The proof is sufficiently constructive to indicate a computational method to find the output z in terms of input a. Thus, the system proves the existence, for any input, of an output that satisfies the specified conditions in a background theory.

2.4.3 Middle-Out Synthesis

Proof planning can help to guide the search at the meta-level of a synthesis proof by making a plan of the object level proof. Synthesis conjectures are expressed using existential variables to represent output values, and allow undefined functions in the conjecture. This makes the verification proof and synthesis process difficult as the required induction cannot be determined. Middle-out reasoning as a part of proof planning was first proposed in [22] to solve this problem by postponing the selection of induction scheme until late in the proof. Middle-out reasoning represents unknown terms that are to be synthesised by meta-variables. It allows the meta-level representation of an object-level term and middle-out reasoning helps proof planning to proceed even though an object is not fully known. The meta-variables may not always be instantiated properly to correct programs.

Middle-Out Reasoning in Functional Program Synthesis

An application of middle-out reasoning with proof planning and rippling to synthesise functional programs in the context of the formula-as-types principle is presented in [91]. A recursive functional program can be synthesised from the inductive proof of a specification of the form $\forall input.\exists output.spec(input, output)$ in constructive type theory derived from Constable's Nuprl [31]. A meta-variable is used to stand for the existential witness term so that it will be subsequently instantiated to an object level-term as the proof progresses. Proof planning is used for induction as the basis of the synthesis approach. Rippling is used to manipulate the goal so that an induction hypothesis can be applied for fertilization in the step case. In [1], Armando et al. present an automatic technique for inductive synthesis of recursive functional programs from non-executable *input/output* specifications of the form $\forall x.\exists y.spec(x, y)$. This technique uses *proof plans* [13] with some extensions and generalizations to guide the synthesis process. The work is based on Martin-Löf's constructive type theory [77] to achieve total correctness of the synthesised program. In their work using middle-out reasoning, meta-variables are used to instantiate unknown programs, so the development of a program and its proof is done hand-in-hand.

Middle-Out Reasoning in Logic Program Synthesis

In [67] and [68], Kraan et al. extensively used middle-out reasoning to synthesise logic programs, and in the selection of an appropriate induction scheme. Synthesis is performed by planning the verification of a program while leaving the program unknown represented by a meta-variable. In this synthesis planning, the proof steps that depend on the program are postponed as long as possible to partially instantiate the program. The base case of the synthesis proof allows the instantiation of the base case of the program, and the step case of the program is obtained from the step case of the proof.

Recursion analysis [9]/ripple analysis [15, 19] can find induction only if all of the functions are defined and required lemmas are available. For example, in the following conjecture, *even* is undefined. Recursion analysis or ripple analysis fails to determine the appropriate induction for this conjecture because the induction scheme may correspond to the recursion scheme of the undefined function.

$$\text{ALL } x : nat.(even(x)) \leftrightarrow (\text{EX } y : nat.y \times Succ(Succ(Zero)) = x)$$

However, middle-out reasoning [22] can find appropriate induction schemes in such cases. The selection of induction is postponed in the planning process, and a schematic step case is formed by replacing the potential induction variable with a constructor represented by a meta-variable applied to this potential induction variable in the induction conclusion. Rippling of the schematic step case fully instantiates the meta-variable with the appropriate induction type after fertilization has been performed.

Kraan et al. identified the possibility of non-termination of the rewriting process in middle-out induction in the presence of meta-variables using rippling. They also identified that speculative rippling may lead to non-termination in failure branches.

2.4.4 Inductive Synthesis

Inductive synthesis uses artificial intelligence techniques to synthesise programs from incomplete information, such as examples, by means of inductive inference [36]. The purpose of inductive synthesis is to formulate general rules. Inductive inference is related to generalization, whereas deductive synthesis is related to specialisation.

The inductive synthesis of logic programs starts with a logic specification expressed with a set of examples, and an intended relation. The synthesised program

must be consistent with respect to the specification, and must also cover the unspecified examples in the case of incomplete specification.

The inductive synthesis methods are classified as traced-based approaches or model-based approaches. The trace-based approach uses folding, matching and generalization to synthesise a generalized program containing loops and recursion. The trace-based approach has received much attention in the context of functional programming [93].

An inductive synthesis method to synthesise recursive functional program from input/output examples based on the recurrence-detection method of Summers [98] is presented in [64].

The model-based approach constructs a finite axiomatisation of a model of the examples. Plotkin's idea of least general generalization [82, 83] is the basis of most model-based approaches of logic program synthesis.

2.5 Inductive Theorem Proving Using Program Transformation

Automatic program transformation techniques, such as supercompilation and distillation, are capable of metacomputation using rules and strategies in metalanguage. These techniques can be used in proving inductive theorems. *Metacomputation* is an alternative to formal logic in automated theorem proving. General propositions which require quantification, and proofs using mathematical induction, can be handled with metacomputation [103, 42, 41].

For a given program defined with a term and recursive definitions, it is possible to show that the given definitions satisfy the specification described by the term. In this section, we give an overview of the use of metacomputation to prove inductive theorems using program transformation techniques.

2.5.1 Metacomputation

Metacomputation is one level higher than ordinary computation, where programs are treated as data objects. For example, program specialization is a metacomputation task. The programs which have the capability to perform metacomputation are *metaprograms*. The application of a metaprogram M to a program *prog* is defined as $<$ M *prog* $>$.

A *metasystem* is defined as a system which integrates, controls and processes

other systems as objects. The step from an initial program *prog* to the application of a metaprogram M to an encoded form $\overline{prog}$ of *prog* is called a *metasystem transition.* A *multi-level metasystem hierarchy* can be obtained by repeated use of metasystem transitions. A formal description of metacomputation and metasystem transitions is given in [42, 41].

2.5.2 Partial Evaluation

In [58], Julia has shown the development of proofs by structural induction about program transformations using *partial evaluation.* The parts of the transformation that depend on static data are unfolded, and those parts that depend on the dynamic data are residualized and simplified using the induction hypothesis.

In [58], Julia has used a partial evaluator to automate inductive proof using *Scheme.* She proved the *associativity of append* theorem by case analysis on the input variable xs, and simplifying the corresponding base and step cases using the definition of the *append* function with the help of a partial evaluator.

$$append\ (append\ xs\ ys)\ zs \ = \ append\ xs\ (append\ ys\ zs)$$

In the base case proof of the above theorem, the static value [] and in the step case proof, the partially static value $x :: xs$ where both x and xs are dynamic, were used to construct the corresponding cases based on the structure of xs. The sub-goals resulting from the substitutions were simplified using Consel's partial evaluator *Schism.* Schism does not unfold an application if all of the arguments are dynamic. The details of this proof system is given in [58].

2.5.3 Supercompilation

Metacomputation provides an alternative method of automated theorem proving using formal logic. In this paradigm, the computation process can be fully mechanized using unfold-fold based program transformation. The ability of the supercompiler [101] to perform a deep transformation of the function definitions can be used in *theorem proving.* To prove that a certain property P holds for all values of x expressed by the logical formula $\forall x.P(x)$, we can transform the original definition of $P(x)$ to $True$ using supercompilation. The use of supercompilation in theorem proving presented in this section is based on the work in [102, 81, 94, 42, 41].

Proving Logical Formulae Using Metasystem Transitions in Conjunction with Supercompilation

Logical formulae which are universally and existentially quantified require, computationally, a metasystem transition. Let **ALL** and **EX** be two functions which use the supercompiler to prove universally and existentially quantified conjectures respectively.

To prove a universally quantified formula $\forall x.prog(x)$, the following metacomputation takes place. If the supercompiler constructs a function where all of the exit points are *True*, it outputs *True*. Otherwise, it outputs $\perp$.

$$< \textbf{ALL} \ \ \overline{prog(x)} >= \begin{cases} True, & \forall x.prog(x) \ is \ proven \\ \perp, & Unproved \end{cases}$$

To prove an existentially quantified formula $\exists x.prog(x)$, the following metacomputation takes place. The function **EX** constructs a potentially infinite process tree by driving *prog* using the breadth-first principle. If it finds any *True* exit point, it outputs *True*. Otherwise, it continually searches until it is stopped.

$$< \textbf{EX} \ \ \overline{prog(x)} >= \begin{cases} True, & \exists x.prog(x) \ is \ proven \\ is \ stopped, & Unproved \end{cases}$$

To prove a conjecture $\forall x.\exists y.f \ x \ y$, the following metasystem transition scheme is constructed.

$$\begin{array}{l} < \textbf{ALL} \ \ \ldots\ldots\ldots\ldots\ldots > \\ \quad < \textbf{EX} \ \ \ldots.x\ldots. > \\ \qquad\quad < \boldsymbol{f} \ \bullet \ y > \end{array}$$

In the above scheme, the function **EX** performs driving of $f \ x \ y$ where x is free. Then, the function **ALL** applies supercompilation on the resulting term obtained.

Example 8

Consider the proof of the *associativity of plus* theorem for natural numbers.

$$\text{ALL } x.\text{ALL } y.\text{ALL } z.eqnum \ (plus \ (plus \ x \ y) \ z) \ (plus \ x \ (plus \ y \ z)) \quad (8.1)$$

The following definition of the function *eqnum* and the definition of *plus* as defined in §2.2.4 are used to prove the above conjecture.

```
eqnum = λx.λy.case x of
                Zero    : case y of
                            Zero    : True
                          | Succ y' : False
              | Succ x' : case y of
                            Zero    : False
                          | Succ y' : eqnum x' y'
```

The supercompiler transforms the body of conjecture (8.1) to obtain the following program.

```
letrec
f0 = λx. case x of
            Zero    : case y of
                        Zero    : case z of
                                    Zero    : True
                                  | Succ z' : letrec
                                                f1 = λz'. case z' of
                                                            Zero     : True
                                                          | Succ z'' : f1 z''
                                              in f1 z'
                      | Succ y' : letrec
                                    f1 = λy'. case y' of
                                                Zero     : case z of
                                                             Zero    : True
                                                           | Succ z' : letrec
                                                                         f2 = λz'. case z' of
                                                                                     Zero     : True
                                                                                   | Succ z'' : f2 z''
                                                                       in f2 z'
                                              | Succ y'' : f1 y''
                                  in f1 y'
          | Succ x' : f0 x'
in f0 x
```

By inspecting the above term, we see that all of the exit points from the term are *True*. Turchin requires that all of these functions are total, and thus guaranteed to terminate. Supercompilation recognizes that the above term is transformable to *True*. This proves the associativity of plus theorem.

2.6 Use of Lemmas and Generalization Techniques

In this section, we discuss how the concept of cut elimination relates to proof and program transformation techniques. Also, we briefly consider the use of intermediate

lemmas and generalization in inductive proofs.

2.6.1 Cut Elimination

The inability to generate and prove a well-founded ordering for the non-trivial recursive data types, and the inability to generate new induction rules based on that order by computer programs, limits the power of automatic inductive theorem provers. The *cut rule* is therefore required to propose intermediate lemmas and for the generalization of conjectures (§1.4.2). Gentzen's original formalisation of sequent calculus contains the cut rule ($Cut(1)$ in Fig. 2.15).

Despite strong arguments in favour of the fact that inductive theories do not admit cut elimination, some research demonstrates cut elimination (cut-free proof) in the presence of induction in first order intuitionistic logic [76, 78, 100] and also in classical logic [10, 12, 11]. Some research [45, 103, 102] in the field of inductive theorem proving in functional programming using metacomputation techniques has been carried out that does not make use of any explicit intermediate lemmas, whereas some other existing inductive proof techniques need to use explicit lemmas in such proofs.

Simon Marlow's research [75] draws a relationship between the ideas of program deforestation and cut elimination. The goal of deforestation is to eliminate the intermediate data structures by reducing it to a normal form. Marlow reformulates first order deforestation making it similar to the formulation of cut elimination, and combined it with the non-recursive cut elimination algorithm. Marlow uses **let** to represent those data structures that will not be removed by deforestation, and **cut** to represent eliminable data structures. The term form corresponding to the cut rule of logic is the *cut construct* ($Cut(2)$).

$$\frac{\Gamma, A \vdash \Delta \qquad \Gamma \vdash A}{\Gamma \vdash \Delta} Cut(1) \qquad\qquad \frac{\Gamma \vdash t \colon A \qquad \Gamma, x \colon A \vdash u \colon B}{\Gamma \vdash \mathbf{cut} \quad x = t \quad \mathbf{in} \quad u \colon B} Cut(2)$$

Figure 2.15: Cut rule ($Cut(1)$) and the cut construct ($Cut(2)$)

In [30], Cockett argued that Wadler's deforestation technique [104] and Burstall/Darlington's unfold/fold transformation [24] are necessarily shadows of an underlying cut elimination procedure, and should be more generally recognized as proof techniques.

2.6.2 Use of Lemmas and Generalization in Induction

Various heuristics are used to suggest intermediate lemmas and for generalization. Rippling [23] uses rich heuristics to conjecture lemmas to design new wave rules to unblock rippling, and to generalize goals so that wave rules apply.

Use of Lemmas in Inductive Proof

To explain the use of intermediate lemmas in inductive proof, we consider the inductive proof of the *commutativity of addition* theorem given by the following conjecture using standard rewriting techniques. The rewrite rules of §1.4.1 are used in this proof.

$$\text{ALL } x : nat.\text{ALL } y : nat.x + y = y + x$$

Both of the variables x and y are universally quantified. The variable x has one *unflawed* and one *flawed* occurrence, and the variable y also has one *unflawed* and one *flawed* occurrence. Recursion analysis suggests both of them as induction variables. However, we use x as the induction variable in this case. We perform a 1-step induction on x. The induction hypothesis is: $x + y = y + x$. The base case is simplified as follows.

$$\begin{array}{lll} \vdash & 0 + y = y + 0 & \text{(by base case premise of induction rule (1.1))} \\ \vdash & y = y + 0 & \text{(by rewrite rule (i))} \end{array}$$

To prove the subgoal $y = y + 0$, we perform 1-step induction on y. We assume the induction hypothesis $y = y + 0$. In the base case, $0 = 0 + 0$, which is simplified to $0 = 0$ by rewrite rule (i). In the step case, the induction conclusion is constructed and it is simplified as follows.

$$\begin{array}{llll} y = y + 0 & \vdash & Succ(y) = Succ(y) + 0 & \text{(by step case premise of} \\ & & & \text{induction rule (1.1))} \\ & \vdash & Succ(y) = Succ(y + 0) & \text{(by rewrite rule (ii))} \\ & \vdash & y = y + 0 & \text{(by rewrite rule (iii))} \end{array}$$

Now, the simplified induction conclusion contains a complete copy of the induction hypothesis. So, strong fertilization can be performed. This simplifies the step case proof of the subgoal $y = y + 0$ to $\top$. The simplification of the induction conclusion for the original conjecture proceeds as follows.

$$x+y=y+x \quad \vdash \quad Succ(x)+y=y+Succ(x) \quad \text{(by step case premise of induction rule (1.1))}$$
$$\vdash \quad Succ(x+y)=y+Succ(x) \quad \text{(by rewrite rule (ii))}$$

The proof is stuck at this point. No further rewriting of this partially simplified induction conclusion is possible. This unsolved goal does not suggest any generalization. To simplify this, we need the following rewrite rule:

$$y+Succ(x)=Succ(y+x) \quad \text{(iv)}$$

The right hand side of $Succ(x+y)=y+Succ(x)$ suggests a lemma of the form $\forall y : nat.\forall x : nat.y+Succ(x) = Succ(y+x)$. This lemma is used to derive the above rewrite rule which is used to simplify the unsolved goal. Before using this rewrite rule, we prove the lemma $\forall y : nat.\forall x : nat.y+Succ(x) = Succ(y+x)$. Recursion analysis suggests a 1-step induction on y. The induction hypothesis is: $y+Succ(x)=Succ(y+x)$. The base case is simplified as follows.

$$\vdash \quad 0+Succ(x)=Succ(0+x) \quad \text{(by base case premise of induction rule (1.1))}$$
$$\vdash \quad Succ(x)=Succ(x) \quad \text{(by rewrite rule (i))}$$
$$\vdash \quad x=x \quad \text{(by rewrite rule (iii))}$$

In the step case, the induction conclusion is constructed and it is simplified using the rewrite rules (ii) and (iii) as follows.

$$y+Succ(x)=Succ(y+x) \quad \vdash \quad Succ(y)+Succ(x)=Succ(Succ(y)+x)$$
$$\vdash \quad Succ(y+Succ(x))=Succ(Succ(y+x))$$
$$\vdash \quad y+Succ(x)=Succ(y+x)$$

Now, the simplified induction conclusion contains a complete copy of the induction hypothesis. So, strong fertilization is performed. This simplifies the step case proof of the lemma to $\top$. Now, we complete the remaining step case proof of the commutativity of addition theorem using the new rewrite rule with the existing set of rewrite rules.

$$x+y=y+x \quad \vdash \quad Succ(x+y)=y+Succ(x)$$
$$\vdash \quad Succ(x+y)=Succ(y+x) \quad \text{(by rewrite rule (iv))}$$
$$\vdash \quad x+y=y+x \quad \text{(by rewrite rule (iii))}$$

Now, the induction conclusion contains a complete copy of the induction hypothesis. By performing strong fertilization, this is simplified to $\top$. This proves the theorem.

Generalization in Inductive Proof

The need for generalization of a goal is also a consequence of the failure of cut elimination in inductive theories. To see how generalization helps to achieve a successful proof in a diverged proof attempt, consider the proof of the following conjecture which is a variant of the associativity of $<>$ theorem using standard rewriting techniques. We adopt this example from [15].

$$\forall xs : list(\tau).xs <> (xs <> xs) = (xs <> xs) <> xs$$

We consider only the step case proof using the following rewrite rules.

$$(H :: T) <> L \Rightarrow H :: (T <> L) \quad \text{(i)}$$
$$X1 :: X2 = Y1 :: Y2 \Rightarrow X1 = Y1 \wedge X2 = Y2 \quad \text{(ii)}$$

Recursion analysis suggests a 1-step list induction rule (1.2) on xs, though the 3rd, 5th and 6th occurrences of xs are *flawed.* The induction hypothesis is:

$$t <> (t <> t) = (t <> t) <> t$$

In the step case proof, the simplification of the induction conclusion proceeds as follows:

$$\vdash \quad (h :: t) <> ((h :: t) <> (h :: t)) = ((h :: t) <> (h :: t)) <> (h :: t)$$
$$\vdash \quad h :: (t <> (h :: (t <> (h :: t)))) = (h :: (t <> (h :: t))) <> (h :: t)$$

No further simplification is possible, which causes the proof procedure to fail. Only *generalizing apart* of the 2nd, 3rd, 5th and 6th positions of the original conjecture by introducing a new universally quantified variable ys can help the proof to go through. This will generate the new conjecture as given below.

$$\forall xs : list(\tau).\forall ys : list(\tau).xs <> (ys <> ys) = (xs <> ys) <> ys$$

Now, recursion analysis will suggest a 1-step list induction on xs, but this time it is unflawed, as both of its occurrences are in the recursive argument position of the function $<>$. The above generalized conjecture can be proved successfully using strong fertilization.

2.7 Conclusion

In this chapter, we have presented the background research on the state of the art work in the areas of program transformation, automatic inductive theorem proving techniques and strategies, program synthesis, and metacomputation based theorem proving using program transformation. We have discussed the limitations of inductive inference and the use of the cut rule to introduce intermediate lemmas and to perform appropriate generalizations while preventing over-generalization, which may cause infinite branching points into the search space.

Burstall and Darlington's unfold/fold program transformation technique is a semiautomatic user-guided transformation system. They use associativity or commutativity properties of primitives as laws when folding is possible to generate efficient programs. Supercompilation is more powerful than partial evaluation and deforestation. Supercompilation is a fully deterministic transformation algorithm. Over-generalization occurs a lot in supercompilation. The unification-based information propagation in supercompilation makes it appropriate for metacomputation-based inductive theorem proving.

Among the various inductive proof techniques, rippling is a powerful knowledge-based theorem proving technique using explicit induction. Rippling provides a useful heuristic guidance in failed proof attempts in the selection of lemmas, induction rule choice and generalization. Ripple analysis can suggest an appropriate induction even if Boyer and Moore's recursion analysis fails, which allows rippling to prove more theorems. In middle-out synthesis, a meta-variable is instantiated to unknown program when the proof is completed. The rewriting process may not always terminate in middle-out induction in the presence of meta-variables using rippling.

Metacomputation provides an alternative to inductive theorem proving using explicit induction. Turchin has shown the use of metasystem transition in theorem proving using supercompilation. This technique has not been studied in depth to prove existential theorems and in the construction of programs involving different data types. The power of a metacomputation-based theorem prover largely depends on the program transformation technique incorporated in its proof technique. To tackle the challenging problem of removing intermediate structures from programs more naturally, the transformation technique must be equipped with strong heuristics. In Chapter 3, we present the more powerful distillation program transformation algorithm [45, 46] for higher order functional programs which can be used for this purpose.

Chapter 3

Distillation

3.1 Introduction

In this chapter, we describe the distillation program transformation technique. Distillation [45, 46] is a powerful program transformation algorithm to remove intermediate data structures from higher order functional programs.

Distillation is more powerful than supercompilation; supercompilation can produce only a linear improvement in run-time performance of programs [101], while distillation can produce superlinear improvement. In supercompilation, matching is performed on flat expressions only; functions are considered to match only if they have the same name. In distillation algorithm, matching is also performed on recursive expressions, which are considered to match if they have the same recursive structure even though they may contain different function names.

Many of the expressions which are extracted using generalization in supercompilation may actually be intermediate within the resulting generalized expression, but will not be transformed away. This will result in an over-generalized expression, which is not desirable. In distillation, if an expression has been generalized, then this generalization is undone and the extracted sub-expressions are substituted back into the expression resulting from the transformation of the remaining generalized expression. The resulting residual program is further transformed to try and remove these intermediate structures.

Two different versions of distillation have been proposed by Hamilton. The first version [45] is implemented in the framework of the theorem prover Poitín [45]. In [45], Hamilton defined the distillation algorithm with a set of 9 transformation rules, and has shown its use in inductive theorem proving [45].

The version of distillation presented in [46] constructs partial process trees by transforming input programs, and constructs residual programs from the resulting partial process trees. We present the distillation transformation technique based on the presentation in [46].

3.2 Program Transformation Using Distillation

In this section, we give an overview of the distillation algorithm. The language for which the transformations are to be performed is a simple higher order functional language as described in §2.2.1.

In our presentation of the distillation algorithm, unlike in [46], we do not separately transform the sub-expressions extracted using generalization. The residual program therefore contains these extracted sub-expressions in their original form. In addition, unlike in [46], rather than constructing a local function using a **letrec** expression from a cycle in the partial process tree, the resulting recursive expression is constructed using new **Node/Repeat** constructs. The construct **Node f:** e [(**Repeat f:** e')$/v$] is equivalent to **letrec** $f = \lambda v_1 \ldots v_n.e[(f\ e_1 \ldots e_n)/v]$ **in** $f\ v_1 \ldots v_n$ where $\{v_1 \ldots v_n\} = fv(e)$.

3.2.1 Folding and Generalization

Folding is performed when the current expression is an instance of a previously encountered expression. In supercompilation, matching is performed on flat terms; functions are matched if they have the same name. In distillation, matching is also performed on recursive terms; different functions are matched if their corresponding recursive definitions also match. If any expression containing a function call or a function node in the redex is an instance of a previously encountered expression within the partially constructed process tree, then a repeat node is created at the occurrence of the current expression. In the case of a successful match for expressions containing a function call or a function node in the redex, the original occurrence of the expression is replaced by a **Node** construct, and the re-occurrence is replaced by a corresponding **Repeat** construct.

A **Node** expression is the process tree representation of a recursive function, and hence it is further transformed in the hope of finding a match with a further **Node** expression. A local function is defined using a **letrec** expression only when the matched expressions are of **Node** type. In transforming a **letrec** expression, a

local function is defined using a **letrec** expression in the residual program from the resulting subtree rooted at a function node α with a function call in the redex and a repeat node which points to the function node.

Generalization is performed to ensure termination if the current expression is an embedding of a previously encountered expression. To perform generalization, sub-expressions are extracted from expressions as described in §2.2.4. Special guidance is needed to control the whole generalization process during transformation. In §2.2.4, we have defined two types of embedding: *strict* and *non-strict*. If there is a strict embedding, the current (embedding) expression is generalized, whereas, the previous (embedded) expression is generalized if there is a non-strict embedding. We extend this non-strict homeomorphic embedding relation and the most specific generalization to **Node** and **letrec** expressions. In supercompilation, the extracted sub-expressions are not transformed away, and therefore the constructed residual program contains these intermediate structures. In distillation, the residual program constructed from supercompilation is further transformed to try to remove these intermediate structures. The extraction of sub-expressions as a result of generalization is only made permanent in distillation when the embedding of a recursive expression is encountered. Thus, generalization will be performed at most twice for each expression; once when the redex is a function and once when the redex is a recursive expression.

3.2.2 Construction of Partial Process Trees

The output of distillation is a partial process tree from which a residual program can be constructed. The distillation algorithm is defined by the rule shown in Fig. 3.1. The normal order reduction rules $\mathcal{N}$ are defined in Fig. 2.4. The residual program construction rules $\mathcal{P}$ (excluding rules ($\mathcal{P}4$) and ($\mathcal{P}5$)) as defined in Fig. 2.6 are used along with the rules of §3.2.3. In this rule, the nodes which contain a function call, function node, repeat node, **let** or a **letrec** expression in the redex, are handled differently than the nodes which do not.

If the current node β contains an expression in which the redex is a function, and this expression is an *instance* of an expression within an ancestor node α, then a *repeat* node is created. A residual program is constructed from the tree rooted at α and this residual program is further transformed to construct a new partial process tree. The sub-tree rooted at α is replaced with this partial process tree. If the expression contained in the current node β is a strict embedding of an expres-

$$
\begin{array}{l}
\mathcal{T}[\![\beta]\!]\ \phi = \textbf{if}\ t(\beta) = con\langle f\rangle \hfill (\mathcal{T}1)\\
\quad\quad \textbf{then if}\ \exists\alpha \in anc(t,\beta).t(\alpha) \lesssim t(\beta)\\
\quad\quad\quad\quad \textbf{then}\ t\{\beta := t(\beta) \dashrightarrow \alpha\}\{\alpha := \mathcal{T}[\![\mathcal{P}[\![\alpha]\!]]\!]\ \phi\}\\
\quad\quad\quad\quad \textbf{else if}\ \ \exists\alpha \in anc(t,\beta).t(\alpha) \trianglelefteq t(\beta)\\
\quad\quad\quad\quad\quad \textbf{then if}\ t(\alpha) \lhd t(\beta)\\
\quad\quad\quad\quad\quad\quad \textbf{then}\ t\{\beta := \mathcal{T}[\![\mathcal{P}[\![\mathcal{T}[\![extract(t(\alpha),t(\beta))]\!]\ \phi]\!]]\!]\ \phi\}\\
\quad\quad\quad\quad\quad\quad \textbf{else}\ t\{\alpha := \mathcal{T}[\![\mathcal{P}[\![\mathcal{T}[\![abstract(t(\alpha),t(\beta))]\!]\ \phi]\!]]\!]\ \phi\}\\
\quad\quad\quad\quad\quad \textbf{else}\ t\{\beta := t(\beta) \rightarrow \mathcal{T}[\![unfold(t(\beta))\ \phi]\!]\ \phi\}\\
\quad\quad \textbf{else if}\ t(\beta) = \textbf{Node f:}\ e\\
\quad\quad\quad \textbf{then if}\ \exists\alpha \in anc(t,\beta).t(\alpha) \lesssim t(\beta)\\
\quad\quad\quad\quad \textbf{then}\ t\{\beta := t(\beta) \dashrightarrow \alpha\}\\
\quad\quad\quad\quad \textbf{else if}\ \ \exists\alpha \in anc(t,\beta).t(\alpha) \trianglelefteq t(\beta)\\
\quad\quad\quad\quad\quad \textbf{then}\ t\{\alpha := \mathcal{T}[\![abstract(t(\alpha),t(\beta))]\!]\ \phi\}\\
\quad\quad\quad\quad\quad \textbf{else}\ t\{\beta := t(\beta) \rightarrow \mathcal{T}[\![e]\!]\ \phi\}\\
\quad\quad \textbf{else if}\ t(\beta) = \textbf{Repeat f:}\ e\\
\quad\quad\quad \textbf{then}\ t\{\beta := \mathcal{T}[\![e]\!]\ \phi\}\\
\quad\quad \textbf{else if}\ t(\beta) = con\langle\textbf{letrec}\ f = e_0\ \textbf{in}\ e_f\rangle\\
\quad\quad\quad \textbf{then}\ t\{\beta := t(\beta) \rightarrow e_f \rightarrow \mathcal{T}[\![unfold(e_f)\ (\phi \cup \{f, e_0\})]\!]\ \phi\}\\
\quad\quad \textbf{else if}\ t(\beta) = \textbf{let}\ v_1 = e_1, \ldots, v_n = e_n\ \textbf{in}\ e_g\\
\quad\quad\quad \textbf{then}\ t\{\beta := t(\beta) \rightarrow \mathcal{T}[\![e_g]\!]\ \phi\}\\
\quad\quad\quad \textbf{else}\ t\{\beta := t(\beta) \rightarrow \mathcal{T}[\![e_1]\!]\ \phi, \ldots, \mathcal{T}[\![e_n]\!]\ \phi\}\\
\quad\quad\quad\quad\quad\quad \text{where}\ \mathcal{N}[\![t(\beta)]\!]\ \ \phi =\ [e_1, \ldots, e_n]
\end{array}
$$

Figure 3.1: Distillation algorithm

sion within an ancestor node α, generalization is performed as described in §2.2.4 using the *extract* operation. The resulting generalized expression is transformed to construct a partial process tree from which a residual program is constructed. The extracted sub-expression is substituted back into this program, which is then further transformed to construct a new partial process tree. This partial process tree is used to replace the sub-tree rooted at β. If the expression contained in the current node β is a non-strict embedding of an expression within an ancestor node α, *most specific generalization* is performed. The generalized form of the expression within the node α is obtained using the *abstract* operation. This generalized expression is transformed to construct a partial process tree from which a residual program is

constructed. The extracted sub-expressions are substituted back into this program, which is then further transformed to construct a new partial process tree which is used to replace the sub-tree rooted at α.

If the expression within the current node β contains an expression **Node f:** e in the redex, which is an instance of an expression within an ancestor node α, then a repeat node is created. If the current expression is a homeomorphic embedding of an expression within an ancestor node α, most specific generalization is performed. However, in this case, generalization is permanent. The partial process tree obtained by transforming the resulting generalized expression is used to replace the sub-tree rooted at α. Otherwise, the sub-expression e is further transformed.

If the expression within the current node β contains an expression **Repeat f:** e in the redex, then the sub-expression e is further transformed.

If the expression within the current node β contains a **letrec** expression in the redex, then the function definition is added to ϕ, and the unfolded function call is further transformed. The resulting sub-tree is added to the sub-tree rooted at β with the function call as the descendant.

If the current node β contains a **let** expression **let** $v_1 = e_1, \ldots, v_n = e_n$ **in** e_g, then the remaining generalized expression e_g is transformed. The resulting sub-tree is added to the sub-tree rooted at β.

For any other expression e, the expressions obtained by normal order reduction of the expression e are transformed separately, and added as children to the sub-tree rooted at e.

3.2.3 Rules for Residual Program Construction

The following rules for program construction are re-defined.

$$\begin{array}{lll}
\mathcal{P}[\![\alpha = (con\langle f\rangle) \rightarrow t]\!] & & (\mathcal{P}4)\\
\quad = \textbf{Node f': } (\mathcal{P}[\![t]\!]), & \text{if } \exists\beta \in t.\beta \dashrightarrow \alpha \text{ and } \beta \equiv \alpha\{v_1 := e_1, \ldots, v_n := e_n\} & \\
\quad = \mathcal{P}[\![t]\!], & \text{otherwise} & \\
\mathcal{P}[\![\beta = (con\langle f\rangle) \dashrightarrow \alpha]\!] & = \textbf{Repeat f': } (con\langle f\rangle) & (\mathcal{P}5)\\
 & \quad \text{where } \beta \equiv \alpha\{v_1 := e_1, \ldots, v_n := e_n\} & \\
\mathcal{P}[\![(\textbf{let } v_1 = e_1, \ldots, v_n = e_n \textbf{ in } e_g) \rightarrow t]\!] & & (\mathcal{P}9)\\
\quad = (\mathcal{P}[\![t]\!])\ \{v_1 := e_1, \ldots, v_n := e_n\} & &
\end{array}$$

Rule ($\mathcal{P}4$) processes a tree rooted at α which contains an expression with a function in the redex and a sub-tree t. If there exists a node β within the sub-tree t such that β is an instance of α, then the result of processing the current tree is **Node f′:** e where e is the expression obtained from the sub-tree t, and **f′** is a new name for this local definition. Otherwise, the residual program is constructed from the sub-tree t.

In rule ($\mathcal{P}5$), a repeat node β containing an expression in which the redex is a function is processed, which has a matching function node α. The result of processing the current tree is **Repeat f′:** e where **f′** is the function which was introduced for the function node and e is the current expression.

In rule ($\mathcal{P}9$), a tree rooted at a **let** expression is processed. The sub-expressions $e_1, \ldots, e_n$ are substituted for the variables $v_1, \ldots, v_n$ within the expression obtained by processing the child sub-tree.

The partial process tree constructed by distillation may contain **Node** expressions within its nodes. Therefore, in addition to the residual program construction rules defined in §2.2.4, the following rules are used to deal with these expressions within the partial process tree.

$$\begin{aligned}
&\mathcal{P}[\![\alpha = (\textbf{Node f: } e) \rightarrow t]\!] && (\mathcal{P}10)\\
&\quad = \textbf{letrec } f = \lambda v_1 \ldots v_n.\mathcal{P}[\![t]\!] \\
&\qquad \textbf{in } f\ v_1 \ldots v_n, \quad \text{if } \exists \beta \in t.\beta \dashrightarrow \alpha \text{ and } \beta \equiv \alpha\{v_1 := e_1, \ldots, v_n := e_n\} \\
&\quad = \mathcal{P}[\![t]\!], \quad \text{otherwise} \\
&\mathcal{P}[\![\beta = (\textbf{Node f: } e) \dashrightarrow \alpha]\!] && (\mathcal{P}11)\\
&\quad = f\ e_1 \ldots e_n \\
&\quad \text{where } \beta \equiv \alpha\{v_1 := e_1, \ldots, v_n := e_n\}
\end{aligned}$$

Rule ($\mathcal{P}10$) processes a tree rooted at α which contains an expression **Node f:** e with sub-tree t. If there exists a node β within the sub-tree t such that β is an instance of α, then the result of processing the current tree is to introduce a local function definition into the residual program. The body of the new function is constructed from the sub-tree t. Otherwise, no local function is defined, and the residual program is constructed from the sub-tree t.

In rule ($\mathcal{P}11$), a repeat node β containing an expression **Node f:** e is processed, which is an instance of an ancestor node α. In this case, an appropriate recursive call to the function f introduced for the ancestor node α is added to the residual program.

The partial process tree constructed by distillation may contain **letrec** expressions within its nodes. The following program construction rules are defined to deal with these expressions within the partial process tree.

$$\begin{array}{ll}
\mathcal{P}[\![\alpha = (con\langle\textbf{letrec}\ f = e_0\ \textbf{in}\ f\ e_1 \ldots e_n\rangle) \rightarrow \beta \rightarrow t]\!] & (\mathcal{P}12)\\
\quad = \textbf{letrec}\ f' = \lambda v_1 \ldots v_n.\mathcal{P}[\![t]\!] & \\
\qquad \textbf{in}\ f'\ v_1 \ldots v_n, \quad \text{if}\ \exists\beta' \in t.\beta' \dashrightarrow \beta\ \text{and}\ \beta' \equiv \beta\{v_1 := e'_1, \ldots, v_n := e'_n\} & \\
\quad = \mathcal{P}[\![t]\!], \qquad\qquad \text{otherwise} & \\
\mathcal{P}[\![\beta = (con\langle f\rangle) \dashrightarrow \alpha]\!] & (\mathcal{P}13)\\
\quad = f'\ e'_1 \ldots e'_n, \qquad \text{if}\ \beta \equiv \alpha\{v_1 := e'_1, \ldots, v_n := e'_n\}\ \text{and} & \\
\qquad\qquad\qquad\qquad \exists\alpha' \in anc(t,\alpha).t(\alpha') = con\langle\textbf{letrec}\ f = e_0\ \textbf{in}\ f\ e_1 \ldots e_n\rangle &
\end{array}$$

Rule ($\mathcal{P}12$) processes a tree rooted at α which contains an expression $con\langle\textbf{letrec}\ f = e_0\ \textbf{in}\ f\ e_1 \ldots e_n\rangle$ with a descendant β and sub-tree t. If there exists a node β' within the sub-tree t such that β' is an instance of β, then the result of processing the current tree is to introduce a local function definition into the residual program. The body of the new function f' is constructed from the sub-tree t. Otherwise, no local function is defined, and the residual program is constructed from the sub-tree t.

In rule ($\mathcal{P}13$), a repeat node β containing an expression $con\langle f\rangle$ is processed, which is an instance of an ancestor node α and the ancestor of α is a **letrec** expression. In this case, an appropriate recursive call to the function f' introduced for the ancestor node α is added to the residual program.

3.3 Examples

In this section, we give several examples to show how distillation can be used to transform input programs. In the examples, we use simplified partial process trees from which nodes which contain expressions of the form $con\langle(\lambda v.e_0)\ e_1\rangle$ and $con\langle\textbf{case}\ (c\ e_1 \ldots e_n)\ \textbf{of}\ \ldots\rangle$ have been omitted for simplicity. We also rename pattern variables during the transformation of a **case** expression with a variable in the redex.

Within the partial process trees shown in the following examples, we represent some of the leaf nodes with expressions of the form $\mathcal{T}[\![e]\!]$ where e is the expression to be transformed. We do this to refer to the transformation of an identical expression of e that has already been performed.

The following example does not require any generalization to complete the transformation, but all other examples require generalization. Examples of *accumulating patterns, accumulating parameters* and *obstructing function calls* are shown in Appendix A.1. Fig. 3.2 shows some of the function definitions which are used along with the definitions of the functions *append, reverse, plus* and *eqnum* as defined in the examples 3, 4, 6 and 8 respectively of Chapter 2 to transform the input expressions.

$$
\begin{array}{lll}
even & = & \lambda x.\,\textbf{case}\ x\ \textbf{of} \\
 & & \quad Zero \ :\ True \\
 & & \quad \mid Succ\ x'\ :\ \textbf{case}\ x'\ \textbf{of} \\
 & & \qquad\qquad Zero \ :\ False \\
 & & \qquad\qquad \mid Succ\ x''\ :\ even\ x'' \\
doublea & = & \lambda x.\lambda y.\,\textbf{case}\ x\ \textbf{of} \\
 & & \quad Zero \ :\ y \\
 & & \quad \mid Succ\ x'\ :\ doublea\ x'\ (Succ\ (Succ\ y)) \\
leq & = & \lambda x.\lambda y.\,\textbf{case}\ x\ \textbf{of} \\
 & & \quad Zero \ :\ \textbf{case}\ y\ \textbf{of} \\
 & & \qquad\qquad Zero \ :\ True \\
 & & \qquad\qquad \mid Succ\ y'\ :\ True \\
 & & \quad \mid Succ\ x'\ :\ \textbf{case}\ y\ \textbf{of} \\
 & & \qquad\qquad Zero \ :\ False \\
 & & \qquad\qquad \mid Succ\ y'\ :\ leq\ x'\ y'
\end{array}
$$

Figure 3.2: Function definitions

Example 9

Consider the transformation of the following expression (9.1) about natural numbers.

$$leq\ x\ (plus\ x\ y) \tag{9.1}$$

During the transformation of expression (9.1), the partial process tree shown in Fig. 3.3 is constructed. Within this partial process tree, expression (9.2) is encountered, which is an instance of expression (9.1). A repeat node is therefore created at the occurrence of expression (9.2).

$$leq\ x'\ (plus\ x'\ y) \tag{9.2}$$

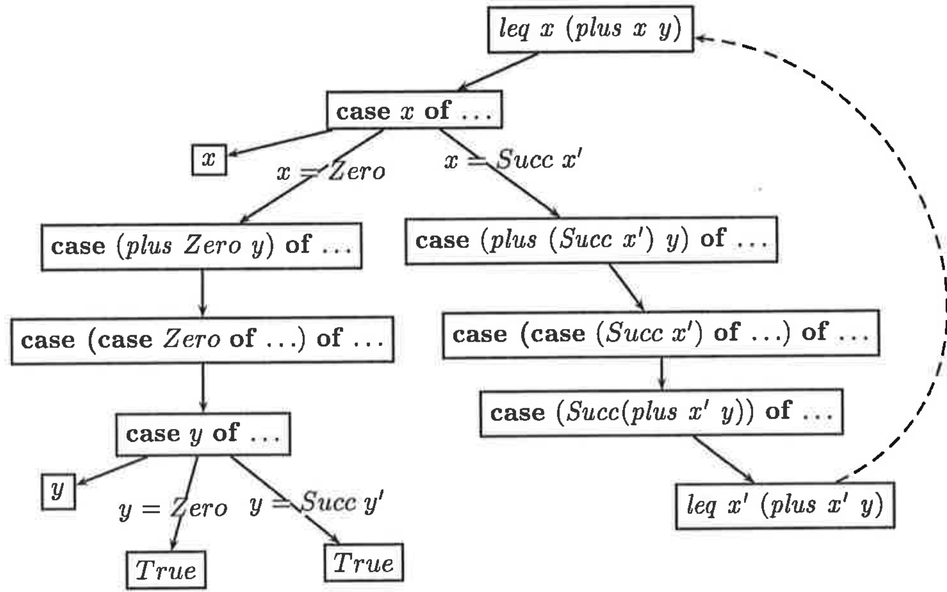

Figure 3.3: Partial process tree (1) for $\mathcal{T}[\![leq\ x\ (plus\ x\ y)]\!]$

Expression (9.3) is constructed from the partial process tree shown in Fig. 3.3.

$$
\begin{aligned}
\textbf{Node f0: case } & x \textbf{ of} \\
& Zero \quad : \textbf{case } y \textbf{ of} \\
& \qquad\qquad Zero \quad : True \\
& \qquad\qquad \mid Succ\ y' : True \\
& \mid Succ\ x' : \textbf{Repeat f0: } leq\ x'\ (plus\ x'\ y)
\end{aligned}
\tag{9.3}
$$

Expression (9.3) is further transformed, which results in the partial process tree shown in Fig. 3.4. Expression (9.4) is constructed from the partial process tree shown in Fig. 3.4.

$$
\begin{aligned}
& \textbf{case } x \textbf{ of} \\
& \quad Zero \quad : \textbf{case } y \textbf{ of} \\
& \qquad\qquad Zero \quad : True \\
& \qquad\qquad \mid Succ\ y' : True \\
& \mid Succ\ x' : \textbf{Node f1: case } x' \textbf{ of} \\
& \qquad\qquad Zero \quad : \textbf{case } y \textbf{ of} \\
& \qquad\qquad\qquad Zero \quad : True \\
& \qquad\qquad\qquad \mid Succ\ y' : True \\
& \qquad\qquad \mid Succ\ x'' : \textbf{Repeat f1: } leq\ x''\ (plus\ x''\ y)
\end{aligned}
\tag{9.4}
$$

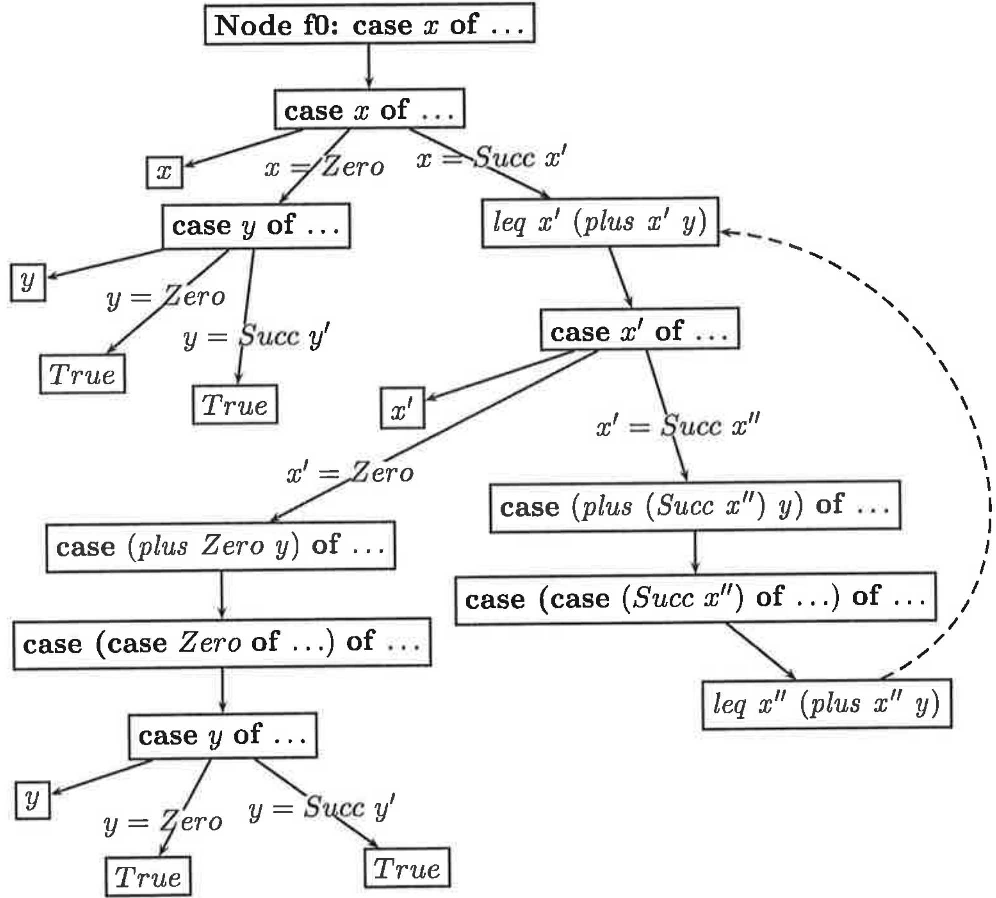

Figure 3.4: Partial process tree (2) for $\mathcal{T}[\![leq\ x\ (plus\ x\ y)]\!]$

We obtain expression (9.5) from the sub-tree rooted at $leq\ x'\ (plus\ x'\ y)$ within Fig. 3.4.

$$\begin{aligned}
&\textbf{Node f1: case } x' \textbf{ of} \\
&\qquad Zero \quad : \textbf{case } y \textbf{ of} \\
&\qquad\qquad Zero \quad : True \\
&\qquad\qquad \mid Succ\ y' : True \\
&\qquad \mid Succ\ x'' : \textbf{Repeat f1: } leq\ x''\ (plus\ x''\ y)
\end{aligned} \tag{9.5}$$

Expression (9.5) is an instance of expression (9.3). A repeat node is therefore created at the occurrence of expression (9.5). This results in the partial process tree which is shown in Fig. 3.5.

We therefore construct the residual program shown in Fig. 3.6 from the partial process tree shown in Fig. 3.5.

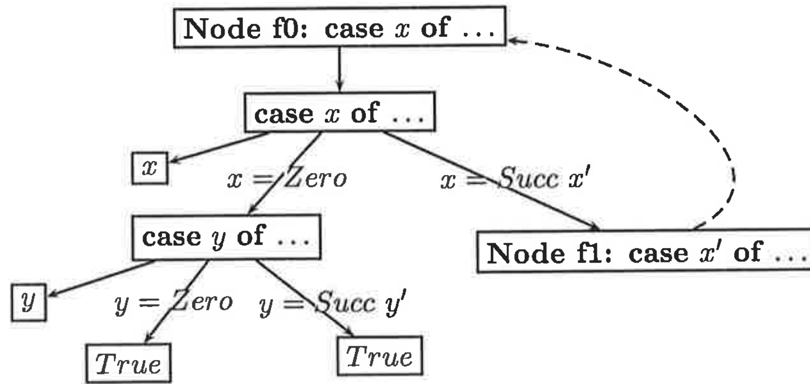

Figure 3.5: Partial process tree (3) for $\mathcal{T}[\![leq\ x\ (plus\ x\ y)]\!]$

$$
\begin{array}{l}
\textbf{letrec}\ f0 = \lambda x.\textbf{case}\ x\ \textbf{of} \\
\qquad\qquad\qquad\qquad \mathit{Zero} \quad : \textbf{case}\ y\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{Zero} \quad : \mathit{True} \\
\qquad\qquad\qquad\qquad\qquad\quad\ \mid \mathit{Succ}\ y' : \mathit{True} \\
\qquad\qquad\qquad\quad\ \mid \mathit{Succ}\ x' : f0\ x' \\
\textbf{in}\ f0\ x
\end{array}
$$

Figure 3.6: Residual program for $\mathcal{T}[\![leq\ x\ (plus\ x\ y)]\!]$

3.4 Termination of the Distillation Algorithm

In this section, we give the proof that distillation algorithm always terminates. This proof of termination is based on the language-independent framework for proving termination of abstract program transformers in [96] in the metric space of trees.

For a transformer to fit into the framework for termination of abstract transformers [96], it is sufficient to ensure that:

1. in the sequence of trees produced by transformation, for any depth d, there must be some point from which every two consecutive trees are identical down to depth d.

2. only finite trees are produced.

We can prove the first property by induction on the depth of the trees produced by virtue of the fact that the algorithm does one of the following:

- adds new leaves to a tree which makes consecutive trees identical at an increasing depth.
- replaces a sub-tree with a node whose label is a **let** expression. Each node can be generalized at most twice in this way: once when the label is a flat expression, and once when the label is a recursive expression.

The second property is ensured by the fact that in every process tree:

- the node that contains a **let** expression has children which are sub-expressions of the **let** expression. So, within a path which consists only of **let** expressions, the size of the nodes strictly decreases.
- all other nodes are not allowed to homeomorphically embed an ancestor.

Now, we give the details of the termination proof of distillation based on the termination proof of an abstract program transformer in the metric space of trees as presented in [96]. We recommend the interested readers to see [96] for the details of an abstract program transformer and metric space of trees.

We consider an abstract program transformer M on a set E. Let t be a tree over E. The elements of dom(t) are called nodes of t. The empty string ε is the root, and for any node α in t, the nodes αi of t (if any) are the children of α and α is the parent of these nodes. leaf(t) denotes the set of all leafs in t. Also, t is finite, if dom(t) is finite and t is singleton if dom(t) $= \{\varepsilon\}$. Two expressions e_1 and e_2 are incommensurable, $e_1 \leftrightarrow e_2$, if $e_1 \sqcap e_2$ is a variable.

$T_\infty(E)$ is the set of all trees over E and $T(E)$ is the set of all finite trees over E. Let $\mathcal{E}_H(V)$ be the set of expressions over symbols H and variables V.

An abstract program transformer on E is a map $M : T(E) \to T(E)$. No more transformation steps will happen when $M(t) = t$. M on E terminates on $t \in T(E)$ if $M^i(t) = M^{i+1}(t)$ for some $i \in \mathbb{N}$ (for $f : A \to A,\ f^0(a) = a,\ f^{i+1}(a) = f^i(f(a))$). M on E terminates if M terminates on all singletons $t \in T(E)$.

Let $M : T(E) \to T(E)$ be an abstract program transformer on E and $p : T_\infty(E) \to \mathbb{B}$ be a predicate. M maintains p if, for every singleton $t \in T(E)$ and $i \in \mathbb{N}$, $p(M^i(t)) = 1$. A predicate $p : T_\infty(E) \to \mathbb{B}$ is finitary if $p(t) = 0$ for all infinite $t \in T_\infty(E)$. An abstract program transformer M on E is Cauchy if, for every singleton $t \in T_\infty(E)$, the sequence $t, M(t), M^2(t), \ldots$ is a Cauchy sequence. Let $M : T(E) \to T(E)$ maintain predicate $p : T_\infty(E) \to \mathbb{B}$. If

1. M is Cauchy, and

2. p is finitary and continuous,

then M terminates.

The condition that M be Cauchy guarantees that only finitely many generalization steps will happen at a given node, and the condition that p be finitary and continuous guarantees that only finitely many unfolding steps will be used to expand the transformation tree. An abstract program transformer is Cauchy if it always either adds some new children to a leaf node by unfolding, or replaces a subtree by a new tree whose root label is strictly smaller than the label of the root of the former subtree by generalize operation.

Now, we prove that distillation M_D terminates. We do this by proving that M_D is Cauchy and that M_D maintains a finitary, continuous predicate. We first prove that M_D is Cauchy by using the following proposition:

Proposition 3.4.1

Let $(E, \leq)$ be a well-founded quasi order and $M : T(E) \to T(E)$ an abstract program transformer such that, for all t, $M(t) = t\{\gamma := t'\}$ for some γ, t' where

1. $\gamma \in \text{leaf}(t)$ and $t(\gamma) = t'(\varepsilon)$ (unfold); or

2. $t(\gamma) > t'(\varepsilon)$ (generalize).

then M_D is Cauchy.

The following shows that a Cauchy transformer terminates if it never introduces a node whose label is larger than an ancestor's label with respect to some well-quasi order.

Proposition 3.4.2

Let $(E, \leq)$ be a well-quasi order. Then a finitary predicate $p : T_\infty(E) \to \mathbb{B}$,

$$p(t) = \begin{cases} 0 \text{ if } \exists \alpha, \alpha i \beta \in \mathrm{dom(t)} : \mathrm{t}(\alpha) \leq \mathrm{t}(\alpha i \beta) \\ 1 \text{ otherwise} \end{cases}$$

is finitary and continuous.

The following shows that a Cauchy transformer terminates if it never introduces a node whose label is not smaller than its immediate ancestor's label with respect to some well-founded quasi order.

Proposition 3.4.3

Let $(E, \leq)$ be a well-founded quasi order. Then a finitary predicate $p : T_\infty(E) \to \mathbb{B}$,

$$p(t) = \begin{cases} 0 \text{ if } \exists \alpha, \alpha i \in \mathrm{dom(t)} : \mathrm{t}(\alpha) \not\geq \mathrm{t}(\alpha i) \\ 1 \text{ otherwise} \end{cases}$$

is finitary and continuous.

M_D always either unfolds of an expression or replaces a subtree by a new leaf whose label is strictly smaller than the expression in the root of the former subtree.

Proposition 3.4.4

M_D is Cauchy.

Proof. We define the relation $\succ$ on the set $\mathcal{L}$ of **let** expressions by:
let $v'_1 = e'_1, \ldots, v'_m = e'_m$ **in** $e \succ$ **let** $v_1 = e_1, \ldots, v_n = e_n$ **in** $e \Leftrightarrow m = 0 \ \& \ n \geq 0$ where $\succeq$ is a well-founded quasi order.

We show that for any $t \in T(\mathcal{L})$, $M_D(t) = t\{\gamma := t'\}$ where for some $\gamma \in \mathrm{dom}(t)$ and $t' \in T_\infty(\mathcal{L})$, either $\gamma \in \mathrm{leaf}(t)$ and $t(\gamma) = t'(\varepsilon)$, or $t(\gamma) \succ t'(\varepsilon)$. We proceed by case analysis of the operation performed by M_D.

1. $M_D(t) = \mathcal{T}(\gamma) = t\{\gamma := t'\}$, where $\gamma \in \mathrm{leaf}(t)$ and, for the expressions $e_1, \ldots, e_n$, $t' = t(\gamma) \to e_1, \ldots, e_n$. Then $t(\gamma) = t'(\varepsilon)$.

2. $M_D(t) = \mathit{abstract}(t(\gamma), t(\alpha)) = t\{\gamma := \textbf{let } v_1 = e_1, \ldots, v_n = e_n \textbf{ in } e \to\}$, where $\alpha \in \mathit{anc}(t, \gamma)$, $t(\alpha) \neq t(\gamma)$, $t(\alpha)$, $t(\gamma) \in \mathcal{E}$ are both non-trivial, $t(\alpha) \leq\!\cdot\ t(\gamma)$, $e = t(\alpha) \sqcap t(\gamma)$, and $t(\gamma) = e\{v_1 := e_1, \ldots, v_n := e_n\}$. Then, $e = t(\alpha)$ and $t(\gamma) = t(\alpha)\{v_1 := e_1, \ldots, v_n := e_n\}$, but $t(\gamma) \neq t(\alpha)$, so $n > 0$. Thus, $t(\gamma) \succ$ **let** $v_1 = e_1, \ldots, v_n = e_n$ **in** $e = t'(\varepsilon)$.

3. $M_D(t) = \mathit{abstract}(t(\gamma), t(\beta)) = t\{\gamma := \textbf{let } v_1 = e_1, \ldots, v_n = e_n \textbf{ in } e \to\}$, where $\gamma \in \mathit{anc}(t, \beta)$, $t(\beta), t(\gamma)$ are both non-trivial, $t(\gamma) \not\leq\!\cdot\ t(\beta)$, $e = t(\gamma) \sqcap t(\beta)$, and $t(\gamma) = e\{v_1 := e_1, \ldots, v_n := e_n\}$. Then, $t(\gamma) \neq e$, but $t(\gamma) = e\{v_1 := e_1, \ldots, v_n := e_n\}$, so $n > 0$. Thus, $t(\gamma) \succ$ **let** $v_1 = e_1, \ldots, v_n = e_n$ **in** $e = t'(\varepsilon)$.

4. $M_D(t) = \mathit{extract}(t(\alpha), t(\gamma)) = t\{\gamma := \textbf{let } v = e_0 \textbf{ in } e\langle v \rangle \to\}$, where $\alpha \in \mathit{anc}(t, \gamma)$, $t(\alpha), t(\beta)$ are non-trivial, $t(\alpha) \trianglelefteq t(\gamma)$, $t(\alpha) \leftrightarrow t(\gamma)$, and also $t(\gamma) = e\langle e_0 \rangle$. Here $n > 0$: if $n = 0$, then $t(\gamma) = e\langle\rangle$, but then $t(\alpha) \not\leftrightarrow t(\beta)$. Thus, $t(\gamma) = e\langle e_0 \rangle \succ$ **let** $v = e_0$ **in** $e\langle v \rangle = t'(\varepsilon)$.

This concludes the proof.

Proposition 3.4.5

M_D maintains a finitary, continuous predicate.

Proof. We define $\mathcal{S}[\![\bullet]\!] : \mathcal{E} \to \mathbb{N}$ by

$$
\begin{array}{ll}
v & = 1 \\
c\ e_1\ \ldots\ e_n & = 1 + \mathcal{S}[\![e_1]\!] + \ \ldots\ + \mathcal{S}[\![e_n]\!] \\
\lambda v.e & = 1 + \mathcal{S}[\![e]\!] \\
f & = 1 \\
e_0\ e_1 & = 1 + \mathcal{S}[\![e_0]\!] + \mathcal{S}[\![e_1]\!] \\
\textbf{case}\ e_0\ \textbf{of}\ p_1 :\ e_1 \mid \ldots \mid p_k :\ e_k & = 1 + \mathcal{S}[\![e_0]\!] + \ldots + \mathcal{S}[\![e_k]\!] \\
\textbf{let}\ v_1 = e_1,\ \ldots,\ v_n = e_n\ \textbf{in}\ e_0 & = 1 + \mathcal{S}[\![e_0]\!] + \ldots + \mathcal{S}[\![e_n]\!] \\
\textbf{letrec}\ f = e_0\ \textbf{in}\ e_1 & = 1 + \mathcal{S}[\![e_0]\!] + \mathcal{S}[\![e_1]\!] \\
\textbf{Node f:}\ e & = 1 + \mathcal{S}[\![e]\!] \\
\textbf{Repeat f:}\ e & = 1 + \mathcal{S}[\![e]\!]
\end{array}
$$

We define $\mathrm{l} : \mathcal{L} \to \mathcal{E}$ by

$$\mathrm{l}(\textbf{let}\ v_1 = e_1,\ \ldots,\ v_n = e_n\ \textbf{in}\ e_0) = e_0\{v_1 := e_1, \ldots, v_n := e_n\} \text{ for } n \geq 0.$$

We define $\sqsupseteq$ on $\mathcal{L}$ by:

$$\ell \sqsupseteq \ell' \Leftrightarrow \mathcal{S}[\![\mathrm{l}(\ell)]\!] > \mathcal{S}[\![\mathrm{l}(\ell')]\!] \text{ or, } \mathcal{S}[\![\mathrm{l}(\ell)]\!] = \mathcal{S}[\![\mathrm{l}(\ell')]\!] \ \&\ \mathrm{l}(\ell) \geq \mathrm{l}(\ell')$$

We consider $\sqsubseteq$ is a well-founded quasi order. Consider the predicate $q : T_\infty(\ell) \to \mathbb{B}$ defined by $q(t) = p(t^0)$ where $p : T_\infty(\ell) \to \mathbb{B}$ is defined by:

$$p(t) = \begin{cases} 0 \text{ if } \exists \alpha, \alpha i \beta \in \mathrm{dom(t)} : \mathrm{t}(\alpha), \mathrm{t}(\alpha i \beta) \text{ are non-trivial } \& \ \mathrm{t}(\alpha) \trianglelefteq \mathrm{t}(\alpha i \beta) \\ 0 \text{ if } \exists \alpha, \alpha i \in \mathrm{dom(t)} : \mathrm{t}(\alpha), \mathrm{t}(\alpha i) \text{ are non-trivial } \& \ \mathrm{t}(\alpha) \not\sqsupset \mathrm{t}(\alpha i) \\ 1 \text{ otherwise} \end{cases}$$

The sets of non-trivial and trivial expressions constitute a partition of $\mathcal{L}$. Also, $\trianglelefteq$ is a well-quasi order on the set of non-trivial expressions (i.e., on all of $\mathcal{E}$) and $\sqsubseteq$ is a well-founded quasi order on the set of trivial expressions (i.e., on all of $\mathcal{L}$). It follows by proposition 3.4.6 that p is finitary and continuous, and by proposition 3.4.7 that q is also finitary and continuous.

The following shows that one can combine well-quasi orders and well-founded quasi orders in a partition.

Proposition 3.4.6

Let (E_1, E_2) be a partition of E and let $\leq_1$ be a well-quasi order on E_1 and $\leq_2$ be a well-founded quasi order on E_2. Then $p : T_\infty(E) \to \mathbb{B}$,

$$p(t) = \begin{cases} 0 \text{ if } \exists \alpha, \alpha i \beta \in \text{dom}(t) : t(\alpha), t(\alpha i \beta) \in E_1 \ \& \ t(\alpha) \leq_1 t(\alpha i \beta) \\ 0 \text{ if } \exists \alpha, \alpha i \in \text{dom}(t) : t(\alpha), t(\alpha i) \in E_2 \ \& \ t(\alpha) \not>_2 t(\alpha i) \\ 1 \text{ otherwise} \end{cases}$$

is finitary and continuous.

The following shows that it suffices to apply a finitary and continuous predicate to the interior part of a tree. For $t \in T_\infty(E)$, we define the interior $t^0 \in T_\infty(E)$ of t by:

$$\text{dom}(t^0) = (\text{dom}(t) \setminus \text{leaf}(t)) \cup \{\varepsilon\}$$
$$t^0(\gamma) = t(\gamma) \text{ for all } \gamma \in \text{dom}(t^0)$$

Proposition 3.4.7

Let $p : T_\infty(E) \to \mathbb{B}$ be finitary and continuous. Then also the map $q : T_\infty(E) \to \mathbb{B}$ defined by $q(t) = p(t^0)$ is finitary and continuous.

Now, one can replace in the proposition $\bullet^0$ by any continuous map which maps infinite trees to infinite trees.

It remains to show that M_D maintains q, i.e., that $q(M_D^i(t_0)) = 1$ for any singleton $t_0 \in T_\infty(\mathcal{L})$. Given any $t \in T_\infty(\mathcal{L})$ and $\beta \in \text{dom}(t)$, we say that β is good in t if the following conditions both hold:

(i) $t(\beta)$ non-trivial & $\beta \notin \text{leaf}(t) \Rightarrow \forall \alpha \in \text{anc}(t, \beta) \setminus \{\beta\} : t(\alpha)$ non-trivial $\Rightarrow t(\alpha) \ntrianglelefteq t(\beta)$

(ii) $\beta = \alpha i$ & $t(\alpha)$ trivial $\Rightarrow t(\alpha) \sqsupset t(\beta)$

We say that t is good if all $\beta \in \text{dom}(t)$ are good in t.

We see that $q(t) = 1$ if t is good (the converse does not hold). It therefore suffices to show for any singleton $t_0 \in T_\infty(L)$ that $M_D^i(t_0)$ is good for all i. We proceed by induction on i.

For $i = 0$, (i)-(ii) are both satisfied since t_0 consists of a single leaf. For $i > 0$, we split into cases according to the operation performed by M_D on $M_D^{i-1}(t_0)$.

Before considering these cases, by the definition of goodness, if $t \in T_\infty(\ell)$ is good, $\gamma \in \text{dom}(t)$, and $t' \in T_\infty(\mathcal{L})$, then $t\{\gamma := t'\}$ is good too, provided $\gamma\delta$ is good in $t\{\gamma := t'\}$ for all $\delta \in \text{dom}(t')$.

Let $t = M_D^{i-1}(t_0)$.

1. $M_D(t) = \mathcal{T}(\gamma) = t\{\gamma := t'\}$, where $\gamma \in \text{leaf}(t)$, $t' = t(\gamma) \to e_1, \ldots, e_n$, and $\{e_1, \ldots, e_n\} = \{e \mid t(\gamma) \Rightarrow e\}$. We show that $\gamma, \gamma 1, \ldots, \gamma n$ are good in $M_D(t)$. To see that γ is good in $M_D(t)$, if $t(\gamma)$ is non-trivial, then the algorithm ensures that condition (i) is satisfied. Condition (ii) follows from the induction hypothesis. To see that γi is good in $M_D(t)$, condition (i) is satisfied. Moreover, when $\ell \Rightarrow e$ and ℓ is trivial, $\ell \sqsupset e$, so condition (ii) holds as well.

2. $M_D(t) = \mathit{abstract}(t(\gamma), t(\alpha)) = t\{\gamma := \textbf{let } v_1 = e_1, \ldots, v_n = e_n \textbf{ in } e \to\}$, where $\alpha \in \mathit{anc}(t, \gamma)$, $t(\alpha) \neq t(\gamma)$, $t(\alpha)$, $t(\gamma) \in \mathcal{E}$ are both non-trivial, $t(\alpha) \leq t(\gamma)$, $e = t(\alpha) \sqcap t(\gamma)$, and $t(\gamma) = e\{v_1 := e_1, \ldots, v_n := e_n\}$.

   We show that γ is good in $M_D(t)$. Condition (i) holds, and (ii) follows from the induction hypothesis and $\text{l}(t(\gamma)) = \text{l}(\textbf{let } v_1 = e_1, \ldots, v_n = e_n \textbf{ in } e)$.

The remaining two cases are similar to the preceding case.

3.5 Correctness of Distillation Algorithm

The transformation of a program is correct if the extensional meaning of the original program is preserved in the transformed program. The proof of correctness of the distillation algorithm is given in [46]. We give an outline of this proof here. To prove that the distillation algorithm produces programs which are equivalent to the original programs, the *improvement theorem* of Sands [89, 90] is used. In order to prove the correctness of the distillation algorithm, we first prove the following lemma.

Lemma 3.5.1 (Efficiency)

The distillation algorithm produces programs which are no less efficient than the original.

Proof (Sketch). To prove that the distillation algorithm does not result in a loss of efficiency, a measure of the cost of expressions related to the operational

semantics of the language is used. This measure indicates the number of reduction steps required to reduce an expression to normal form. In [89, 90], the one-step reduction relation for a call-by-name semantics is denoted by $\mapsto$, and a closed expression e is said to converge to weak head normal form w denoted by $e \Downarrow w$, if and only if $e \mapsto^* w$, where $\mapsto^*$ denotes the transitive closure of $\mapsto$. For any expression e, $C[e] \Downarrow^n w$ denotes that a closed expression e converges to weak head normal form w in n reduction steps, if $e \mapsto_n w$ where $\mapsto_n$ denotes a sequence of n reductions.

The following notion of improvement is defined in [89, 90] for a call-by-name semantics as follows.

Definition 3.5.2 (Improvement) *An expression e is improved by e', denoted by $e \succsim e'$ if, for all contexts C such that $C[e]$ and $C[e']$ are closed, if $C[e] \Downarrow^n$, $C[e'] \Downarrow^m$ and $m \leq n$.*

This notion of improvement was used to prove that there is no efficiency loss with respect to a call-by-name semantics resulting from supercompilation in [89]. In order to prove this for the new rules of distillation for transforming nodes which contain recursive expressions, we need to show that each new function call which is introduced by transformation comes together with an unfolding step in the body of that function definition. To ensure this, after first encountering a node which contains a **Node** expression, the function is unfolded. When a node with a matching **Node** expression is subsequently encountered, folding is performed, but an unfolding step will have been performed in the body of the constructed function.

The following lemma is required to prove the correctness of the distillation algorithm.

Lemma 3.5.3 (Correctness)

The distillation algorithm produces programs which are equivalent to the original.

Proof (Sketch). The proof of correctness of the distillation algorithm follows the work of Sands [89, 90], which makes use of an *improvement theorem*. The improvement theorem states that if a transformation repeatedly applies a set of transformation rules to a program, where each transformation step is equivalence preserving, then a transformation which replaces a program e by e' is totally correct if e is improved by e'. The equivalence of each transformation step can be proved

with respect to the operational semantics of the language. The improvement of the expression e by e' follows immediately from the preservation of efficiency in the distillation algorithm (Lemma 3.5.1).

3.6 Distilled Form

Distillation transforms an input program to a *normal form* which we call *distilled form* as shown in Fig. 3.7. As we can see, in distilled form, all functions are tail recursive.

$$
\begin{array}{rcl}
dt & := & v\ dt_1 \ldots dt_n \\
 & | & c\ dt_1 \ldots dt_n \\
 & | & \lambda v.dt \\
 & | & \mathbf{case}\ v\ \mathbf{of}\ p_1\ :\ dt'_1 \mid \ldots \mid p_k\ :\ dt'_k \\
 & | & \mathbf{let}\ v\ =\ dt_0\ \mathbf{in}\ dt_1 \\
 & | & \mathbf{letrec}\ f\ =\ \lambda v_1 \ldots v_n.dt\ \mathbf{in}\ f\ v_1 \ldots v_n \\
 & | & f\ dt_1 \ldots dt_n
\end{array}
$$

Figure 3.7: Distilled form

3.7 Conclusion

In this chapter, we have given an overview of the distillation program transformation algorithm. We have given several examples to demonstrate the application of distillation to transform input programs to output programs which are in *distilled form*. We have shown how distillation can be used to cope with different non-termination problems caused by *accumulating patterns*, *accumulating parameters*, and *obstructing function calls*. Distillation is more powerful than existing program transformation algorithms such as supercompilation and partial evaluation. These previous algorithms can produce only a linear speedup in programs, whereas distillation can produce a superlinear speedup. For example, it is possible to transform the naive quadratic reverse function into the linear accumulating version. This extra power is obtained through the use of more powerful matching prior to folding. In previous techniques, matching is performed on flat terms only; functions are considered to match only if they have the same name. In distillation, matching is also

applied to recursive terms, so different functions are considered to match if their corresponding recursive definitions also match. Distillation is guaranteed to terminate and constructs a distilled output program with the same semantic meaning as the input program while preserving efficiency. These features make distillation algorithm applicable to inductive theorem proving and program construction, which we present in Chapter 4 and Chapter 5.

Chapter 4

Theorem Proving in Poitín

4.1 Introduction

In this chapter, we present our inductive theorem proving techniques, and show how the Poitín theorem prover [45] can be extended to handle explicit quantification to prove universally and existentially quantified conjectures. In the previous chapter, we have given an overview of the distillation program transformation algorithm [46], and have shown how distillation can be used to transform input programs to a normal form called *distilled form*. In order to use distillation within the theorem prover Poitín, distillation is applied to the input conjecture. The inductive proof rules are then applied to the resulting distilled expression to prove it. The distillation rules are therefore extended to handle explicit quantification. We present proof rules for universal and existential quantification to prove inductive conjectures.

The proof of a universally quantified conjecture in Poitín does not require any intermediate lemmas. The usual approach to proving existential theorems is to constructively find the witness, and then show that this witness satisfies the required inductive property. This requires the use of higher order unification, which is in general undecidable. We present an alternative approach to prove inductive existential conjectures, which gives a pure existence proof of the conjecture. An earlier version of the work presented in this chapter can be found in [47, 61, 60].

4.2 Pre-Processing Phase

In order to use the distillation algorithm within our inductive theorem prover Poitín, we apply distillation to the input conjecture. The result of this transformation will

be a boolean expression which is in distilled form as described in §3.6. A further pre-processing phase is applied to the resulting distilled expression before being passed on to the theorem prover. In this phase, the distilled program obtained from the input conjecture is processed to obtain a *proof expression* (Fig. 4.3 of §4.3). This phase performs the following tasks:

- adds all free variables appearing within the body of a local function defined using a **letrec** expression as parameters to the function definition, so that no variable remains free within the body of the local function.
- removes the non-terminating functions and replaces them with $\perp$.
- removes all **let** expressions and replaces them with $\perp$.

Definition 4.2.1 (Decreasing parameter) *A parameter is decreasing from value e to e', denoted by $e' \sqsubset e$, if e' is a sub-component of e.*

Definition 4.2.2 (Non-Decreasing parameter) *A parameter is non-decreasing from value e to e', denoted by $e' \sqsupseteq e$, if $e \sqsubset e'$ or $e = e'$.*

If all of the parameters in the recursive call(s) of a function are *non-decreasing*, then the function is potentially non-terminating, so the recursive call is replaced by $\perp$. A non-terminating function loops infinitely, which will never terminate. If at least one of the parameters in a recursive call of the function is *decreasing*, then the recursive call remains. Otherwise, the recursive call of the function is unfolded. As the **let** expressions contain intermediate data structures, our proof rules cannot prove them. So, all **let** expressions are removed by the pre-processing phase.

The original distillation algorithm as presented in the previous chapter is therefore extended to include a second pass to perform the pre-processing tasks as described above. The pre-processing phase $\mathcal{R}$ as defined in Fig. 4.1 is applied to the term resulting from distillation. Within these rules, the set ρ contains the initial calls of functions, and ϕ is the function variable environment.

4.3 Explicit Quantification in Poitín

In this section, we extend the Poitín theorem prover to handle explicit quantification. To facilitate explicit quantification, two first-order quantifiers are added to the higher order functional language defined in §2.2.1 as shown in Fig. 4.2.

$$\mathcal{R}[\![v\ e_1 \ldots e_n]\!]\ \rho\ \phi \ = \ v\ (\mathcal{R}[\![e_1]\!]\ \rho\ \phi) \ldots (\mathcal{R}[\![e_n]\!]\ \rho\ \phi) \qquad (\mathcal{R}1)$$

$$\mathcal{R}[\![c\ e_1 \ldots e_n]\!]\ \rho\ \phi \ = \ c\ (\mathcal{R}[\![e_1]\!]\ \rho\ \phi) \ldots (\mathcal{R}[\![e_n]\!]\ \rho\ \phi) \qquad (\mathcal{R}2)$$

$$\mathcal{R}[\![\textbf{case}\ v\ \textbf{of}\ p_1\ :\ e_1 \mid \ldots \mid p_n\ :\ e_n]\!]\ \rho\ \phi \qquad (\mathcal{R}3)$$
$$= \ \textbf{case}\ v\ \textbf{of}\ p_1\ :\ (\mathcal{R}[\![e_1]\!]\ \rho\ \phi) \mid \ldots \mid p_n\ :\ (\mathcal{R}[\![e_n]\!]\ \rho\ \phi)$$

$$\mathcal{R}[\![\textbf{let}\ v_1 = e_1, \ldots, v_n = e_n\ \textbf{in}\ e_g]\!]\ \rho\ \phi \ = \ \bot \qquad (\mathcal{R}4)$$

$$\mathcal{R}[\![\textbf{letrec}\ f \ = \ \lambda v_1 \ldots v_n.e_0\ \textbf{in}\ f\ v_1 \ldots v_n]\!]\ \rho\ \phi \qquad (\mathcal{R}5)$$
$$= \ \textbf{letrec}\ f \ = \ \lambda v_1 \ldots v_n\ v'_1 \ldots v'_k.e'_0\ \textbf{in}\ f\ v_1 \ldots v_n\ v'_1 \ldots v'_k$$

where

$$e'_0 \ = \ \mathcal{R}[\![e_0]\!]\ (\rho \cup \{f\ v_1 \ldots v_n\})\ (\phi \cup \{f, e_0\})$$
$$\{v'_1 \ldots v'_k\} = fv(\lambda v_1 \ldots v_n.e_0)$$

$$\mathcal{R}[\![f\ e_1 \ldots e_n]\!]\ \rho\ \phi \qquad (\mathcal{R}6)$$
$$= \ \bot, \qquad \text{if } \exists (f\ v_1 \ldots v_n) \in \rho. \forall i \in \{1 \ldots n\}.e_i \sqsupseteq v_i$$
$$= \ f\ e_1 \ldots e_n\ v_1 \ldots v_k, \qquad \text{if } \exists (f\ v_1 \ldots v_n) \in \rho. \exists i \in \{1 \ldots n\}.e_i \sqsubset v_i$$
$$= \ \mathcal{R}[\![e]\!]\ \rho\ \phi\ , \quad \text{otherwise}$$

where

$$\phi(f) = \lambda v_1 \ldots v_n.e$$
$$\{v_1 \ldots v_k\} = fv(\lambda v_1 \ldots v_n.e)$$

Figure 4.1: Distillation pre-processing rules

$$\begin{array}{rll} e ::= & \text{ALL}\ v.e & \text{universally quantified expression} \\ \mid & \text{EX}\ v.e & \text{existentially quantified expression} \end{array}$$

Figure 4.2: Form of input conjecture

The input conjectures can be entered into the system in any of the quantified forms of expressions as shown in Fig. 4.2. The body of the quantified expression can be any expression in the language.

The grammar of redexes is extended as follows to handle quantified expressions.

$$\begin{array}{rll} \mathit{red} & ::= & f \\ & \mid & (\lambda v.e_0)\ e_1 \\ & \mid & \text{case}\ (v\ e_1 \ldots e_n)\ \text{of}\ p_1\ :\ e'_1 \mid \ldots \mid p_k\ :\ e'_k \\ & \mid & \text{case}\ (c\ e_1 \ldots e_n)\ \text{of}\ p_1\ :\ e'_1 \mid \ldots \mid p_k\ :\ e'_k \\ & \mid & \text{ALL}\ v_1 \ldots v_n.e \\ & \mid & \text{EX}\ v_1 \ldots v_n.e \end{array}$$

We define a set of rules $\mathcal{A}$ for handling universal quantifiers, and a set of rules $\mathcal{E}$ for handling existential quantifiers. A proof expression can be obtained by preprocessing the distilled form of expression described in §3.6. For proof expressions which are to be proved using these inductive proof rules, the output from applying these proof rules will be either $True$ if the input conjecture is true, or else $\perp$ which provides no information about the input conjecture. Within the proof expression, all free variables will be first order and the result type of the proof expression will be boolean. The proof expressions must therefore satisfy the form as shown in Fig. 4.3.

$$
\begin{array}{rcl}
bdt & := & v \\
 & | & True \\
 & | & False \\
 & | & \perp \\
 & | & \mathbf{case}\ v\ \mathbf{of}\ p_1\ :\ bdt_1 \mid \ldots \mid p_k\ :\ bdt_k \\
 & | & \mathbf{letrec}\ f\ =\ \lambda v_1 \ldots v_n.bdt\ \mathbf{in}\ f\ v_1 \ldots v_n \\
 & | & f\ bdt_1 \ldots bdt_n
\end{array}
$$

Figure 4.3: Form of proof expressions

The proof rules $\mathcal{A}$ and $\mathcal{E}$ will only be applied to expressions which are already in the form as shown in Fig. 4.3. The transformation rules $\mathcal{T}$ for distillation are extended to be able to handle explicit quantification as shown in Fig. 4.4.

The distillation rules $\mathcal{T}2$ and $\mathcal{T}3$ for quantifiers guide the whole proof process for two types of explicit quantifications: **ALL** and **EX**. This meta-level guidance essentially constructs a hierarchy of transformations which resembles *metasystem transitions* [103, 42].

Within the rules $\mathcal{T}2$ and $\mathcal{T}3$, the parameter ρ represents the set of the previously encountered expressions, and ϕ is the set of function definitions used within the current expression.

Rule ($\mathcal{T}2$) handles universally quantified expressions of the form $c\langle$ALL $v_1 \ldots v_n.e\rangle$ where the context $c\langle\rangle$ may be empty. The sub-expression e is transformed first using distillation (the quantified variables $v_1 \ldots v_n$ are treated as free variables). The proof rules $\mathcal{A}$ which are described in §4.4.1, are then applied to the resulting distilled expression. The set $\{v_1 \ldots v_n\}$ of the universally quantified

$$\mathcal{T}[\![c\langle \text{ALL } v_1 \ldots v_n.e\rangle]\!] \ \rho \ \phi \ = \mathcal{T}[\![c\langle e''\rangle]\!] \ \rho \ \phi \qquad (\mathcal{T}2)$$
where
$$e' = \mathcal{T}[\![e]\!] \ \{\} \ \phi$$
$$e'' = \mathcal{A}[\![e']\!] \ \{\} \ \{v_1 \ldots v_n\}$$

$$\mathcal{T}[\![c\langle \text{EX } v_1 \ldots v_n.e\rangle]\!] \ \rho \ \phi \ = \mathcal{T}[\![c\langle e''\rangle]\!] \ \rho \ \phi \qquad (\mathcal{T}3)$$
where
$$e' = \mathcal{T}[\![e]\!] \ \{\} \ \phi$$
$$e'' = \mathcal{E}[\![e']\!] \ \{\} \ \{v_1 \ldots v_n\}$$

Figure 4.4: Distillation rules for quantifiers

variables is passed as a parameter in the application of $\mathcal{A}$. Finally, the expression obtained from the application of $\mathcal{A}$ is transformed within the context (i.e., $c\langle e''\rangle$) using $\mathcal{T}$.

An existentially quantified expression $c\langle \text{EX } v_1 \ldots v_n.e\rangle$ is handled by rule ($\mathcal{T}3$) in a similar way. The sub-expression e is transformed first using the distillation rules $\mathcal{T}$. The proof rules $\mathcal{E}$ which are described in §4.4.2, are then applied to the resulting expression. The set $\{v_1 \ldots v_n\}$ of the existentially quantified variables is passed as a parameter in the application of $\mathcal{E}$. The expression obtained from the application of $\mathcal{E}$ is then transformed within the context using $\mathcal{T}$.

The distillation rules ($\mathcal{T}2$) and ($\mathcal{T}3$) can handle input conjectures containing quantifiers at any level of nesting. If a conjecture contains a number of nested quantifiers of different types (ALL and EX), then the proof rules will be applied to the innermost quantified expression first.

4.4 Inductive Theorem Proving in Poitín

In this section, we formally present the proof rules $\mathcal{A}$ and $\mathcal{E}$ for universal and existential quantification, which are used in Poitín to prove inductive conjectures.

4.4.1 Proving Universally Quantified Conjectures

The rules for proving universally quantified conjectures are defined by $\mathcal{A}[\![e]\!] \ \rho \ \phi$ as shown in Fig. 4.5, where the expression e is in the form of proof expressions shown

in Fig. 4.3, the parameter ρ is the set of previously encountered function calls and ϕ is the set of universally quantified variables.

For a universally quantified expression ALL $v_1 \ldots v_n.e$, the transformation rules $\mathcal{T}$ for distillation are first of all applied to the sub-expression e. Pre-processing is applied to the resulting expression to obtain a proof expression. The function calls within this proof expression are all potential inductive hypotheses. At least one of the parameters in the recursive call(s) of a function must be *decreasing*, and if all of the variables in the recursive call are universally quantified, then the inductive hypothesis can be applied, and the value *True* returned. Rule ($\mathcal{R}6$) in Fig. 4.1 tests whether a function call is an instance of a previous function call, and ensures that at least one of the parameters decreases in the recursive call within the resulting proof expression. The truth value of the conjecture is given by the final value obtained by applying the proof rules to the proof expression.

The proof rules $\mathcal{A}$ can be explained as follows. In rule ($\mathcal{A}1$), a variable v is encountered, which must be a Boolean. If v is universally quantified (i.e., v is contained in ϕ), then the undefined value $\perp$ is returned as v cannot always be $True$. If v is not universally quantified (i.e., v is not contained in ϕ), then it is free, so v remains unchanged. In rule ($\mathcal{A}2$), if the boolean value $True$ is encountered, we simply return it. In rule ($\mathcal{A}3$), if the boolean value $False$ is encountered, the undefined value $\perp$ is returned. In rule ($\mathcal{A}4$), the undefined value $\perp$ is encountered, which is returned unchanged. Rule ($\mathcal{A}5$) deals with a **case** expression, where the redex is a variable v. If v is universally quantified (i.e., v is contained in ϕ), then we try to prove the expression for all possible values of v within the expression. The different values of v are the patterns within the branches of the **case** expression. A *case split* is therefore performed in which the expression is separately proved for all values of v, and the *conjunction* of the resulting values is further transformed using distillation ($\mathcal{T}$). The different values of the expression for different values of v are the corresponding branches. The pattern variables are the sub-components of the universally quantified variable v, so they will also be universally quantified. Before applying the proof rules to each branch, the corresponding pattern variables are therefore added to ϕ. This rule simplifies the **case** expression by universal variable elimination from the selector. If v is not universally quantified (i.e., v is not contained in ϕ), then it is free, so it remains within the expression. The proof rules are then applied to the branches of the **case** expression.

In rule ($\mathcal{A}6$), a **letrec** expression is encountered. If all of the variables $v_1 \ldots v_n$ in

$$\mathcal{A}[\![v]\!]\ \rho\ \phi \ = \ \bot, \qquad \text{if } v \in \phi \qquad (\mathcal{A}1)$$
$$= \ v, \qquad \text{otherwise}$$

$$\mathcal{A}[\![True]\!]\ \rho\ \phi \ = True \qquad (\mathcal{A}2)$$

$$\mathcal{A}[\![False]\!]\ \rho\ \phi \ = \bot \qquad (\mathcal{A}3)$$

$$\mathcal{A}[\![\bot]\!]\ \rho\ \phi \ = \bot \qquad (\mathcal{A}4)$$

$$\mathcal{A}[\![\mathbf{case}\ v\ \mathbf{of}\ p_1\ :\ e_1 \mid \ldots \mid p_n\ :\ e_n]\!]\ \rho\ \phi \qquad (\mathcal{A}5)$$
$$= \ \mathcal{T}[\![(\mathcal{A}[\![e_1]\!]\ (\rho\ (\phi \cup \{v_{11} \ldots v_{1k_1}\}))) \ \wedge \ldots \wedge$$
$$(\mathcal{A}[\![e_n]\!]\ (\rho\ (\phi \cup \{v_{n1} \ldots v_{nk_n}\})))]\!]\ \{\}\ \{\}, \quad \text{if } v \in \phi$$
$$= \ \mathbf{case}\ v\ \mathbf{of}\ p_1\ :\ (\mathcal{A}[\![e_1]\!]\ \rho\ \phi) \mid \ldots \mid p_n\ :\ (\mathcal{A}[\![e_n]\!]\ \rho\ \phi), \qquad \text{otherwise}$$

where

$$p_i \ = \ c_i\ v_{i1} \ldots v_{ik_i}$$

$$\mathcal{A}[\![\mathbf{letrec}\ f \ = \ \lambda v_1 \ldots v_n.e_0\ \mathbf{in}\ f\ v_1 \ldots v_n]\!]\ \rho\ \phi \qquad (\mathcal{A}6)$$
$$= \ \mathcal{A}[\![e_0]\!]\ (\rho \cup \{f\ v_1 \ldots v_n\})\ \phi, \qquad \text{if } \{v_1 \ldots v_n\} \subseteq \phi$$
$$= \ \mathbf{letrec}\ f \ = \ \lambda v'_1 \ldots v'_k.(\mathcal{A}[\![e_0]\!]\ (\rho \cup \{f\ v_1 \ldots v_n\})\ \phi)\ \ \mathbf{in}\ f\ v'_1 \ldots v'_k,$$
$$\text{otherwise}$$

where

$$\{v'_1 \ldots v'_k\} = \{v_1 \ldots v_n\} \setminus \phi$$

$$\mathcal{A}[\![f\ e_1 \ldots e_n]\!]\ \rho\ \phi \qquad (\mathcal{A}7)$$
$$= \ True, \qquad \text{if } \{v_1 \ldots v_n\} \subseteq \phi$$
$$= \ (f\ v'_1 \ldots v'_k)[e_1/v_1 \ldots e_n/v_n], \qquad \text{otherwise}$$

where

$$(f\ v_1 \ldots v_n) \in \rho.(f\ v_1 \ldots v_n) \preceq (f\ e_1 \ldots e_n)$$
$$\{v'_1 \ldots v'_k\} = \{v_1 \ldots v_n\} \setminus \phi$$

Figure 4.5: Proof rules for universal quantification

the function application $f\ v_1 \ldots v_n$ within the expression are universally quantified, and therefore contained in ϕ, then the function application $f\ v_1 \ldots v_n$ is an *inductive hypothesis*. Since at least one of the variables $v_1 \ldots v_n$ must be decreasing and it is also universally quantified, this variable can be used as an *induction variable* during the proof of this expression. The proof rules are applied to the unfolded function call until the recursive call to function f is encountered. This recursive call is the re-occurrence of the inductive hypothesis $f\ v_1 \ldots v_n$. *Strong fertilization* can therefore be performed at this point by applying the inductive hypothesis. The expression resulting from the application of the proof rules to the body of the **letrec** expression is returned. If any of the variables $v_1 \ldots v_n$ in the function application $f\ v_1 \ldots v_n$ are not contained in ϕ, then the function application contains free variables, and the inductive hypothesis cannot be applied at this point. The function f is therefore redefined in terms of these free variables with a **letrec** expression using the result of applying the proof rules $\mathcal{A}$ to the unfolded function call. The function application $f\ v_1 \ldots v_n$ is added to the environment ρ before applying the proof rules $\mathcal{A}$ to the body e_0.

Rule ($\mathcal{A}7$) searches for the potential application of an inductive hypothesis. If there exists a function application $f\ v_1 \ldots v_n$ in ρ such that the recursive call $f\ e_1 \ldots e_n$ is an instance of $f\ v_1 \ldots v_n$, and all of the variables in the initial function application $f\ v_1 \ldots v_n$ are universally quantified (i.e., they are all contained in ϕ), then the inductive hypothesis is applied, and the value *True* returned. This corresponds to *strong fertilization*. At least one of the parameters within the application $f\ e_1 \ldots e_n$ must be decreasing, which is ensured by the pre-processing phase. If, on the other hand, the function application $f\ v_1 \ldots v_n$ contains free variables (i.e., not contained in ϕ), then strong fertilization cannot be performed. The recursive call $f\ e_1 \ldots e_n$ is therefore simplified to be defined over the arguments corresponding to these free variables.

Example 10

Consider the following conjecture (10.1) which states the *commutativity of plus* theorem for natural numbers.

$$\text{ALL } x.\text{ALL } y.eqnum\ (plus\ x\ y)\ (plus\ y\ x) \tag{10.1}$$

The proof of conjecture (10.1) is guided by distillation rule ($\mathcal{T}2$) (Fig. 4.4). Rule ($\mathcal{T}2$) applies distillation to expression (10.2).

$$eqnum\ (plus\ x\ y)\ (plus\ y\ x) \tag{10.2}$$

The transformation of expression (10.2) using distillation is shown in Example 17 in Appendix A.1. Rule ($\mathcal{T}2$) then applies the proof rules $\mathcal{A}$ for universal quantification to the proof expression resulting from the pre-processing of the distilled form of expression (10.2) as shown below.

$$
\begin{array}{l}
\mathcal{A}[\![\mathbf{letrec}\ f0 = \lambda x.\lambda y.\mathbf{case}\ x\ \mathbf{of} \\
\quad\quad Zero : \mathbf{case}\ y\ \mathbf{of} \\
\quad\quad\quad\quad Zero : True \\
\quad\quad\quad\quad \mid Succ\ y' : \mathbf{letrec}\ f1 = \lambda y'.\mathbf{case}\ y'\ \mathbf{of} \\
\quad\quad\quad\quad\quad\quad Zero : True \\
\quad\quad\quad\quad\quad\quad \mid Succ\ y'' : f1\ y'' \\
\quad\quad\quad\quad\quad \mathbf{in}\ f1\ y' \\
\quad\quad \mid Succ\ x' : \mathbf{case}\ y\ \mathbf{of} \\
\quad\quad\quad\quad Zero : \mathbf{letrec}\ f1 = \lambda x'.\mathbf{case}\ x'\ \mathbf{of} \\
\quad\quad\quad\quad\quad\quad Zero : True \\
\quad\quad\quad\quad\quad\quad \mid Succ\ x'' : f1\ x'' \\
\quad\quad\quad\quad\quad \mathbf{in}\ f1\ x' \\
\quad\quad\quad\quad \mid Succ\ y' : f0\ x'\ y' \\
\quad \mathbf{in}\ f0\ x\ y]\!]\ \{\}\ \{x,y\}
\end{array}
$$

(by $\mathcal{T}2$)

The proof using the rules $\mathcal{A}$ proceeds as shown below.

$$
\begin{array}{l}
= \mathcal{A}[\![\mathbf{case}\ x\ \mathbf{of} \quad\quad \text{(by } \mathcal{A}6\text{)} \\
\quad\quad Zero : \mathbf{case}\ y\ \mathbf{of} \\
\quad\quad\quad\quad Zero : True \\
\quad\quad\quad\quad \mid Succ\ y' : \mathbf{letrec}\ f1 = \lambda y'.\mathbf{case}\ y'\ \mathbf{of} \\
\quad\quad\quad\quad\quad\quad Zero : True \\
\quad\quad\quad\quad\quad\quad \mid Succ\ y'' : f1\ y'' \\
\quad\quad\quad\quad\quad \mathbf{in}\ f1\ y' \\
\quad\quad \mid Succ\ x' : \mathbf{case}\ y\ \mathbf{of} \\
\quad\quad\quad\quad Zero : \mathbf{letrec}\ f1 = \lambda x'.\mathbf{case}\ x'\ \mathbf{of} \\
\quad\quad\quad\quad\quad\quad Zero : True \\
\quad\quad\quad\quad\quad\quad \mid Succ\ x'' : f1\ x'' \\
\quad\quad\quad\quad\quad \mathbf{in}\ f1\ x' \\
\quad\quad\quad\quad \mid Succ\ y' : f0\ x'\ y']\!]\ \{f0\ x\ y\}\ \{x,y\}
\end{array}
$$

$$
\begin{array}{ll}
= \mathcal{T}[\![(\mathcal{A}[\![\textbf{case } y \textbf{ of} & \text{(by } \mathcal{A}5)\\
\quad \quad \quad \mathit{Zero} \quad : \mathit{True} & \\
\quad \quad \mid \mathit{Succ}\ y' : \ \textbf{letrec } \mathit{f1} \ = \ \lambda y'.\textbf{case } y' \textbf{ of} & \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \mathit{Zero} \quad : \mathit{True} & \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \mid \mathit{Succ}\ y'' : \mathit{f1}\ y'' & \\
\quad \quad \quad \quad \textbf{in } \mathit{f1}\ y']\!]\ \{\mathit{f0}\ x\ y\}\ \{x, y\}) & \\
\quad \quad \quad \quad \quad \wedge\ (\mathcal{A}[\![\textbf{case } y \textbf{ of} & \\
\quad \quad \quad \quad \quad \quad \quad \mathit{Zero} \quad : \ \textbf{letrec } \mathit{f1} \ = \ \lambda x'.\textbf{case } x' \textbf{ of} & \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \mathit{Zero} \quad : \mathit{True} & \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \mid \mathit{Succ}\ x'' : \mathit{f1}\ x'' & \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \textbf{in } \mathit{f1}\ x' & \\
\quad \quad \quad \quad \quad \quad \mid \mathit{Succ}\ y' : \mathit{f0}\ x'\ y']\!]\ \{\mathit{f0}\ x\ y\}\ \{x, y, x'\})]\!]\ \{\}\ \{\} &
\end{array}
$$

$$
\begin{array}{ll}
= \mathcal{T}[\![(\mathcal{T}[\![(\mathcal{A}[\![\mathit{True}]\!]\ \{\mathit{f0}\ x\ y\}\ \{x, y\}) \wedge (\mathcal{A}[\![\ \textbf{letrec } \mathit{f1} \ = \ \lambda y'.\textbf{case } y' \textbf{ of} & \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \mathit{Zero} \quad : \mathit{True} & \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \mid \mathit{Succ}\ y'' : \mathit{f1}\ y'' & \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \textbf{in } \mathit{f1}\ y']\!]\ \{\mathit{f0}\ x\ y\}\ \{x, y, y'\}))]\!]\ \{\}\ \{\} & \\
\quad \quad \quad \quad \quad \wedge\ (\mathcal{A}[\![\textbf{case } y \textbf{ of} & \\
\quad \quad \quad \quad \quad \quad \quad \mathit{Zero} \quad : \ \textbf{letrec } \mathit{f1} \ = \ \lambda x'.\textbf{case } x' \textbf{ of} & \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \mathit{Zero} \quad : \mathit{True} & \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \mid \mathit{Succ}\ x'' : \mathit{f1}\ x'' & \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \textbf{in } \mathit{f1}\ x' & \\
\quad \quad \quad \quad \quad \quad \mid \mathit{Succ}\ y' : \mathit{f0}\ x'\ y']\!]\ \{\mathit{f0}\ x\ y\}\ \{x, y, x'\})]\!]\ \{\}\ \{\} & \\
& \text{(by } \mathcal{A}5)
\end{array}
$$

$$
\begin{array}{ll}
= \mathcal{T}[\![(\mathcal{T}[\![\mathit{True}\ \wedge\ (\mathcal{A}[\![\textbf{case } y' \textbf{ of} & \\
\quad \quad \quad \quad \quad \quad \mathit{Zero} \quad : \mathit{True} & \\
\quad \quad \quad \quad \quad \mid \mathit{Succ}\ y'' : \mathit{f1}\ y'']\!]\ \{\mathit{f0}\ x\ y, \mathit{f1}\ y'\}\ \{x, y, y'\})]\!]\ \{\}\ \{\}) & \\
\quad \quad \quad \quad \quad \wedge\ (\mathcal{A}[\![\textbf{case } y \textbf{ of} & \\
\quad \quad \quad \quad \quad \quad \quad \mathit{Zero} \quad : \ \textbf{letrec } \mathit{f1} \ = \ \lambda x'.\textbf{case } x' \textbf{ of} & \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \mathit{Zero} \quad : \mathit{True} & \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \mid \mathit{Succ}\ x'' : \mathit{f1}\ x'' & \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \textbf{in } \mathit{f1}\ x' & \\
\quad \quad \quad \quad \quad \quad \mid \mathit{Succ}\ y' : \mathit{f0}\ x'\ y']\!]\ \{\mathit{f0}\ x\ y\}\ \{x, y, x'\})]\!]\ \{\}\ \{\} & \\
& \text{(by } \mathcal{A}2, \mathcal{A}6)
\end{array}
$$

$$
\begin{aligned}
= \mathcal{T}[\![(\mathcal{T}[\![True \ \wedge \ (\mathcal{T}[\![(\mathcal{A}[\![True]\!] \ \{f0 \ x \ y, f1 \ y'\} \ \{x,y,y'\}) \\
\wedge \ (\mathcal{A}[\![f1 \ y'']\!] \ \{f0 \ x \ y, f1 \ y'\} \ \{x,y,y',y''\})]\!] \ \{\} \ \{\})]\!] \ \{\} \ \{\}) \\
\wedge \ (\mathcal{A}[\![\mathbf{case} \ y \ \mathbf{of} \\
Zero \quad : \mathbf{letrec} \ f1 \ = \ \lambda x'.\mathbf{case} \ x' \ \mathbf{of} \\
Zero \quad : True \\
\mid Succ \ x'' : f1 \ x'' \\
\mathbf{in} \ f1 \ x' \\
\mid Succ \ y' : f0 \ x' \ y']\!] \ \{f0 \ x \ y\} \ \{x,y,x'\})]\!] \ \{\} \ \{\}
\end{aligned}
$$

(by $\mathcal{A}5$)

$$
\begin{aligned}
= \mathcal{T}[\![(\mathcal{T}[\![True \ \wedge (\mathcal{T}[\![True \ \wedge (\mathcal{A}[\![f1 \ y'']\!] \ \{f0 \ x \ y, f1 \ y'\} \ \{x,y,y',y''\})]\!] \ \{\} \ \{\}))]\!] \ \{\}\{\} \\
\wedge \ (\mathcal{A}[\![\mathbf{case} \ y \ \mathbf{of} \\
Zero \quad : \mathbf{letrec} \ f1 \ = \ \lambda x'.\mathbf{case} \ x' \ \mathbf{of} \\
Zero \quad : True \\
\mid Succ \ x'' : f1 \ x'' \\
\mathbf{in} \ f1 \ x' \\
\mid Succ \ y' : f0 \ x' \ y']\!] \ \{f0 \ x \ y\} \ \{x,y,x'\})]\!] \ \{\}\{\}
\end{aligned}
$$

(by $\mathcal{A}2$)

$$
\begin{aligned}
= \mathcal{T}[\![(\mathcal{T}[\![True \ \wedge \ (\mathcal{T}[\![True \ \wedge \ True]\!] \ \{\} \ \{\})]\!] \ \{\} \ \{\}) \qquad \text{(by } \mathcal{A}7) \\
\wedge \ (\mathcal{A}[\![\mathbf{case} \ y \ \mathbf{of} \\
Zero \quad : \mathbf{letrec} \ f1 \ = \ \lambda x'.\mathbf{case} \ x' \ \mathbf{of} \\
Zero \quad : True \\
\mid Succ \ x'' : f1 \ x'' \\
\mathbf{in} \ f1 \ x' \\
\mid Succ \ y' : f0 \ x' \ y']\!] \ \{f0 \ x \ y\} \ \{x,y,x'\})]\!] \ \{\} \ \{\}
\end{aligned}
$$

$$
\begin{aligned}
= \mathcal{T}[\![(\mathcal{T}[\![True \ \wedge True]\!] \ \{\} \ \{\}) \ \wedge \ (\mathcal{A}[\![\mathbf{case} \ y \ \mathbf{of} \\
Zero \quad : \mathbf{letrec} \\
f1 \ = \ \lambda x'.\mathbf{case} \ x' \ \mathbf{of} \\
Zero \quad : True \\
\mid Succ \ x'' : f1 \ x'' \\
\mathbf{in} \ f1 \ x' \\
\mid Succ \ y' : f0 \ x' \ y']\!] \ \{f0 \ x \ y\} \ \{x,y,x'\})]\!] \ \{\}\{\}
\end{aligned}
$$

(by $\mathcal{T}1$,$\mathcal{P}$)

$$
\begin{aligned}
&= \mathcal{T}[\![True \wedge (\mathcal{A}[\![\textbf{case } y \textbf{ of} && \text{(by } \mathcal{T}1,\mathcal{P})\\
&\qquad \textit{Zero} \quad : \textbf{letrec } \textit{f1} = \lambda x'.\textbf{case } x' \textbf{ of}\\
&\qquad\qquad\qquad\qquad \textit{Zero} \quad : \textit{True}\\
&\qquad\qquad\qquad\qquad \mid \textit{Succ } x'' : \textit{f1 } x''\\
&\qquad\qquad \textbf{in } \textit{f1 } x'\\
&\qquad \mid \textit{Succ } y' : \textit{f0 } x' \ y']\!] \ \{\textit{f0 } x \ y\} \ \{x,y,x'\})]\!] \ \{\} \ \{\}\\
&= \mathcal{T}[\![True \wedge (\mathcal{T}[\![(\mathcal{A}[\![\textbf{ letrec } \textit{f1} = \lambda x'.\textbf{case } x' \textbf{ of}\\
&\qquad\qquad\qquad\qquad \textit{Zero} \quad : \textit{True}\\
&\qquad\qquad\qquad\qquad \mid \textit{Succ } x'' : \textit{f1 } x''\\
&\qquad\qquad \textbf{in } \textit{f1 } x']\!] \ \{\textit{f0 } x \ y\} \ \{x,y,x'\})\\
&\qquad\qquad \wedge (\mathcal{A}[\![\textit{f0 } x' \ y']\!] \ \{\textit{f0 } x \ y\} \ \{x,y,x',y'\})]\!] \ \{\} \ \{\})]\!] \ \{\} \ \{\}\\
& && \text{(by } \mathcal{A}5)\\
&= \mathcal{T}[\![True \wedge (\mathcal{T}[\![(\mathcal{A}[\![\textbf{case } x' \textbf{ of} && \text{(by } \mathcal{A}6)\\
&\qquad\qquad \textit{Zero} \quad : \textit{True}\\
&\qquad\qquad \mid \textit{Succ } x'' : \textit{f1 } x'']\!] \ \{\textit{f0 } x \ y, \textit{f1 } x'\} \ \{x,y,x'\})\\
&\qquad\qquad \wedge \ (\mathcal{A}[\![\textit{f0 } x' \ y']\!] \ \{\textit{f0 } x \ y\} \ \{x,y,x',y'\})]\!] \ \{\} \ \{\})]\!] \ \{\} \ \{\}\\
&= \mathcal{T}[\![True \wedge (\mathcal{T}[\![(\mathcal{T}[\![(\mathcal{A}[\![True]\!] \ \{\textit{f0 } x \ y, \textit{f1 } x'\} \ \{x,y,x'\}) && \text{(by } \mathcal{A}5)\\
&\qquad\qquad \wedge (\mathcal{A}[\![\textit{f1 } x'']\!] \ \{\textit{f0 } x \ y, \textit{f1 } x'\} \ \{x,y,x',x''\})]\!] \ \{\} \ \{\})\\
&\qquad\qquad \wedge (\mathcal{A}[\![\textit{f0 } x' \ y']\!] \ \{\textit{f0 } x \ y\} \ \{x,y,x',y'\})]\!] \ \{\} \ \{\})]\!] \ \{\} \ \{\}\\
&= \mathcal{T}[\![True \wedge (\mathcal{T}[\![(\mathcal{T}[\![True \wedge True]\!] \ \{\} \ \{\})\\
&\qquad \wedge (\mathcal{A}[\![\textit{f0 } x' \ y']\!] \ \{\textit{f0 } x \ y\} \ \{x,y,x',y'\})]\!] \ \{\} \ \{\})]\!] \ \{\} \ \{\} && \text{(by } \mathcal{A}2,\mathcal{A}7)\\
&= \mathcal{T}[\![True \wedge (\mathcal{T}[\![True \wedge (\mathcal{A}[\![\textit{f0 } x' \ y']\!] \ \{\textit{f0 } x \ y\} \ \{x,y,x',y'\})]\!] \ \{\} \ \{\})]\!] \ \{\} \ \{\}\\
& && \text{(by } \mathcal{T}1,\mathcal{P})\\
&= \mathcal{T}[\![True \wedge (\mathcal{T}[\![True \wedge True]\!] \ \{\} \ \{\})]\!] \ \{\} \ \{\} && \text{(by } \mathcal{A}7)\\
&= \mathcal{T}[\![True \wedge True]\!] \ \{\} \ \{\} && \text{(by } \mathcal{T}1,\mathcal{P})\\
&= True && \text{(by } \mathcal{T}1,\mathcal{P})
\end{aligned}
$$

We obtain the truth value *True* by simplifying conjecture (10.1) as required. This completes the proof of the *commutativity* of plus theorem for natural numbers. The proof of this conjecture is particularly troublesome for most inductive theorem provers. The proof of the commutativity theorem is given in §2.6.2 using 1-step induction for *nat* on the induction variable x considering x as the primary induction variable. Recursion analysis/ripple analysis suggest both of the variables x and y as induction variables even though both of them have unflawed as well as flawed

occurrences. No unflawed induction variable is available. Multiple induction variables complicate the preconditions of the method for induction strategy of a proof planning-based theorem prover, e.g., CLAM [14]. In this case, no choice of induction variable fully meets the preconditions. The generation of the induction rule and the control loop for induction and rewriting of the induction conclusion with multiple induction variables become more complicated for non-proof planning-based inductive theorem provers.

4.4.2 Proving Existentially Quantified Conjectures

The rules for proving existentially quantified conjectures are defined with a set of rules $\mathcal{E}$ by $\mathcal{E}[\![e]\!]\ \rho\ \phi$ as shown in Fig. 4.6, where the expression e is in the form of proof expressions shown in Fig. 4.3, the parameter ρ is the set of previously encountered function calls and the environment ϕ is the set of existentially quantified variables appearing within the conjecture.

The proof rules $\mathcal{E}$ can be explained as follows. In rule ($\mathcal{E}1$), a variable v is encountered, which must be a Boolean. If v is existentially quantified (i.e., v is contained in ϕ), then $True$ is returned as the value of v can be $True$. If v is not existentially quantified, then it is free, so we return it unchanged. In rule ($\mathcal{E}2$), if the boolean value $True$ is encountered, we simply return it. In rule ($\mathcal{E}3$), if the boolean value $False$ is encountered, the undefined value $\bot$ is returned.

In rule ($\mathcal{E}4$), the undefined value $\bot$ is encountered, which is returned unchanged. Rule ($\mathcal{E}5$) deals with a **case** expression, where the redex is a variable v. If v is existentially quantified (i.e., v is contained in ϕ), then we try to find some value of v for which the expression can be proved $True$. The different values of v which are used to prove the expression are the patterns within the branches of the **case** expression. A *case split* is therefore performed in which the expression is separately proved for each value of v. The *disjunction* of the resulting values is further transformed using distillation ($\mathcal{T}$). The different values of the expression for different values of v are the corresponding branches. The pattern variables are the sub-components of the existentially quantified variable v, so they will also be existentially quantified. Before applying the proof rules to each branch, the corresponding pattern variables are therefore added to ϕ. This rule simplifies the **case** expression by existential variable elimination from the selector. If v is not existentially quantified (i.e., v is not contained in ϕ), then it is free, so it remains within the expression. The proof rules are then applied to the branches of the **case** expression.

$$\mathcal{E}[\![v]\!]\ \rho\ \phi = True, \qquad \text{if } v \in \phi \qquad (\mathcal{E}1)$$
$$= v, \qquad \text{otherwise}$$

$$\mathcal{E}[\![True]\!]\ \rho\ \phi\ = True \qquad (\mathcal{E}2)$$

$$\mathcal{E}[\![False]\!]\ \rho\ \phi\ = \bot \qquad (\mathcal{E}3)$$

$$\mathcal{E}[\![\bot]\!]\ \rho\ \phi\ = \bot \qquad (\mathcal{E}4)$$

$$\mathcal{E}[\![\textbf{case}\ v\ \textbf{of}\ p_1\ :\ e_1 \mid \ldots \mid p_n\ :\ e_n]\!]\ \rho\ \phi \qquad (\mathcal{E}5)$$
$$= \mathcal{T}[\![(\mathcal{E}[\![e_1]\!]\ (\rho\ (\phi \cup \{v_{11} \ldots v_{1k_1}\}))) \vee \ldots \vee$$
$$(\mathcal{E}[\![e_n]\!]\ (\rho\ (\phi \cup \{v_{n1} \ldots v_{nk_n}\})))]\!]\ \{\}\ \{\}, \qquad \text{if } v \in \phi$$
$$= \textbf{case}\ v\ \textbf{of}\ p_1\ :\ (\mathcal{E}[\![e_1]\!]\ \rho\ \phi) \mid \ldots \mid p_n\ :\ (\mathcal{E}[\![e_n]\!]\ \rho\ \phi), \qquad \text{otherwise}$$

where

$$p_i\ =\ c_i\ v_{i1} \ldots v_{ik_i}$$

$$\mathcal{E}[\![\textbf{letrec}\ f\ =\ \lambda v_1 \ldots v_n.e_0\ \textbf{in}\ f\ v_1 \ldots v_n]\!]\ \rho\ \phi \qquad (\mathcal{E}6)$$
$$= \mathcal{E}[\![e_0]\!]\ (\rho \cup \{f\ v_1 \ldots v_n\})\ \phi, \qquad \text{if } \{v_1 \ldots v_n\} \subseteq \phi$$
$$= \textbf{letrec}\ f\ =\ \lambda v'_1 \ldots v'_k.(\mathcal{E}[\![e_0]\!]\ (\rho \cup \{f\ v_1 \ldots v_n\})\ \phi)\ \textbf{in}\ f\ v'_1 \ldots v'_k, \qquad \text{otherwise}$$

where

$$\{v'_1 \ldots v'_k\} = \{v_1 \ldots v_n\} \setminus \phi$$

$$\mathcal{E}[\![f\ e_1 \ldots e_n]\!]\ \rho\ \phi \qquad (\mathcal{E}7)$$
$$=\ \bot, \qquad \text{if } \{v_1 \ldots v_n\} \subseteq \phi$$
$$=\ (f\ v'_1 \ldots v'_k)[e_1/v_1 \ldots e_n/v_n], \qquad \text{otherwise}$$

where

$$(f\ v_1 \ldots v_n) \in \rho.(f\ v_1 \ldots v_n) \lessdot (f\ e_1 \ldots e_n)$$
$$\{v'_1 \ldots v'_k\} = \{v_1 \ldots v_n\} \setminus \phi$$

Figure 4.6: Proof rules for existential quantification

Rule ($\mathcal{E}6$) deals with a **letrec** expression. If all of the variables $v_1 \ldots v_n$ in the function application $f\ v_1 \ldots v_n$ within the expression are existentially quantified (i.e., $v_1 \ldots v_n$ are contained in ϕ), then the function application $f\ v_1 \ldots v_n$ does not contain any free variables. In this case, the function definition is removed, and the result of applying the proof rules to the unfolded function call is returned. If any of the variables $v_1 \ldots v_n$ in the function application $f\ v_1 \ldots v_n$ are not contained in ϕ, then the function application contains free variables. The function f is therefore redefined in terms of these free variables with a **letrec** expression using the result of applying the proof rules $\mathcal{E}$ to the body of the function. The function application $f\ v_1 \ldots v_n$ is added to the environment ρ before applying the proof rules $\mathcal{E}$ to the body e_0.

In rule ($\mathcal{E}7$), a recursive function call $f\ e_1 \ldots e_n$ is encountered. If there exists a function application $f\ v_1 \ldots v_n$ in ρ such that the recursive call $f\ e_1 \ldots e_n$ is an instance of $f\ v_1 \ldots v_n$, and all of the variables $v_1 \ldots v_n$ are existentially quantified (i.e., they are all contained in ϕ), the value $\bot$ is returned as the search space associated with the existential variables $v_1 \ldots v_n$ is exhausted. If, on the other hand, the function application $f\ v_1 \ldots v_n$ contains free variables (i.e., not contained in ϕ), then the recursive call $f\ e_1 \ldots e_n$ is simplified to be defined over the arguments corresponding to these free variables.

Example 11

Consider the proof of the following conjecture (11.1) about natural numbers, which states that for every value of x, there exists a y such that x is even if and only if the double of y equals x. We adopted this example from [68] rearranging the existential quantifier for our purposes.

$$\text{ALL } x.\text{EX } y.\mathit{iff}\ (\mathit{even}\ x)\ (\mathit{eqnum}\ (\mathit{double}\ y)\ x) \tag{11.1}$$

The proof process is guided by distillation rules ($\mathcal{T}2$) and ($\mathcal{T}3$) (Fig. 4.4) when conjecture (11.1) is input to the theorem prover Poitín. The existential proof rules $\mathcal{E}$ will be applied to the innermost existential quantifier first by rule ($\mathcal{T}3$) to conjecture (11.1). Rule ($\mathcal{T}3$) applies distillation to expression (11.2).

$$\mathit{iff}\ (\mathit{even}\ x)\ (\mathit{eqnum}\ (\mathit{double}\ y)\ x) \tag{11.2}$$

The transformation of expression (11.2) using distillation is shown in Example 18 in Appendix A.1. The distilled expression has been converted to a proof expression by adding the free variables within the function body as parameters in the local

function by pre-processing. Rule ($\mathcal{T}3$) applies the proof rules $\mathcal{E}$ for existential quantification to the proof expression obtained by pre-processing the distilled expression resulting from the transformation of expression (11.2).

$$
\begin{array}{l}
\mathcal{E}[\![\textbf{ letrec } f0 \; = \; \lambda x.\lambda y.\textbf{case } x \textbf{ of} \\
\qquad\qquad\qquad\qquad \begin{array}{l} \textit{Zero} \quad : \textbf{case } y \textbf{ of} \\ \qquad\qquad \begin{array}{l} \textit{Zero} \quad : \textit{True} \\ \mid \textit{Succ } y' : \textit{False} \end{array} \\ \mid \textit{Succ } x' : \textbf{case } x' \textbf{ of} \\ \qquad\qquad \begin{array}{l} \textit{Zero} \quad : \textbf{case } y \textbf{ of} \\ \qquad\qquad \begin{array}{l} \textit{Zero} \quad : \textit{True} \\ \mid \textit{Succ } y' : \textit{True} \end{array} \\ \mid \textit{Succ } x'' : f0 \; x'' \; y \end{array} \end{array} \\
\quad \textbf{in } f0 \; x \; y]\!] \; \{\} \; \{y\}
\end{array}
\qquad (\text{by } \mathcal{T}3)
$$

The function application $f0 \; x \; y$ within the **letrec** expression contains the free variable x (universally quantified in the outer scope) and an existential variable y. The function $f0$ will be redefined with the result of applying the proof rules $\mathcal{E}$ to the body of the function by rule $\mathcal{E}6$. The original function call $f0 \; x \; y$ is simplified to $f0 \; x$ by removing the existential variable y during the application of rule $\mathcal{E}6$. The application of the proof rules $\mathcal{E}$ to the unfolded function call proceeds as shown below.

$$
\begin{array}{l}
\textbf{letrec } f0 \; = \; \lambda x.\mathcal{E}[\![\textbf{case } x \textbf{ of} \\
\qquad\qquad\qquad\qquad \begin{array}{l} \textit{Zero} \quad : \textbf{case } y \textbf{ of} \\ \qquad\qquad \begin{array}{l} \textit{Zero} \quad : \textit{True} \\ \mid \textit{Succ } y' : \textit{False} \end{array} \\ \mid \textit{Succ } x' : \textbf{case } x' \textbf{ of} \\ \qquad\qquad \begin{array}{l} \textit{Zero} \quad : \textbf{case } y \textbf{ of} \\ \qquad\qquad \begin{array}{l} \textit{Zero} \quad : \textit{True} \\ \mid \textit{Succ } y' : \textit{True} \end{array} \\ \mid \textit{Succ } x'' : f0 \; x'' \; y]\!] \; \{f0 \; x \; y\} \; \{y\} \end{array} \end{array} \\
\textbf{in } f0 \; x
\end{array}
\qquad (\text{by } \mathcal{E}6)
$$

$$
\begin{array}{lll}
= & \textbf{letrec}\ f0\ =\ \lambda x.\textbf{case}\ x\ \textbf{of} & (\text{by } \mathcal{E}5)\\
& \quad \textit{Zero} \quad : \mathcal{E}[\![\textbf{case}\ y\ \textbf{of} & \\
& \qquad \textit{Zero} \quad : \textit{True} & \\
& \qquad \mid \textit{Succ}\ y' : \textit{False}]\!]\ \{f0\ x\ y\}\ \{y\} & \\
& \quad \mid \textit{Succ}\ x' : \mathcal{E}[\![\textbf{case}\ x'\ \textbf{of} & \\
& \qquad \textit{Zero} \quad : \textbf{case}\ y\ \textbf{of} & \\
& \qquad\quad \textit{Zero} \quad : \textit{True} & \\
& \qquad\quad \mid \textit{Succ}\ y' : \textit{True} & \\
& \qquad \mid \textit{Succ}\ x'' : f0\ x''\ y]\!]\ \{f0\ x\ y\}\ \{y\} & \\
& \textbf{in}\ f0\ x &
\end{array}
$$

$$
\begin{array}{lll}
= & \textbf{letrec} & (\text{by } \mathcal{E}5)\\
& f0\ =\ \lambda x.\textbf{case}\ x\ \textbf{of} & \\
& \quad \textit{Zero} \quad : \mathcal{T}[\![(\mathcal{E}[\![\textit{True}]\!]\ \{f0\ x\ y\}\ \{y\}) & \\
& \qquad \vee (\mathcal{E}[\![\textit{False}]\!]\ \{f0\ x\ y\}\ \{y, y'\})]\!]\ \{\}\ \{\} & \\
& \quad \mid \textit{Succ}\ x' : \mathcal{E}[\![\textbf{case}\ x'\ \textbf{of} & \\
& \qquad \textit{Zero} \quad : \textbf{case}\ y\ \textbf{of} & \\
& \qquad\quad \textit{Zero} \quad : \textit{True} & \\
& \qquad\quad \mid \textit{Succ}\ y' : \textit{True} & \\
& \qquad \mid \textit{Succ}\ x'' : f0\ x''\ y]\!]\ \{f0\ x\ y\}\ \{y\} & \\
& \textbf{in}\ f0\ x &
\end{array}
$$

$$
\begin{array}{lll}
= & \textbf{letrec} & (\text{by } \mathcal{E}2, \mathcal{E}3)\\
& f0\ =\ \lambda x.\textbf{case}\ x\ \textbf{of} & \\
& \quad \textit{Zero} \quad : \mathcal{T}[\![\textit{True} \vee \bot]\!]\ \{\}\ \{\} & \\
& \quad \mid \textit{Succ}\ x' : \mathcal{E}[\![\textbf{case}\ x'\ \textbf{of} & \\
& \qquad \textit{Zero} \quad : \textbf{case}\ y\ \textbf{of} & \\
& \qquad\quad \textit{Zero} \quad : \textit{True} & \\
& \qquad\quad \mid \textit{Succ}\ y' : \textit{True} & \\
& \qquad \mid \textit{Succ}\ x'' : f0\ x''\ y]\!]\ \{f0\ x\ y\}\ \{y\} & \\
& \textbf{in}\ f0\ x &
\end{array}
$$

$$
\begin{array}{rl}
= & \textbf{letrec } \mathit{f0} \ = \ \lambda x.\textbf{case } x \textbf{ of} \\
& \qquad \mathit{Zero} \quad : \mathit{True} \\
& \qquad \mid \mathit{Succ}\ x' : \textbf{case } x' \textbf{ of} \\
& \qquad\qquad \mathit{Zero} \quad : \mathcal{E}[\![\textbf{case } y \textbf{ of} \\
& \qquad\qquad\qquad \mathit{Zero} \quad : \mathit{True} \\
& \qquad\qquad\qquad \mid \mathit{Succ}\ y' : \mathit{True}]\!]\ \{\mathit{f0}\ x\ y\}\ \{y\} \\
& \qquad\qquad \mid \mathit{Succ}\ x'' : \mathcal{E}[\![\mathit{f0}\ x''\ y]\!]\ \{\mathit{f0}\ x\ y\}\ \{y\} \\
& \textbf{in } \mathit{f0}\ x
\end{array}
$$

(by $(\mathcal{T}1,\mathcal{P}),\mathcal{E}5$)

$$
\begin{array}{rl}
= & \textbf{letrec } \mathit{f0} \ = \ \lambda x.\textbf{case } x \textbf{ of} \\
& \qquad \mathit{Zero} \quad : \mathit{True} \\
& \qquad \mid \mathit{Succ}\ x' : \textbf{case } x' \textbf{ of} \\
& \qquad\qquad \mathit{Zero} \quad : \mathit{True} \\
& \qquad\qquad \mid \mathit{Succ}\ x'' : \mathcal{E}[\![\mathit{f0}\ x''\ y]\!]\ \{\mathit{f0}\ x\ y\}\ \{y\} \\
& \textbf{in } \mathit{f0}\ x
\end{array}
$$

(by $\mathcal{E}5,\mathcal{E}2,\mathcal{E}2,(\mathcal{T}1,\mathcal{P})$)

The recursive function application $\mathit{f0}\ x''\ y$ contains the free variable x''. The function application is simplified to define in terms of the free variable x'' by removing the existential variable y by rule $\mathcal{E}7$. This results in the following expression.

$$
\begin{array}{l}
\textbf{letrec } \mathit{f0} \ = \ \lambda x.\textbf{case } x \textbf{ of} \qquad\qquad (\text{by } \mathcal{E}7) \\
\qquad \mathit{Zero} \quad : \mathit{True} \\
\qquad \mid \mathit{Succ}\ x' : \textbf{case } x' \textbf{ of} \\
\qquad\qquad \mathit{Zero} \quad : \mathit{True} \\
\qquad\qquad \mid \mathit{Succ}\ x'' : \mathit{f0}\ x'' \\
\textbf{in } \mathit{f0}\ x
\end{array}
$$

The proof rules $\mathcal{A}$ for universal quantification are now applied to this expression. During the application of the rules $\mathcal{A}$, the universally quantified variable x within expression (11.1) is passed as a singleton set of universally quantified variables $\{x\}$. The proof of the above expression proceeds as shown below.

$$
\begin{array}{l}
\mathcal{A}[\![\ \textbf{letrec}\ \mathit{f0}\ =\ \lambda x.\textbf{case}\ x\ \textbf{of} \\
\qquad\qquad\qquad\qquad \mathit{Zero} \quad : \mathit{True} \\
\qquad\qquad\qquad\quad\ \mid \mathit{Succ}\ x' : \textbf{case}\ x'\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{Zero} \quad : \mathit{True} \\
\qquad\qquad\qquad\qquad\qquad\quad\ \mid \mathit{Succ}\ x'' : \mathit{f0}\ x'' \\
\quad \textbf{in}\ \mathit{f0}\ x]\!]\ \{\}\ \{x\}
\end{array}
\qquad \text{(by } \mathcal{T}2\text{)}
$$

$$
\begin{array}{l}
= \mathcal{A}[\![\textbf{case}\ x\ \textbf{of} \\
\qquad \mathit{Zero} \quad : \mathit{True} \\
\quad\ \mid \mathit{Succ}\ x' : \textbf{case}\ x'\ \textbf{of} \\
\qquad\qquad\quad \mathit{Zero} \quad : \mathit{True} \\
\qquad\qquad\ \mid \mathit{Succ}\ x'' : \mathit{f0}\ x'']\!]\ \{\mathit{f0}\ x\}\ \{x\}
\end{array}
\qquad \text{(by } \mathcal{A}6\text{)}
$$

$$
\begin{array}{l}
= \mathcal{T}[\![(\mathcal{A}[\![True]\!]\ \{\mathit{f0}\ x\}\ \{x\})\ \wedge\ (\mathcal{A}[\![\textbf{case}\ x'\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{Zero} \quad : \mathit{True} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad\ \mid \mathit{Succ}\ x'' : \mathit{f0}\ x'']\!]\ \{\mathit{f0}\ x\}\ \{x, x'\})]\!]\ \{\}\ \{\} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(by } \mathcal{A}5\text{)} \\
= \mathcal{T}[\![True\ \wedge\ (\mathcal{T}[\![(\mathcal{A}[\![True]\!]\ \{\mathit{f0}\ x\}\ \{x, x'\}) \\
\qquad\qquad\qquad\qquad \wedge\ (\mathcal{A}[\![f0\ x'']\!]\ \{\mathit{f0}\ x\}\ \{x, x', x''\})]\!]\ \{\}\ \{\})]\!]\ \{\}\ \{\} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(by } \mathcal{A}2,\mathcal{A}5\text{)}
\end{array}
$$

$$
\begin{array}{ll}
= \mathcal{T}[\![True\ \wedge (\mathcal{T}[\![True \wedge\ (\mathcal{A}[\![f0\ x'']\!]\ \{\mathit{f0}\ x\}\ \{x, x', x''\})]\!]\ \{\}\ \{\})]\!]\ \{\}\ \{\} & \text{(by } \mathcal{A}2\text{)} \\
= \mathcal{T}[\![True\ \wedge (\mathcal{T}[\![True\ \wedge\ True]\!]\ \{\}\ \{\})]\!]\ \{\}\ \{\} & \text{(by } \mathcal{A}7\text{)} \\
= \mathcal{T}[\![True\ \wedge\ True]\!]\ \{\}\ \{\} & \text{(by } \mathcal{T}1,\mathcal{P}\text{)} \\
= True & \text{(by } \mathcal{T}1,\mathcal{P}\text{)}
\end{array}
$$

We obtain the truth value $True$ by simplifying conjecture (11.1) as required. This completes the proof of conjecture (11.1) and demonstrates that it is a theorem.

4.5 Soundness of Proof Techniques

In order to show that our proof rules are sound, we need to show that for every conjecture which is found to be $True$ in our proof rules, there is a corresponding logical proof of the conjecture. To facilitate this, we define sequent calculus rules for the proof expression obtained from the distilled form of input conjecture as shown in Fig. 4.7.

$$\frac{}{A \vdash A}\ (\text{Id})$$

$$\frac{}{\Gamma \vdash True, \Delta}\ (\text{True-R})$$

$$\frac{}{False, \Gamma \vdash \Delta}\ (\text{False-L})$$

$$\frac{\Gamma \vdash e[x], \Delta}{\Gamma \vdash ALL\ v.e\ [v/x], \Delta}\ (\text{ALL-R})$$

$$\frac{\Gamma, e_0[e_1] \vdash \Delta}{\Gamma, ALL\ v.e_0\ [v/e_1] \vdash \Delta}\ (\text{ALL-L})$$

$$\frac{\Gamma \vdash e_0[e_1], \Delta}{\Gamma \vdash EX\ v.e_0[v/e_1], \Delta}\ (\text{EX-R})$$

$$\frac{\Gamma, e[x] \vdash \Delta}{\Gamma, EX\ v.e\ [v/x] \vdash \Delta}\ (\text{EX-L})$$

$$\frac{\Gamma, f\ v_1 \ldots v_n \vdash e_0, \Delta}{\Gamma \vdash \mathbf{letrec}\ f\ =\ \lambda v_1 \ldots v_n.e_0\ \mathbf{in}\ f\ v_1 \ldots v_n, \Delta}\ (\text{Ind})$$

$$\frac{\Gamma, v = p_1 \vdash e_1, \Delta\ \ldots\ \Gamma, v = p_k \vdash e_k, \Delta}{\Gamma \vdash \mathbf{case}\ v\ \mathbf{of}\ p_1\ :\ e_1\ |\ \ldots\ |\ p_k\ :\ e_k, \Delta}\ (\text{Case})$$

$$\frac{}{\Gamma, ALL\ \ v_1 \ldots v_n.f\ v_1 \ldots v_n \vdash f\ e_1 \ldots e_n, \Delta}\ (\text{IndHyp})$$

Figure 4.7: Sequent calculus rules for language

In these rules, the (Id) rule and the quantifier rules are standard sequent calculus rules for first order logic by Gentzen. The rest of the rules are defined for some of the expressions in our language. In this formulation, there is no need for a cut rule as all the intermediate structures in the input program will have been eliminated.

The (Id) rule states that from the assumption A, one can deduce A. The (True-R) rule states that from the assumption Γ, one can deduce $True$ and Δ. The (False-L) rule states that from the assumption $False$ and Γ, one can deduce Δ. The (ALL-R) rule states that $ALL\ v.e[v/x]$ holds only if $e[x]$ is true (i.e., x must not be free within Γ, $e[v/x]$ or Δ). The (ALL-L) rule states that if one knows that $ALL\ v.e_0\ [v/e_1]$ is true, then it can be proved for any term e_1 (i.e., $e_0[e_1]$). The (EX-R) rule states that $EX\ v.e_0[v/e_1]$ holds only if $e_0[e_1]$ is true for any term e_1. The (EX-L) rule states that if one knows that $EX\ v.e[v/x]$ is true, then it can be proved for any value x (i.e., x must not be free within $\Gamma, e[v/x]$ or Δ). The (Ind) rule states that the expression **letrec** $f = \lambda v_1 \ldots v_n.e_0$ **in** $f\ v_1 \ldots v_n$ holds only if one can deduce e_0 from the assumptions Γ and $f\ v_1 \ldots v_n$. The (Case) rule states that the expression **case** v **of** $p_1 : e_1 \mid \ldots \mid p_k : e_k$ holds only if one can deduce $e_1, \ldots, e_k$ from the assumptions Γ, and $p_1, \ldots, p_k$ as the different values of v. The (IndHyp) rule states that if one knows that $ALL\ \ v_1 \ldots v_n.f\ v_1 \ldots v_n$ is true, then one can deduce $f\ e_1 \ldots e_n$.

Theorem 4.5.1 (Soundness of proof rules) *The proof rules $\mathcal{A}$ and $\mathcal{E}$ are sound with respect to the sequent calculus rules defined in Fig. 4.7.*

Proof. (Theorem 4.5.1)
The proof of this theorem follows immediately from lemmata 4.5.2 and 4.5.3.

Lemma 4.5.2 (Soundness of universal proof rules)

$$\mathcal{A}[\![e]\!]\ \rho\ \{v_1 \ldots v_n\} \ = \ True \ \Rightarrow \ \rho \ \vdash \ ALL\ v_1 \ldots v_n.e$$

Lemma 4.5.3 (Soundness of existential proof rules)

$$\mathcal{E}[\![e]\!]\ \rho\ \{v_1 \ldots v_n\} \ = \ True \ \Rightarrow \ \rho \ \vdash \ EX\ v_1 \ldots v_n.e$$

Proof. (Lemma 4.5.2)
The proof of this is by recursion induction on the proof rules $\mathcal{A}$.

Base Cases

Case for Rule $\mathcal{A}1$:

If $v \in \phi$:

$\mathcal{A}[\![v]\!]\ \rho\ \phi \ = \ \bot \ \neq \ True$

In this case, the proof expression is equivalent to $ALL\ v.v$, for which there is no corresponding sequent calculus proof.

If $v \notin \phi$, then v is free.

$\mathcal{A}[\![v]\!]\ \rho\ \phi \ = \ v \ \neq \ True$

Case for Rule $\mathcal{A}2$:

$\mathcal{A}[\![True]\!]\ \rho\ \phi \ = \ True$

The corresponding sequent calculus proof fragment is as follows:

$$\frac{}{\Gamma \vdash True}\ \text{(True-R)}$$

Case for Rule $\mathcal{A}3$:

$\mathcal{A}[\![False]\!]\ \rho\ \phi \ = \ \bot \ \neq \ True$

As the expression is $False$, there is no corresponding sequent calculus proof.

Case for Rule $\mathcal{A}4$:

$\mathcal{A}[\![\bot]\!]\ \rho\ \phi \ = \ \bot \ \neq \ True$

As the expression is undefined, there is no corresponding sequent calculus proof.

Case for Rule $\mathcal{A}7$:

If $\{v_1 \ldots v_n\} \subseteq \phi$, then there must exist an inductive hypothesis of the form $f\ v_1 \ldots v_n$.

$\mathcal{A}[\![f\ e_1 \ldots e_n]\!]\ \rho\ \phi \ = \ True$

The corresponding sequent calculus proof fragment is as follows:

$$\frac{}{\Gamma, ALL\ \ v_1 \ldots v_n . f\ v_1 \ldots v_n\ \vdash\ f\ e_1 \ldots e_n}\ \text{(IndHyp)}$$

If $\{v_1 \ldots v_n\} \not\subseteq \phi$, then the function call remains.

$\mathcal{A}[\![f\ e_1 \ldots e_n]\!]\ \rho\ \phi\ =\ f\ v'_1 \ldots v'_k \neq True$

Inductive cases

Case for Rule $\mathcal{A}5$:

By the inductive hypothesis:

$$\forall i \in \{1 \ldots k\}.\mathcal{A}[\![e_i]\!]\ \rho\ \{v_1 \ldots v_n\}\ =\ True\ \Rightarrow\ \rho\ \vdash\ ALL\ v_1 \ldots v_n.e_i$$

If $v \in \phi$, then the proof expression is equivalent to

$$ALL\ v.\textbf{case}\ v\ \textbf{of}\ p_1\ :\ e_1\ |\ \ldots\ |\ p_k\ :\ e_k$$

$\mathcal{A}[\![\textbf{case}\ v\ \textbf{of}\ p_1\ :\ e_1|\ \ldots\ |\ p_k\ :\ e_k]\!]\ \rho\ \phi$

$=\ \mathcal{T}[\![(\mathcal{A}[\![e_1]\!]\ \rho\ (\phi \cup \{v_{11} \ldots v_{1k_1}\}))\ \wedge \ldots \wedge\ (\mathcal{A}[\![e_k]\!]\ \rho\ (\phi \cup \{v_{n1} \ldots v_{nk_n}\}))]\!]\ \{\}\ \{\}$

The corresponding sequent calculus proof fragment is as follows:

$$\frac{\dfrac{\Gamma, v = p_1\ \vdash\ e_1\ \ldots\ \Gamma, v = p_k\ \vdash\ e_k}{\Gamma \vdash \textbf{case}\ v\ \textbf{of}\ p_1\ :\ e_1\ |\ \ldots\ |\ p_k\ :\ e_k}\ \text{(Case)}}{\Gamma \vdash ALL\ v.\textbf{case}\ v\ \textbf{of}\ p_1\ :\ e_1\ |\ \ldots\ |\ p_k\ :\ e_k}\ \text{(ALL-R)}$$

If $v \notin \phi$, then v remains in the resulting term, and the proof rules are further applied to the branches $e_1 \ldots e_k$ of the **case** term.

$\mathcal{A}[\![\textbf{case}\ v\ \textbf{of}\ p_1\ :\ e_1|\ \ldots\ |\ p_k\ :\ e_k]\!]\ \rho\ \phi$

$=\ \ \textbf{case}\ v\ \textbf{of}\ p_1\ :\ (\mathcal{A}[\![e_1]\!]\ \rho\ \phi)\ |\ \ldots\ |\ p_k\ :\ (\mathcal{A}[\![e_k]\!]\ \rho\ \phi) \neq True$

Case for Rule $\mathcal{A}6$:

By the inductive hypothesis:

$$\mathcal{A}[\![e_0]\!]\ \rho\ \{v_1 \ldots v_n\}\ =\ True\ \Rightarrow\ \vdash\ ALL\ v_1 \ldots v_n.e_0$$

If $\{v_1 \ldots v_n\} \subseteq \phi$, then the body of the function e_0 is further transformed.

$\mathcal{A}[\![\textbf{letrec}\ f\ =\ \lambda v_1 \ldots v_n.e_0\ \textbf{in}\ f\ v_1 \ldots v_n]\!]\ \rho\ \phi\ \ =\ \mathcal{A}[\![e_0]\!]\ \rho\ \phi$

The corresponding sequent calculus proof fragment is as follows:

$$\frac{\Gamma, f\ v_1 \ldots v_n\ \vdash\ e_0}{\Gamma \vdash \textbf{letrec}\ f\ =\ \lambda v_1 \ldots v_n.e_0\ \textbf{in}\ f\ v_1 \ldots v_n}\ (\text{Ind})$$

If $\{v_1 \ldots v_n\} \not\subseteq \phi$, then the definition of the function remains, and the body e_0 is further transformed:

$\mathcal{A}[\![\textbf{letrec}\ f\ =\ \lambda v_1 \ldots v_n.e_0\ \textbf{in}\ f\ v_1 \ldots v_n]\!]\ \rho\ \phi$

$=\ \textbf{letrec}\ f\ =\ \lambda v_1 \ldots v_k.(\mathcal{A}[\![e_0]\!]\ \rho\ \phi)\ \textbf{in}\ f\ v_1 \ldots v_k \neq True$

where $\{v_1 \ldots v_k\} = \{v_1 \ldots v_n\} \setminus \phi$.

Proof. (Lemma 4.5.3)

The proof of this is by recursion induction on the proof rules $\mathcal{E}$.

Base Cases

Case for Rule $\mathcal{E}1$:

If $v \in \phi$:

$\mathcal{E}[\![v]\!]\ \rho\ \phi\ =\ True$

In this case, the proof expression is equivalent to $EX\ v.v$. The corresponding sequent calculus proof fragment is as follows:

$$\frac{\dfrac{}{\Gamma\ \vdash\ True}\ (\textbf{True-R})}{\Gamma\ \vdash\ EX\ v.v}\ (\text{EX-R})$$

If $v \notin \phi$, then v is free.

$\mathcal{E}[\![v]\!]\ \rho\ \phi\ =\ v\ \neq\ True$

Case for Rule $\mathcal{E}2$:

$\mathcal{E}[\![True]\!]\ \rho\ \phi\ =\ True$

The corresponding sequent calculus proof fragment is as follows:

$$\frac{}{\Gamma\ \vdash\ True}\ (\text{True-R})$$

Case for Rule $\mathcal{E}3$:

$\mathcal{E}[\![False]\!]\ \rho\ \phi\ =\ \bot\ \neq\ True$

As the expression is *False*, there is no corresponding sequent calculus proof.

Case for Rule $\mathcal{E}4$:

$\mathcal{E}[\![\bot]\!]\ \rho\ \phi\ =\ \bot\ \neq\ True$

As the expression is undefined, there is no corresponding sequent calculus proof.

Case for Rule $\mathcal{E}7$:

If $\{v_1 \ldots v_n\} \subseteq \phi$, then the search space of the existential variables has been exhausted and there is no corresponding sequent calculus proof.

$\mathcal{E}[\![f\ e_1 \ldots e_n]\!]\ \rho\ \phi\ =\ \bot\ \neq\ True$

If $\{v_1 \ldots v_n\} \not\subseteq \phi$, then the function call remains.

$\mathcal{E}[\![f\ e_1 \ldots e_n]\!]\ \rho\ \phi\ =\ f\ v'_1 \ldots v'_k \neq True$

Inductive cases

Case for Rule $\mathcal{E}5$:

By the inductive hypothesis:

$$\forall i \in \{1 \ldots k\}.\mathcal{E}[\![e_i]\!]\ \rho\ \{v_1 \ldots v_n\}\ =\ True\ \Rightarrow\ \rho\ \vdash\ EX\ v_1 \ldots v_n.e_i$$

If $v \in \phi$, then the proof expression is equivalent to

$$EX\ v.\mathbf{case}\ v\ \mathbf{of}\ p_1\ :\ e_1\ |\ \ldots\ |\ p_k\ :\ e_k$$

$\mathcal{E}[\![\mathbf{case}\ v\ \mathbf{of}\ p_1\ :\ e_1\ |\ \ldots\ |\ p_k\ :\ e_k]\!]\ \rho\ \phi$

$=\ \mathcal{T}[\![(\mathcal{E}[\![e_1]\!]\ \rho\ (\phi \cup \{v_{11} \ldots v_{1k_1}\}))\ \vee \ldots \vee\ (\mathcal{E}[\![e_k]\!]\ \rho\ (\phi \cup \{v_{n1} \ldots v_{nk_n}\}))]\!]\ \{\}\ \{\}$

The corresponding sequent calculus proof fragment is as follows:

$$\frac{\Gamma \vdash e_1, \ldots, e_k}{\Gamma \vdash EX\ v.\mathbf{case}\ v\ \mathbf{of}\ p_1\ :\ e_1\ |\ \ldots\ |\ p_k\ :\ e_k}\ \text{(EX-R)}$$

If $v \notin \phi$, then v remains in the resulting term, and the proof rules are further applied to the branches $e_1 \dots e_k$ of the **case** term.

$\mathcal{E}[\![\textbf{case } v \textbf{ of } p_1 : e_1 \mid \dots \mid p_k : e_k]\!] \; \rho \; \phi$
$= \textbf{case } v \textbf{ of } p_1 : (\mathcal{E}[\![e_1]\!] \; \rho \; \phi) \mid \dots \mid p_k : (\mathcal{E}[\![e_k]\!] \; \rho \; \phi) \neq True$

Case for Rule $\mathcal{E}6$:

By the inductive hypothesis:

$$\mathcal{E}[\![e_0]\!] \; \rho \; \{v_1 \dots v_n\} = True \Rightarrow \; \vdash EX \; v_1 \dots v_n.e_0$$

If $\{v_1 \dots v_n\} \subseteq \phi$, then the body of the function e_0 is further transformed.

$\mathcal{E}[\![\textbf{letrec } f = \lambda v_1 \dots v_n.e_0 \textbf{ in } f \; v_1 \dots v_n]\!] \; \rho \; \phi \quad = \mathcal{E}[\![e_0]\!] \; \rho \; \phi$

The corresponding sequent calculus proof fragment is as follows:

$$\frac{\Gamma, f \; v_1 \dots v_n \vdash e_0}{\Gamma \vdash \textbf{letrec } f = \lambda v_1 \dots v_n.e_0 \textbf{ in } f \; v_1 \dots v_n} \text{(Ind)}$$

If $\{v_1 \dots v_n\} \nsubseteq \phi$, then the definition of the function remains, and the body e_0 is further transformed:

$\mathcal{E}[\![\textbf{letrec } f = \lambda v_1 \dots v_n.e_0 \textbf{ in } f \; v_1 \dots v_n]\!] \; \rho \; \phi$
$= \textbf{letrec } f = \lambda v_1 \dots v_k.(\mathcal{E}[\![e_0]\!] \; \rho \; \phi) \textbf{ in } f \; v_1 \dots v_k \neq True$
where $\{v_1 \dots v_k\} = \{v_1 \dots v_n\} \setminus \phi$.

4.6 Completeness

Neil D. Jones has shown that a function f is computable by a cons-free first-order functional program if and only if f is in PTIME (polynomial time) [57]. By analogy, our proof techniques can prove the inductive conjectures which can be defined with cons-free programs without using any intermediate lemmas. This cons-free program corresponds to program without any intermediate data structures. Therefore, our theorem prover should be able to prove any conjecture which is in PTIME, so long as it has been expressed in this cons-free form. We therefore argue that Poitín is complete for conjectures which belong to PTIME.

4.7 Conclusion

In this chapter, we have presented a novel approach to prove inductive conjectures which contain universal and existential quantification using the program transformation algorithm distillation [46]. We have extended the distillation rules to handle explicit quantification. A form of proof expressions has been defined to which these proof rules can be applied. We have formally presented the proof rules $\mathcal{A}$ for universal quantification and the proof rules $\mathcal{E}$ for existential quantification, and have shown how these proof rules can be used to prove inductive conjectures. These proof techniques do not require any intermediate lemmas. The existential theorem proving technique presented in this chapter gives a pure existence proof, which is an alternative to the usual constructive approach using higher order unification. The soundness of the proof techniques has been shown with respect to a logical proof system using the sequent calculus.

We have implemented the theorem proving technique presented in this chapter, and added it to the theorem prover Poitín. In Chapter 5, we present the program construction technique used in Poitín to construct correct programs from input specifications.

Chapter 5

Program Construction in Poitín

5.1 Introduction

In this chapter, we present a novel program construction method to construct correct programs from input program specifications. The programs are generated from the proofs of existential theorems in the theorem prover Poitín. The constructed program essentially computes the existential witness of the existential theorem.

In our program construction method, distillation is first applied to the input specification. Rules for program construction are then applied to the resulting distilled expression to construct a program. We present the program construction rules, and then give some examples to show how these rules can be used to construct correct programs. We also prove that the programs constructed using our technique are totally correct. Some of the work presented in this chapter can be found in [59].

5.2 Form of Input Specification

To facilitate program construction in Poitín, a first-order quantifier ANY is added to the higher order functional language defined in §2.2.1 as shown in Fig. 5.1.

e ::= ANY $v : \tau . e$ ANY-quantified expression

Figure 5.1: Form of input specification for program construction

The sub-expression e of the specification may contain free variables which are

implicitly universally quantified. The existential variable within the input specification is quantified with the ANY quantifier. The grammar of redexes is extended as follows to handle ANY-quantified expressions.

$$
\begin{array}{rcl}
red & ::= & f \\
& | & (\lambda v.e_0)\ e_1 \\
& | & \text{case } (v\ e_1 \ldots e_n) \text{ of } p_1\ :\ e'_1 \mid \ldots \mid p_k\ :\ e'_k \\
& | & \text{case } (c\ e_1 \ldots e_n) \text{ of } p_1\ :\ e'_1 \mid \ldots \mid p_k\ :\ e'_k \\
& | & \text{ALL } v_1 \ldots v_n.e \\
& | & \text{EX } v_1 \ldots v_n.e \\
& | & \text{ANY } v : \tau.e
\end{array}
$$

We define a set of rules $\mathcal{C}$ to deal with ANY-quantified expressions. The rules $\mathcal{C}$ for program construction will only be applied to the expressions which are in the form of proof expressions as shown in Fig. 4.3 (§4.3).

5.3 Construction of Program in Poitín

The construction of a program in Poitín involves: i) writing a specification of the form ANY $v : \tau.spec(x_1 \ldots x_n, v)$, which expresses the input/output relation for which the program is to be constructed ii) construction of a program from the specification using the rules $\mathcal{C}$. Within the specification, v is the output variable of type τ; *spec* is the relation between the input and output data expressed using predicates, functions and implication; and $x_1 \ldots x_n$ are the implicitly universally quantified input variables.

In the following sections, we present the distillation rule, the program construction steps and the program construction rules for the construction of a program from an input specification.

5.3.1 Distillation Rule for Program Construction

The transformation rules $\mathcal{T}$ for distillation are extended to be able to handle ANY-quantified expression as shown in Fig. 5.2. The application of this rule to an input specification results in the construction of a recursive functional program which computes the existential witness. Within rule $\mathcal{T}4$, the parameter ρ represents the set of previously encountered expressions, and ϕ is the set of function definitions used within the current expression. In transforming an expression of the form $c\langle$ANY v :

$\tau.e\rangle$, the sub-expression e is transformed first using distillation (the ANY-quantified variable v is free within e). An empty set $\{\}$ is passed to $\mathcal{T}$, which indicates that the initial set of previously encountered expressions is empty. The rules $\mathcal{C}$ for program construction are then applied to the proof expression obtained by pre-processing (§4.2) of the resulting distilled expression. The ANY-quantified variable v is passed as the variable under construction to the object level program construction rules $\mathcal{C}$. In addition, the set of implicitly universally quantified variables is passed as a parameter in the application of $\mathcal{C}$. Finally, the program obtained by applying the program construction rules $\mathcal{C}$ is transformed within the context (i.e., $c\langle e''\rangle$) using $\mathcal{T}$ to give the output program.

$$\mathcal{T}[\![c\langle \text{ANY } v : \tau.e\rangle]\!]\ \rho\ \phi \ = \mathcal{T}[\![c\langle e''\rangle]\!]\ \rho\ \phi \qquad (\mathcal{T}4)$$

where

$$e' = \mathcal{T}[\![e]\!]\ \{\}\ \phi$$

$$e'' = \mathcal{C}[\![e']\!]\ [\![v]\!]\ \{\}\ (fv(e) \setminus \{v\})$$

Figure 5.2: Distillation rule for program construction

5.3.2 Precondition and Postcondition Analysis

The input specification for program construction can be expressed in any of the following forms:

i)	ANY $v : \tau.e$	No precondition
ii)	ANY $v : \tau.pre \rightarrow post$	Precondition with implication

Specifications which satisfy form (i) do not contain any precondition within the conjecture. For example, the conjecture

$$\text{ANY } y : nat.(eqnum\ x\ Zero) \vee (eqnum\ x\ (Succ\ y))$$

satisfies this form. The program constructed from this specification can be used to compute an output for each value of the input variable x. Specifications which satisfy form (ii) contain a precondition and postcondition. The precondition specifies the properties of the input variables; this corresponds to the *computationally irrelevant* part of the specification as defined in [32]. The postcondition specifies the value of the output in relation to the input variables; this corresponds to the *computationally relevant* part of the specification as defined in [32]. We construct programs solely

from the computationally relevant part of the specification (i.e., the postcondition). The programs which we construct may return $\perp$, but this will only be for values of the input variables which do not satisfy the precondition. The precondition of the specification is therefore replaced by *True* in our approach so that it can be transformed away and programs are then constructed from the resulting postcondition. As an example, the specification ANY y : $nat.(even\ x) \rightarrow (eqnum\ (double\ y)\ x)$ has the precondition $(even\ x)$ and the postcondition $(eqnum\ (double\ y)\ x)$. In our approach, we construct a program from the postcondition $(eqnum\ (double\ y)\ x)$. This program may return $\perp$. However, if we also show that the existential conjecture ALL x.EX $y.(even\ x) \rightarrow (eqnum\ (double\ y)\ x)$ is *True*, then we know that the original specification is satisfiable, and that the constructed program will only return $\perp$ for values of the input variable x which do not satisfy the precondition $(even\ x)$.

5.3.3 Construction Process

Program construction from an input specification in Poitín is guided by the steps as described in Fig. 5.3. These program construction steps ensure the construction of programs which are correct with respect to the input specification.

5.3.4 Program Construction Rules $\mathcal{C}$

The program construction rules for an ANY-quantified specification are defined with a set of rules $\mathcal{C}$ by $\mathcal{C}[\![e]\!]\ [\![e']\!]\ \rho\ \phi$ as shown in Figs 5.4 and 5.5, where the expression e is the proof expression obtained from the postcondition of the specification. e' is the existential witness which may be a variable v or a constructor application $c\ e_1 \ldots e_n$. The environment ρ is the set of the previously encountered function calls, and ϕ is the set of implicitly universally quantified variables.

The rules $\mathcal{C}$ for program construction can be explained as follows. In rule ($\mathcal{C}1$), a variable v is encountered which must be a Boolean, and the existential witness is also a variable. If v is universally quantified (and therefore in ϕ), then the undefined value $\perp$ is returned as the value of v cannot always be *True*. Otherwise, v is implicitly existentially quantified and must be the existential witness, so the value *True* is returned as the only possible value of this witness. In rule ($\mathcal{C}2$), we encounter a variable v where the constructor application $c\ e_1 \ldots e_n$ is the existential witness. If v is universally quantified (and therefore in ϕ), then the value $\perp$ is returned as the value of v cannot always be *True*. Otherwise, v is implicitly existentially quantified, so the

1. Construct an existential conjecture from the program construction specification. Let ANY $v : \tau.e$ be the input program specification. Then, the resulting existential conjecture will be ALL $v_1 \ldots v_n$.EX $v.e$ where $v_1 \ldots v_n$ are the implicitly universally quantified variables in e.

2. Construct a proof of this existential conjecture using Poitín to verify that the input specification is satisfiable.

3. If the conjecture is proved, then follow step 4. Otherwise, return $\perp$ (i.e., the input specification is not satisfiable).

4. For input specifications satisfying form (i) (§5.3.2), construct a program from the input specification. For input specifications satisfying form (ii), follow step 5.

5. Construct an existential conjecture using the precondition within the input specification by existentially quantifying all of the free variables within the precondition.

6. If the proof of the existential conjecture is *True*, then construct a new program specification by replacing the precondition with *True* within the original specification. Construct a program from this new specification. Otherwise, return $\perp$.

Figure 5.3: Program construction steps

arguments $e_1 \ldots e_n$ are further constructed separately and the existential witness is given by the application of the constructor c to these constructed arguments.

In rule $(\mathcal{C}3)$, we encounter the value $True$ where the existential witness is a variable. The existential witness is constructed using a non-recursive constructor of the existential witness type. In rule $(\mathcal{C}4)$, we encounter the value $True$ where the constructor application $c\ e_1 \ldots e_n$ is the existential witness. The arguments $e_1 \ldots e_n$ are further constructed separately and the existential witness is given by the application of the constructor c to these constructed arguments. In rule $(\mathcal{C}5)$, we encounter the value $False$. In this case, there is no existential witness, so the undefined value $\perp$ is returned. In rule $(\mathcal{C}6)$, the undefined value $\perp$ is encountered. The existential witness for this expression is therefore also $\perp$.

In rule $(\mathcal{C}7)$, we encounter a **case** expression where the redex must be a variable

$$\mathcal{C}[\![v]\!]\ [\![v']\!]\ \rho\ \phi \;=\; \bot, \qquad \text{if } v \in \phi \qquad (\mathcal{C}1)$$
$$\;=\; True, \qquad \text{otherwise}$$

$$\mathcal{C}[\![v]\!]\ [\![c\ e_1 \ldots e_n]\!]\ \rho\ \phi \;=\; \bot, \qquad \text{if } v \in \phi$$
$$\;=\; c\ (\mathcal{C}[\![v]\!]\ [\![e_1]\!]\ \rho\ \phi) \ldots (\mathcal{C}[\![v]\!]\ [\![e_n]\!]\ \rho\ \phi), \qquad \text{otherwise} \qquad (\mathcal{C}2)$$

$$\mathcal{C}[\![True]\!]\ [\![v]\!]\ \rho\ \phi \;=\mathcal{C}[\![True]\!]\ [\![c_i\ v_{i1} \ldots v_{ik_i}]\!]\ \rho\ \phi \qquad (\mathcal{C}3)$$
where v is of type $\tau = c_1\ \tau_{11} \ldots \tau_{1k_1} \mid \ldots \mid c_m\ \tau_{m1} \ldots \tau_{mk_m}$
and $\exists i \in \{1 \ldots m\}.\tau \notin \{\tau_{i1} \ldots \tau_{ik_i}\}$

$$\mathcal{C}[\![True]\!]\ [\![c\ e_1 \ldots e_n]\!]\ \rho\ \phi \;=\; c\ (\mathcal{C}[\![True]\!]\ [\![e_1]\!]\ \rho\ \phi) \ldots (\mathcal{C}[\![True]\!]\ [\![e_n]\!]\ \rho\ \phi) \qquad (\mathcal{C}4)$$

$$\mathcal{C}[\![False]\!]\ [\![e]\!]\ \rho\ \phi \;=\bot \qquad (\mathcal{C}5)$$

$$\mathcal{C}[\![\bot]\!]\ [\![e]\!]\ \rho\ \phi \;=\bot \qquad (\mathcal{C}6)$$

$$\mathcal{C}[\![\textbf{case}\ v\ \textbf{of}\ p_1\ :\ e_1 | \ \ldots\ | p_n\ :\ e_n]\!]\ [\![e]\!]\ \rho\ \phi \qquad (\mathcal{C}7)$$
$$= \textbf{case}\ v\ \textbf{of}\ p_1\ :\ (\mathcal{C}[\![e_1]\!]\ [\![e]\!]\ \rho\ \phi_1) \mid \ldots \mid p_n\ :\ (\mathcal{C}[\![e_n]\!]\ [\![e]\!]\ \rho\ \phi_n), \quad \text{if } v \in \phi$$
$$= \mathcal{T}[\![(\mathcal{C}[\![e_1]\!]\ [\![e[p_1/v]]\!]\ \rho\ \phi)\ \sqcup \ldots \sqcup\ (\mathcal{C}[\![e_n]\!]\ [\![e[p_n/v]]\!]\ \rho\ \phi)]\!]\ \{\}\ \{\}, \text{otherwise}$$
where
$$\phi_i \;=\; \phi \cup fv(p_i)$$

$$\mathcal{C}[\![\textbf{letrec}\ f\ =\ \lambda v_1 \ldots v_n.e_0\ \textbf{in}\ f\ v_1 \ldots v_n]\!]\ [\![v]\!]\ \rho\ \phi \qquad (\mathcal{C}8)$$
$$= \textbf{letrec}\ f\ =\ \lambda v'_1 \ldots v'_k.e'_0\ \textbf{in}\ f\ v'_1 \ldots v'_k, \qquad \text{if } \exists x \in \{v_1 \ldots v_n\}.x \in \phi$$
$$= e'_0, \qquad \text{otherwise}$$
where
$$e'_0 = \mathcal{C}[\![e_0]\!]\ [\![v]\!]\ (\rho \cup \{f\ v_1 \ldots v_n\})\ \phi$$
$$\{v'_1 \ldots v'_k\} = \{v_1 \ldots v_n\} \cap \phi$$

$$\mathcal{C}[\![\textbf{letrec}\ f\ =\ \lambda v_1 \ldots v_n.e_0\ \textbf{in}\ f\ v_1 \ldots v_n]\!]\ [\![c\ e_1 \ldots e_k]\!]\ \rho\ \phi \qquad (\mathcal{C}9)$$
$$= c\ (\mathcal{C}[\![\textbf{letrec}\ f\ =\ \lambda v_1 \ldots v_n.e_0\ \textbf{in}\ f\ v_1 \ldots v_n]\!]\ [\![e_1]\!]\ \rho\ \phi) \ldots$$
$$(\mathcal{C}[\![\textbf{letrec}\ f\ =\ \lambda v_1 \ldots v_n.e_0\ \textbf{in}\ f\ v_1 \ldots v_n]\!]\ [\![e_k]\!]\ \rho\ \phi),$$
$$\text{if } \exists x \in \{v_1 \ldots v_n\}.x \in \phi$$
$$= \mathcal{C}[\![e_0]\!]\ [\![c\ e_1 \ldots e_k]\!]\ (\rho \cup \{f\ v_1 \ldots v_n\})\ \phi, \qquad \text{otherwise}$$

Figure 5.4: Program construction rules $\mathcal{C}$

$$\mathcal{C}[\![f\ e_1 \ldots e_n]\!]\ [\![v]\!]\ \rho\ \phi \;=\; f\ v'_1 \ldots v'_k[e_1/v_1 \ldots e_n/v_n], \quad \text{if } \exists x \in \{e_1 \ldots e_n\}.x \in \phi \qquad (\mathcal{C}10)$$
$$= \;\bot, \quad \text{otherwise}$$
$$\text{where } (f\ v_1 \ldots v_n) \in \rho.(f\ v_1 \ldots v_n) \leq (f\ e_1 \ldots e_n)$$
$$\{v'_1 \ldots v'_k\} = \{v_1 \ldots v_n\} \cap \phi$$

$$\mathcal{C}[\![f\ e_1 \ldots e_n]\!]\ [\![c\ e'_1 \ldots e'_k]\!]\ \rho\ \phi \qquad (\mathcal{C}11)$$
$$= c\ (\mathcal{C}[\![f\ e_1 \ldots e_n]\!]\ [\![e'_1]\!]\ \rho\ \phi) \ldots (\mathcal{C}[\![f\ e_1 \ldots e_n]\!]\ [\![e'_k]\!]\ \rho\ \phi), \quad \text{if } \exists x \in \{e_1 \ldots e_n\}.x \in \phi$$
$$= \;\bot, \quad \text{otherwise}$$

Figure 5.5: Program construction rules $\mathcal{C}$ (Continued)

as the expression is in distilled form. If v is universally quantified (and therefore in ϕ), then it remains within the expression. The program construction rules are then further applied to the branches of the **case** expression. Before transforming each branch, the corresponding pattern variables are added to ϕ as they are also implicitly universally quantified. If v is implicitly existentially quantified (and therefore not contained in ϕ), existential witnesses are constructed for each of the branches separately. These witnesses will be constructed using the corresponding patterns which give the value of the redex within the branch. The existential witness for the overall expression is then given by the *least upper bound* ($\sqcup$) of these existential witnesses for each branch.

In rule ($\mathcal{C}8$), we encounter a **letrec** expression where the existential witness is a variable. If at least one of the variables $v_1 \ldots v_n$ within the function application $f\ v_1 \ldots v_n$ is universally quantified (and therefore in ϕ), then the function definition is simplified to be defined over these universally quantified variables. The program construction rules are then further applied to the body of the function. If the function application does not contain any universally quantified variables, then all of the variables within the expression are implicitly existentially quantified. In this case, the function definition is removed, and the program construction rules are then further applied to the unfolded function call. The function application $f\ v_1 \ldots v_n$ is added to the environment ρ before applying the program construction rules to the body of the function. In rule ($\mathcal{C}9$), we encounter a **letrec** expression where the constructor application $c\ e_1 \ldots e_k$ is the existential witness. If at least one of the

variables $v_1 \ldots v_n$ within the function application $f\ v_1 \ldots v_n$ is universally quantified (and therefore in ϕ), then each of the arguments $e_1 \ldots e_k$ is further constructed separately and the existential witness is given by the application of the constructor c to these constructed arguments. If the function application $f\ v_1 \ldots v_n$ does not contain any universally quantified variables, then the program construction rules are applied to the function body to give the existential witness. The function application $f\ v_1 \ldots v_n$ is added to the environment ρ before applying the program construction rules to the function body.

In rule ($\mathcal{C}10$), we encounter a recursive function call $f\ e_1 \ldots e_n$ where the existential witness is a variable. If there is a function application $f\ v_1 \ldots v_n$ in ρ such that the recursive call $f\ e_1 \ldots e_n$ is an instance of $f\ v_1 \ldots v_n$, and at least one of the arguments $e_1 \ldots e_n$ in the recursive call is a universally quantified variable (and therefore in ϕ), then the function application is simplified to be defined over the arguments of the recursive call corresponding to the universally quantified variables within the initial function application $f\ v_1 \ldots v_n$. If, on the other hand, the function application does not contain any universal variables, then all of the variables are implicitly existentially quantified, so the undefined value $\perp$ is returned as the search space of these existential variables has been exhausted. In rule ($\mathcal{C}11$), we encounter a recursive function call $f\ e_1 \ldots e_n$ where the constructor application $c\ e'_1 \ldots e'_k$ is the existential witness. If at least one of the arguments $e_1 \ldots e_n$ in the recursive call is a universally quantified variable (and therefore in ϕ), then each of the arguments $e'_1 \ldots e'_k$ is constructed separately and the existential witness is given by the application of the constructor c to these constructed arguments. If, on the other hand, the recursive call does not contain any universal variables, then the undefined value $\perp$ is returned.

5.4 Examples

In this section, we give two examples of program construction to demonstrate how the rules C can be used in the construction of programs.

Example 12

Consider the input specification (12.1), which requires the construction of a natural number y such that for every natural number x, either the double of y is equal to x, or the successor of the double of y is equal to x. For even values of x, the constructed

program should compute the value of y as half of x. For odd values of x, the value of y should be computed as half of the predecessor of x.

$$\text{ANY } y : nat.or\ (eqnum\ (double\ y)\ x)\ (eqnum\ (Succ\ (double\ y))\ x) \quad (12.1)$$

The program construction from specification (12.1) is guided by the construction steps as shown in Fig. 5.3. Step 1 generates a theorem proving conjecture stating the satisfiability of specification (12.1), which is given by expression (12.2).

$$\text{ALL } x.\text{EX } y.or\ (eqnum\ (double\ y)\ x)\ (eqnum\ (Succ\ (double\ y))\ x) \quad (12.2)$$

The function *eqnum* is defined in §2.5.3 and *double* is defined in Appendix A.1. These definitions are used with the following definitions of the functions *or* and *lub* ($\sqcup$- least upper bound- in this case for natural numbers).

$$\begin{array}{lcl} or & = & \lambda x.\lambda y.\textbf{case}\ x\ \textbf{of} \\ & & \quad True : True \\ & & \quad \mid False : y \\ lub & = & \lambda x.\lambda y.\textbf{case}\ x\ \textbf{of} \\ & & \quad Zero \quad : Zero \\ & & \quad \mid Succ\ x' \ : Succ\ x' \\ & & \quad \mid \bot \ : y \end{array}$$

Poitín proves conjecture (12.2)- the details of this proof are not given here. This ensures that specification (12.1) is satisfiable. Rule ($\mathcal{T}4$) (Fig. 5.2) therefore applies the rules $\mathcal{C}$ to the proof expression obtained by pre-processing the distilled expression resulting from the transformation of expression (12.1) as shown below.

$$\begin{array}{l} \mathcal{C}[\![\ \textbf{letrec}\ f0 = \lambda y.\lambda x.\textbf{case}\ y\ \textbf{of} \qquad \text{(by } \mathcal{T}4) \\ \qquad Zero \quad : \textbf{case}\ x\ \textbf{of} \\ \qquad\qquad Zero \quad : True \\ \qquad\qquad \mid Succ\ x' : \textbf{case}\ x'\ \textbf{of} \\ \qquad\qquad\qquad Zero \quad : True \\ \qquad\qquad\qquad \mid Succ\ x'' : False \\ \qquad \mid Succ\ y' : \textbf{case}\ x\ \textbf{of} \\ \qquad\qquad Zero \quad : False \\ \qquad\qquad \mid Succ\ x' : \textbf{case}\ x'\ \textbf{of} \\ \qquad\qquad\qquad Zero \quad : False \\ \qquad\qquad\qquad \mid Succ\ x'' : f0\ y'\ x'' \\ \textbf{in}\ f0\ y\ x]\!]\ [\![y]\!]\ \{\}\ \{x\} \end{array}$$

During the application of the rules $\mathcal{C}$ in the above term, rule $\mathcal{C}8$ simplifies the **letrec** expression. The function application $f0\ y\ x$ is simplified to be defined over the universally quantified variable x as $f0\ x$ where the implicitly existentially quantified variable y is removed. The application of the rules $\mathcal{C}$ to the unfolded function call proceeds as shown below.

$$
\begin{array}{l}
\textbf{letrec} \hspace{20em} (\text{by } \mathcal{C}8)\\
f0 = \lambda x.\mathcal{C}[\![\textbf{case } y \textbf{ of}\\
\qquad\qquad Zero \quad : \textbf{case } x \textbf{ of}\\
\qquad\qquad\qquad\qquad Zero \quad : True\\
\qquad\qquad\qquad\quad \mid Succ\ x' : \textbf{case } x' \textbf{ of}\\
\qquad\qquad\qquad\qquad\qquad\qquad Zero \quad : True\\
\qquad\qquad\qquad\qquad\qquad\quad \mid Succ\ x'' : False\\
\qquad\quad \mid Succ\ y' : \textbf{case } x \textbf{ of}\\
\qquad\qquad\qquad\qquad Zero \quad : False\\
\qquad\qquad\qquad\quad \mid Succ\ x' : \textbf{case } x' \textbf{ of}\\
\qquad\qquad\qquad\qquad\qquad\qquad Zero \quad : False\\
\qquad\qquad\qquad\qquad\qquad\quad \mid Succ\ x'' : f0\ y'\ x'']\!]\ [\![y]\!]\ \{f0\ y\ x\}\ \{x\}\\
\textbf{in } f0\ x
\end{array}
$$

$$
\begin{array}{l}
= \textbf{letrec} \hspace{20em} (\text{by } \mathcal{C}7)\\
\quad f0 = \lambda x.\mathcal{T}[\![(\mathcal{C}[\![\textbf{case } x \textbf{ of}\\
\qquad\qquad\qquad Zero \quad : True\\
\qquad\qquad\quad \mid Succ\ x' : \textbf{case } x' \textbf{ of}\\
\qquad\qquad\qquad\qquad\qquad Zero \quad : True\\
\qquad\qquad\qquad\qquad \mid Succ\ x'' : False]\!]\ [\![Zero]\!]\ \{f0\ y\ x\}\ \{x\})\\
\qquad\qquad \sqcup (\mathcal{C}[\![\textbf{case } x \textbf{ of}\\
\qquad\qquad\qquad Zero \quad : False\\
\qquad\qquad\quad \mid Succ\ x' : \textbf{case } x' \textbf{ of}\\
\qquad\qquad\qquad\qquad\qquad Zero \quad : False\\
\qquad\qquad\qquad\qquad \mid Succ\ x'' : f0\ y'\ x'']\!]\ [\![Succ\ y']\!]\ \{f0\ y\ x\}\ \{x\})]\!]\ \{\}\ \{\}\\
\quad \textbf{in } f0\ x
\end{array}
$$

$$
\begin{array}{ll}
= \textbf{letrec} & \text{(by } \mathcal{C}7) \\
\quad f0 = \lambda x.\mathcal{T}[\![(\textbf{case } x \textbf{ of} & \\
\qquad\qquad Zero \quad : \mathcal{C}[\![True]\!]\ [\![Zero]\!]\ \{f0\ y\ x\}\ \{x\} & \\
\qquad\qquad \mid Succ\ x' : \mathcal{C}[\![\textbf{case } x' \textbf{ of} & \\
\qquad\qquad\qquad Zero \quad : True & \\
\qquad\qquad\qquad \mid Succ\ x'' : False]\!]\ [\![Zero]\!]\ \{f0\ y\ x\}\ \{x, x'\}) & \\
\qquad \sqcup (\mathcal{C}[\![\textbf{case } x \textbf{ of} & \\
\qquad\qquad Zero \quad : False & \\
\qquad\qquad \mid Succ\ x' : \textbf{case } x' \textbf{ of} & \\
\qquad\qquad\qquad Zero \quad : False & \\
\qquad\qquad\qquad \mid Succ\ x'' : f0\ y'\ x'']\!]\ [\![Succ\ y']\!]\ \{f0\ y\ x\}\ \{x\})]\!]\ \{\}\ \{\} & \\
\quad \textbf{in } f0\ x &
\end{array}
$$

$$
\begin{array}{ll}
= \textbf{letrec} & \text{(by } \mathcal{C}4, \mathcal{C}7) \\
\quad f0 = \lambda x.\mathcal{T}[\![(\textbf{case } x \textbf{ of} & \\
\qquad\qquad Zero \quad : Zero & \\
\qquad\qquad \mid Succ\ x' : \textbf{case } x' \textbf{ of} & \\
\qquad\qquad\qquad Zero \quad : \mathcal{C}[\![True]\!]\ [\![Zero]\!]\ \{f0\ y\ x\}\ \{x, x'\} & \\
\qquad\qquad\qquad \mid Succ\ x'' : \mathcal{C}[\![False]\!]\ [\![Zero]\!]\ \{f0\ y\ x\}\ \{x, x', x''\}) & \\
\qquad \sqcup (\mathcal{C}[\![\textbf{case } x \textbf{ of} & \\
\qquad\qquad Zero \quad : False & \\
\qquad\qquad \mid Succ\ x' : \textbf{case } x' \textbf{ of} & \\
\qquad\qquad\qquad Zero \quad : False & \\
\qquad\qquad\qquad \mid Succ\ x'' : f0\ y'\ x'']\!]\ [\![Succ\ y']\!]\ \{f0\ y\ x\}\ \{x\})]\!]\ \{\}\ \{\} & \\
\quad \textbf{in } f0\ x &
\end{array}
$$

$$
\begin{aligned}
= {} & \textbf{letrec} & \text{(by } \mathcal{C}4,\mathcal{C}5,\mathcal{C}7,\mathcal{C}5,\mathcal{C}7,\mathcal{C}5) \\
& \quad f0 = \lambda x.\mathcal{T}[\![(\textbf{case } x \textbf{ of} \\
& \qquad\qquad Zero \quad : Zero \\
& \qquad\qquad \mid Succ\ x' : \textbf{case } x' \textbf{ of} \\
& \qquad\qquad\qquad Zero \quad : Zero \\
& \qquad\qquad\qquad \mid Succ\ x'' : \bot\ ) \\
& \qquad \sqcup\ (\textbf{case } x \textbf{ of} \\
& \qquad\qquad Zero \quad : \bot \\
& \qquad\qquad \mid Succ\ x' : \ \textbf{case } x' \textbf{ of} \\
& \qquad\qquad\qquad Zero \quad : \ \bot \\
& \qquad\qquad\qquad \mid Succ\ x'' \quad : \\
& \qquad\qquad\qquad\quad \mathcal{C}[\![f0\ y'\ x'']\!]\ [\![Succ\ y']\!]\ \{f0\ y\ x\}\ \{x, x', x''\})]\!]\ \{\}\ \{\} \\
& \quad \textbf{in } f0\ x
\end{aligned}
$$

$$
\begin{aligned}
= {} & \textbf{letrec} & \text{(by } \mathcal{C}11) \\
& \quad f0 = \lambda x.\mathcal{T}[\![(\textbf{case } x \textbf{ of} \\
& \qquad\qquad Zero \quad : Zero \\
& \qquad\qquad \mid Succ\ x' : \textbf{case } x' \textbf{ of} \\
& \qquad\qquad\qquad Zero \quad : Zero \\
& \qquad\qquad\qquad \mid Succ\ x'' : \bot\ ) \\
& \qquad \sqcup\ (\textbf{case } x \textbf{ of} \\
& \qquad\qquad Zero \quad : \bot \\
& \qquad\qquad \mid Succ\ x' : \ \textbf{case } x' \textbf{ of} \\
& \qquad\qquad\qquad Zero \quad : \ \bot \\
& \qquad\qquad\qquad \mid Succ\ x'' \quad : \\
& \qquad\qquad\qquad\quad Succ\ (\mathcal{C}[\![f0\ y'\ x'']\!]\ [\![y']\!]\ \{f0\ y\ x\}\ \{x, x', x''\}))]\!]\ \{\}\ \{\} \\
& \quad \textbf{in } f0\ x
\end{aligned}
$$

$$
\begin{array}{ll}
= \textbf{letrec} & \text{(by } \mathcal{C}10\text{)} \\
\quad f0 = \lambda x.\mathcal{T}[\![(\textbf{case } x \textbf{ of} & \\
\qquad\qquad Zero \;\; : Zero & \\
\qquad\quad \mid Succ\ x' : \textbf{case } x' \textbf{ of} & \\
\qquad\qquad\qquad Zero \;\; : Zero & \\
\qquad\qquad\quad \mid Succ\ x'' : \bot\ ) & \\
\qquad \sqcup\ (\textbf{case } x \textbf{ of} & \\
\qquad\qquad Zero \;\; : \bot & \\
\qquad\quad \mid Succ\ x' : \textbf{case } x' \textbf{ of} & \\
\qquad\qquad\qquad Zero \;\; : \bot & \\
\qquad\qquad\quad \mid Succ\ x'' : Succ\ (f0\ x''))]\!]\ \{\}\ \{\} & \\
\quad \textbf{in } f0\ x &
\end{array}
$$

$$
\begin{array}{ll}
= \textbf{letrec } f0 = \lambda x.\textbf{case } x \textbf{ of} & \text{(by } \mathcal{T}1,\mathcal{P}\text{)} \\
\qquad\qquad Zero \;\; : Zero & \\
\qquad\quad \mid Succ\ x' : \textbf{case } x' \textbf{ of} & \\
\qquad\qquad\qquad Zero \;\; : Zero & \\
\qquad\qquad\quad \mid Succ\ x'' : Succ\ (f0\ x'') & \\
\quad \textbf{in } f0\ x &
\end{array}
$$

This program is further transformed using distillation, which results in the same output program. The constructed program computes an output for each value of the input variable x. The constructed program is totally correct, and it satisfies the input specification as required.

Example 13

Consider the following program specification (13.1), which requires the construction of a natural number z such that for every value of the natural numbers x and y, if $x < y$, then the sum of x and z is equal to y.

$$\text{ANY } z : nat.implies\ (less\ x\ y)\ (eqnum\ (plus\ x\ z)\ y) \tag{13.1}$$

In step 1 of the construction process (Fig. 5.3), the following existential conjecture is generated from specification (13.1).

$$\text{ALL } x.\text{ALL } y.\text{EX } z.implies\ (less\ x\ y)\ (eqnum\ (plus\ x\ z)\ y) \tag{13.2}$$

The functions *eqnum* and *plus* used within the specification have the same definitions as given in Chapter 2. The functions *implies* and *less* are defined as follows.

$$
\begin{array}{l}
implies \;=\; \lambda x.\lambda y.\,\mathbf{case}\ x\ \mathbf{of} \\
\qquad\qquad\qquad\qquad True : y \\
\qquad\qquad\qquad\quad\;\; \mid False : True \\
less = \lambda x.\lambda y.\,\mathbf{case}\ x\ \mathbf{of} \\
\qquad\qquad\quad Zero \quad\;\, : \mathbf{case}\ y\ \mathbf{of} \\
\qquad\qquad\qquad\qquad\qquad Zero \quad\; : False \\
\qquad\qquad\qquad\qquad\quad\;\; \mid Succ\ y' : True \\
\qquad\qquad\;\; \mid Succ\ x' : \mathbf{case}\ y\ \mathbf{of} \\
\qquad\qquad\qquad\qquad\qquad Zero \quad\; : False \\
\qquad\qquad\qquad\qquad\quad\;\; \mid Succ\ y' : less\ x'\ y'
\end{array}
$$

Poitín proves conjecture (13.2)- the details of this proof are not given here. Specification (13.1) is therefore a satisfiable program specification, which is of form (ii) as defined in §5.3.2. The existential conjecture EX x.EX $y.less\ x\ y$ is constructed from the precondition $less\ x\ y$ of specification (13.1) according to step 5 of the construction process (Fig. 5.3). Poitín proves this conjecture. A new specification is therefore constructed by replacing the precondition $less\ x\ y$ with *True* in step 6 of the construction process, which results in the following expression (13.3).

$$implies\ (True)\ (eqnum\ (plus\ x\ z)\ y) \tag{13.3}$$

Rule ($\mathcal{T}4$) applies the program construction rules $\mathcal{C}$ to the proof expression obtained from the distilled expression resulting from the transformation of expression (13.3) as shown below.

```
C[[letrec
  f0 = λx.λz.λy. case x of
                 Zero    : case z of
                             Zero    : case y of
                                         Zero    : True
                                       | Succ y' : False
                           | Succ z' : case y of
                                         Zero    : False
                                       | Succ y' : letrec
                                                   f1 = λz'.λy'. case z' of
                                                                  Zero     : case y' of
                                                                               Zero     : True
                                                                             | Succ y'' : False
                                                                | Succ z'' : case y' of
                                                                               Zero     : False
                                                                             | Succ y'' : f1 z'' y''
                                                   in f1 z' y'
               | Succ x' : case y of
                             Zero    : False
                           | Succ y' : f0 x' z y'
  in f0 x z y]] [[z]] {} {x,y}
```

(by $\mathcal{T}4$)

The details of the construction steps is out of scope because of larger expression size. Finally, we obtain the following program by applying the program construction rules $\mathcal{C}$.

```
letrec
f0 = λx.λy. case x of
              Zero    : case y of
                          Zero    : Zero
                        | Succ y' : Succ ( letrec
                                           f1 = λy'. case y' of
                                                       Zero     : Zero
                                                     | Succ y'' : Succ (f1 y'')
                                           in f1 y')
            | Succ x' : case y of
                          Zero    : ⊥
                        | Succ y' : f0 x' y'
in f0 x y
```

The constructed program computes the existential witness z as a function of the universally quantified variables x and y. For $x \leq y$, it returns $y - x$; and for

$x > y$, it returns $\perp$ (*Bottom*). The constructed program therefore satisfies the input specification.

5.5 Proof of Correctness

The program construction method of Poitín constructs executable functional programs from input specifications. The construction method extracts a *program* from the *proof* of the input specification.

In order to prove that for every program specification which is found to be satisfiable, the programs constructed by our program construction rules are correct with respect to original specifications, we need to show the following:

1. $\mathcal{C}[\![post]\!]\ [\![e]\!]\ \rho\ \{v_1 \ldots v_n\} \ = \ e[e_1/v'_1, \ldots, e_k/v'_k]$
   $\Rightarrow\ \mathcal{T}[\![ALL\ v_1 \ldots v_n.post[e_1/v'_1, \ldots, e_k/v'_k]]\!]\ \{\}\ \{\} \ = \ True$

2. (a) for specifications of the form $ANY\ v : \tau.pre \ \Rightarrow\ post$:
   $\mathcal{C}[\![post]\!]\ [\![e]\!]\ \rho\ \{v_1 \ldots v_n\} \ = \ \perp$
   $\wedge\ \mathcal{T}[\![ALL\ v_1 \ldots v_n.EX\ v.pre\ \Rightarrow\ post]\!]\ \{\}\ \{\} \ = \ True$
   $\Rightarrow\ \mathcal{T}[\![ALL\ v_1 \ldots v_n.pre]\!]\ \{\}\ \{\} \ = \ \perp$

   (b) for specifications of the form $ANY\ v : \tau.post$:
   $\mathcal{C}[\![post]\!]\ [\![e]\!]\ \rho\ \{v_1 \ldots v_n\} \ = \ \perp$
   $\Rightarrow\ \mathcal{T}[\![ALL\ v_1 \ldots v_n.EX\ v.post]\!]\ \{\}\ \{\} \ = \ \perp$

In order to prove the correctness of the constructed program using the program construction rules $\mathcal{C}$ from specifications of the form $ANY\ v : \tau.post$, we need to show (1) and 2(b), and from specifications of the form $ANY\ v : \tau.pre\ \Rightarrow\ post$, we need to show (1) and 2(a).

Proof.
The proof of this is by recursion induction on the proof rules $\mathcal{C}$.

Base Cases

Case for Rule $\mathcal{C}1$:
If $v \in \phi$:
$\mathcal{C}[\![v]\!]\ [\![v']\!]\ \rho\ \{v_1 \ldots v_n\} = \perp$

2. (a) $\mathcal{T}[\![ALL\ v_1 \dots v_n.EX\ v'.v_i]\!]\ \{\}\ \{\} \ = \ \bot$ (by ($\mathcal{A}1$))

$\mathcal{T}[\![ALL\ v_1 \dots v_n.EX\ v'.pre \ \Rightarrow \ v_i]\!]\ \{\}\ \{\} \ = \ True$ (by assumption)

$\Rightarrow \ \mathcal{T}[\![ALL\ v_1 \dots v_n.EX\ v'.pre]\!]\ \{\}\ \{\} \ = \ \bot$

(b) $\mathcal{T}[\![ALL\ v_1 \dots v_n.EX\ v'.v_i]\!]\ \{\}\ \{\} \ = \ \bot$ (by ($\mathcal{A}1$))

If $v \notin \phi$:

1. $\mathcal{C}[\![v]\!]\ [\![v]\!]\ \rho\ \{v_1 \dots v_n\} = v[True/v]$

$\mathcal{T}[\![ALL\ v_1 \dots v_n.v[True/v]]\!]\ \{\}\ \{\} \ = \ True$ (by ($\mathcal{A}2$))

Case for Rule $\mathcal{C}2$:

If $v \in \phi$:

$\mathcal{C}[\![v]\!]\ [\![c\ e_1 \dots e_n]\!]\ \rho\ \{v_1 \dots v_n\} = \bot$

2. (a) $\mathcal{T}[\![ALL\ v_1 \dots v_n.EX\ v'.v_i]\!]\ \{\}\ \{\} \ = \ \bot$ (by ($\mathcal{A}1$))

$\mathcal{T}[\![ALL\ v_1 \dots v_n.EX\ v'.pre \ \Rightarrow \ v_i]\!]\ \{\}\ \{\} \ = \ True$ (by assumption)

$\Rightarrow \ \mathcal{T}[\![ALL\ v_1 \dots v_n.EX\ v'.pre]\!]\ \{\}\ \{\} \ = \ \bot$

(b) $\mathcal{T}[\![ALL\ v_1 \dots v_n.EX\ v'.v_i]\!]\ \{\}\ \{\} \ = \ \bot$ (by ($\mathcal{A}1$))

If $v \notin \phi$:

1. $\mathcal{C}[\![v'_i]\!]\ [\![c\ e_1 \dots e_n]\!]\ \rho\ \{v_1 \dots v_n\} = c\ e_1 \dots e_n[True/v'_1 \dots True/v'_k]$ (by ($\mathcal{C}1$))

$\mathcal{T}[\![ALL\ v_1 \dots v_n.v'_i[True/v'_1 \dots True/v'_k]]\!]\ \{\}\ \{\} \ = \ True$ (by ($\mathcal{A}2$))

Case for Rule $\mathcal{C}3$:

1. $\mathcal{C}[\![True]\!]\ [\![v]\!]\ \rho\ \{v_1 \dots v_n\} = v[(c_i\ v_{i1} \dots v_{ik_i})/v]$

$\mathcal{T}[\![ALL\ v_1 \dots v_n.True[(c_i\ v_{i1} \dots v_{ik_i})/v]]\!]\ \{\}\ \{\} \ = \ True$ (by ($\mathcal{A}2$))

Case for Rule $\mathcal{C}4$:

1. $\mathcal{C}[\![True]\!]\ [\![c\ e_1 \dots e_n]\!]\ \rho\ \{v_1 \dots v_n\} = (c\ e_1 \dots e_n)[e'_1/v'_1 \dots e'_k/v'_k]$

$\mathcal{T}[\![ALL\ v_1 \dots v_n.True[e'_1/v'_1 \dots e'_k/v'_k]]\!]\ \{\}\ \{\} \ = \ True$ (by ($\mathcal{A}2$))

Case for Rule $\mathcal{C}5$:

$\mathcal{C}[\![False]\!]\ [\![e]\!]\ \rho\ \{v_1 \dots v_n\} = \bot$

2. (a) $\mathcal{T}[\![ALL\ v_1 \dots v_n.EX\ v.False]\!]\ \{\}\ \{\} \ = \ \bot$ (by ($\mathcal{E}3$))

$\mathcal{T}[\![ALL\ v_1 \dots v_n.EX\ v.pre \ \Rightarrow \ False]\!]\ \{\}\ \{\} \ = \ True$ (by assumption)

$\Rightarrow \ \mathcal{T}[\![ALL\ v_1 \dots v_n.EX\ v.pre]\!]\ \{\}\ \{\} \ = \ \bot$

(b) $\mathcal{T}[\![ALL\ v_1 \dots v_n.EX\ v.False]\!]\ \{\}\ \{\} \ = \ \bot$ (by ($\mathcal{E}3$))

Case for Rule $\mathcal{C}6$:

$\mathcal{C}[\![\bot]\!]\ [\![e]\!]\ \rho\ \{v_1 \ldots v_n\} = \bot$

2. (a) $\mathcal{T}[\![ALL\ v_1 \ldots v_n.EX\ v.\ \bot]\!]\ \{\}\ \{\}\ =\ \bot$ (by ($\mathcal{E}4$))

$\mathcal{T}[\![ALL\ v_1 \ldots v_n.EX\ v.pre\ \Rightarrow\ \bot]\!]\ \{\}\ \{\}\ =\ True$ (by assumption)

$\Rightarrow\ \mathcal{T}[\![ALL\ v_1 \ldots v_n.EX\ v.pre]\!]\ \{\}\ \{\}\ =\ \bot$

(b) $\mathcal{T}[\![ALL\ v_1 \ldots v_n.EX\ v.\ \bot]\!]\ \{\}\ \{\}\ =\ \bot$ (by ($\mathcal{E}4$))

Case for Rule $\mathcal{C}10$:

If $\exists x \in \{e_1 \ldots e_m\}.x \in \{v_1 \ldots v_n\}$:

1. $\mathcal{C}[\![f\ e_1 \ldots e_m]\!]\ [\![v]\!]\ \rho\ \{v_1 \ldots v_n\} = v[f\ e'_1 \ldots e'_k/v]$

$\mathcal{T}[\![ALL\ v_1 \ldots v_n.(f\ v_1 \ldots v_m[f\ e'_1 \ldots e'_k/v])]\!]\ \{\}\ \{\}\ =\ True$ (by ($\mathcal{A}7$))

If $\nexists x \in \{e_1 \ldots e_m\}.x \in \{v_1 \ldots v_n\}$:

$\mathcal{C}[\![f\ e_1 \ldots e_m]\!]\ [\![v]\!]\ \rho\ \{v_1 \ldots v_n\} = \bot$

2. (a) $\mathcal{T}[\![EX\ v'_1 \ldots v'_k.f\ e_1 \ldots e_m]\!]\ \{\}\ \{\}\ =\ \bot$ (by ($\mathcal{E}7$))

$\mathcal{T}[\![EX\ v'_1 \ldots v'_k.pre \Rightarrow f\ v_1 \ldots v_m]\!]\ \{\}\ \{\}\ =\ True$ (by assumption)

$\Rightarrow\ \mathcal{T}[\![EX\ v'_1 \ldots v'_k.pre]\!]\ \{\}\ \{\}\ =\ \bot$

(b) $\mathcal{T}[\![EX\ v'_1 \ldots v'_k.f\ e_1 \ldots e_m]\!]\ \{\}\ \{\}\ =\ \bot$ (by ($\mathcal{E}7$))

Case for Rule $\mathcal{C}11$:

If $\exists x \in \{e_1 \ldots e_m\}.x \in \{v_1 \ldots v_n\}$:

1. $\mathcal{C}[\![f\ e_1 \ldots e_m]\!]\ [\![c\ e'_1 \ldots e'_l]\!]\ \rho\ \{v_1 \ldots v_n\} = (c\ e'_1 \ldots e'_l)[e''_1/v'_1 \ldots e''_k/v'_k]$

$\mathcal{T}[\![ALL\ v_1 \ldots v_n.(f\ e_1 \ldots e_m[e''_1/v'_1 \ldots e''_k/v'_k])]\!]\ \{\}\ \{\}\ =\ True$ (by ($\mathcal{A}7$))

If $\nexists x \in \{e_1 \ldots e_m\}.x \in \{v_1 \ldots v_n\}$:

$\mathcal{C}[\![f\ e_1 \ldots e_m]\!]\ [\![c\ e'_1 \ldots e'_l]\!]\ \rho\ \{v_1 \ldots v_n\} = \bot$

2. (a) $\mathcal{T}[\![EX\ v'_1 \ldots v'_k.f\ e_1 \ldots e_m]\!]\ \{\}\ \{\}\ =\ \bot$ (by ($\mathcal{E}7$))

$\mathcal{T}[\![EX\ v'_1 \ldots v'_k.pre \Rightarrow f\ e_1 \ldots e_m]\!]\ \{\}\ \{\}\ =\ True$ (by assumption)

$\Rightarrow\ \mathcal{T}[\![EX\ v'_1 \ldots v'_k.pre]\!]\ \{\}\ \{\}\ =\ \bot$

(b) $\mathcal{T}[\![EX\ v'_1 \ldots v'_k.f\ e_1 \ldots e_m]\!]\ \{\}\ \{\}\ =\ \bot$ (by ($\mathcal{E}7$))

Inductive cases

Case for Rule $\mathcal{C}7$:

If $v \in \phi$:

$$\mathcal{C}[\![\textbf{case } v \textbf{ of } p_1 : e_1 \mid \ldots \mid p_n : e_n]\!]\ [\![e]\!]\ \rho\ \phi$$
$$= \textbf{case } v \textbf{ of } p_1 : (\mathcal{C}[\![e_1]\!]\ [\![e]\!]\ \rho\ \phi_1) \mid \ldots \mid p_n : (\mathcal{C}[\![e_n]\!]\ [\![e]\!]\ \rho\ \phi_n)$$

By the inductive hypothesis, $\forall i \in \{1 \ldots n\}$:

1. $\mathcal{C}[\![e_i]\!]\ [\![e]\!]\ \rho\ \{v_1 \ldots v_n\} = e[e'_1/v'_1 \ldots e'_k/v'_k]$
   $\Rightarrow \mathcal{T}[\![ALL\ v_1 \ldots v_n.e_i[e'_1/v'_1 \ldots e'_k/v'_k]]\!]\ \{\}\ \{\} = True$

2. (a) $\mathcal{C}[\![e_i]\!]\ [\![e]\!]\ \rho\ \{v_1 \ldots v_n\} = \bot$
   $\wedge\ \mathcal{T}[\![ALL\ v_1 \ldots v_n.EX\ v.pre \Rightarrow e_i]\!]\ \{\}\ \{\} = True$
   $\Rightarrow \mathcal{T}[\![ALL\ v_1 \ldots v_n.pre]\!]\ \{\}\ \{\} = \bot$
   (b) $\mathcal{T}[\![ALL\ v_1 \ldots v_n.EX\ v.e_i]\!]\ \{\}\ \{\} = \bot$

If $v \notin \phi$:

$$\mathcal{C}[\![\textbf{case } v \textbf{ of } p_1 : e_1 \mid \ldots \mid p_n : e_n]\!]\ [\![e]\!]\ \rho\ \phi$$
$$= \mathcal{T}[\![(\mathcal{C}[\![e_1]\!]\ [\![e[p_1/v]]\!]\ \rho\ \phi) \sqcup \ldots \sqcup (\mathcal{C}[\![e_n]\!]\ [\![e[p_n/v]]\!]\ \rho\ \phi)]\!]\ \{\}\ \{\}$$

If $\exists i \in \{1 \ldots n\}.\mathcal{C}[\![e_i]\!]\ [\![e]\!]\ \rho\ \phi \neq \bot$, then by the inductive hypothesis:

1. $\mathcal{C}[\![e_i]\!]\ [\![e]\!]\ \rho\ \{v_1 \ldots v_n\} = e[e'_1/v'_1 \ldots e'_k/v'_k]$
   $\Rightarrow \mathcal{T}[\![ALL\ v_1 \ldots v_n.e_i[e'_1/v'_1 \ldots e'_k/v'_k]]\!]\ \{\}\ \{\} = True$

$\forall i \in \{1 \ldots n\}.\mathcal{C}[\![e_i]\!]\ [\![e]\!]\ \rho\ \phi = \bot$

$\mathcal{C}[\![\textbf{case } v \textbf{ of } p_1 : e_1 \mid \ldots \mid p_n : e_n]\!]\ [\![e]\!]\ \rho\ \phi = \bot$

2. (a) $\mathcal{T}[\![ALL\ v_1 \ldots v_n.EX\ v.pre \Rightarrow \textbf{case } v \textbf{ of } p_1 : e_1 \mid \ldots \mid p_n : e_n]\!]\ \{\}\ \{\}$
   $= True$ (by assumption)
   $\Rightarrow \mathcal{T}[\![ALL\ v_1 \ldots v_n.EX\ v.pre]\!]\ \{\}\ \{\} = \bot$
   (b) $\mathcal{T}[\![ALL\ v_1 \ldots v_n.EX\ v.\textbf{case } v \textbf{ of } p_1 : e_1 \mid \ldots \mid p_n : e_n]\!]\ \{\}\ \{\} = \bot$
   (by assumption)

Case for Rule $\mathcal{C}8$:

If $\exists x \in \{v_1 \ldots v_m\}.x \in \{v'_1 \ldots v'_n\}$:

$$\mathcal{C}[\![\textbf{letrec } f = \lambda v_1 \ldots v_m.e_0 \textbf{ in } f\ v_1 \ldots v_m]\!]\ [\![v]\!]\ \rho\ \{v'_1 \ldots v'_n\}$$
$$= \textbf{letrec } f = \lambda v''_1 \ldots v''_k.e'_0 \textbf{ in } f\ v''_1 \ldots v''_k$$

where
$e'_0 \;=\; \mathcal{C}[\![e_0]\!]\;[\![v]\!]\;(\rho \cup \{f\ v_1 \ldots v_n\})\;\{v'_1 \ldots v'_n\}$

By the inductive hypothesis:

1. $\mathcal{C}[\![e_0]\!]\;[\![e]\!]\;\rho\;\{v'_1 \ldots v'_n\} \;=\; e[e''_1/v''_1 \ldots e''_k/v''_k]$
   $\Rightarrow\; \mathcal{T}[\![ALL\ v'_1 \ldots v'_n.e_0[e''_1/v''_1 \ldots e''_k/v''_k]]\!]\;\{\}\;\{\} \;=\; True$

2. (a) $\mathcal{C}[\![e_0]\!]\;[\![e]\!]\;\rho\;\{v'_1 \ldots v'_n\} \;=\; \bot$
   $\wedge\; \mathcal{T}[\![ALL\ v'_1 \ldots v'_n.EX\ v.pre \;\Rightarrow\; e_0]\!]\;\{\}\;\{\} \;=\; True$
   $\Rightarrow\; \mathcal{T}[\![ALL\ v'_1 \ldots v'_n.pre]\!]\;\{\}\;\{\} \;=\; \bot$
   (b) $\mathcal{C}[\![e_0]\!]\;[\![e]\!]\;\rho\;\{v'_1 \ldots v'_n\} \;=\; \bot$
   $\Rightarrow\; \mathcal{T}[\![ALL\ v'_1 \ldots v'_n.EX\ v.e_0]\!]\;\{\}\;\{\} \;=\; \bot$

If $\nexists x \in \{v_1 \ldots v_m\}.x \in \{v'_1 \ldots v'_n\}$:
$\mathcal{C}[\![\textbf{letrec}\ f \;=\; \lambda v_1 \ldots v_m.e_0\ \textbf{in}\ f\ v_1 \ldots v_m]\!]\;[\![v]\!]\;\rho\;\{v'_1 \ldots v'_n\} \;=\; e'_0$
where
$e'_0 \;=\; \mathcal{C}[\![e_0]\!]\;[\![v]\!]\;(\rho \cup \{f\ v_1 \ldots v_n\})\;\{v'_1 \ldots v'_n\}$

By the inductive hypothesis:

1. $\mathcal{C}[\![e_0]\!]\;[\![e]\!]\;\rho\;\{v'_1 \ldots v'_n\} \;=\; e[e''_1/v''_1 \ldots e''_k/v''_k]$
   $\Rightarrow\; \mathcal{T}[\![ALL\ v'_1 \ldots v'_n.e_0[e''_1/v''_1 \ldots e''_k/v''_k]]\!]\;\{\}\;\{\} \;=\; True$

2. (a) $\mathcal{C}[\![e_0]\!]\;[\![e]\!]\;\rho\;\{v'_1 \ldots v'_n\} \;=\; \bot$
   $\wedge\; \mathcal{T}[\![ALL\ v'_1 \ldots v'_n.EX\ v.pre \;\Rightarrow\; e_0]\!]\;\{\}\;\{\} \;=\; True$
   $\Rightarrow\; \mathcal{T}[\![ALL\ v'_1 \ldots v'_n.pre]\!]\;\{\}\;\{\} \;=\; \bot$
   (b) $\mathcal{C}[\![e_0]\!]\;[\![e]\!]\;\rho\;\{v'_1 \ldots v'_n\} \;=\; \bot$
   $\Rightarrow\; \mathcal{T}[\![ALL\ v'_1 \ldots v'_n.EX\ v.e_0]\!]\;\{\}\;\{\} \;=\; \bot$

Case for Rule $\mathcal{C}9$:
If $\exists x \in \{v_1 \ldots v_m\}.x \in \{v'_1 \ldots v'_n\}$:
$\mathcal{C}[\![\textbf{letrec}\ f \;=\; \lambda v_1 \ldots v_m.e_0\ \textbf{in}\ f\ v_1 \ldots v_m]\!]\;[\![c\ e'_1 \ldots e'_l]\!]\;\rho\;\{v'_1 \ldots v'_n\}$
$= (c\ e'_1 \ldots e'_l)[(\mathcal{C}[\![\textbf{letrec}\ f \;=\; \lambda v_1 \ldots v_m.e_0\ \textbf{in}\ f\ v_1 \ldots v_m]\!]\;[\![e'_i]\!]\;\rho\;\{v'_1 \ldots v'_n\}/v''_i)]$

By the inductive hypothesis, $\forall i \in \{1 \ldots l\}$:

1. $\mathcal{C}[\![\textbf{letrec}\ f = \lambda v_1 \ldots v_m.e_0\ \textbf{in}\ f\ v_1 \ldots v_m]\!]\;[\![e'_i]\!]\;\rho\;\{v'_1 \ldots v'_n\} = \;e[e''_1/v''_1 \ldots e''_k/v''_k]$
   $\Rightarrow \mathcal{T}[\![ALL\ v'_1 \ldots v'_n.\textbf{letrec}\ f = \lambda v_1 \ldots v_m.e_0\ \textbf{in}\ f\ v_1 \ldots v_m[e''_1/v''_1 \ldots e''_k/v''_k]]\!]\;\{\}\;\{\}$
   $= \;True$

If $\nexists x \in \{v_1 \ldots v_m\}.x \in \{v'_1 \ldots v'_n\}$:

$$\mathcal{C}[\![\textbf{letrec } f \ = \ \lambda v_1 \ldots v_m.e_0 \textbf{ in } f \ v_1 \ldots v_m]\!] \ [\![e'_i]\!] \ \rho \ \{v'_1 \ldots v'_n\}$$
$$= \ \mathcal{C}[\![e_0]\!] \ [\![c \ e'_1 \ldots e'_l]\!] \ \rho \ \{v'_1 \ldots v'_n\}$$

By the inductive hypothesis: $\forall i \in \{1 \ldots l\}$:

1. $\mathcal{C}[\![e_0]\!] \ [\![e_i]\!] \ \rho \ \{v'_1 \ldots v'_n\} \ = \ e[e''_1/v''_1 \ldots e''_k/v''_k]$
   $\Rightarrow \ \mathcal{T}[\![ALL \ v'_1 \ldots v'_n.e_0[e''_1/v''_1 \ldots e''_k/v''_k]]\!] \ \{\} \ \{\} \ = \ True$

2. (a) $\mathcal{C}[\![e_0]\!] \ [\![e_i]\!] \ \rho \ \{v'_1 \ldots v'_n\} \ = \ \perp$
   $\wedge \ \mathcal{T}[\![ALL \ v'_1 \ldots v'_n.EX \ v.pre \ \Rightarrow \ e_0]\!] \ \{\} \ \{\} \ = \ True$
   $\Rightarrow \ \mathcal{T}[\![ALL \ v'_1 \ldots v'_n.pre]\!] \ \{\} \ \{\} \ = \ \perp$

   (b) $\mathcal{C}[\![e_0]\!] \ [\![e_i]\!] \ \rho \ \{v'_1 \ldots v'_n\} \ = \ \perp$
   $\Rightarrow \ \mathcal{T}[\![ALL \ v'_1 \ldots v'_n.EX \ v.e_0]\!] \ \{\} \ \{\} \ = \ \perp$

5.6 Conclusion

In this chapter, a novel program construction method has been presented which can be used to construct correct programs from input specifications. We have extended our language and distillation rules to handle input specifications with ANY-quantification. Unsatisfiable specifications are rejected during the construction process if we are unable to prove the existential conjectures generated from the input specification. We have then shown how a correct program can be constructed by removing the precondition from the input specification.

We have formally defined the program construction rules $\mathcal{C}$ which can be used to construct programs from specifications. The application of these rules has been demonstrated with two examples. The constructed program is executable in the source language. The examples show that the constructed programs are efficient and correct with respect to the input specifications. We have implemented this program construction technique and added it to the theorem prover Poitín.

Chapter 6

Implementation and Results

6.1 Introduction

In this chapter, we briefly overview the implementation of the theorem prover Poitín, and present some results of our research. The theorem prover is implemented using the functional programming language `Standard ML of New Jersey v110.60`. The strong type system of `Standard ML` allows the definition of appropriate data types to represent input conjectures and specifications as data objects of these types. The implementation of the Poitín theorem prover consists of three main modules: `Toplevel`, `ATP` and `Distill`. The `Toplevel` module implements the main interface to the theorem prover, which consists of several menu options. The `ATP` module implements the data types to express input conjectures and program specifications, and implements functions that operate on these expressions. The module `Distill` is the main module of the theorem prover, which implements the distillation rules for quantification, program transformation, and the proof and program construction rules.

6.2 Poitín: a Prototype Version

In this section, we present the data types and main functions of each module of the theorem prover. The `Toplevel` module and the distillation program transformer were implemented jointly by myself and my supervisor Geoff Hamilton. I have improved the implementation of the embedding detection algorithm and the generalization techniques of distillation. In addition, I have implemented the preprocessing phase, the distillation rules for quantifiers, the universal and existential

proof rules, the program construction steps, and the program construction rules.

6.2.1 Module Toplevel

The `Toplevel` module consists of the function `toplevel` which implements the interface to the Poitín theorem prover. The function `toplevel` has the following signature.

```
val toplevel:  unit -> unit
```

On execution of the function `toplevel` in the `SML` prompt, the following prompt appears to interact with the Poitín theorem prover:

`POT>`

The available commands at this prompt are: `load, save, distill, step, show, showprog, graph, help` and `quit`. One may learn about these commands by using the `help` command. The input conjecture or program specification is stored with function definitions as a program defined in the language in a `.pot` file, and can be loaded using the command `load` *filename*. To prove or construct programs, the command `distill` is used, and the output can be viewed using the command `show`. Using the `step` command, one can switch to step mode distillation after loading the input file.

6.2.2 Module ATP

The module `ATP` defines the data type `t` as shown in Fig. 6.1 to represent any expression in the higher order functional language described in §2.2.1, input conjecture to be proved and input specification for program construction. Some of the main functions of this module which operate on the expressions defined using the data type `t` are listed in Fig. 6.2.

The function `inForm` parses the expressions defined in the higher order functional language, the operands of the infix operators $\rightarrow, \leftrightarrow, \wedge, \vee$, and the operands of the quantifiers `ALL, EX` and `ANY`. The function `readTerm` returns an expression of data type `t` by processing a string consisting of an expression of data type `t`. For example, the conjecture ALL x.EX $y.(even\ x) \leftrightarrow (eqnum\ (double\ y)\ x)$ is processed by the function `readTerm` to give the following:

```
ALL("x",EX("y",Apply(Apply(Fun "iff",Apply(Free "even",Free "x")),
 Apply(Apply(Free "eqnum",Apply(Free "double",Free "y")),Free "x"))))
```

```
datatype t =  Free of string
            | Bound of int
            | Fun of string
            | Let of string * t * t
            | Letrec of string * t * t
            | Abs of string * t
            | Con of string * t list
            | Apply of t * t
            | Case of t * (string * string list * t) list
            | Node of string * t * (string * t) list
            | Repeat of string * t * (string * t) list
            | ALL of string * t
            | EX of string * t
            | ANY of string * t * t
```

Figure 6.1: Data type of ATP module

```
val  freevars:  t -> string list
val  abstract:  int -> string -> t -> t
val  shift:  int -> int -> t -> t
val  subst:  int -> t -> t -> t
val  inst:  t StringDict.t -> t -> t
val  rename:  string list -> string -> string
val  absList:  string list * t -> t
val  allList:  string list * t -> t
val  exList:  string list * t -> t
val  anyList:  (string * t) list * t -> t
val  inForm:  ATPParsing.token list -> t * ATPParsing.token list
val  outTerm:  t -> Pretty.t
val  readTerm:  string -> t
val  rdInput:  string -> (t * (string * t) list)
val  outTree:  t -> Pretty.t
```

Figure 6.2: Functions of ATP module

The input specification ANY z : $nat.(less\ x\ y) \rightarrow (eqnum\ (plus\ x\ z)\ y)$ is processed by the function `readTerm` to give the following:

```
ANY("z",Nat "nat",Apply(Apply(Fun "implies",Apply(Apply(Free "less",
 Free "x"),Free "y")),Apply(Apply(Free "eqnum",Apply(Apply(Free "plus",
 Free "x"),Free "z")),Free "y")))
```

The function `rdInput` converts a string consisting of an expression of data type `t` with function definitions in the form of a program to a pair consisting of an expression and a list of function definitions. For example, the program

$$
\begin{array}{l}
append\ xs\ ys \\
\text{where} \\
append = \lambda xs.\lambda ys.\mathbf{case}\ xs\ \mathbf{of} \\
\qquad\qquad\qquad Nil \qquad\quad : ys \\
\qquad\qquad\quad\ \mid Cons\ x\ xs' : Cons\ x\ (append\ xs'\ ys);
\end{array}
$$

is processed by the function `rdInput` to give the following:

```
(Apply(Apply(Fun "append",Free "xs"),Free "ys"),[("append",Abs("xs",
Abs("ys",Case(Free "xs",[("Nil",[],Free "ys"),("Cons",[Free "x",
Free "xs'"],Con("Cons",[Free "x",Apply(Apply(Fun "append",Free "xs'"),
Free "ys")]))]))))])
```

6.2.3 Module Distill

The module `Distill` consists of six main functions: `distill`, `pass1`, `pass2`, `forall`, `exist` and `constructany`.

The function `distill` implements the distillation rule $\mathcal{T}1$ using the function `pass1` to implement the distillation program transformation algorithm as defined in §3.2, the pre-processing phase as described in Chapter 4 using the function `pass2`, rule $\mathcal{T}2$ for universal quantification and rule $\mathcal{T}3$ for existential quantification as defined in Chapter 4, and rule $\mathcal{T}4$ for ANY-quantification as defined in Chapter 5. The implementation of rule $\mathcal{T}4$ includes the implementation of the program construction steps as described in §5.3.3. A verification proof of the input specification is performed before the application of rule $\mathcal{T}4$. As the input specification contains a precondition, a new specification is constructed by removing the precondition. The

functions **forall**, **exist** and **constructany** implement the proof rules $\mathcal{A}$, $\mathcal{E}$, and the program construction rules $\mathcal{C}$, which are discussed in §6.2.4 and §6.2.5, respectively.

The `distill` function has the form *distill* s $[\![e]\!]$ ρ ϕ, where s is the step mode indicator, and e is the input expression to which the distillation rules will be applied. The parameter ρ represents the set of previously encountered expressions, and ϕ represents the set of function definitions used within the expression e. The function `distill` has the following signature.

```
val distill :  bool -> ATP.t * (string * ATP.t) list -> ATP.t
```

The first argument of this function is a SML boolean value: `true` or `false`, which indicates whether the `step` mode is on or off. The second argument is a pair of an ATP term of type `t` and a set of function definitions. Each function definition consists of a function name and the function body. The outcome of this function for an input conjecture may be `True` or `Bottom`. For an input specification with `ANY`-quantification, the outcome is a program which computes the existential witness. If the input expression does not contain any sort of quantification, the `distill` function transforms the input program to an equivalent and efficient output program.

The function `pass1` implements the distillation algorithm as defined by rule $\mathcal{T}1$ while the function `pass2` reconstructs the residual program resulting from the function `pass1` by performing the pre-processing tasks (§4.2).

The function `pass1` implements the normal order reduction rules $\mathcal{N}$ as described in §2.2.4. This function implements the reduction rules, which decompose the first argument to this function into a unique *context* and a *redex* based on the *unique decomposition property*. The function *findMatch* tries to find a match of its second argument with any of the expressions of type `t` within the matrix in the first argument. The functions **findembed**, **finddive** and **findCouple** implement the *homeomorphic embedding* detection algorithm to detect whether any of the expressions of type `t` within the matrix in the second argument is embedded within the expression in the third argument or not. The function `extract` performs generalization of an obstructing function call in the case of *strict* embedding, and the function *generalise* performs the *most specific generalization* if the embedding is *non-strict*. The `unfold` function implements the unfolding operation of a function call.

The `construct` function implements the residual program construction rules $\mathcal{P}$ as described in §2.2.4 and §3.2.3. We recall Example 9 of Chapter 3; one of the

expressions which was encountered in demonstrating the example is used to explain the use of the `construct` function as shown below.

$$
\begin{array}{l}
\texttt{construct}\ []\ \textbf{Node f0:} \\
\qquad\qquad \textbf{case}\ x\ \textbf{of} \\
\qquad\qquad\quad \textit{Zero} \quad : \textbf{case}\ y\ \textbf{of} \\
\qquad\qquad\qquad\qquad \textit{Zero} \quad : \textit{True} \\
\qquad\qquad\qquad\qquad \mid \textit{Succ}\ y' : \textit{True} \\
\qquad\qquad \mid \textit{Succ}\ x' : \textbf{Repeat f0: Node f1: case}\ x'\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{Zero} \quad : \textbf{case}\ y\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textit{Zero} \quad : \textit{True} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid \textit{Succ}\ y' : \textit{True} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid \textit{Succ}\ x'' : \textbf{Repeat f1:}\ \textit{leq}\ x''\ (\textit{plus}\ x''\ y)
\end{array}
$$

The `construct` function returns the following residual program.

$$
\begin{array}{lcl}
\textbf{letrec}\ f0 & = & \lambda x.\textbf{case}\ x\ \textbf{of} \\
& & \quad \textit{Zero} \quad : \textbf{case}\ y\ \textbf{of} \\
& & \qquad\qquad \textit{Zero} \quad : \textit{True} \\
& & \qquad\qquad \mid \textit{Succ}\ y' : \textit{True} \\
& & \quad \mid \textit{Succ}\ x' : f0\ x' \\
\textbf{in}\ f0\ x & &
\end{array}
$$

6.2.4 Implementation of the Proof Rules $\mathcal{A}$ and $\mathcal{E}$

The function `forall` implements the proof rules $\mathcal{A}$ for universal quantification, and the function `exist` implements the proof rules $\mathcal{E}$ for existential quantification. These functions are invoked by the function `distill` to prove input conjectures.

The function `forall` has the form $forall\ [\![e]\!]\ \rho\ \phi$, where e is the proof expression to which the proof rules will be applied and the parameter ρ is the set of the previously encountered function calls, and ϕ is the set of universally quantified variables. The function `forall` has the following signature:

```
val forall :  ATP.t -> ATP.t list -> string list ->
              ATP.t * ATP.t list * string list
```

The outcome of this function is a simplified proof expression, a set of function calls and a set of universally quantified variables.

The function `exist` has the form $exist\ [\![e]\!]\ \rho\ \phi$, where e is the proof expression to which the proof rules will be applied and the parameter ρ is the set of the previously

encountered function calls, and ϕ is the set of existentially quantified variables. The function `exist` has the following signature.

```
val exist :  ATP.t -> ATP.t list -> string list ->
             ATP.t * ATP.t list * string list
```

The outcome of this function is a simplified proof expression, a set of function calls and a set of existentially quantified variables.

6.2.5 Implementation of the Program Construction Rules $\mathcal{C}$

The function `constructany` implements the proof rules $\mathcal{C}$ which perform the constructive proof of a distilled expression. The function `constructany` has the form *constructany* $[\![e]\!]$ $[\![e']\!]$ τ ρ ϕ. This function is invoked by the function `distill` to construct a program from the proof expression obtained from an input specification. The function `constructany` has the following signature.

```
val constructany :  ATP.t -> ATP.t -> ATP.t -> ATP.t list ->
                    string list -> ATP.t * ATP.t list * string list
```

The first argument of the function `constructany` is a proof expression e of type `t` to which the constructive proof rules $\mathcal{C}$ will be applied. The second argument e' is the current existential witness and the third argument τ is the existential witness type. The parameter ρ is the set of the previously encountered function calls, and ϕ is the set of universally quantified variables within e. The outcome of this function is the simplified constructed program, a set of function calls and a set of universally quantified variables.

6.3 Results

We have applied the theorem prover Poitín to a large number of inductive theorems and program specifications. Poitín can prove these theorems without using any intermediate lemmas by performing only generalization, whereas some other inductive theorem provers require lemmas and generalizations to prove some of these theorems. Some of the conjectures listed in Table 6.1 were proved by SPIKE [66] using a divergence critic [107], NQTHM [8, 9], ACL2 [63], CLAM [21, 23] using rippling, and Periwinkle [68] by proposing lemmas or performing generalizations.

No.	Conjecture	Time (in Seconds)
1.	ALL x.ALL y.*eqnum* (*plus* x y) (*plus* y x)	0.0094
2.	ALL x.*eqnum* (*plus* x (*Succ* x)) (*Succ* (*plus* x x))	0.0016
3.	ALL x.ALL y.ALL z.*eqnum* (*plus* (*plus* x y) z) (*plus* x (*plus* y z))	0.0032
4.	ALL x.*eqnum* (*plus* (*plus* x x) x) (*plus* x (*plus* x x))	0.0046
5.	ALL x.*eqnum* (*gcd* x x) x	0.0016
6.	ALL x.ALL y.*eqnum* (*sub* (*plus* x y) x) y	0.0016
7.	ALL x.ALL y.*eqnum* (*plus* x (*Succ* y)) (*Succ* (*plus* x y))	0.0015
8.	ALL x.*even* (*plus* x x)	0.0016
9.	ALL x.*even* (*doublea* x *Zero*)	0.0016
10.	ALL x.ALL y.((*even* x) $\wedge$ (*even* y)) $\rightarrow$ (*even* (*plus* x y))	0.0078
11.	ALL x.(*eqbool* (*even* x) (*True*)) $\rightarrow$ (*eqbool* (*odd* x) (*False*))	0.0015
12.	ALL x.EX y.(*even* x) $\leftrightarrow$ (*eqnum* (*double* y) x)	0.0077
13.	ALL x.EX y.(*even* x) $\leftrightarrow$ (*eqnum* (*mult* y (*Succ* (*Succ* *Zero*))) x)	0.0016
14.	ALL x.ALL y.EX z.(*less* x y) $\rightarrow$ (*eqnum* (*plus* x z) y)	0.0032
15.	ALL xs.ALL ys.*eqnum* (*length* (*append* xs ys)) (*length* (*append* ys xs))	0.0109
16.	ALL xs.ALL ys.*eqnum* (*length* (*append* xs ys)) (*plus* (*length* xs) (*length* ys))	0.0016
17.	ALL xs.ALL ys.ALL zs.*eqlist* (*append* xs (*append* ys zs)) (*append* (*append* xs ys) zs)	0.0063
18.	ALL xs.ALL ys.(*even* (*length* (*append* xs ys))) $\leftrightarrow$ (*even* (*length* (*append* ys xs)))	0.0548

Table 6.1: Some conjectures proved in Poitín

Poitín can prove all of these conjectures fully automatically without requiring any intermediate lemmas. Conjectures 3, 6, 7, 16 and 17 do not require any generalization to be performed. All other conjectures require generalization to be performed during distillation. The times listed to prove the conjectures are given by the average of 10 runs for each conjecture on an Intel Pentium 4 PC with 2.40 GHz and 512 MB RAM. As these times are very low, the results are encouraging. The proof of conjecture 1 is troublesome for a lot of inductive theorem provers. This conjecture has two *unflawed* induction variables, which make the proof complicated for explicit inductive provers as discussed in Example 10 (§4.4.1). Poitín proves this conjecture by generalization of accumulating patterns during distillation. Conjecture 2 is also difficult to prove using previous proof techniques. SPIKE diverges in an attempt to prove conjecture 8. Divergence critic takes 5.4 seconds to suggest a lemma to prove this conjecture. The proof of conjecture 9 in the explicit induction method involves the introduction of a new universally quantified variable in place of the accumulating parameter [45, 54], which over-generalizes the conjecture by generating the new conjecture ALL x.ALL $y.even$ $(doublea\ x\ y)$. Poitín has also been successfully used in proving existential theorems (e.g., conjectures 12, 13 and 14 in Table 6.1). SPIKE fails to prove conjectures 15 and 16. The divergence critic avoids the divergence by proposing two lemmas in each case in 3.6 and 7.2 seconds respectively. Poitín proves conjecture 15 by generalization of accumulating patterns during distillation.

The following Table 6.2 shows some universally quantified conjectures which cannot currently be proved in Poitín. The function definitions in Fig. 6.3 were used along with the definitions of the functions *eqnum, double, plus, length, eqlist, reva, append* and *reverse* as defined in the previous chapters.

Among these conjectures, conjecture 3 states the commutativity of multiplication and conjecture 5 uses a mutually recursive function. The distillation of most of these conjectures suffers from non-termination due to successively growing patterns or the existence of more than one of the three different forms of non-termination. For example, the transformation of conjectures 3, 7 and 8 encounters the occurrence of both accumulating patterns and obstructing function calls. In the current version of Poitín, we consider only one form of embedding of two expressions: *strict* or *non-strict*, and the corresponding generalization. The distillation of conjecture 6 encounters the embedding of both accumulating patterns and accumulating parameters. The distillation of all other conjectures suffers from non-termination due to successively growing patterns because of *unification-based information propagation.*

No.	Conjecture
1.	ALL $x.eqnum\ (double\ x)\ (plus\ x\ x)$
2.	ALL $x.eqnum\ (half\ (plus\ x\ x))\ x$
3.	ALL x.ALL $y.eqnum\ (mult\ x\ y)\ (mult\ y\ x)$
4.	ALL x.ALL y.ALL z.ALL v.ALL $w.eqnum\ (plus\ x\ (plus\ y\ (plus\ z\ (plus\ v\ w))))\ (plus\ w\ (plus\ x\ (plus\ y\ (plus\ z\ v))))$
5.	ALL $xs.leq\ (length\ (evenlist\ xs))\ (length\ xs)$
6.	ALL xs.ALL $ys.eqlist\ (reva\ xs\ ys)\ (append\ (reverse\ xs)\ ys)$
7.	ALL $xs.eqlist\ (rotate\ (length\ xs)\ xs)\ xs$
8.	ALL $xs.eqlist\ (reverse\ (reverse\ xs))\ xs$
9.	ALL $xs.eqnum\ (length\ (append\ xs\ xs))\ (double\ (length\ xs))$

Table 6.2: Some of Poitín's failures

```
half     =  λx. case x of
               Zero     : Zero
             | Succ x'  : case x' of
                            Zero      : Zero
                          | Succ x''  : Succ (half x'')
mult     =  λx.λy. case x of
                Zero     : Zero
              | Succ x'  : plus y (mult x' y)
evenlist =  λxs. case xs of
               Nil          : Nil
             | Cons x xs'   : oddlist xs'
oddlist  =  λxs. case xs of
               Nil          : Nil
             | Cons x xs'   : Cons x (evenlist xs')
rotate   =  λx.λys. case x of
                 Zero     : ys
               | Succ x'  : case ys of
                              Nil         : Nil
                            | Cons y ys'  : rotate x' (append ys' (Cons y Nil))
```

Figure 6.3: Some function definitions for failed proofs

Table 6.3 shows how SPIKE using the divergence critic [107] and rippling [23, 17] deals with these conjectures. The symbol – indicates that the proof example could not be found using the indicated method.

No.	SPIKE (Divergence Critic)		Rippling	
	Proved	Lemma required	Proved	Lemma required
1.	√	√	–	–
2.	√	√	√	√
3.	×	×	–	–
4.	√	√	–	–
5.	×	×	–	–
6.	√	√	√	√
7.	√	√	√	√
8.	√	√	√	√
9.	√	√	–	–

Table 6.3: Conjectures of Table 6.2 proved by SPIKE using divergence critic and rippling

Poitín has been used to construct programs from input specifications. The constructed programs are efficient and correct with respect to the input specifications. In Table 6.4, some input specifications are listed, which were used to construct programs. The constructed programs from these specifications are shown in Fig. 6.4.

No.	**Specification**	Time (in Seconds)
1.	ANY $y : nat.(even\ x) \rightarrow (eqnum\ (double\ y)\ x)$	0.0095
2.	ANY $y : nat.(even\ x) \rightarrow$ $(eqnum\ (mult\ y\ (Succ\ (Succ\ Zero)))\ x)$	0.0031
3.	ANY $y : nat.(eqnum\ (double\ y)\ x)\ \vee$ $(eqnum\ (Succ\ (double\ y))\ x)$	0.0031
4.	ANY $z : nat.(less\ x\ y) \rightarrow (eqnum\ (plus\ x\ z)\ y)$	0.0047
5.	ANY $y : nat.(eqnum\ x\ (Zero)) \vee (eqnum\ x\ (Succ\ y))$	0.0046
6.	ANY $y : nat.eqnum\ y\ (plus\ x\ (Succ\ Zero))$	0.0046

Table 6.4: Some specifications for program construction

Specification No.	Constructed Program
1.	**letrec** $f0 = \lambda x.$**case** x **of** $Zero \quad : 0$ $\mid Succ\ x' :$ **case** x' **of** $Zero \quad : \bot$ $\mid Succ\ x'' : Succ\ (f0\ x'')$ **in** $f0\ x$
2.	**letrec** $f0 = \lambda x.$**case** x **of** $Zero \quad : 0$ $\mid Succ\ x' :$ **case** x' **of** $Zero \quad : \bot$ $\mid Succ\ x'' : Succ\ (f0\ x'')$ **in** $f0\ x$
3.	**letrec** $f0 = \lambda x.$**case** x **of** $Zero \quad : 0$ $\mid Succ\ x' :$ **case** x' **of** $Zero \quad : 0$ $\mid Succ\ x'' : Succ\ (f0\ x'')$ **in** $f0\ x$
4.	As shown in Example 13 of §5.4
5.	**case** x **of** $Zero \quad : 0$ $\mid Succ\ x' :$ **letrec** $f0 = \lambda x'.$**case** x' **of** $Zero \quad : 0$ $\mid Succ\ x'' : Succ\ (f0\ x'')$ **in** $f0\ x'$
6.	**letrec** $f0 = \lambda x.$**case** x **of** $Zero \quad : 1$ $\mid Succ\ x' : Succ\ (f0\ x')$ **in** $f0\ x$

Figure 6.4: Constructed programs for specifications of Table 6.4

In Fig. 6.4, the programs constructed from the specifications are totally correct.

The results show that some difficult theorems about natural numbers and lists were proved by Poitín. Only the definitions of logical connectives $\wedge, \vee, \rightarrow, \leftrightarrow$ etc. are provided as built-in functions. In theory, Poitín should be able to prove any conjectures about functions which are defined over inductive types such as sets, integers, rationals, trees, etc.

6.4 Conclusion

In this chapter, we have presented the implementation of the automatic theorem prover Poitín. The implementation includes four top-level distillation rules, inductive theorem proving rules, and program construction rules. The four top-level distillation rules control the functioning of the theorem prover. Rule $\mathcal{T}1$ implements the distillation program transformer, which is at the heart of the theorem prover. Rules $\mathcal{T}2$ and $\mathcal{T}3$ implement the inductive theorem prover, and rule $\mathcal{T}4$ implements the program construction method. The implementation of rule $\mathcal{T}4$ includes the implementation of the program construction steps as described in §5.3.3. In rule $\mathcal{T}4$, a verification proof of the input specification is performed before each application of this rule, so that incorrect specifications are rejected in the construction process. The construction process removes the precondition part from the input specification by generating a new specification for program construction. This ensures that only correct programs are constructed in Poitín.

We have presented some results of the application of the Poitín theorem prover to inductive theorems and program specifications. The results are encouraging, although some straightforward conjectures cannot be proved using the current implementation of Poitín. The main outcome is that the proof techniques of Poitín can be used to prove inductive conjectures fully automatically without the need for conjecturing any intermediate lemmas, whereas most inductive theorem provers require intermediate lemmas to prove these conjectures. Poitín also reduces over-generalization and generation of non-theorems. Our program construction techniques can be used to construct totally correct programs from input specifications. The future development plans for the theorem prover include the implementation of a graphical user interface, and improving the performance of the theorem prover.

Chapter 7

Conclusion and Future Work

In this thesis, we have shown how automatic program transformation can be used in a novel way in metacomputation-based inductive theorem proving and program construction methods. The work presented in this thesis is an extension of the theorem prover Poitín to handle explicit quantification. Our inductive proof technique is an alternative to standard inductive proof methods using induction rules in explicit induction. The theorem proving and program construction techniques of Poitín do not require any intermediate lemmas, and therefore remove the need for a search in a vast collection of lemmas which is required in the axiomatic approach. The associated search space is very small and is restricted to the set of expressions encountered during distillation, which constitute the set of inductive hypotheses.

The program associated with an input conjecture or program specification can be transformed with distillation to an equivalent and efficient output program which is in a normal form called *distilled form*. The distilled expression can then be simplified to a proof expression, which is used in theorem proving and program construction. Proof rules for universal and existential quantification were defined to prove these proof expressions. The existential proof rules perform pure existence proof of a proof expression. To construct a program from an input program specification, a constructive proof method has been presented. The constructed program is executable in the source language, and can compute the unknown values as specified by the input specification. We have proved that the constructed program will be correct with respect to the input specification.

To conclude the thesis, we first summarise the work presented in previous chapters, and then give some directions for future research.

7.1 Summary of Thesis

In this section, we summarise the main chapters of the thesis.

7.1.1 Background

In Chapter 2, we briefly surveyed the state of the art in the areas of program transformation, inductive theorem proving techniques and strategies using explicit induction, e.g. rippling, and program synthesis methods. A higher order functional language was defined which is used throughout the thesis. The higher order formulation of the supercompilation algorithm was given based on the presentation in [46]. We reviewed the recursion analysis technique used in the Boyer-Moore Theorem Prover to show how the required induction scheme for an inductive proof can be constructed from the recursive definitions of functions used within the inductive conjecture. We reviewed Turchin's metacomputation-based inductive theorem proving technique to prove logical formulas using supercompilation. We also showed the relationship of cut elimination to the removal of intermediate data structures from programs.

7.1.2 Distillation

In Chapter 3, an overview of the distillation algorithm was given based on the presentation in [46]. It was shown how the supercompilation algorithm can be extended to develop the more powerful distillation algorithm. The distillation algorithm was devised with the aim to remove intermediate data structures from higher order functional programs. The unification-based information propagation and the more powerful matching technique adopted in distillation have made this algorithm very suited to the metacomputation-based inductive theorem prover Poitín. The transformation rules for distillation were presented, and generalization methods were described based on homeomorphic embedding to ensure on-line termination. The distilled form of expressions resulting from distillation was also defined. The proof of termination of the distillation algorithm was given based on the termination proof of a language independent framework of an abstract program transformer [96], and the correctness proof was given based on the improvement theorem of Sands [89, 90].

7.1.3 Theorem Proving in Poitín

In Chapter 4, the inductive theorem proving techniques of Poitín were presented. The language and the distillation rules were extended to deal with explicit quantification. Two distillation rules were defined for universal and existential quantification to deal with quantifiers at the meta-level, and two sets of object level proof rules have been formalised to prove proof expressions which contain universal and existential variables. A set of potential inductive hypotheses is maintained during universal proof. An inductive hypothesis is only applied if a recursive function call is an instance of the inductive hypothesis, and at least one of the universally quantified variables in this application is decreasing. The existential proof rules use a pure existence proof technique. The great advantage of the proof techniques of Poitín is that these techniques do not require any intermediate lemmas, and therefore help to reduce the search required in an inductive proof. The soundness of the proof rules was shown with respect to a logical proof system using sequent calculus.

7.1.4 Program Construction in Poitín

In Chapter 5, a constructive proof method was presented to construct a higher order functional program from an input program specification. The language and the distillation rules were extended to handle ANY-quantified input specifications. A distillation rule was defined for ANY quantification, and constructive proof rules were defined for program construction from proof expressions. The program construction process was described, which includes a verification proof of the input specification to reject unsatisfiable specifications, so that programs are constructed only from satisfiable specifications. As the input specification contains a precondition, a new specification is generated by removing this precondition as it does not help to define the output data. A proof of correctness of the construction method was also given.

7.1.5 Implementation and Results

In Chapter 6, the prototype implementation of the theorem prover Poitín was presented. The distillation program transformer, distillation rules for the quantifiers ALL, EX and ANY, the proof rules and the program construction rules were implemented using `Standard ML`, and added to Poitín. The prototype of Poitín is an integrated environment for inductive theorem proving and program construction us-

ing higher order functional programs. Some results of the application of the theorem prover to inductive theorems and program specifications were presented. The results are encouraging although some straightforward conjectures cannot be proved using the current implementation of the theorem prover. Poitín proved these theorems fully automatically without requiring any intermediate lemmas, whereas the most inductive theorem provers require intermediate lemmas to prove some of these theorems. We proved that the programs constructed from the program specifications will be correct with respect to the specifications.

7.2 Research Contributions

This thesis mainly contributes to the fields of metacomputation-based inductive theorem proving and program construction. We have developed an inductive theorem proving and program construction framework to deal with explicit quantification. This framework can be used in conjunction with many existing program transformation algorithms. We have chosen to use distillation as it is the most powerful program transformation algorithm currently available.

Our work is a significant improvement over the theorem proving technique of Poitín [45] using distillation. In [45], all free variables of the input conjectures are considered implicitly universally quantified, and there is no explicit quantification. The theorem prover is not capable of any program construction from specifications. We have extended the theorem proving technique of Poitín to handle explicit universal and existential quantifications to prove explicitly quantified inductive conjectures fully automatically. We have defined distillation rules for quantifiers and the proof rules for universal and existential quantifications. We have developed a program construction method to construct correct, efficient and executable functional programs from the proofs of non-executable input specifications using program construction rules.

Our inductive proof method does not require any intermediate lemmas, which helps to avoid infinite branch points in the search space. The existential proof rules perform a pure existence proof of the existential conjecture without requiring to construct any witness. This is an alternative to the usual constructive approach to prove existential theorems using higher order unification. The inclusion of the distillation program transformation algorithm within the inductive theorem proving techniques has reduced over-generalization and generation of non-theorems. The

soundness of the proof techniques was shown with respect to a logical proof system using sequent calculus.

We have formalised a program construction method to construct programs from input specifications. The constructed program is correct with respect to the input specification, and executable in the source language. Though the programs developed in this method are still limited to small problems, it can help to reduce the burden of a programmer to some extent by automating the process of writing programs. This is the only method we know of which constructs programs from specifications fully automatically. The proof of correctness of the program construction method was also given. We also argue that the programs which are constructed using our techniques are likely to be more efficient than those which are generated by other constructive methods, as they are generated using distillation which has the main aim of making programs more efficient.

The theorem proving and program construction techniques have been implemented and added to the theorem prover Poitín. The use of distillation within the framework of Poitín has eased the automation of the inductive proof and program construction techniques to make Poitín a fully automatic and efficient theorem prover.

7.3 Future Work

There are many directions for future research that may arise from this thesis, which are described below.

7.3.1 Distillation

Distillation algorithm is at the heart of the theorem prover Poitín, and the range of inductive theorems that can be proved by Poitín depends on the power of distillation. Special techniques are needed to deal with conjectures involving functions defined with *mutual recursion.* For example, expression 1 is one such example where the function *evenlist* is mutually defined with another recursive function *oddlist.* Work is under way to develop techniques to transform some difficult expressions as shown below to obtain proof expressions which can be used in proving the respective theorems.

1. *leq (length (evenlist xs)) (length xs)*
2. *eqnum (double x) (plus x x)*
3. *eqnum (mult x y) (mult y x)*
4. *eqnum (length xs) (length (reverse xs))*
5. *eqlist (reverse (reverse xs)) xs*

The transformation of expression 1 encounters successively larger sub-expressions in the second occurrence of xs due to accumulating patterns. The transformation of expression 2 suffers from accumulating patterns, which cannot be solved with the current generalization technique. One possible solution to this problem is extending Poitín to be able to allow the use of intermediate lemmas where such failures are detected. The transformation of expressions 3-5 encounters the occurrence of both accumulating patterns and obstructing function calls as discussed in §6.3. One possible solution to this problem is to ignore the embedding of accumulating patterns, and performing generalization of obstructing function calls, which are under investigation. We are working to extend the power of distillation.

The distillation algorithm has already been implemented. The future development includes a re-implementation in its own input language which will allow the transformer to be self-applicable. Distillation algorithm will also be incorporated into a full programming language, which will allow a lot of powerful optimisations to be performed on programs in the language, and will also allow the verification of properties about these programs using Poitín.

7.3.2 Inductive Theorem Proving

Poitín can prove a wide range of inductive theorems. As the functions within the output residual programs obtained with distillation are parameterised with all of the unique free variables appearing in a recursive expression in the pre-processing phase, the recursive call to this function may contain non-decreasing variables. One major problem is caused by the substitution of patterns for these non-decreasing variables during proof rule application on some existential conjectures. We are working to resolve this problem. The conjectures below suffer from this problem.

$$\text{EX } y.\text{ALL } x.(\mathit{even}\ x) \leftrightarrow (\mathit{eqnum}\ (\mathit{double}\ y)\ x)$$

$$\text{ALL } x.\text{ALL } y.\text{EX } q.\text{EX } r.(\mathit{neq}\ x\ \mathit{Zero}) \rightarrow ((\mathit{eqnum}\ (\mathit{plus}\ (\mathit{mult}\ q\ x)\ r)\ y) \wedge (\mathit{less}\ r\ x))$$

To solve this problem, we propose a single set of proof rules for both universal and existential quantifications by merging the two sets of separate proof rules $\mathcal{A}$ and $\mathcal{E}$. The distillation rules $\mathcal{T}2$ and $\mathcal{T}3$ for quantifiers and the proof rules must include parameters for the current quantification scope, universal and existential variables.

Even though the results for our theorem proving techniques may appear somewhat disappointing, the main reason for these disappointing results is the performance of distillation. This is because Poitín can be used to try and prove only those conjectures which can be distilled successfully. The distillation algorithm can be improved to solve the problems as described in §7.3.1. We have however developed a framework for the proof of quantified conjectures using program transformation. Thus, any future improvements to the program transformation algorithm distillation will also feed in to the theorem proving to make improvements in this area too.

7.3.3 Program Verification

The inductive proof techniques can be used in program verification. To prove a property P about a program, P is expressed as an input conjecture in the form of a program in the language. The theorem proving techniques can then be used to prove P. An application of inductive proof rules in program verification can be found in [47].

7.3.4 Program Construction

The program construction method presented in Chapter 5 can deal with input specifications that contain an existential variable which is ANY-quantified. The constructive proof of the input specification results in a program which is a function that computes the witness. One possible extension to this method is to deal with input specifications that contain multiple existential variables which are ANY-quantified. For each existential variable, the constructive proof will be performed separately to construct a function to compute the witness. Thus, the extended program construction method will construct n separate functions to compute n witnesses. This method is applicable if the existential variables are independent of each other within the specification where each witness can be computed only in terms of the input variables using the program construction rules. An alternative method to handle multiple existential variables is shown below.

$$\mathcal{T}[\![c\langle \text{ANY } e : \tau.e' \rangle]\!]\ \rho\ \phi\ = \mathcal{T}[\![c\langle e''' \rangle]\!]\ \rho\ \phi$$
where
$$e'' = \mathcal{T}[\![e']\!]\ \{\}\ \phi$$
$$e''' = \mathcal{C}[\![e'']\!]\ [\![e]\!]\ \{\}\ (fv(e') \setminus fv(e))$$

In the specification, e is the existential witness, which is a constructor applied to the existential variables contained in the expression e'. A separate function will be constructed for each existential variable.

The results for our program construction technique may also appear somewhat disappointing. The main reason for these disappointing results is again the performance of distillation. We have however developed a framework for the program construction using program transformation. Thus, any future improvements to the program transformation algorithm distillation will also feed in to the program construction to make improvements in this area too.

7.3.5 Implementation

The future development of the theorem prover includes the implementation of a graphical user interface. Refinement of the implementation of the distillation algorithm is also in progress to enhance its power. The generalization technique could also be extended and implemented to deal with generalizations of expressions involving multiple embeddings of *obstructing function calls*, *accumulating patterns* and *accumulating parameters* at the same time as proposed in §7.3.1. A possible solution to the non-termination problem due to pattern substitution during proof as proposed in §7.3.2 will also be implemented, and added to the theorem prover.

Bibliography

[1] A. Armando, A. Smaill, and I. Green. Automatic synthesis of recursive programs: The proof-planning paradigm. In *12th IEEE International Automated Software Engineering Conference, IEEE Computer Society*, 1997.

[2] D. Basin and T. Walsh. Annotated rewriting in inductive theorem proving. Technical report, Max-Planck-Institute für Informatik, 1994.

[3] D. Basin and T. Walsh. A calculus for and termination of rippling. *Journal of Automated Reasoning*, 16(1-2), 1996.

[4] David Basin, Yves Deville, Pierre Flenner, Andreas Hamfelt, and Jørgen Fischer Nilsson. Synthesis of programs in computational logic. In *Program Development in Computational Logic, Lecture Notes in Computer Science LNCS 3049, Springer*, pages 30–65, 2004.

[5] A. Bouhoula, E. Kounalis, and M. Rusinowitch. Automated mathematical induction. *Journal of Logic and Computation*, 5:631–668, 1995.

[6] A. Bouhoula and M. Rusinowitch. Implicit induction in conditional theories. *Journal of Automated Reasoning*, 1995.

[7] R. S. Boyer and J. S. Moore. Proving theorems about lisp functions. *Journal of the ACM (JACM), ACM Press New York, NY*, 22, 1975.

[8] R. S. Boyer and J. S. Moore. *A Computational Logic.* Academic Press, ACM monograph series, 1979.

[9] R. S. Boyer and J. S. Moore. *A Computational Logic Handbook*, volume 23. Academic Press, Perspectives in Computing, 1988.

[10] James Brotherston. Cyclic-proofs for first-order logic with inductive definitions. In *Automated Reasoning with Analytic Tableaux and Related Methods:*

Proceedings of TABLEAUX 2005, volume 3702 of *Lecture Notes in Artificial Intelligence*, pages 78–92. Springer-Verlag, 2005.

[11] James Brotherston. *Sequent calculus proof systems for inductive definitions.* PhD thesis, Laboratory for Foundations of Computer Science, School of Informatics, University of Edinburgh, 2006.

[12] James Brotherston and Alex Simpson. Complete sequent calculi for induction and infinite descent. In *Proceedings of the Twenty-Second Annual IEEE Symposium on Logic in Computer Science (LICS 2007)*, pages 51–62, 2007.

[13] Alan Bundy. The use of explicit plans to guide inductive proofs. In R. Lusk and R. Oberbeek, editors, *9th Conference on Automated Deduction, CADE-9*, volume 310 of *Lecture Notes in Computer Science*, pages 111–120. Springer-Verlag, 1988.

[14] Alan Bundy. A science of reasoning. *Computational Logic: Essays in Honor of Alan Robinson, MIT Press*, pages 178–198, 1991.

[15] Alan Bundy. The automation of proof by mathematical induction. *Elsevier Science B.V.*, 1995.

[16] Alan Bundy. A survey of automated deduction. *Artificial Intelligence Today. Recent Trends and Developments*, 1999.

[17] Alan Bundy. *Handbook of Automated Reasoning*, volume I. Elsevier Science B.V., Amsterdam, 2001.

[18] Alan Bundy, David Basin, Dieter Hutter, and Andrew Ireland. *Rippling: Meta-level Guidance for Mathematical Reasoning.* Cambridge University Press, 2005.

[19] Alan Bundy, F. V. Harmelen, J. Hesketh, A. Smaill, and A. Stevens. A rational reconstruction and extension of recursion analysis. In N. S. Sridharan, editor, *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 359–365. Morgan Kaufmann, 1989.

[20] Alan Bundy, Frank Van Harmelen, Jane Hesketh, and Alan Smaill. Experiments with proof plans for induction. *Journal of Automated Reasoning*, 7:303–324, 1991.

[21] Alan Bundy, Frank Van Harmelen, Christian Horn, and Alan Smaill. The Oyster-Clam system. In M. E. Stickel, editor, *Proceedings of the 10th International Conference on Automated Deduction*, pages 647–648. Springer-Verlag, 1990.

[22] Alan Bundy, A. Smaill, and J. Hesketh. Turning eureka steps into calculations in automatic program synthesis. In S. L. H. Clarke, editor, *Proceedings of UKIT*, pages 111–120. Springer-Verlag: London, 1990.

[23] Alan Bundy, Andrew Stevens, Frank Van Harmelen, Andrew Ireland, and Alan Smaill. Rippling: A heuristic for guiding inductive proofs. *AI Journal*, 62:185–253, 1993.

[24] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24:44–67, 1977.

[25] Wei-Ngan Chin. *Automatic Methods for Program Transformation*. PhD thesis, Imperial College, University of London, October 1990.

[26] A. R. Choudhury. Induction proofs by program transformations. Technical report, School of Computing, National University of Singapore, 2003.

[27] A. Church. A set of postulates for the foundation of logic (1). *Annals of Mathematics*, 33:346–366, 1932.

[28] A. Church. A set of postulates for the foundation of logic (2). *Annals of Mathematics*, 34:839–864, 1933.

[29] A. Church. A formulation of the simple theory of types. *The journal of symbolic logic*, 5:56–68, 1940.

[30] J. Robin B. Cockett. Deforestation, program transformation and cut-elimination. *Electronics Notes in Theoretical Computer Science*, 44(1):88–127, 2001.

[31] R. L. Constable, S. F. Allen, and H. M Bromley. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice Hall, 1986.

[32] Luís Cruz-Filipe and Bas Spitters. Program extraction from large proof developments. *Lecture Notes in Computer Science*, 2758:205–220, 2003.

[33] Haskell B. Curry and Robert Feys. Combinatory logic. I, 1958. North-Holland.

[34] L. Damas and R. Milner. Principal type schemes for functional programs. In *Proceedings of the Ninth ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[35] N. Dershowitz and J. P. Jouannaud. Rewrite systems. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 244–320. Elsevier, 1992.

[36] Yves Deville and Kung-Kiu Lau. Logic program synthesis. *The Journal of Logic Programming, Elsevier Science Publishing Co., Inc.*, 19-20:321–350, 1994.

[37] L. Dixon and J. D. Fleuriot. IsaPlanner: A prototype proof planner in isabelle. In *Proceedings of CADE'03*, LNCS, pages 279–283, 2003.

[38] L. Dixon and J. D. Fleuriot. Higher order rippling in IsaPlanner. In *Theorem proving in higher order logics 2004 (TPHOL's2004)*, volume 3223 of *LNCS*, pages 83–98. Springer, 2004.

[39] G. Gentzen. Investigations into logical deduction (1934). In M. E. Szabo, editor, *The Collected Works of Gerhard Gentzen*, North-Holland Publishing Company, 1969, pages 68–131.

[40] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types.* Cambridge Tracts in Theoretical Computer Science 7, Cambridge University press, 1993.

[41] Robert Glück. Towards multiple self application. In *PEPM '1: Proceedings of the 1991 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 309–320. ACM Press, New Haven, Connecticut, United States, 1991.

[42] Robert Glück and Morten Heine Sørensen. A roadmap to metacomputation by supercompilation. In Oliver Danvy, Robert Glück, and Peter Thiemann, editors, *Selected papers of the International Seminar- Partial Evaluation*, volume 1110 of *Lecture Notes in Computer Science*, pages 137–160. Springer-Verlag, 1996.

[43] K. Godel. *ber formal unentscheidbare Stze der Principia Mathematica.* 1931.

[44] G. W. Hamilton. *Compile Time Optimisation of Store Usage in Lazy Functional Programs.* PhD thesis, Department of Computing Science and Mathematics, University of Stirling, U.K., October 1993.

[45] G. W. Hamilton. Poitín: Distilling theorems from conjectures. *Electronic Notes in Theoretical Computer Science*, 151(1):143–160, 2006.

[46] G. W. Hamilton. Distillation: Extracting the essence of programs. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 61–70. 2007.

[47] G. W. Hamilton. Distilling programs for verification. In *Proceedings of the 6th International Workshop on Compiler Optimization meets Compiler Verification, ETAPS 2007*, pages 21–35. 2007.

[48] G. Higman. Ordering by divisibility in abstract algebras. *Proc. London Math. Soc.*, (3):2:326–336, 1972.

[49] R. Hindley. The principal type scheme of an object in combinatory logic. *Trans. Am. Math. Soc.*, 146, 1969.

[50] W. A. Howard. The formulae-as-types notion of construction. *To: H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490, 1980.

[51] D. Hutter and Alan Bundy. The design of the CADE-16 inductive theorem prover contest.

[52] A. Ireland. The use of planning critics in mechanizing inductive proofs. In A. Voronkov, editor, *International Conference on Logic Programming and Automated Reasoning - LPAR'92*, volume 624 of *Lecture Notes in Artificial Intelligence*, pages 178–189. Springer-Verlag, 1992.

[53] A. Ireland and Alan Bundy. Productive use of failure in inductive proof. *Journal of Automated Reasoning on Inductive Proof, special edition*, 16(1-2):79–111, 1996.

[54] A. Ireland and Alan Bundy. Automatic verification of functions with accumulating parameters. *Journal of Functional Programming*, 9:225–245, 1999.

[55] N. D. Jones. An introduction to partial evaluation. *ACM Computing Surveys (CSUR)*, 28:480–503, 1996.

[56] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall International, 1993.

[57] Neil D. Jones. The expressive power of higher-order-types or life without CONS. *Journal of Functional Programming*, 11:55–94, 2001.

[58] L. Julia. Proofs by structural induction using partial evaluation. In *Proceedings of the 1993 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 155–166, 1993.

[59] H. Kabir and G. W. Hamilton. Constructing programs from metasystem transition proofs, 2007. School of Computing, Dublin City University, Working Paper CA-0207.

[60] H. Kabir and G. W. Hamilton. Extending Poitín to handle explicit quantification. In Silvio Ranise, editor, *Proceedings of the 6th International Workshop on First-Order Theorem Proving FTP 2007*, pages 20–34, 2007.

[61] H. Kabir and G. W. Hamilton. Extending Poitín to handle explicit quantifiers, 2007. School of Computing, Dublin City University, Working Paper CA-0107.

[62] M. Kaufmann and J. S. Moore. An industrial strength theorem prover for a logic based on common LISP. *IEEE transactions on software engineering*, 23(4):203–213, 1997.

[63] Matt Kaufmann, Panagiotis Manolios, and J. Strother Moore. *Computer-Aided Reasoning- An Approach.* Kluwer Academic Publishers, 2000.

[64] Emanuel Kitzelmann, Ute Schmid, Martin Mühlpfordt, and Fritz Wysotzki. Inductive synthesis of functional programs. *Lecture Notes in Computer Science*, 2385:26–37, 2002.

[65] L. Kott. About a transformation system: A theoretical study. In *Proceedings of the 3rd Symposium on Programming*, pages 232–267, 1978.

[66] E. Kounalis, A. Bouhoula, and M. Rusinowitch. SPIKE: An automatic theorem prover. In *Proceedings of LPAR'92, Lecture Notes in Artificial Intelligence*, volume 624. Springer-Verlag, 1992.

[67] Ina Kraan, David Basin, and Alan Bundy. Logic program synthesis via proof planning. *LOPSTAR-92*, pages 1–14, 1992.

[68] Ina Kraan, David Basin, and Alan Bundy. Middle-out reasoning for synthesis and induction. *Journal of Automated Reasoning*, 16(1-2):113–145, 1996.

[69] G. Kreisel. Mathematical logic. In T. L. Satty, editor, *Lectures in Modern Mathematics*, volume III, pages 95–195. John Wiley, New York, 1965.

[70] J.B. Kruskal. The theory of well-quasi-ordering: A frequently discovered concept. *Journal of Combinatorial Theory*, A(13):297–305, 1972.

[71] K. K. Lau and S. D. Prestwich. Top-down synthesis of recursive logic procedures from first-order logic specifications. In *Proceedings of the ICLP*, pages 667–684. Springer, 1990.

[72] P. Madden, Alan Bundy, and A. Smaill. Recursive program optimization through inductive synthesis proof transformation. *Journal of Automated Reasoning*, 22:65–115, 1999.

[73] Zohar Manna and Richard Waldinger. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2:90–121, 1980.

[74] Zohar Manna and Richard Waldinger. Fundamentals of deductive program synthesis, 1992. Manuscript, Computer Science Department, Stanford University.

[75] S. Marlow. *Deforestation for Higher-Order Functional Programs*. PhD thesis, University of Glasgow, September 1995.

[76] P. Martin-Lof. Haupstatz for the intuitionistic theory of iterated inductive definitions. In J. E. Fenstad, editor, *Proceeding of the Second Scandinavian Logic Symposium*, pages 179–216. North-Holland, 1971.

[77] P. Martin-Lof. Constructive mathematics and computer programming. In *6th International Congress for Logic, Methodology, and Philosophy of Science, 1979*, volume VI, pages 153–175. North-Holland, 1982.

[78] R. McDowell and D. Miller. Cut-elimination for a logic with definitions and induction. *Theoretical Computer Science*, 232:91–119, 2000.

[79] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Science*, 17:29–60, 1978.

[80] Augustus De Morgan. Induction (mathematics). In *Penny Cyclopaedia*, volume XII, pages 465–466. The Society for the Diffusion of Useful Knowledge, Charles Knight & Co., 1838.

[81] R. M. Nirenberg, D. V. Turchin, and V. F. Turchin. Experiments with a supercompiler. In *Proceedings of the 1982 ACM symposium on LISP and functional programming, Conference on LISP and Functional Programming, Pittsburgh, Pennsylvania, United States*, pages 47 – 55, 1982.

[82] Gordon D. Plotkin. A note on inductive generalization. *Machine Intelligence 5*, pages 153–163, 1970.

[83] Gordon D. Plotkin. A further note on inductive generalization. *Machine Intelligence 6*, pages 101–124, 1971.

[84] Uday S. Reddy. Term rewriting induction. In *Proceedings of the tenth international conference on Automated deduction*, pages 162–177. Springer-Verlag, New York, 1990.

[85] J. D. C. Richardson. *The Use of Proof Plans for Transformation of Functional Programs by Changes of Data Type.* PhD thesis, Department of Artificial Intelligence, University of Edinburgh, January 1996.

[86] J. D. C. Richardson. Proof planning and program synthesis: a survey. *American Association for Artificial Intelligence*, 2002.

[87] J. D. C. Richardson, A. Smaill, and I. Green. System description: proof-planning in higher-order logic with Lambda-Clam. In C. Kirchner and H. Kirchner, editors, *15th International Conference on Automated Deduction*, volume 1421 of *Lecture Notes in Artificial Intelligence*, pages 129–133. Lindau, Germany, 1998.

[88] D. Sands. Total correctness and improvement in the transformation of functional programs. Technical report, DIKU, University of Copenhagen, Denmark, 1994.

[89] D. Sands. Proving the correctness of recursion-based automatic program transformations. *Theoretical Computer Science*, 1-2(167):193–233, 1996.

[90] D. Sands. Total correctness by local improvement in the transformation of functional programs. *ACM Transactions on Programming Languages and Systems*, 18(2):175–234, March 1996.

[91] A. Smaill and I. Green. Automating the synthesis of functional programs, 1995. Research paper 777, Dept. of Artificial Intelligence, University of Edinburgh.

[92] A. Smaill and I. Green. Higher-order annotated terms for proof search. In *Proceedings of the International Conference on Theorem Proving in Higher order Logics*, pages 399–413, 1996.

[93] D. R. Smith. The synthesis of lisp programs from examples. *Automatic program construction techniques, chapter 15*, pages 307–324, 1984.

[94] M. H. Sorensen. *Turchin's Supercompiler Revisited.* 1994. Master's Thesis, University of Copenhagen, Denmark.

[95] Morten H. Sørensen and Robert Glück. An algorithm of generalization in positive supercompilation. In *Logic Programming: Proceedings of the 1995 International Symposium*, pages 465–479, 1995.

[96] Morten Heine Sørensen. Convergence of program transformers in the metric space of trees. *Lecture Notes in Computer Science*, 1422:315–337, 1998.

[97] A. Stevens. A rational reconstruction of Boyer and Moore's technique for constructing induction formulas. In *Proceedings of European Conference on Artificial Intelligence (ECAI-88)*, pages 565–570, 1988.

[98] Philip D. Summers. A methodology for lisp program construction from examples. *Journal of the ACM*, 24:161–175, 1977.

[99] Simon Thompson. *Type Theory and Functional Programming.* Addison-Wesley Publishing Company, 1991.

[100] A. Tiu. *A Logical Framework For Reasoning About Logical Specifications.* PhD thesis, Pane State University, 2004.

[101] V. F. Turchin. The concept of a supercompiler. *ACM Transaction on Programming Languages and Systems*, 8:292–325, 1986.

[102] Valentin F. Turchin. The use of metasystem transition in theorem proving and program optimization. In *Proceedings of the 7th Colloquium on Automata,*

Languages and Programming, volume 85 of *Lecture Notes in Computer Science*, pages 645 – 657. Springer-Verlag, 1980.

[103] Valentin F. Turchin. Metacomputation: Metasystem transitions plus supercompilation. In *Dagstuhl Seminar on Partial Evaluation*, pages 481 – 509, 1996.

[104] P. Wadler. Deforestation: Transforming programs to eliminate trees. In *European Symposium on Programming, ESOP '88*, volume 300 of *Lecture Notes in Computer Science*, pages 344–358. Nancy, France, 1988.

[105] P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.

[106] Philip Wadler. Proofs are programs: 19th century logic and 21st century computing. *Dr Dobbs Journal.* Special Supplement on Software in the 21st century, December 2000.

[107] T. Walsh. A divergence critic for inductive proof. *Journal of Artificial Intelligence Research*, 4:209–235, 1996.

[108] C. Walther. *Mechanising mathematical induction, Handbook of Logic in Artificial Intelligence and Logic programming.* Oxford University Press, Oxford, 1992.

[109] C. P. Wirth. Descente infinie + deduction. *Logic Journal of the IGPL*, 12(1):1–96, 2004. Oxford University Press.

[110] H. Zhang. *Reduction, superposition and induction: Automated reasoning in an equational logic.* PhD thesis, Rensselaer Polytechnic Institute, Schenectady, New York, 1988.

[111] H. Zhang, D. Kapur, and M. S. Krishnamoorthy. A mechanizable induction principle for equational specifications. In *Proceedings of the 9th International Conference on Automated Deduction*, Lecture Notes In Computer Science, volume 310, pages 162 – 181. Springer-Verlag, London, UK, 1988.

Appendix A

Distillation

A.1 Examples

A.1.1 Accumulating Patterns

In the following example, we demonstrate how distillation avoids the non-termination problem due to *accumulating patterns.*

Example 14

Consider the transformation of expression (14.1).

$$\textit{even}\ (\textit{plus}\ x\ x) \tag{14.1}$$

During this transformation, we encounter expression (14.2) which is a non-strict embedding of expression (14.1).

$$\textit{even}\ (\textit{plus}\ x''\ (\textit{Succ}\ (\textit{Succ}\ x''))) \tag{14.2}$$

Further transformation of expression (14.2) will cause non-termination because of successively larger expressions and folding cannot be performed. The most specific generalization of these two expressions is therefore performed to achieve termination of the distillation process, which results in the following triple:

$$(\textit{even}\ (\textit{plus}\ x\ v), \{v := x\}, \{v := \textit{Succ}\ (\textit{Succ}\ x'')\})$$

The generalized form of expression (14.1) is given by expression (14.3).

$$\textbf{let}\ v = x\ \textbf{in}\ \textit{even}\ (\textit{plus}\ x\ v) \tag{14.3}$$

The sub-tree rooted at expression (14.1) is replaced with the result of transforming expression (14.3). The transformation of expression (14.3) results in the partial process tree which is shown in Fig. A.1. We obtain expression (14.4) from the sub-tree rooted at *even* v'' within the partial process tree shown in Fig. A.1.

$$\begin{array}{l} \textbf{Node f2: case } v'' \textbf{ of} \\ \quad\quad\quad\quad \mathit{Zero} \quad\quad : \mathit{True} \\ \quad\quad\quad\quad \mid \mathit{Succ}\ v''' : \textbf{case } v''' \textbf{ of} \\ \quad\quad\quad\quad\quad\quad\quad\quad \mathit{Zero} \quad\quad : \mathit{False} \\ \quad\quad\quad\quad\quad\quad\quad\quad \mid \mathit{Succ}\ v'''' : \textbf{Repeat f2: } \mathit{even}\ v'''' \end{array} \tag{14.4}$$

Expression (14.4) is further transformed, which results in the partial process tree shown in Fig. A.2. We obtain expression (14.5) from the sub-tree rooted at *even* v'''' within the partial process tree shown in Fig. A.2.

$$\begin{array}{l} \textbf{Node f3: case } v'''' \textbf{ of} \\ \quad\quad\quad\quad \mathit{Zero} \quad\quad : \mathit{True} \\ \quad\quad\quad\quad \mid \mathit{Succ}\ v''''' : \textbf{case } v''''' \textbf{ of} \\ \quad\quad\quad\quad\quad\quad\quad\quad \mathit{Zero} \quad\quad : \mathit{False} \\ \quad\quad\quad\quad\quad\quad\quad\quad \mid \mathit{Succ}\ v'''''' : \textbf{Repeat f3: } \mathit{even}\ v'''''' \end{array} \tag{14.5}$$

Expression (14.5) is an instance of expression (14.4). A repeat node is therefore created at the occurrence of expression (14.5). This results in the partial process tree which is shown in Fig. A.3. The residual program given by expression (14.6) is constructed from the partial process tree shown in Fig. A.3.

$$\begin{array}{l} \textbf{letrec } f2 \ = \ \lambda v''.\textbf{case } v'' \textbf{ of} \\ \quad\quad\quad\quad\quad\quad\quad\quad \mathit{Zero} \quad\quad : \mathit{True} \\ \quad\quad\quad\quad\quad\quad\quad\quad \mid \mathit{Succ}\ v''' : \textbf{case } v''' \textbf{ of} \\ \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mathit{Zero} \quad\quad : \mathit{False} \\ \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mid \mathit{Succ}\ v'''' : \mathit{f2}\ v'''' \\ \textbf{in } \mathit{f2}\ v'' \end{array} \tag{14.6}$$

In a similar way, we obtain the residual program given by expression (14.7) from the sub-tree rooted at *even* v' within the partial process tree shown in Fig. A.1.

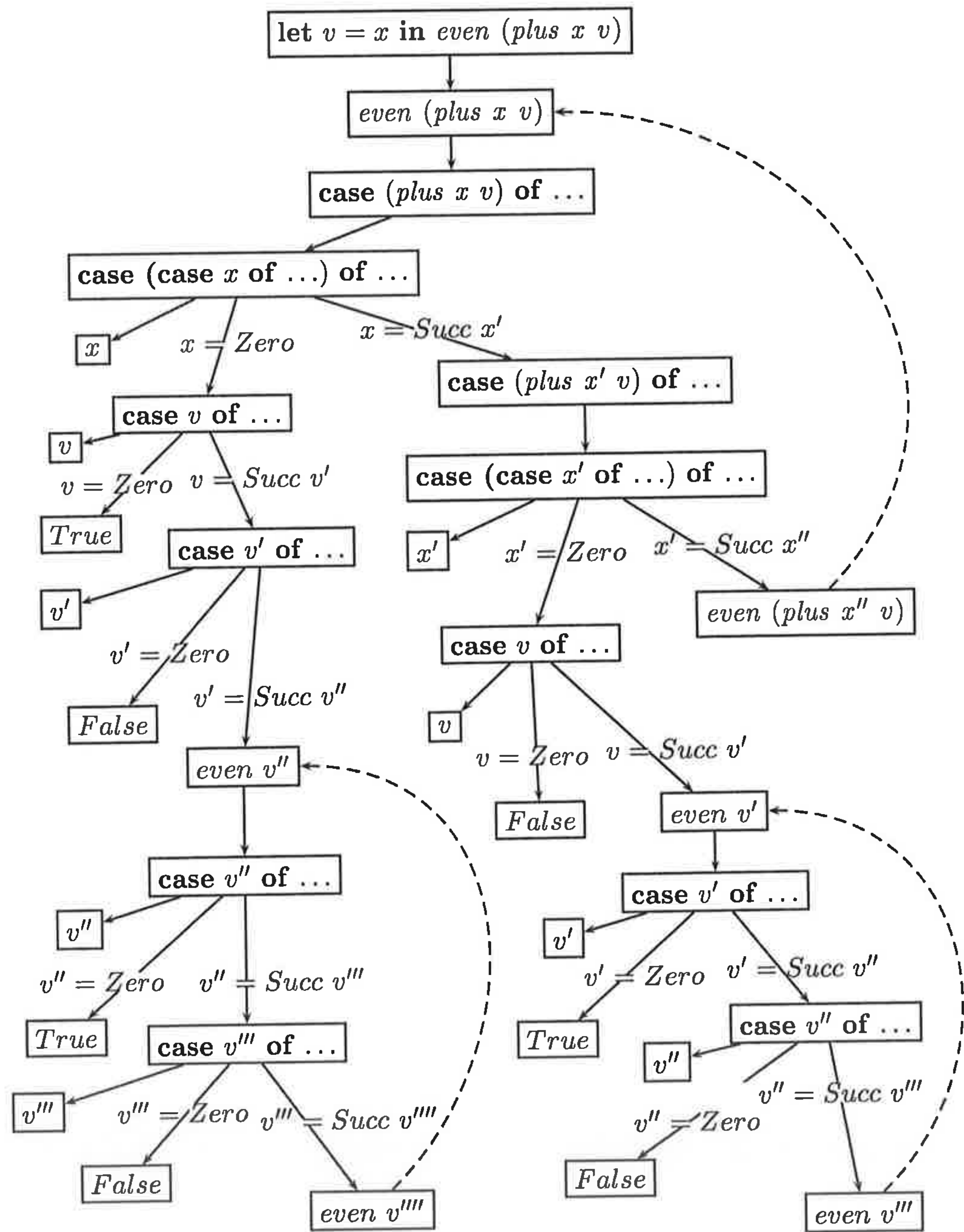

Figure A.1: Partial process tree (1) for $\mathcal{T}[\![even\ (plus\ x\ x)]\!]$

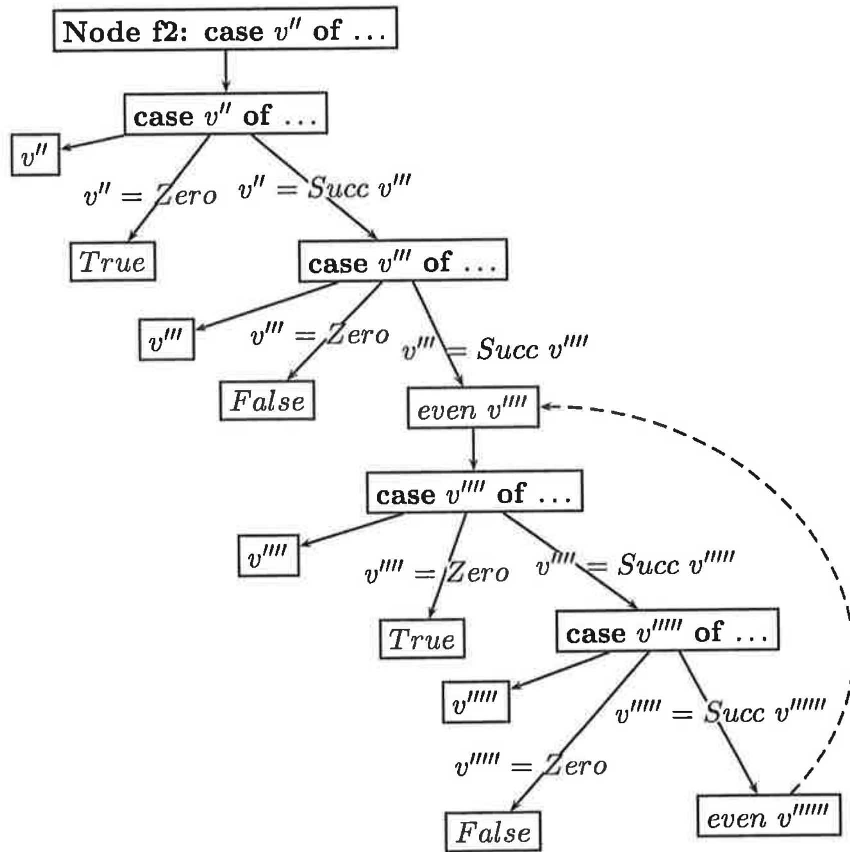

Figure A.2: Partial process tree (2) for $\mathcal{T}[\![even\ (plus\ x\ x)]\!]$

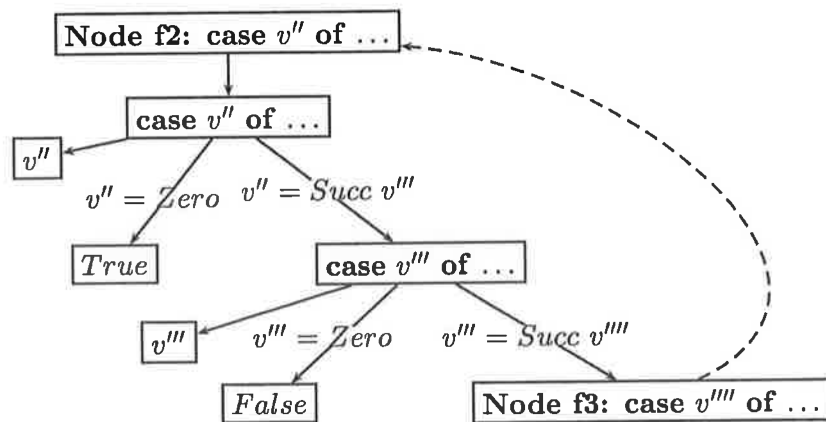

Figure A.3: Partial process tree (3) for $\mathcal{T}[\![even\ (plus\ x\ x)]\!]$

$$
\begin{aligned}
\textbf{letrec}\ f3\ &=\ \lambda v'.\textbf{case}\ v'\ \textbf{of} \\
&\qquad\qquad \mathit{Zero}\quad :\ \mathit{True} \\
&\qquad\qquad \mid \mathit{Succ}\ v'' :\ \textbf{case}\ v''\ \textbf{of} \\
&\qquad\qquad\qquad\qquad \mathit{Zero}\quad :\ \mathit{False} \\
&\qquad\qquad\qquad\qquad \mid \mathit{Succ}\ v''' :\ f3\ v''' \\
\textbf{in}\ f3\ v'
\end{aligned}
\tag{14.7}
$$

We obtain expression (14.8) from the sub-tree rooted at *even* (*plus* x v) within the partial process tree shown in Fig. A.1.

$$
\begin{aligned}
&\textbf{Node f0:}\ \textbf{case}\ x\ \textbf{of} \\
&\quad \mathit{Zero}\quad :\ \textbf{case}\ v\ \textbf{of} \\
&\qquad\quad \mathit{Zero}\quad :\ \mathit{True} \\
&\qquad\quad \mid \mathit{Succ}\ v' :\ \textbf{case}\ v'\ \textbf{of} \\
&\qquad\qquad\quad \mathit{Zero}\quad :\ \mathit{False} \\
&\qquad\qquad\quad \mid \mathit{Succ}\ v'' :\ \textbf{letrec} \\
&\qquad\qquad\qquad\quad f2\ =\ \lambda v''.\textbf{case}\ v''\ \textbf{of} \\
&\qquad\qquad\qquad\qquad\quad \mathit{Zero}\quad :\ \mathit{True} \\
&\qquad\qquad\qquad\qquad\quad \mid \mathit{Succ}\ v''' :\ \textbf{case}\ v'''\ \textbf{of} \\
&\qquad\qquad\qquad\qquad\qquad\quad \mathit{Zero}\quad :\ \mathit{False} \\
&\qquad\qquad\qquad\qquad\qquad\quad \mid \mathit{Succ}\ v'''' :\ f2\ v'''' \\
&\qquad\qquad\qquad\quad \textbf{in}\ f2\ v'' \\
&\quad \mid \mathit{Succ}\ x' :\ \textbf{case}\ x'\ \textbf{of} \\
&\qquad\quad \mathit{Zero}\quad :\ \textbf{case}\ v\ \textbf{of} \\
&\qquad\qquad\quad \mathit{Zero}\quad :\ \mathit{False} \\
&\qquad\qquad\quad \mid \mathit{Succ}\ v' :\ \textbf{letrec} \\
&\qquad\qquad\qquad\quad f3\ =\ \lambda v'.\textbf{case}\ v'\ \textbf{of} \\
&\qquad\qquad\qquad\qquad\quad \mathit{Zero}\quad :\ \mathit{True} \\
&\qquad\qquad\qquad\qquad\quad \mid \mathit{Succ}\ v'' :\ \textbf{case}\ v''\ \textbf{of} \\
&\qquad\qquad\qquad\qquad\qquad\quad \mathit{Zero}\quad :\ \mathit{False} \\
&\qquad\qquad\qquad\qquad\qquad\quad \mid \mathit{Succ}\ v''' :\ f3\ v''' \\
&\qquad\qquad\qquad\quad \textbf{in}\ f3\ v' \\
&\qquad\quad \mid \mathit{Succ}\ x'' :\ \textbf{Repeat f0:}\ \mathit{even}\ (\mathit{plus}\ x''\ v)
\end{aligned}
\tag{14.8}
$$

The transformation of expression (14.8) proceeds as shown in the partial process tree of Fig. A.4.

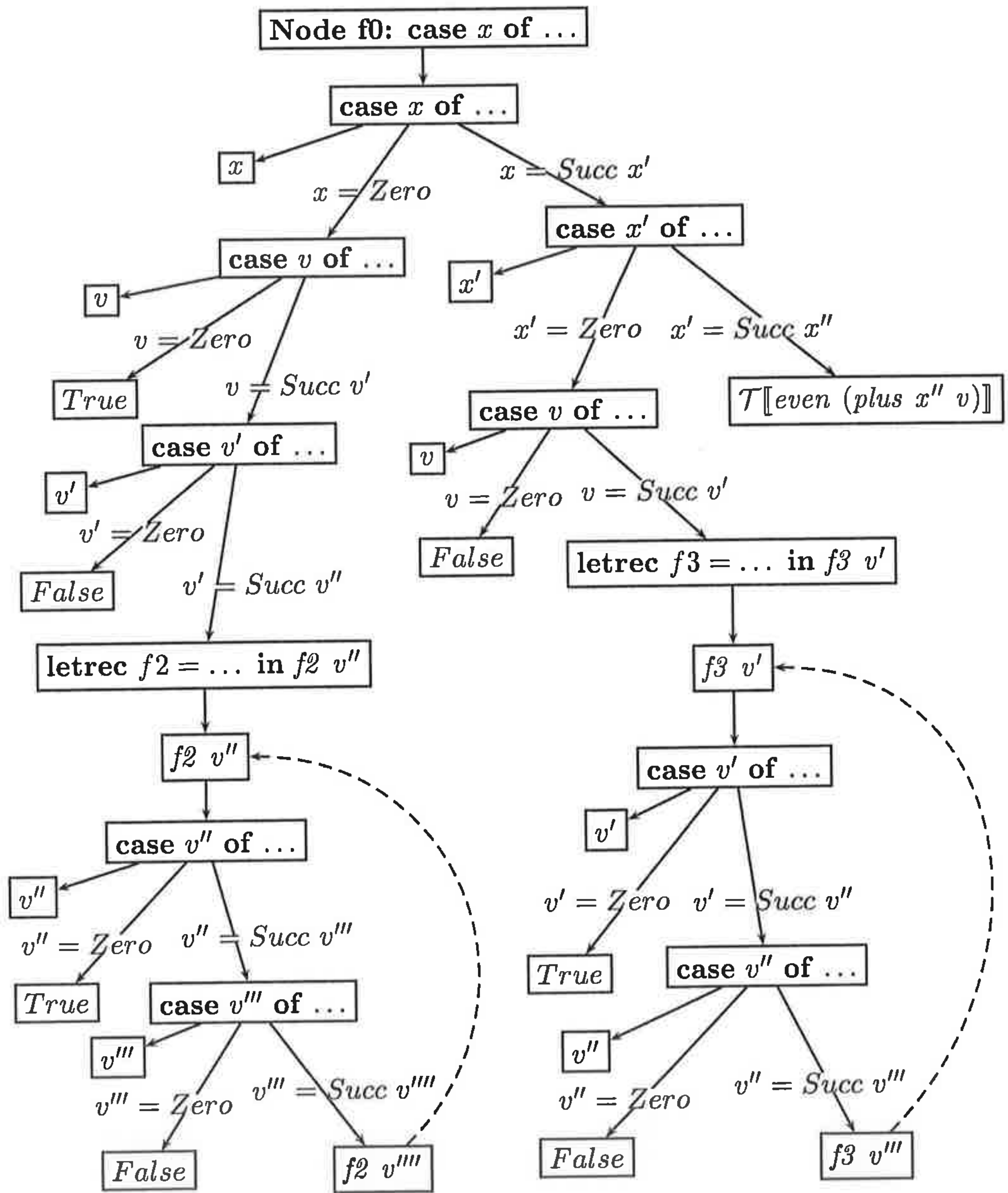

Figure A.4: Partial process tree (4) for $\mathcal{T}[\![even\ (plus\ x\ x)]\!]$

We construct residual programs given by the expressions (14.9) and (14.10) from the sub-trees rooted at $f2\ v''$ and $f3\ v'$ respectively within the partial process tree shown in Fig. A.4.

$$\begin{array}{lll} \textbf{letrec}\ f1 & = & \lambda v''.\textbf{case}\ v''\ \textbf{of} \\ & & \quad Zero \quad : True \\ & & \quad \mid Succ\ v''' : \textbf{case}\ v'''\ \textbf{of} \\ & & \qquad\qquad Zero \quad : False \\ & & \qquad\qquad \mid Succ\ v'''' : f1\ v'''' \\ \textbf{in}\ f1\ v'' & & \end{array} \tag{14.9}$$

$$\begin{array}{lll} \textbf{letrec}\ f1 & = & \lambda v'.\textbf{case}\ v'\ \textbf{of} \\ & & \quad Zero \quad : True \\ & & \quad \mid Succ\ v'' : \textbf{case}\ v''\ \textbf{of} \\ & & \qquad\qquad Zero \quad : False \\ & & \qquad\qquad \mid Succ\ v''' : f1\ v''' \\ \textbf{in}\ f1\ v' & & \end{array} \tag{14.10}$$

The transformation of expression (14.11) within the partial process tree shown in Fig. A.4 is performed in a similar way to that of the expression $even\ (plus\ x\ v)$ in the partial process tree shown in Fig. A.1.

$$even\ (plus\ x''\ v) \tag{14.11}$$

During this transformation, expression (14.12) is encountered which is an instance of expression (14.11). A repeat node is therefore created at the occurrence of expression (14.12).

$$even\ (plus\ x''''\ v) \tag{14.12}$$

The transformation of expression (14.11) therefore results in expression (14.13).

Node f1:

```
case x'' of
   Zero      : case v of
                   Zero     : True
                 | Succ v'  : case v' of
                                  Zero     : False
                                | Succ v'' : letrec
                                               f3 = λv''.case v'' of
                                                             Zero      : True
                                                           | Succ v''' : case v''' of
                                                                             Zero       : False
                                                                           | Succ v'''' : f3 v''''
                                             in f3 v''
 | Succ x''' : case x''' of
                   Zero       : case v of
                                    Zero    : False
                                  | Succ v' : letrec
                                                f4 = λv'.case v' of
                                                              Zero     : True
                                                            | Succ v'' : case v'' of
                                                                             Zero      : False
                                                                           | Succ v''' : f4 v'''
                                              in f4 v'
                 | Succ x'''' : Repeat f1: even (plus x'''' v)
```

(14.13)

Expression (14.13) is an instance of expression (14.8). A repeat node is therefore created at the occurrence of expression (14.13). This results in the partial process tree shown in Fig. A.5. We construct the residual program given by expression (14.14) from the partial process tree shown in Fig. A.5.

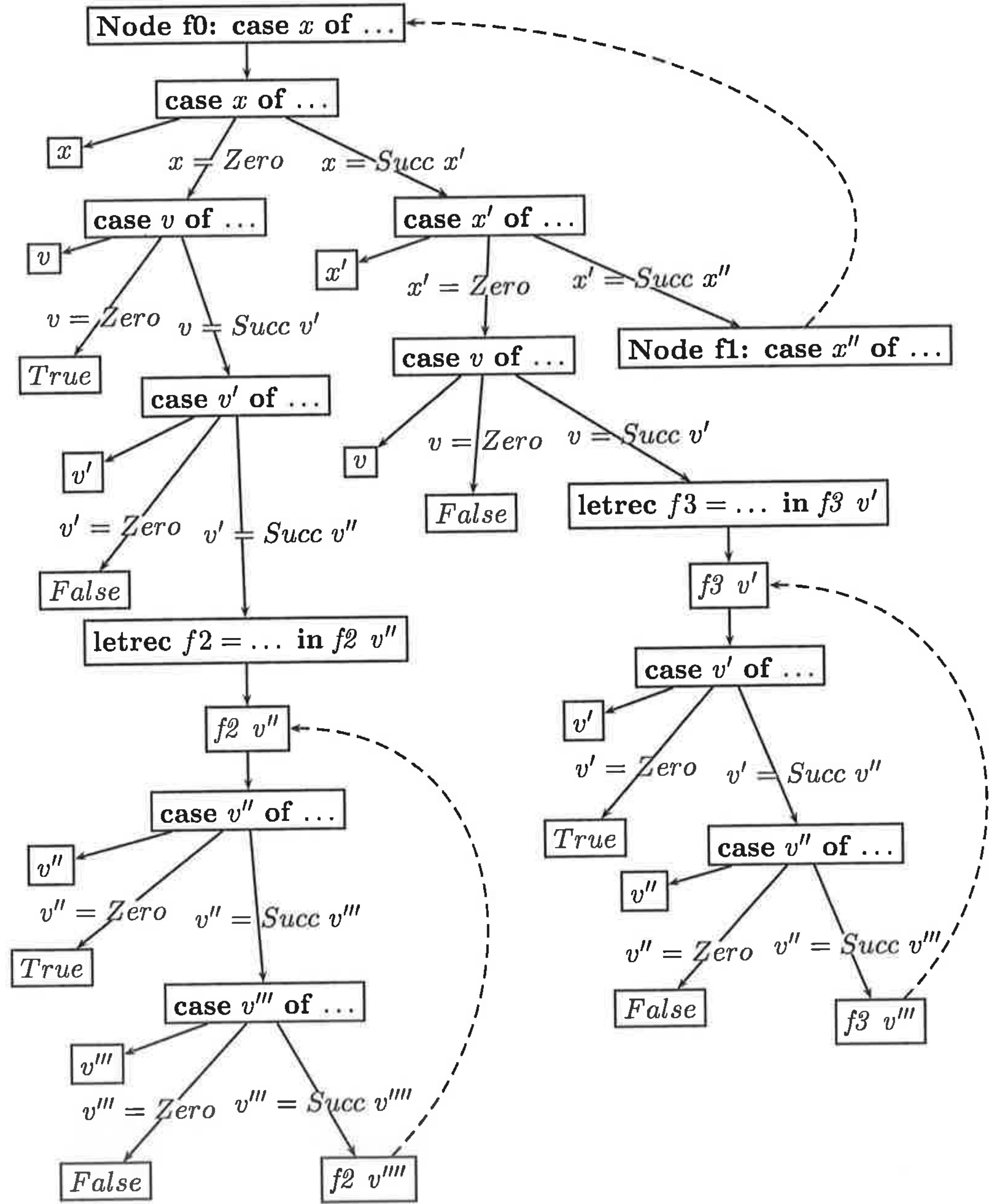

Figure A.5: Partial process tree (5) for $\mathcal{T}[\![even\ (plus\ x\ x)]\!]$

```
letrec
f0 = λx. case x of
           Zero    : case v of
                       Zero    : True
                     | Succ v' : case v' of
                                   Zero     : False
                                 | Succ v'' : letrec
                                                f1 = λv''. case v'' of
                                                             Zero      : True
                                                           | Succ v''' : case v''' of
                                                                           Zero       : False
                                                                         | Succ v'''' : f1 v''''
                                              in f1 v''
         | Succ x' : case x' of
                       Zero     : case v of
                                    Zero    : False
                                  | Succ v' : letrec
                                                f1 = λv'. case v' of
                                                            Zero     : True
                                                          | Succ v'' : case v'' of
                                                                         Zero      : False
                                                                       | Succ v''' : f1 v'''
                                              in f1 v'
                     | Succ x'' : f0 x''
in f0 x
```

(14.14)

By substituting back the extracted variable x for the variable v within expression (14.14), we obtain the residual program given by expression (14.15) from the transformation of expression (14.3).

```
letrec
f0 = λx'. case x' of
          Zero    : case x of
                      Zero    : True
                    | Succ v' : case v' of
                                  Zero     : False
                                | Succ v'' : letrec
                                               f1 = λv''. case v'' of
                                                            Zero      : True
                                                          | Succ v''' : case v''' of
                                                                          Zero       : False
                                                                        | Succ v'''' : f1 v''''
                                             in f1 v''
        | Succ x' : case x' of
                      Zero     : case x of
                                   Zero    : False
                                 | Succ v' : letrec
                                               f1 = λv'. case v' of
                                                           Zero     : True
                                                         | Succ v'' : case v'' of
                                                                        Zero      : False
                                                                      | Succ v''' : f1 v'''
                                             in f1 v'
                    | Succ x'' : f0 x''
in f0 x
```

(14.15)

The partial process tree shown in Fig. A.6 is obtained by transforming expression (14.15). We construct the residual program shown in Fig. A.7 from the partial process tree of Fig. A.6.

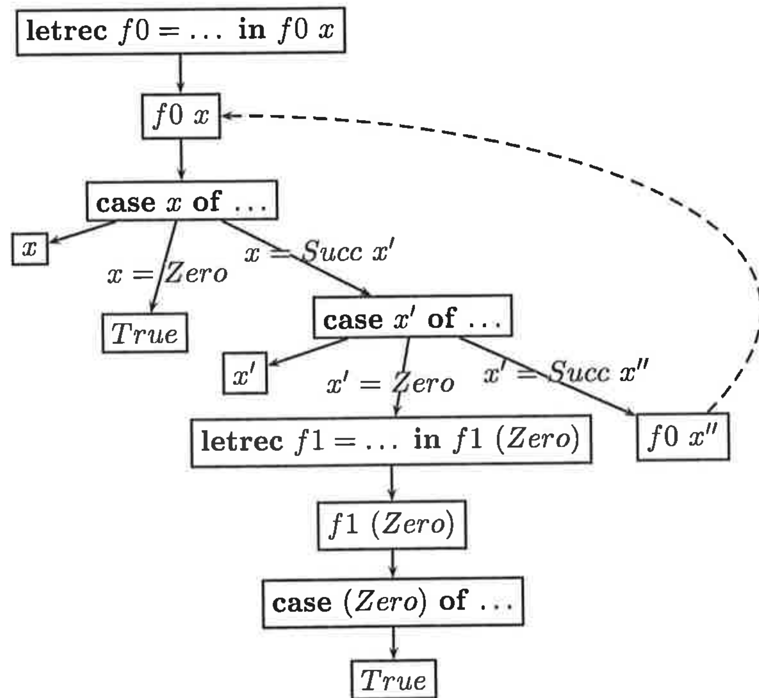

Figure A.6: Partial process tree (6) for $\mathcal{T}[\![even\ (plus\ x\ x)]\!]$

$$
\begin{array}{lcl}
\textbf{letrec } f0 & = & \lambda x.\textbf{case } x \textbf{ of} \\
 & & \quad Zero \quad : True \\
 & & \quad \mid Succ\ x' : \textbf{case } x' \textbf{ of} \\
 & & \qquad\qquad Zero \quad : True \\
 & & \qquad\qquad \mid Succ\ x'' : f0\ x'' \\
\textbf{in } f0\ x & &
\end{array}
$$

Figure A.7: Residual program for $\mathcal{T}[\![even\ (plus\ x\ x)]\!]$

A.1.2 Accumulating Parameters

In the following example, we demonstrate how distillation avoids the non-termination problem due to *accumulating parameters.*

Example 15

Consider the transformation of expression (15.1) using the accumulating version of the double function given by *doublea* as shown in Fig. 3.2. This example is adopted from [45].

$$even\ (doublea\ x\ Zero) \tag{15.1}$$

We obtain expression (15.2) by unfolding the function *even* within the input expression (15.1).

$$\begin{aligned}
&\textbf{case}\ (doublea\ x\ Zero)\ \textbf{of} \\
&\quad Zero \quad : True \\
&\quad |\ Succ\ x' : \textbf{case}\ x'\ \textbf{of} \\
&\qquad\qquad\qquad Zero \quad : False \\
&\qquad\qquad\qquad |\ Succ\ x'' : even\ x''
\end{aligned} \tag{15.2}$$

After a few further steps, we encounter expression (15.3) which is a non-strict homeomorphic embedding of expression (15.2).

$$\begin{aligned}
&\textbf{case}\ (doublea\ x'\ (Succ\ (Succ\ Zero)))\ \textbf{of} \\
&\quad Zero \quad : True \\
&\quad |\ Succ\ x' : \textbf{case}\ x'\ \textbf{of} \\
&\qquad\qquad\qquad Zero \quad : False \\
&\qquad\qquad\qquad |\ Succ\ x'' : even\ x''
\end{aligned} \tag{15.3}$$

The most specific generalization of the expressions (15.2) and (15.3) is therefore performed. The generalized form of expression (15.2) is given by the following expression (15.4).

$$\begin{aligned}
&\textbf{let}\ v = Zero \\
&\textbf{in}\ \textbf{case}\ (doublea\ x\ v)\ \textbf{of} \\
&\qquad Zero \quad : True \\
&\qquad |\ Succ\ x' : \textbf{case}\ x'\ \textbf{of} \\
&\qquad\qquad\qquad\quad Zero \quad : False \\
&\qquad\qquad\qquad\quad |\ Succ\ x'' : even\ x''
\end{aligned} \tag{15.4}$$

The remaining generalized expression is given by expression (15.5).

$$\begin{aligned}
&\textbf{case}\ (doublea\ x\ v)\ \textbf{of} \\
&\quad Zero \quad : True \\
&\quad |\ Succ\ x' : \textbf{case}\ x'\ \textbf{of} \\
&\qquad\qquad\qquad Zero \quad : False \\
&\qquad\qquad\qquad |\ Succ\ x'' : even\ x''
\end{aligned} \tag{15.5}$$

The sub-tree rooted at expression (15.2) is replaced with the result of transforming expression (15.4), which results in the partial process tree shown in Fig. A.8.

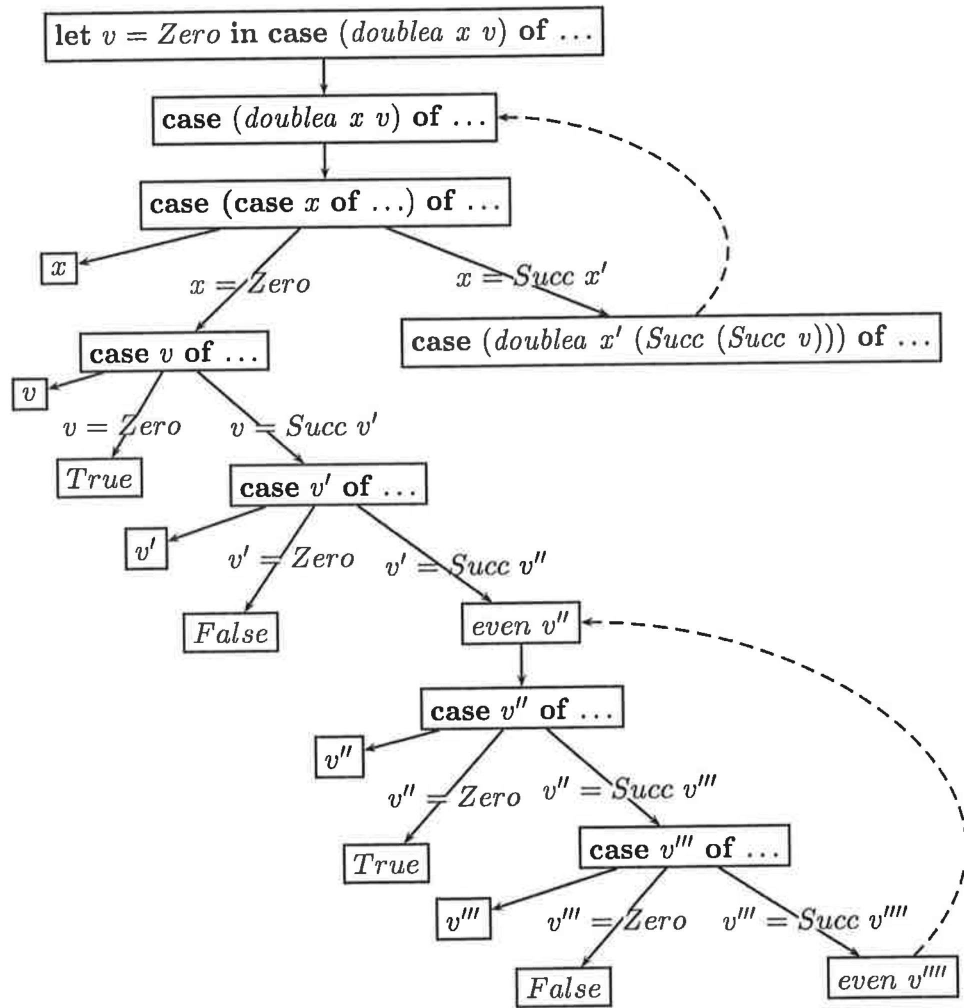

Figure A.8: Partial process tree (1) for $\mathcal{T}[\![even\ (doublea\ x\ Zero)]\!]$

Within this partial process tree, the expression *even* v'''' is encountered which is an instance of the expression *even* v''. A repeat node is therefore created at the occurrence of expression *even* v''''. We obtain expression (15.6) from the sub-tree rooted at *even* v'' within the partial process tree shown in Fig. A.8.

$$
\begin{array}{l}
\textbf{Node f2: case } v'' \textbf{ of} \\
\qquad Zero \quad : True \\
\qquad |\ Succ\ v''' : \textbf{case } v''' \textbf{ of} \\
\qquad\qquad Zero \quad : False \\
\qquad\qquad |\ Succ\ v'''' : \textbf{Repeat f2: } even\ v''''
\end{array}
\tag{15.6}
$$

Expression (15.6) is further transformed. The details of this transformation are shown in the previous example. During the transformation of expression (15.6), we

encounter expression (15.7).

$$\textbf{Node f3: case } v'''' \textbf{ of} \qquad (15.7)$$
$$\begin{array}{ll} Zero & : True \\ \mid Succ\ v''''' & : \textbf{case } v''''' \textbf{ of} \\ & \quad \begin{array}{ll} Zero & : False \\ \mid Succ\ v'''''' & : \textbf{Repeat f3: } even\ v'''''' \end{array} \end{array}$$

Expression (15.7) is an instance of expression (15.6). So, a repeat node is created at the occurrence of expression (15.7). This results in the partial process tree shown in Fig. A.9.

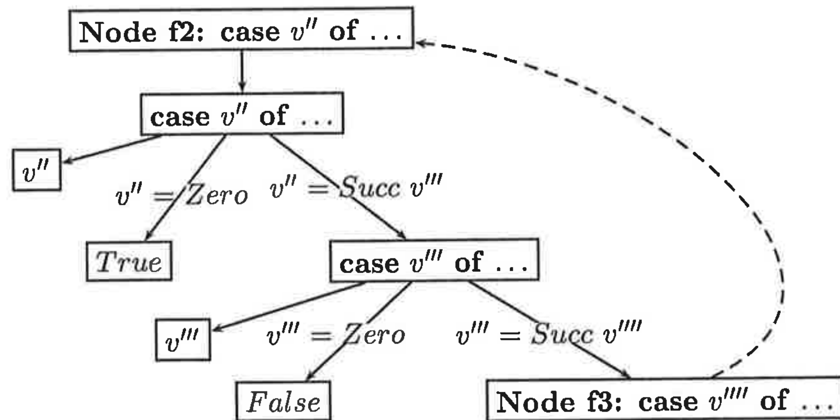

Figure A.9: Partial process tree (2) for $\mathcal{T}[\![even\ (doublea\ x\ Zero)]\!]$

The residual program given by expression (15.8) is constructed from the partial process tree shown in Fig. A.9.

$$\begin{array}{lll} \textbf{letrec } f2 & = & \lambda v''.\textbf{case } v'' \textbf{ of} \\ & & \quad \begin{array}{ll} Zero & : True \\ \mid Succ\ v''' & : \textbf{case } v''' \textbf{ of} \\ & \quad \begin{array}{ll} Zero & : False \\ \mid Succ\ v'''' & : f2\ v'''' \end{array} \end{array} \\ \textbf{in } f2\ v'' & & \end{array} \qquad (15.8)$$

Within the partial process tree shown in Fig. A.8, expression (15.9) is encountered, which is an instance of expression (15.5). A repeat node is therefore created at the occurrence of expression (15.9).

$$\begin{array}{l} \textbf{case}\ (\mathit{doublea}\ x'\ (\mathit{Succ}\ (\mathit{Succ}\ v)))\ \textbf{of} \\ \quad \mathit{Zero} \quad : \mathit{True} \\ \mid \mathit{Succ}\ x' : \textbf{case}\ x'\ \textbf{of} \\ \qquad\qquad \mathit{Zero} \quad : \mathit{False} \\ \qquad\qquad \mid \mathit{Succ}\ x'' : \mathit{even}\ x'' \end{array} \tag{15.9}$$

We obtain expression (15.10) from the partial process trees shown in Fig. A.8 and A.9.

$$\begin{array}{l} \textbf{Node f1: case}\ x\ \textbf{of} \\ \qquad \mathit{Zero} \quad : \textbf{case}\ v\ \textbf{of} \\ \qquad\qquad\qquad \mathit{Zero} \quad : \mathit{True} \\ \qquad\qquad\qquad \mid \mathit{Succ}\ v' : \textbf{case}\ v'\ \textbf{of} \\ \qquad\qquad\qquad\qquad\qquad \mathit{Zero} \quad : \mathit{False} \\ \qquad\qquad\qquad\qquad\qquad \mid \mathit{Succ}\ v'' : \ \textbf{letrec} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad f2 = \lambda v''.\textbf{case}\ v''\ \textbf{of} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{Zero} \quad : \mathit{True} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathit{Succ}\ v''' : \textbf{case}\ v'''\ \textbf{of} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{Zero} \quad : \mathit{False} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathit{Succ}\ v'''' : f2\ v'''' \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{in}\ f2\ v'' \\ \qquad \mid \mathit{Succ}\ x' : \textbf{Repeat f1:}\ \textbf{case}\ (\mathit{doublea}\ x'\ (\mathit{Succ}\ (\mathit{Succ}\ v)))\ \textbf{of} \\ \qquad\qquad\qquad\qquad\qquad \mathit{Zero} \quad : \mathit{True} \\ \qquad\qquad\qquad\qquad\qquad \mid \mathit{Succ}\ x' : \textbf{case}\ x'\ \textbf{of} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{Zero} \quad : \mathit{False} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathit{Succ}\ x'' : \mathit{even}\ x'' \end{array} \tag{15.10}$$

Expression (15.10) is further transformed. This results in the partial process tree shown in Fig. A.10. Within this partial process tree (Fig. A.10), expression (15.11) is encountered which is an instance of the expression (15.9). A repeat node is therefore created at the occurrence of expression (15.11).

$$\begin{array}{l} \textbf{case}\ (\mathit{doublea}\ x''\ (\mathit{Succ}\ (\mathit{Succ}\ (\mathit{Succ}\ (\mathit{Succ}\ v)))))\ \textbf{of} \\ \quad \mathit{Zero} \quad : \mathit{True} \\ \mid \mathit{Succ}\ x' : \textbf{case}\ x'\ \textbf{of} \\ \qquad\qquad \mathit{Zero} \quad : \mathit{False} \\ \qquad\qquad \mid \mathit{Succ}\ x'' : \mathit{even}\ x'' \end{array} \tag{15.11}$$

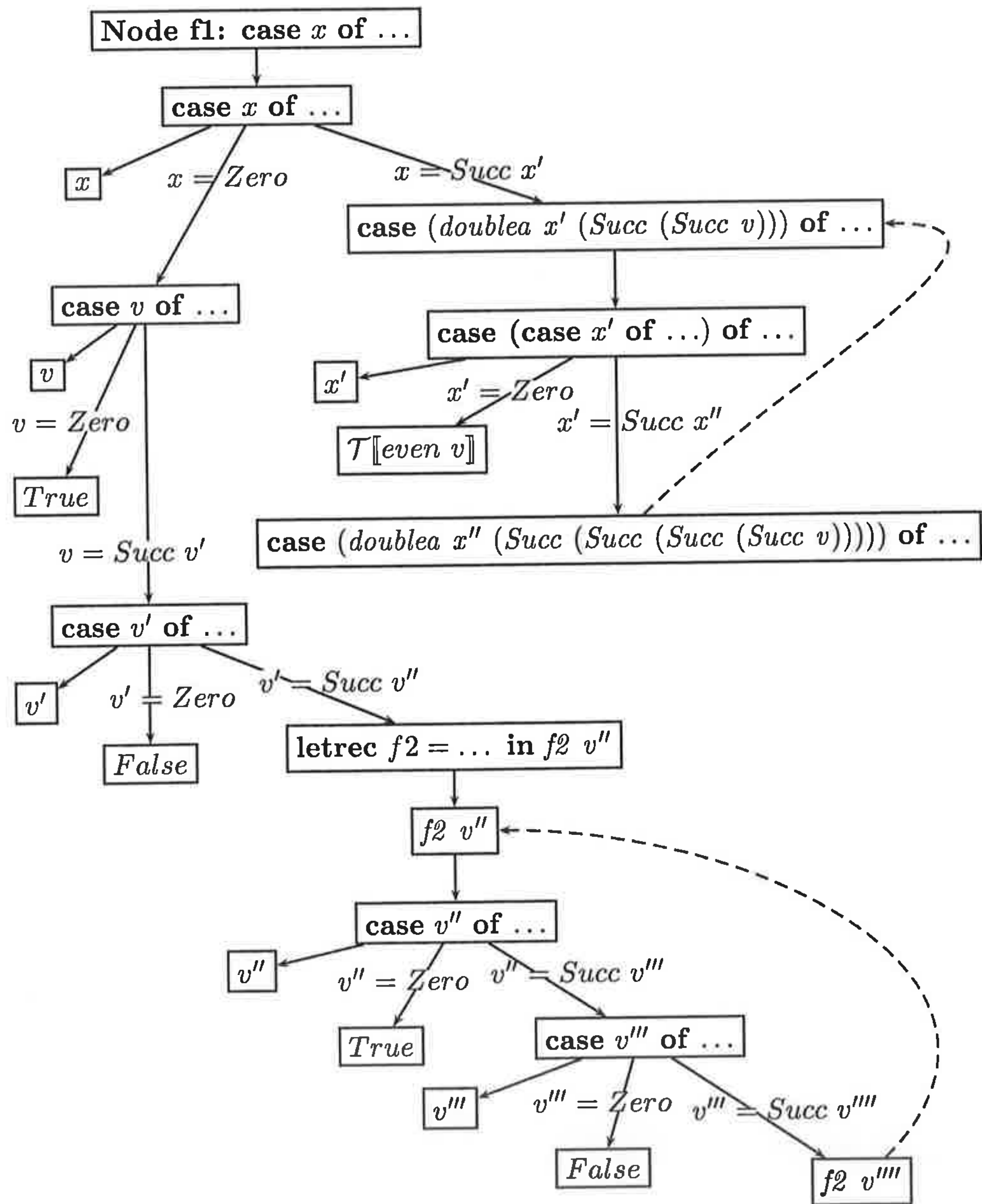

Figure A.10: Partial process tree (3) for $\mathcal{T}[\![even\ (doublea\ x\ Zero)]\!]$

Expression (15.12) is therefore constructed from the sub-tree rooted at expression (15.9) within the partial process tree shown in Fig. A.10.

$$
\begin{array}{l}
\textbf{Node f2: case } x' \textbf{ of} \\
\quad \textit{Zero} \quad : \textbf{letrec } f3 = \lambda v. \textbf{case } v \textbf{ of} \\
\qquad\qquad \textit{Zero} \quad : \textit{True} \\
\qquad\qquad \mid \textit{Succ } v' : \textbf{case } v' \textbf{ of} \\
\qquad\qquad\qquad \textit{Zero} \quad : \textit{False} \\
\qquad\qquad\qquad \mid \textit{Succ } v'' : \textit{f3 } v'' \\
\qquad \textbf{in } \textit{f3 } v \\
\quad \mid \textit{Succ } x'' : \textbf{Repeat f2: case } (\textit{doublea } x'' \ (\textit{Succ } (\textit{Succ } (\textit{Succ } (\textit{Succ } v))))) \textbf{ of} \\
\qquad\qquad \textit{Zero} \quad : \textit{True} \\
\qquad\qquad \mid \textit{Succ } x' : \textbf{case } x' \textbf{ of} \\
\qquad\qquad\qquad \textit{Zero} \quad : \textit{False} \\
\qquad\qquad\qquad \mid \textit{Succ } x'' : \textit{even } x''
\end{array}
\tag{15.12}
$$

Expression (15.12) is further transformed. During this transformation, the partial process tree as shown in Fig. A.11 is constructed. We obtain expression (15.13) from the sub-tree rooted at expression (15.11) within the partial process tree shown in Fig. A.11.

$$
\begin{array}{l}
\textbf{Node f3:} \\
\textbf{case } x'' \textbf{ of} \\
\quad \textit{Zero} \quad : \textbf{letrec } f5 = \lambda v. \textbf{case } v \textbf{ of} \\
\qquad\qquad \textit{Zero} \quad : \textit{True} \\
\qquad\qquad \mid \textit{Succ } v' : \textbf{case } v' \textbf{ of} \\
\qquad\qquad\qquad \textit{Zero} \quad : \textit{False} \\
\qquad\qquad\qquad \mid \textit{Succ } v'' : \textit{f5 } v'' \\
\qquad \textbf{in } \textit{f5 } v \\
\quad \mid \textit{Succ } x''' : \textbf{Repeat f3: case } (\textit{doublea } x''' \ (\textit{Succ } (\textit{Succ } (\textit{Succ } (\textit{Succ } (\textit{Succ } (\textit{Succ } v))))))) \textbf{ of} \\
\qquad\qquad \textit{Zero} \quad : \textit{True} \\
\qquad\qquad \mid \textit{Succ } x' : \textbf{case } x' \textbf{ of} \\
\qquad\qquad\qquad \textit{Zero} \quad : \textit{False} \\
\qquad\qquad\qquad \mid \textit{Succ } x'' : \textit{even } x''
\end{array}
\tag{15.13}
$$

The labels **f2** and **f3** of the **Node** expressions (15.6) and (15.7) are reused in defining the labels of the **Node** expressions (15.12) and (15.13). The transformation of expression (15.5) results in expression (15.10). The partial process tree resulting from the transformation of expression (15.10) is used to replace the subtree rooted

at expression (15.5) in Fig. A.8. So, the reuse of labels does not cause any inconsistency as the transformation of expression (15.5) and expression (15.10) involves two separate passes. The function node at the root is labelled with a smaller natural number preceded with the character **f**, whereas the labels in the subtrees are defined with increasing values in each pass.

Expression (15.13) is an instance of expression (15.12). A repeat node is therefore created at the occurrence of expression (15.13). This results in the partial process tree shown in Fig. A.12.

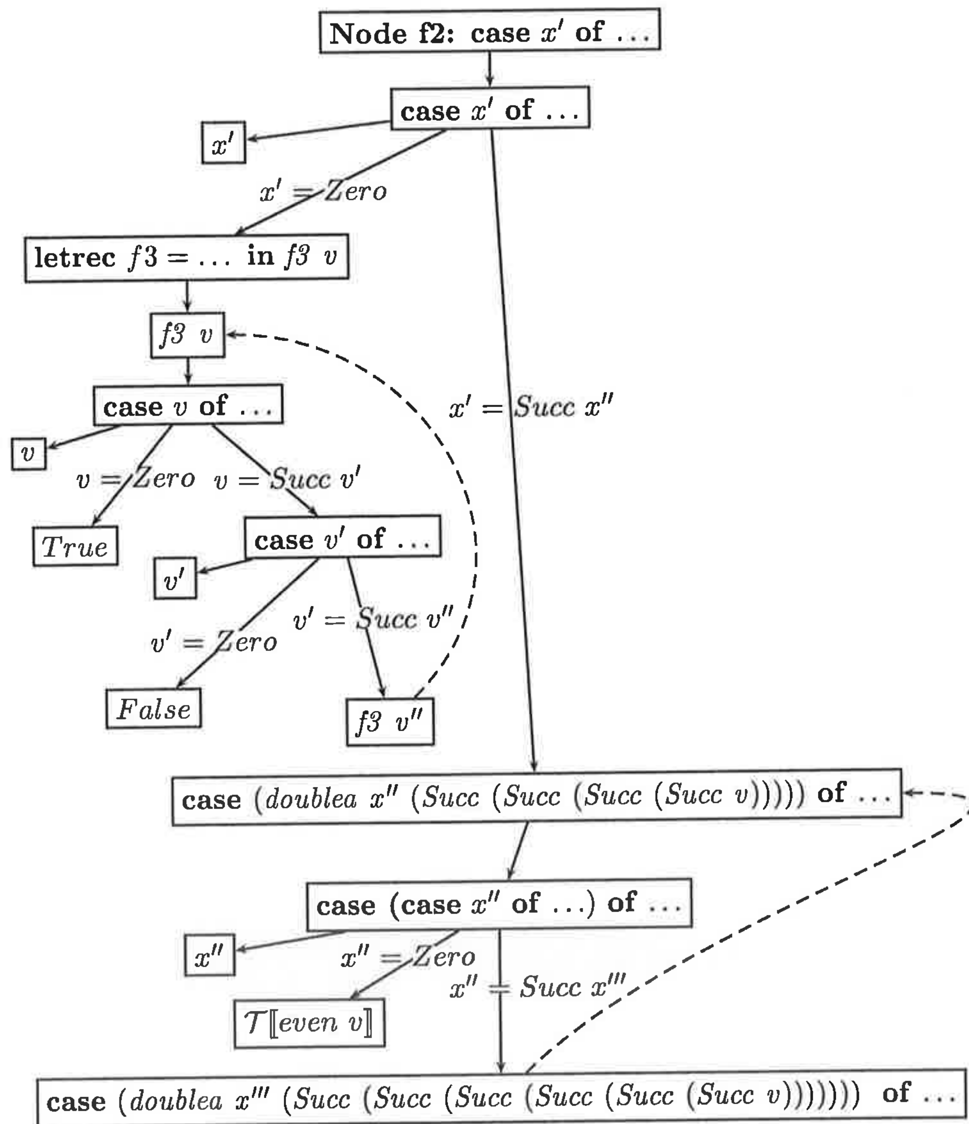

Figure A.11: Partial process tree (4) for $\mathcal{T}[\![even\ (doublea\ x\ Zero)]\!]$

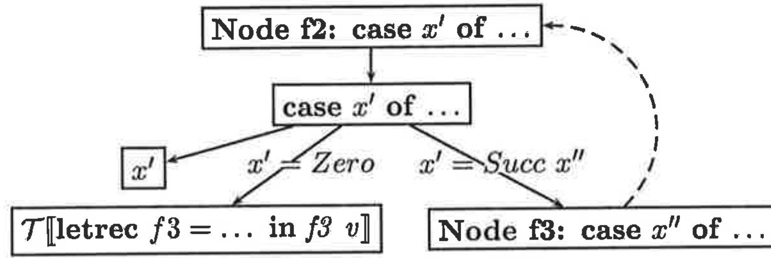

Figure A.12: Partial process tree (5) for $\mathcal{T}[\![even\ (doublea\ x\ Zero)]\!]$

We construct expression (15.14) from the partial process tree shown in Fig. A.12.

$$\begin{array}{l}
\textbf{letrec}\ f2\ =\ \lambda x'.\textbf{case}\ x'\ \textbf{of} \\
\qquad\qquad Zero\ :\ \textbf{letrec}\ f3\ =\ \lambda v.\textbf{case}\ v\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad Zero\ :\ True \\
\qquad\qquad\qquad\qquad\qquad \mid Succ\ v' : \textbf{case}\ v'\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad Zero\ :\ False \\
\qquad\qquad\qquad\qquad\qquad\qquad \mid Succ\ v'' : f3\ v'' \\
\qquad\qquad\qquad \textbf{in}\ f3\ v \\
\qquad\qquad \mid Succ\ x'' : f2\ x'' \\
\textbf{in}\ f2\ x'
\end{array} \tag{15.14}$$

From all of the above transformation of the original expression, we obtain the following expression (15.15).

$$
\begin{array}{l}
\textbf{case } x \textbf{ of} \\
\quad \mathit{Zero} \quad : \textbf{case } v \textbf{ of} \\
\qquad\qquad \mathit{Zero} \quad : \mathit{True} \\
\qquad\qquad \mid \mathit{Succ}\ v' : \textbf{case } v' \textbf{ of} \\
\qquad\qquad\qquad \mathit{Zero} \quad : \mathit{False} \\
\qquad\qquad\qquad \mid \mathit{Succ}\ v'' : \textbf{letrec } f2 = \lambda v''.\textbf{case } v'' \textbf{ of} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{Zero} \quad : \mathit{True} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathit{Succ}\ v''' : \textbf{case } v''' \textbf{ of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{Zero} \quad : \mathit{False} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathit{Succ}\ v'''' : f2\ v'''' \\
\qquad\qquad\qquad\qquad \textbf{in } f2\ v'' \\
\quad \mid \mathit{Succ}\ x' : \textbf{letrec } f2 = \lambda x'.\textbf{case } x' \textbf{ of} \\
\qquad\qquad\qquad \mathit{Zero} \quad : \textbf{letrec } f3 = \lambda v.\textbf{case } v \textbf{ of} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{Zero} \quad : \mathit{True} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathit{Succ}\ v' : \textbf{case } v' \textbf{ of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{Zero} \quad : \mathit{False} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathit{Succ}\ v'' : f3\ v'' \\
\qquad\qquad\qquad\qquad \textbf{in } f3\ v \\
\qquad\qquad\qquad \mid \mathit{Succ}\ x'' : f2\ x'' \\
\qquad\qquad \textbf{in } f2\ x'
\end{array}
\tag{15.15}
$$

The extracted sub-expression *Zero* is substituted back in to expression (15.15) for the variable v, which results in expression (15.16).

$$
\begin{array}{l}
\textbf{case } x \textbf{ of} \\
\quad \mathit{Zero} \quad : \textbf{case } \mathit{Zero} \textbf{ of} \\
\qquad\qquad \mathit{Zero} \quad : \mathit{True} \\
\qquad\qquad \mid \mathit{Succ}\ v' : \textbf{case } v' \textbf{ of} \\
\qquad\qquad\qquad \mathit{Zero} \quad : \mathit{False} \\
\qquad\qquad\qquad \mid \mathit{Succ}\ v'' : \textbf{letrec } f2 = \lambda v''. \textbf{case } v'' \textbf{ of} \\
\qquad\qquad\qquad\qquad\qquad \mathit{Zero} \quad : \mathit{True} \\
\qquad\qquad\qquad\qquad\qquad \mid \mathit{Succ}\ v''' : \textbf{case } v''' \textbf{ of} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{Zero} \quad : \mathit{False} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathit{Succ}\ v'''' : f2\ v'''' \\
\qquad\qquad\qquad\qquad \textbf{in } f2\ v'' \\
\mid \mathit{Succ}\ x' : \textbf{letrec } f2 = \lambda x'. \textbf{case } x' \textbf{ of} \\
\qquad\qquad\qquad \mathit{Zero} \quad : \textbf{letrec } f3 = \lambda v. \textbf{case } v \textbf{ of} \\
\qquad\qquad\qquad\qquad\qquad \mathit{Zero} \quad : \mathit{True} \\
\qquad\qquad\qquad\qquad\qquad \mid \mathit{Succ}\ v' : \textbf{case } v' \textbf{ of} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{Zero} \quad : \mathit{False} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathit{Succ}\ v'' : f3\ v'' \\
\qquad\qquad\qquad\qquad \textbf{in } f3\ \mathit{Zero} \\
\qquad\qquad\qquad \mid \mathit{Succ}\ x'' : f2\ x'' \\
\qquad\quad \textbf{in } f2\ x'
\end{array}
\tag{15.16}
$$

Expression (15.16) is further transformed which results in the partial process tree shown in Fig. A.13.

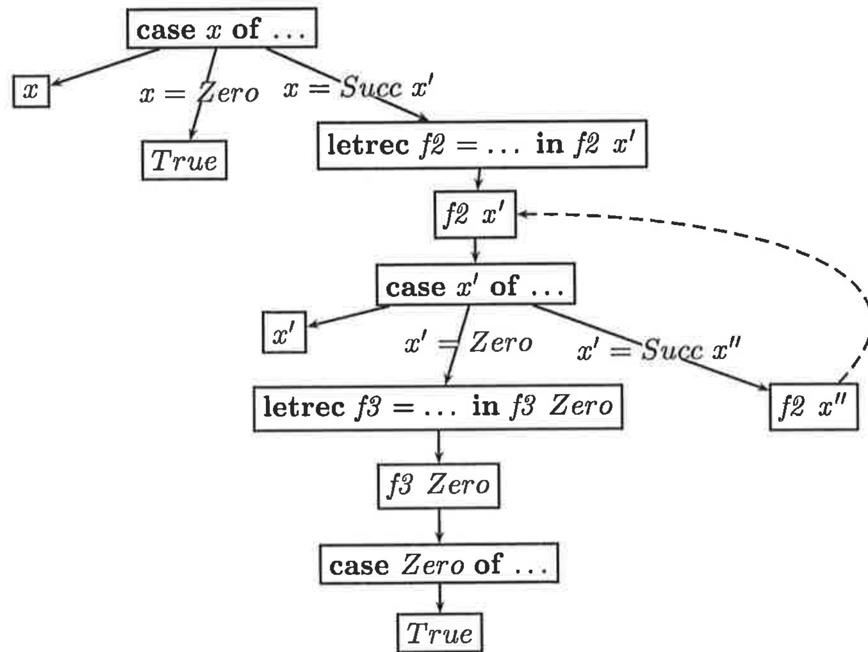

Figure A.13: Partial process tree (6) for $\mathcal{T}[\![even\ (doublea\ x\ Zero)]\!]$

The residual program shown in Fig. A.14 is constructed from the partial process tree shown in Fig. A.13.

$$
\begin{array}{l}
\textbf{case}\ x\ \textbf{of} \\
\quad Zero \quad : True \\
\mid Succ\ x' : \ \textbf{letrec}\ f1 \ = \ \lambda x'.\textbf{case}\ x'\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad Zero \quad : True \\
\qquad\qquad\qquad\qquad\qquad\quad \mid Succ\ x'' : f1\ \ x'' \\
\qquad\qquad\quad \textbf{in}\ f1\ \ x'
\end{array}
$$

Figure A.14: Residual program for $\mathcal{T}[\![even\ (doublea\ x\ Zero)]\!]$

A.1.3 Obstructing Function Calls

In the following example, we demonstrate how distillation performs generalization of an obstructing function call to obtain a successful transformation. An obstructing function call is detected when the current expression is a strict embedding of a previously encountered expression. In this case, the obstructing function call is extracted from the embedding expression.

Example 16

Consider the transformation of the following expression (16.1) using distillation.

$$\mathit{append}\ (\mathit{reverse}\ xs)\ ys \tag{16.1}$$

After a couple of steps, we obtain expression (16.2).

$$\begin{array}{l}\textbf{case}\ (\mathit{reverse}\ xs)\ \textbf{of}\\ \quad \mathit{Nil} \qquad\quad : ys\\ \quad |\ \mathit{Cons}\ x\ xs' : \mathit{Cons}\ x\ (\mathit{append}\ xs'\ ys)\end{array} \tag{16.2}$$

We encounter expression (16.3) during further transformation of expression (16.2).

$$\begin{array}{l}\textbf{case}\ (\mathit{append}\ (\mathit{reverse}\ xs')\ (\mathit{Cons}\ x\ \mathit{Nil}))\ \textbf{of}\\ \quad \mathit{Nil} \qquad\quad : ys\\ \quad |\ \mathit{Cons}\ x\ xs' : \mathit{Cons}\ x\ (\mathit{append}\ xs'\ ys)\end{array} \tag{16.3}$$

Expression (16.3) is a *strict* embedding of expression (16.2). The embedded sub-expression is therefore extracted from expression (16.3) to give the generalized expression (16.4).

$$\begin{array}{l}\textbf{let}\ v = \mathit{reverse}\ xs'\\ \textbf{in case}\ (\textbf{case}\ v\ \textbf{of}\\ \qquad\qquad \mathit{Nil} \qquad\quad : \mathit{Cons}\ x\ \mathit{Nil}\\ \qquad\qquad |\ \mathit{Cons}\ x'\ xs' : \mathit{Cons}\ x'\ (\mathit{append}\ xs'\ (\mathit{Cons}\ x\ \mathit{Nil})))\ \textbf{of}\\ \qquad \mathit{Nil} \qquad\quad : ys\\ \qquad |\ \mathit{Cons}\ x\ xs' : \mathit{Cons}\ x\ (\mathit{append}\ xs'\ ys)\end{array} \tag{16.4}$$

The sub-tree rooted at expression (16.3) is replaced with the result of transforming expression (16.4). The transformation proceeds as shown in Fig. A.15.

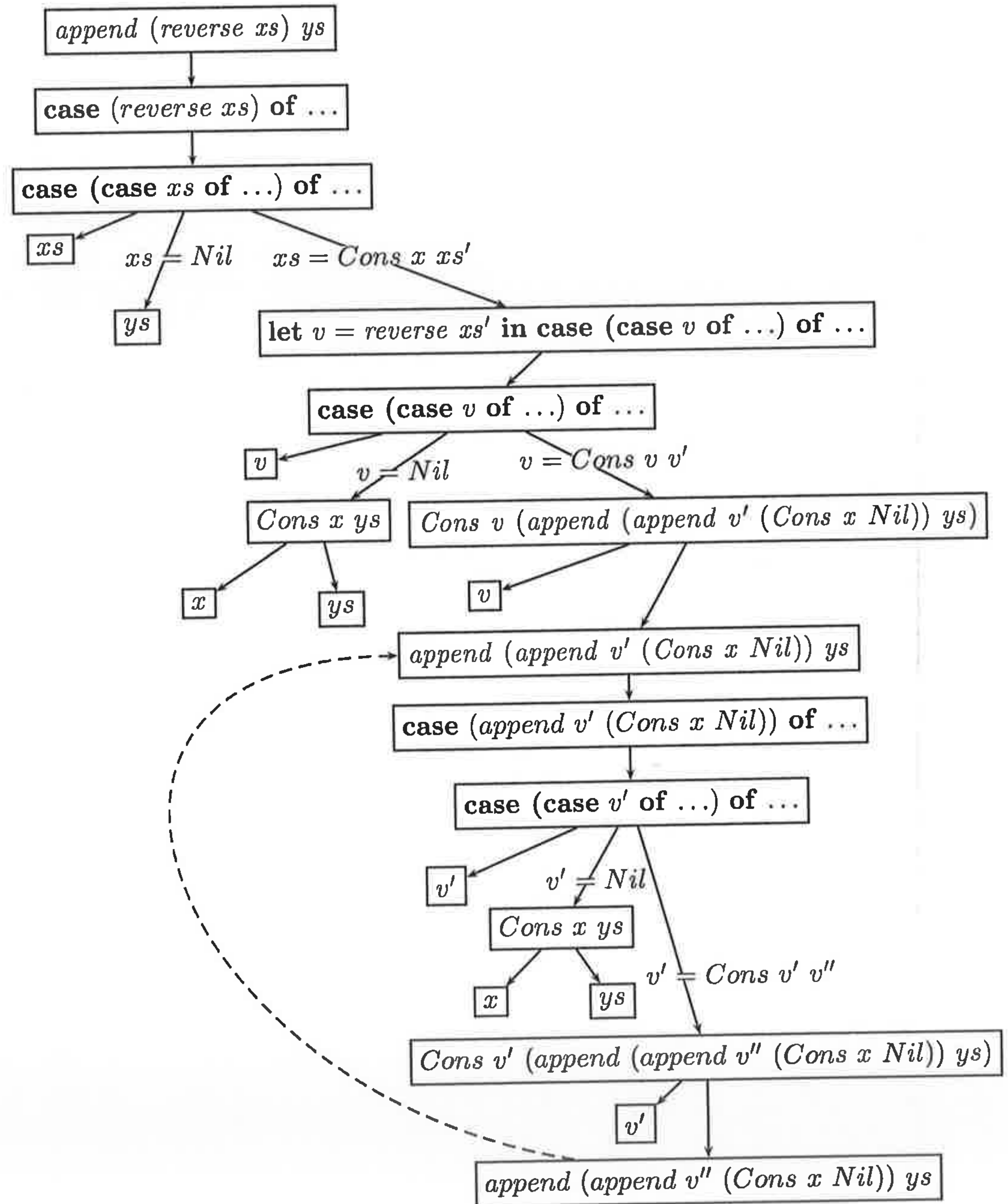

Figure A.15: Partial process tree (1) for $\mathcal{T}[\![append\ (reverse\ xs)\ ys]\!]$

The following residual program (16.5) is constructed from the sub-tree rooted at *append (append v′ (Cons x Nil)) ys*.

$$\textbf{Node f0: case } v' \textbf{ of} \\ \quad Nil \quad : Cons\ x\ ys \\ \quad \mid Cons\ v'\ v'' : Cons\ v'\ (\textbf{Repeat f0}: append\ (append\ v''\ (Cons\ x\ Nil))\ ys) \tag{16.5}$$

This program is further transformed. On further transformation of the repeat node, we obtain the following expression (16.6):

$$\textbf{Node f1: case } v'' \textbf{ of} \\ \quad Nil \quad : Cons\ x\ ys \\ \quad \mid Cons\ v''\ v''' : Cons\ v''\ (\textbf{Repeat f1}: append\ (append\ v'''\ (Cons\ x\ Nil))\ ys) \tag{16.6}$$

We can see that expression (16.6) is an instance of expression (16.5). A repeat node is created at the occurrence of expression (16.6). This results in the partial process tree shown in Fig. A.16.

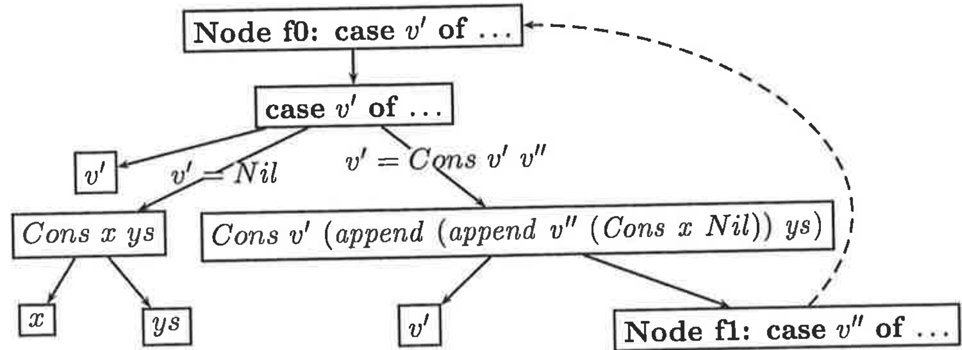

Figure A.16: Partial process tree (2) for $\mathcal{T}[\![append\ (reverse\ xs)\ ys]\!]$

This partial process sub-tree replaces the sub-tree which was rooted at *append (append v′ (Cons x Nil)) ys* in the previous partial process tree shown in Fig. A.15. The following residual program (16.7) is generated from the partial process sub-tree shown in Fig. A.16.

$$\begin{aligned} \textbf{letrec } f0 \ &= \ \lambda v'.\textbf{case } v' \textbf{ of} \\ &\qquad Nil \quad : Cons\ x\ ys \\ &\qquad \mid Cons\ v'\ v'' : Cons\ v'\ (f0\ v'') \\ \textbf{in } f0\ v' \end{aligned} \tag{16.7}$$

The transformation of the original generalized expression therefore results in the following residual program (16.8):

$$\begin{array}{l} \textbf{case } v \textbf{ of} \\ \quad Nil \qquad\quad : Cons\ x\ ys \\ \mid Cons\ v\ v' : Cons\ v\ (\textbf{letrec } f0 = \lambda v'.\textbf{case } v' \textbf{ of} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad Nil \qquad\quad : Cons\ x\ ys \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid Cons\ v'\ v'' : Cons\ v'\ (f0\ v'') \\ \qquad\qquad\qquad\qquad \textbf{in } f0\ v') \end{array} \tag{16.8}$$

After substituting the extracted sub-expression $reverse\ xs'$ for v, we obtain the following expression (16.9):

$$\begin{array}{l} \textbf{case } (reverse\ xs') \textbf{ of} \\ \quad Nil \qquad\quad : Cons\ x\ ys \\ \mid Cons\ v\ v' : Cons\ v\ (\textbf{letrec } f0 = \lambda v'.\textbf{case } v' \textbf{ of} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad Nil \qquad\quad : Cons\ x\ ys \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid Cons\ v'\ v'' : Cons\ v'\ (f0\ v'') \\ \qquad\qquad\qquad\qquad \textbf{in } f0\ v') \end{array} \tag{16.9}$$

After further transformation, we obtain the following expression (16.10):

$$\begin{array}{l} \textbf{case } (append\ (reverse\ xs'')\ (Cons\ x'\ Nil)) \textbf{ of} \\ \quad Nil \qquad\quad : Cons\ x\ ys \\ \mid Cons\ v\ v' : Cons\ v\ (\textbf{letrec } f0 = \lambda v'.\textbf{case } v' \textbf{ of} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad Nil \qquad\quad : Cons\ x\ ys \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid Cons\ v'\ v'' : Cons\ v'\ (f0\ v'') \\ \qquad\qquad\qquad\qquad \textbf{in } f0\ v') \end{array} \tag{16.10}$$

We can see that expression (16.10) is a strict embedding of expression (16.9). The embedded sub-expression is therefore extracted from expression (16.10) to give the generalized expression (16.11).

$$\begin{array}{l}
\textbf{let } v = \textit{reverse } xs'' \\
\textbf{in case } (\textbf{case } v \textbf{ of} \\
\qquad\qquad \textit{Nil} \qquad\qquad : \textit{Cons } x' \textit{ Nil} \\
\qquad\qquad \mid \textit{Cons } x \; xs' : \textit{Cons } x \; (\textit{append } xs' \; (\textit{Cons } x' \textit{ Nil}))) \textbf{ of} \\
\qquad \textit{Nil} \qquad\qquad : \textit{Cons } x \; ys \\
\quad \mid \textit{Cons } v \; v' : \textit{Cons } v \; (\textbf{letrec } f0 \; = \; \lambda v'.\textbf{case } v' \textbf{ of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{Nil} \qquad\qquad : \textit{Cons } x \; ys \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid \textit{Cons } v' \; v'' : \textit{Cons } v' \; (f0 \; v'') \\
\qquad\qquad\qquad\qquad \textbf{in } f0 \; v')
\end{array} \tag{16.11}$$

The following residual program (16.12) is generated from the transformation of the above generalized expression.

$$\begin{array}{l}
\textbf{case } v \textbf{ of} \\
\quad \textit{Nil} \qquad\qquad : \textit{Cons } x' \; (\textit{Cons } x \; ys) \\
\mid \textit{Cons } v \; v' : \textit{Cons } v \; (\textbf{ letrec } f2 \; = \; \lambda v'.\textbf{case } v' \textbf{ of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{Nil} \qquad\qquad : \textit{Cons } x' \; (\textit{Cons } x \; ys) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid \textit{Cons } v' \; v'' : \textit{Cons } v' \; (f2 \; v'') \\
\qquad\qquad\qquad\qquad \textbf{in } f2 \; v')
\end{array} \tag{16.12}$$

After substituting the extracted sub-expression *reverse* xs'' for v, we obtain the following expression (16.13):

$$\begin{array}{l}
\textbf{case } (\textit{reverse } xs'') \textbf{ of} \\
\quad \textit{Nil} \qquad\qquad : \textit{Cons } x' \; (\textit{Cons } x \; ys) \\
\mid \textit{Cons } v \; v' : \textit{Cons } v \; (\textbf{ letrec } f2 \; = \; \lambda v'.\textbf{case } v' \textbf{ of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{Nil} \qquad\qquad : \textit{Cons } x' \; (\textit{Cons } x \; ys) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid \textit{Cons } v' \; v'' : \textit{Cons } v' \; (f2 \; v'') \\
\qquad\qquad\qquad\qquad \textbf{in } f2 \; v')
\end{array} \tag{16.13}$$

Expression (16.13) is an instance of expression (16.9). A repeat node is therefore created at the occurrence of expression (16.13). The following residual program (16.14) is constructed from the transformation of expression (16.9).

Node f0: (16.14)

$$
\begin{array}{l}
\textbf{case } xs' \textbf{ of} \\
\quad \begin{array}{ll}
\;\; Nil & : Cons\ x\ ys \\
\mid Cons\ x'\ xs'' & : \textbf{Repeat f0:} \\
& \quad \textbf{case } (reverse\ xs'') \textbf{ of} \\
& \qquad \begin{array}{ll}
\;\; Nil & : Cons\ x'\ (Cons\ x\ ys) \\
\mid Cons\ v\ v' & : Cons\ v\ (\textbf{letrec} \\
& \qquad f0 = \lambda v'.\textbf{case } v' \textbf{ of} \\
& \qquad\qquad \begin{array}{ll}
\;\; Nil & : Cons\ x'\ (Cons\ x\ ys) \\
\mid Cons\ v'\ v'' & : Cons\ v'\ (f0\ v'')
\end{array} \\
& \qquad \textbf{in } f0\ v')
\end{array}
\end{array}
\end{array}
$$

Expression (16.14) is further transformed. On further transformation of the repeat node, we obtain expression (16.15).

Node f1: (16.15)

$$
\begin{array}{l}
\textbf{case } xs'' \textbf{ of} \\
\quad \begin{array}{ll}
\;\; Nil & : Cons\ x'\ (Cons\ x\ ys) \\
\mid Cons\ x''\ xs''' & : \textbf{Repeat f1:}
\end{array} \\
\qquad \textbf{case } (reverse\ xs''') \textbf{ of} \\
\qquad\quad \begin{array}{ll}
\;\; Nil & : Cons\ x''\ (Cons\ x'\ (Cons\ x\ ys)) \\
\mid Cons\ v\ v' & : Cons\ v\ (\textbf{letrec} \\
& \qquad f0 = \lambda v'.\textbf{case } v' \textbf{ of} \\
& \qquad\qquad \begin{array}{ll}
\;\; Nil & : Cons\ x''\ (Cons\ x'\ (Cons\ x\ ys)) \\
\mid Cons\ v'\ v'' & : Cons\ v'\ (f0\ v'')
\end{array} \\
& \qquad \textbf{in } f0\ v')
\end{array}
\end{array}
$$

Expression (16.15) is an instance of expression (16.14). A repeat node is created at the occurrence of expression (16.15). This results in the partial process tree shown in Fig. A.17.

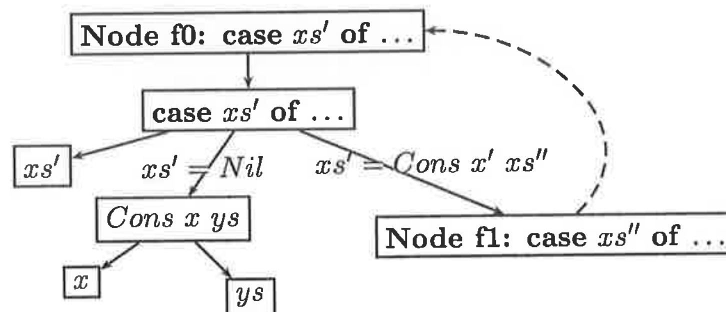

Figure A.17: Partial process tree (3) for $\mathcal{T}[\![append\ (reverse\ xs)\ ys]\!]$

The following residual program shown in Fig. A.18 is eventually constructed from the partial process tree shown in Figs A.15 and A.17.

$$
\begin{array}{l}
\textbf{case } xs \textbf{ of} \\
\quad Nil \quad : ys \\
\mid Cons\ x\ xs' : \textbf{letrec } f0 = \lambda xs'.\lambda x.\lambda ys.\textbf{case } xs' \textbf{ of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad Nil \quad : Cons\ x\ ys \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid Cons\ x'\ xs'' : f0\ xs''\ x'\ (Cons\ x\ ys) \\
\qquad\qquad\quad \textbf{in } f0\ xs'\ x\ ys
\end{array}
$$

Figure A.18: Residual program for $\mathcal{T}[\![append\ (reverse\ xs)\ ys]\!]$

This program does not create any intermediate structures. This transformation demonstrates the significant improvement in program transformation using distillation over other transformation techniques.

A.1.4 Examples for Theorem Proving

We give some examples using distillation algorithm to transform input expressions, which can then be used to prove the respective theorems.

Fig. A.19 shows some of the function definitions which are used with the definitions of the functions *eqnum, plus* of Chapter 2 and the definition of the function *even* of Chapter 3 to transform the input expressions.

$$
\begin{array}{lll}
iff & = & \lambda x.\lambda y.\textbf{case } x \textbf{ of} \\
& & \qquad True : y \\
& & \quad \mid False : \textbf{case } y \textbf{ of} \\
& & \qquad\qquad\quad True : False \\
& & \qquad\qquad \mid False : True \\
double & = & \lambda x.\textbf{case } x \textbf{ of} \\
& & \qquad Zero \quad : Zero \\
& & \quad \mid Succ\ x' : Succ\ (Succ\ (double\ x'))
\end{array}
$$

Figure A.19: Function definitions

In these examples, if a partial process tree does not fit into a single page, we refer to the subtree to be connected by labelling the root node as $\boxed{n}$.

Example 17

Consider the transformation of the following expression (17.1). In this example, we do not give the partial process trees constructed during the transformation. Expression (17.1) states the *commutativity of plus* theorem for natural numbers.

$$\mathit{eqnum}\ (\mathit{plus}\ x\ y)\ (\mathit{plus}\ y\ x) \tag{17.1}$$

Expression (17.2) is obtained by unfolding the function *eqnum* within expression (17.1).

$$\begin{array}{l} \textbf{case}\ (\mathit{plus}\ x\ y)\ \textbf{of} \\ \quad \begin{array}{ll} \mathit{Zero} & : \textbf{case}\ (\mathit{plus}\ y\ x)\ \textbf{of} \\ & \quad \begin{array}{ll} \mathit{Zero} & : \mathit{True} \\ \mid \mathit{Succ}\ y' & : \mathit{False} \end{array} \\ \mid \mathit{Succ}\ x' & : \textbf{case}\ (\mathit{plus}\ y\ x)\ \textbf{of} \\ & \quad \begin{array}{ll} \mathit{Zero} & : \mathit{False} \\ \mid \mathit{Succ}\ y' & : \mathit{eqnum}\ x'\ y' \end{array} \end{array} \end{array} \tag{17.2}$$

During the transformation of expression (17.2), expression (17.3) is encountered.

$$\mathit{eqnum}\ (\mathit{plus}\ x'\ (\mathit{Succ}\ y'))\ (\mathit{plus}\ y'\ (\mathit{Succ}\ x')) \tag{17.3}$$

Expression (17.3) is a non-strict homeomorphic embedding of expression (17.1). The most specific generalization of the expressions (17.1) and (17.3) is therefore performed. We obtain expression (17.4) from the generalization of expression (17.1).

$$\textbf{let}\ v = y,\ v' = x\ \textbf{in}\ \mathit{eqnum}\ (\mathit{plus}\ x\ v)\ (\mathit{plus}\ y\ v') \tag{17.4}$$

By transforming expression (17.4), we obtain the residual program which is shown in Fig. A.20.

Example 18

Consider the transformation of the following expression (18.1).

$$\mathit{iff}\ (\mathit{even}\ x)\ (\mathit{eqnum}\ (\mathit{double}\ y)\ x) \tag{18.1}$$

We obtain expression (18.2) by unfolding *iff* within expression (18.1).

$$
\begin{array}{l}
\textbf{letrec } f0 \;=\; \lambda x.\lambda y.\textbf{case } x \textbf{ of} \\
\qquad\qquad\qquad \mathit{Zero} \quad : \textbf{case } y \textbf{ of} \\
\qquad\qquad\qquad\qquad\qquad \mathit{Zero} \quad : \mathit{True} \\
\qquad\qquad\qquad\qquad\qquad \mid \mathit{Succ}\ y' : \textbf{letrec } f1 \;=\; \lambda y'.\textbf{case } y' \textbf{ of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{Zero} \quad : \mathit{True} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathit{Succ}\ y'' : \mathit{f1}\ y'' \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{in } \mathit{f1}\ y' \\
\qquad\qquad\qquad \mid \mathit{Succ}\ x' : \textbf{case } y \textbf{ of} \\
\qquad\qquad\qquad\qquad\qquad \mathit{Zero} \quad : \textbf{letrec } f1 \;=\; \lambda x'.\textbf{case } x' \textbf{ of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{Zero} \quad : \mathit{True} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathit{Succ}\ x'' : \mathit{f1}\ x'' \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{in } \mathit{f1}\ x' \\
\qquad\qquad\qquad\qquad\qquad \mid \mathit{Succ}\ y' : \mathit{f0}\ x'\ y' \\
\textbf{in } \mathit{f0}\ x\ y
\end{array}
$$

Figure A.20: Residual program for $\mathcal{T}[\![\mathit{eqnum}\ (\mathit{plus}\ x\ y)\ (\mathit{plus}\ y\ x)]\!]$

$$
\begin{array}{l}
\textbf{case } (\mathit{even}\ x) \textbf{ of} \\
\quad \mathit{True} : \mathit{eqnum}\ (\mathit{double}\ y)\ x \\
\mid \mathit{False} : \textbf{case } (\mathit{eqnum}\ (\mathit{double}\ y)\ x) \textbf{ of} \\
\qquad\qquad \mathit{True} : \mathit{False} \\
\qquad\qquad \mid \mathit{False} : \mathit{True}
\end{array} \tag{18.2}
$$

After a few steps, we encounter expression (18.3) which is a non-strict homeomorphic embedding of expression (18.2).

$$
\begin{array}{l}
\textbf{case } (\mathit{even}\ x'') \textbf{ of} \\
\quad \mathit{True} : \mathit{eqnum}\ (\mathit{double}\ y)\ (\mathit{Succ}\ (\mathit{Succ}\ x'')) \\
\mid \mathit{False} : \textbf{case } (\mathit{eqnum}\ (\mathit{double}\ y)\ (\mathit{Succ}\ (\mathit{Succ}\ x''))) \textbf{ of} \\
\qquad\qquad \mathit{True} : \mathit{False} \\
\qquad\qquad \mid \mathit{False} : \mathit{True}
\end{array} \tag{18.3}
$$

The most specific generalization of the expressions (18.2) and (18.3) is therefore performed. The generalization of expression (18.2) results in the expression (18.4).

$$\textbf{let } v = x \textbf{ in case } (\mathit{even}\ x) \textbf{ of} \quad (18.4)$$
$$\begin{array}{l} \mathit{True} : \mathit{eqnum}\ (\mathit{double}\ y)\ v \\ \mid \mathit{False} : \textbf{case } (\mathit{eqnum}\ (\mathit{double}\ y)\ v) \textbf{ of} \\ \qquad \mathit{True} : \mathit{False} \\ \qquad \mid \mathit{False} : \mathit{True} \end{array}$$

The subtree rooted at expression (18.2) is replaced with the result of transforming the generalized form of expression (18.2). The transformation of expression (18.4) proceeds as shown in Fig. A.21.

We obtain expression (18.5) from the subtree rooted at *eqnum* (*double* y) v within the partial process tree shown in Fig. A.21.

$$\begin{array}{l} \textbf{Node f2: case } y \textbf{ of} \\ \quad \mathit{Zero} \quad : \textbf{case } v \textbf{ of} \\ \qquad\quad \mathit{Zero} \quad : \mathit{True} \\ \qquad\quad \mid \mathit{Succ}\ v' : \mathit{False} \\ \quad \mid \mathit{Succ}\ y' : \textbf{case } v \textbf{ of} \\ \qquad\quad \mathit{Zero} \quad : \mathit{False} \\ \qquad\quad \mid \mathit{Succ}\ v' : \textbf{case } v' \textbf{ of} \\ \qquad\qquad\quad \mathit{Zero} \quad : \mathit{False} \\ \qquad\qquad\quad \mid \mathit{Succ}\ v'' : \textbf{Repeat f2: } \mathit{eqnum}\ (\mathit{double}\ y')\ v'' \end{array} \quad (18.5)$$

Expression (18.5) is further transformed which results in the partial process tree shown in Fig. A.22.

During this transformation, the transformation of expression (18.6) within the partial process tree shown in Fig. A.22 is performed in a similar way to that of the expression *eqnum* (*double* y) v within the partial process tree shown in Fig. A.21.

$$\mathit{eqnum}\ (\mathit{double}\ y')\ v'' \quad (18.6)$$

During the transformation of expression (18.6), expression (18.7) is encountered which is an instance of expression (18.6). A repeat node is therefore created at the occurrence of expression (18.7).

$$\mathit{eqnum}\ (\mathit{double}\ y'')\ v'''' \quad (18.7)$$

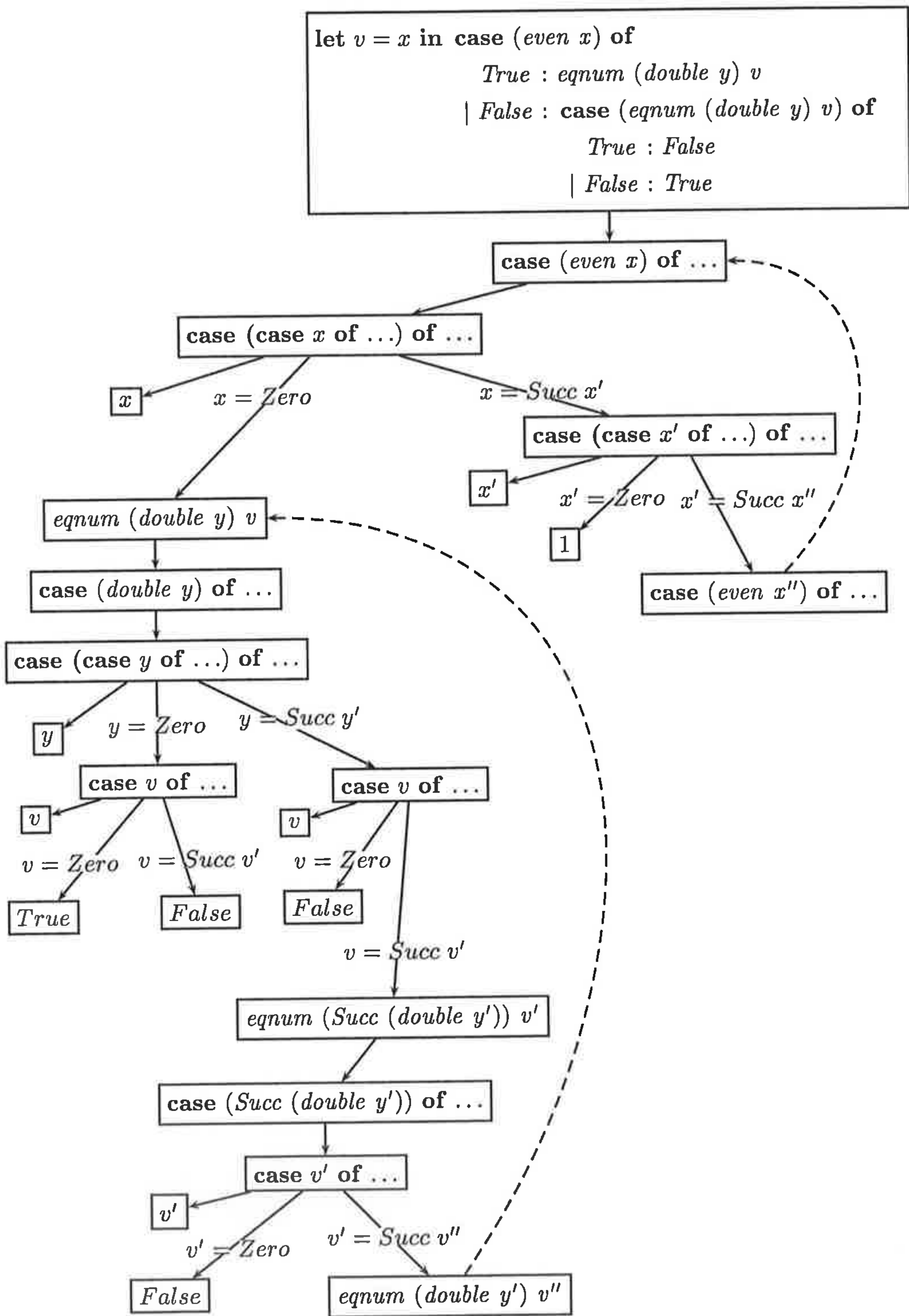

Figure A.21: Partial process tree (1) for $\mathcal{T}[\![$*iff* (*even* x) (*eqnum* (*double* y) x)$]\!]$

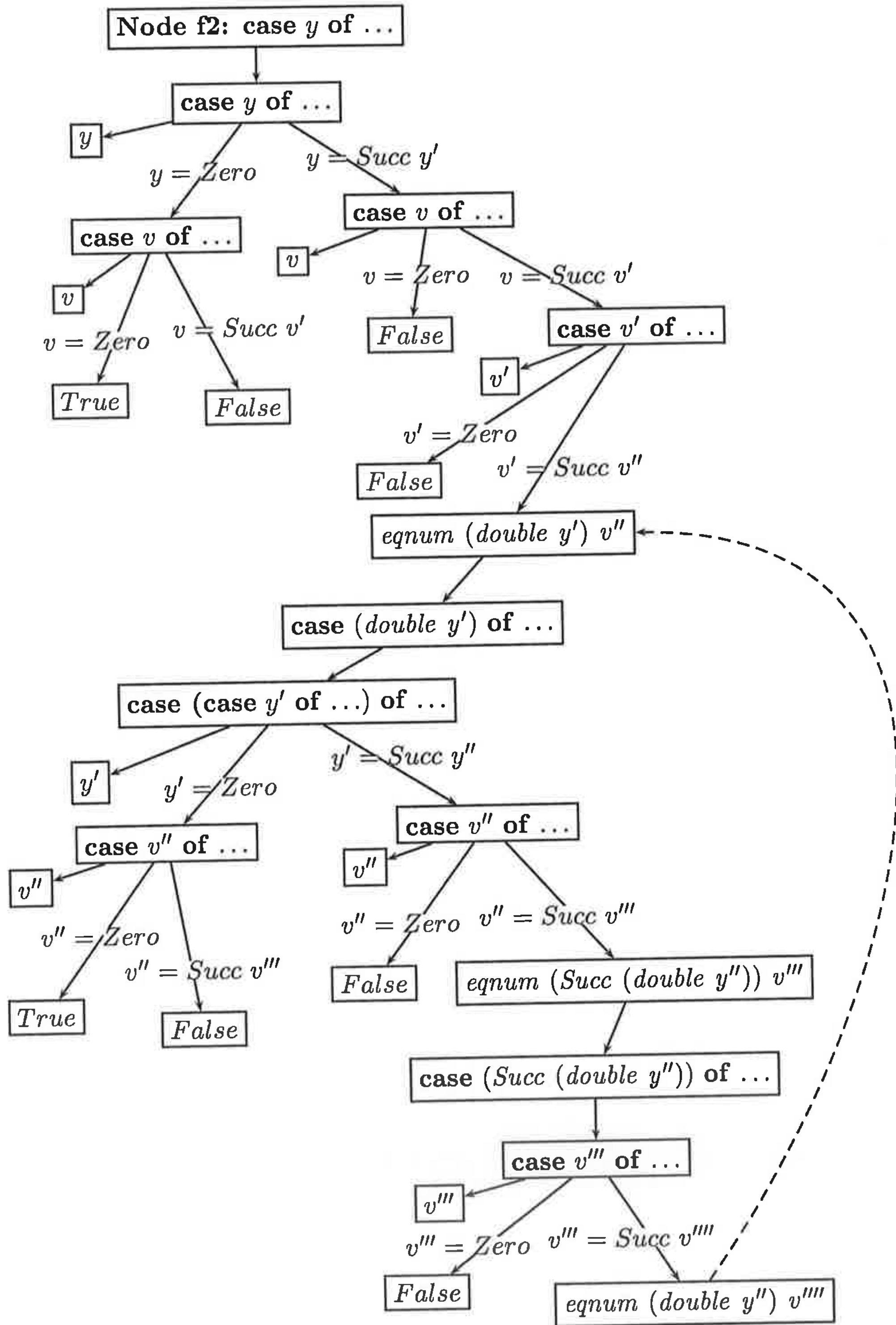

Figure A.22: Partial process tree (2) for $\mathcal{T}[\![\mathit{iff}\ (\mathit{even}\ x)\ (\mathit{eqnum}\ (\mathit{double}\ y)\ x)]\!]$

We obtain expression (18.8) from the subtree rooted at *eqnum* (*double* y') v'' within the partial process tree shown in Fig. A.22.

$$
\begin{array}{l}
\textbf{Node f3: case } y' \textbf{ of} \\
\qquad\qquad \textit{Zero} \quad : \textbf{case } v'' \textbf{ of} \\
\qquad\qquad\qquad\qquad \textit{Zero} \quad : \textit{True} \\
\qquad\qquad\qquad\quad\ \mid \textit{Succ } v''' : \textit{False} \\
\qquad\quad \mid \textit{Succ } y'' : \textbf{case } v'' \textbf{ of} \\
\qquad\qquad\qquad\qquad \textit{Zero} \quad : \textit{False} \\
\qquad\qquad\qquad\quad\ \mid \textit{Succ } v''' : \textbf{case } v''' \textbf{ of} \\
\qquad\qquad\qquad\qquad\qquad\qquad \textit{Zero} \quad : \textit{False} \\
\qquad\qquad\qquad\qquad\qquad\quad \mid \textit{Succ } v'''' : \textbf{Repeat f3}: \textit{eqnum } (\textit{double } y'') \ v''''
\end{array} \tag{18.8}
$$

Expression (18.8) is an instance of expression (18.5). A repeat node is therefore created at the occurrence of expression (18.8). This results in the partial process tree shown in Fig. A.23.

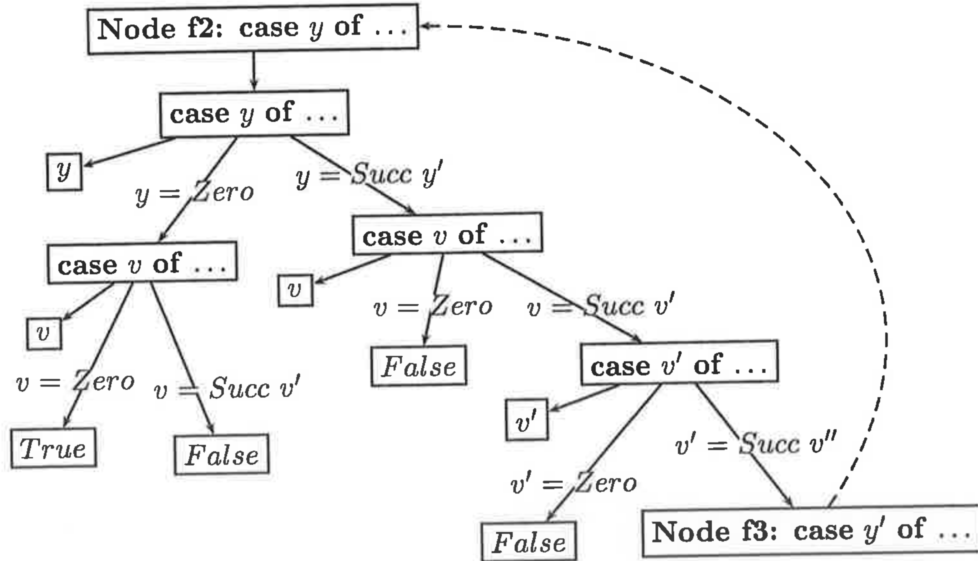

Figure A.23: Partial process tree (3) for $\mathcal{T}[\![$*iff* (*even* x) (*eqnum* (*double* y) x)$]\!]$

The residual program given by expression (18.9) is constructed from the partial process tree shown in Fig. A.23.

$$
\begin{aligned}
\textbf{letrec}\ f2 \ = \ & \lambda y.\lambda v.\,\textbf{case}\ y\ \textbf{of} \\
& \quad \mathit{Zero} \quad : \textbf{case}\ v\ \textbf{of} \\
& \qquad\qquad \mathit{Zero} \quad : \mathit{True} \\
& \qquad\qquad \mid \mathit{Succ}\ v' : \mathit{False} \\
& \quad \mid \mathit{Succ}\ y' : \textbf{case}\ v\ \textbf{of} \\
& \qquad\qquad \mathit{Zero} \quad : \mathit{False} \\
& \qquad\qquad \mid \mathit{Succ}\ v' : \textbf{case}\ v'\ \textbf{of} \\
& \qquad\qquad\qquad \mathit{Zero} \quad : \mathit{False} \\
& \qquad\qquad\qquad \mid \mathit{Succ}\ v'' : \mathit{f2}\ y'\ v'' \\
\textbf{in}\ & \mathit{f2}\ y\ v
\end{aligned} \tag{18.9}
$$

As shown in Fig. A.21, at terminal $\boxed{1}$, expression (18.10) is encountered.

$$
\begin{aligned}
& \textbf{case}\ (\mathit{eqnum}\ (\mathit{double}\ y)\ v)\ \textbf{of} \\
& \quad \mathit{True} : \mathit{False} \\
& \mid \mathit{False} : \mathit{True}
\end{aligned} \tag{18.10}
$$

The partial process tree shown in Fig. A.24 is obtained by transforming expression (18.10).

During the transformation of expression (18.10), expression (18.11) is encountered which is an instance of expression (18.10). A repeat node is therefore created at the occurrence of expression (18.11).

$$
\begin{aligned}
& \textbf{case}\ (\mathit{eqnum}\ (\mathit{double}\ y')\ v'')\ \textbf{of} \\
& \quad \mathit{True} : \mathit{False} \\
& \mid \mathit{False} : \mathit{True}
\end{aligned} \tag{18.11}
$$

We obtain expression (18.12) from the partial process tree shown in Fig. A.24.

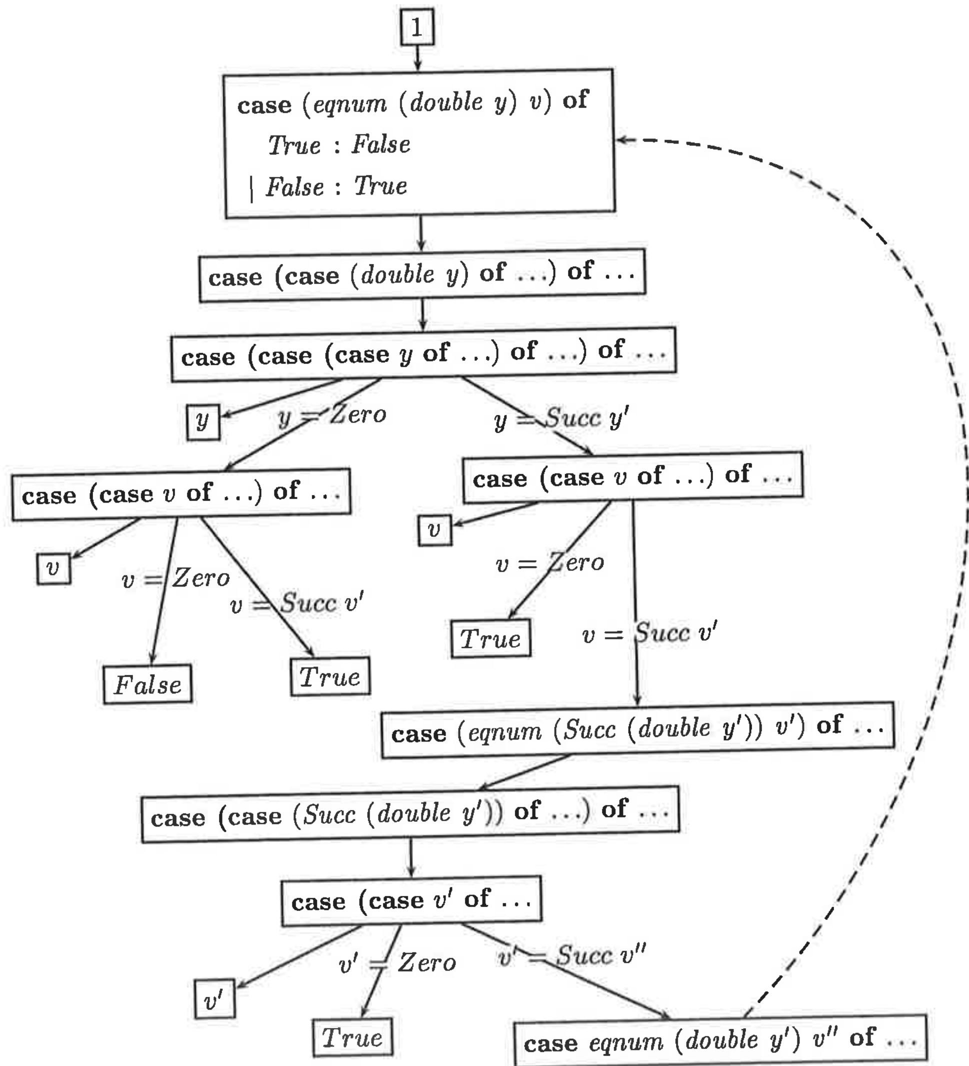

Figure A.24: Partial process tree (4) for $\mathcal{T}[\![iff\ (even\ x)\ (eqnum\ (double\ y)\ x)]\!]$ (Cont. of Fig. A.21)

Node f2: (18.12)

$$
\begin{array}{l}
\textbf{case } y \textbf{ of} \\
\quad \mathit{Zero} \quad : \textbf{case } v \textbf{ of} \\
\qquad\qquad\qquad \mathit{Zero} \quad : \mathit{False} \\
\qquad\qquad\qquad \mid \mathit{Succ}\ v' : \mathit{True} \\
\mid \mathit{Succ}\ y' : \textbf{case } v \textbf{ of} \\
\qquad\qquad\qquad \mathit{Zero} \quad : \mathit{True} \\
\qquad\qquad\qquad \mid \mathit{Succ}\ v' : \textbf{case } v' \textbf{ of} \\
\qquad\qquad\qquad\qquad\qquad \mathit{Zero} \quad : \mathit{True} \\
\qquad\qquad\qquad\qquad\qquad \mid \mathit{Succ}\ v'' : \textbf{Repeat f2: case } (\mathit{eqnum}\ (\mathit{double}\ y')\ v'') \textbf{ of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{True} : \mathit{False} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathit{False} : \mathit{True}
\end{array}
$$

Expression (18.12) is further transformed. The transformation proceeds as shown in Fig. A.25.

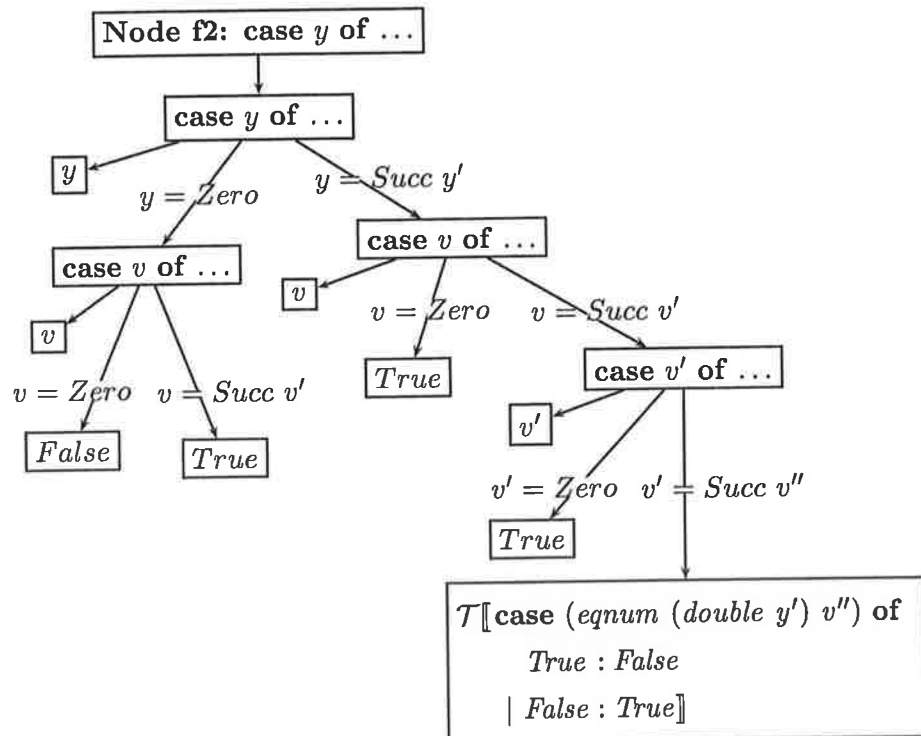

Figure A.25: Partial process tree (5) for $\mathcal{T}[\![\mathit{iff}\ (\mathit{even}\ x)\ (\mathit{eqnum}\ (\mathit{double}\ y)\ x)]\!]$

During this transformation, the transformation of expression (18.13) is performed in a similar way to that of expression (18.10) as shown in Fig. A.24.

$$\begin{aligned}&\textbf{case}\ (\mathit{eqnum}\ (\mathit{double}\ y')\ v'')\ \textbf{of}\\ &\quad \mathit{True} : \mathit{False}\\ &\mid \mathit{False} : \mathit{True}\end{aligned} \tag{18.13}$$

During the transformation of expression (18.13), expression (18.14) is encountered, which is an instance of expression (18.13). A repeat node is therefore created at the occurrence of expression (18.14).

$$\begin{aligned}&\textbf{case}\ (\mathit{eqnum}\ (\mathit{double}\ y'')\ v'''')\ \textbf{of}\\ &\quad \mathit{True} : \mathit{False}\\ &\mid \mathit{False} : \mathit{True}\end{aligned} \tag{18.14}$$

The transformation of the expression (18.13) therefore results in the following expression (18.15).

$$\begin{array}{l}\textbf{Node f3:}\\ \textbf{case}\ y'\ \textbf{of}\\ \quad \mathit{Zero} \qquad : \textbf{case}\ v''\ \textbf{of}\\ \qquad\qquad\qquad \mathit{Zero} \qquad : \mathit{False}\\ \qquad\qquad\quad\ \mid \mathit{Succ}\ v''' : \mathit{True}\\ \mid \mathit{Succ}\ y'' : \textbf{case}\ v''\ \textbf{of}\\ \qquad\qquad\qquad \mathit{Zero} \qquad : \mathit{True}\\ \qquad\qquad\quad\ \mid \mathit{Succ}\ v''' : \textbf{case}\ v'''\ \textbf{of}\\ \qquad\qquad\qquad\qquad\qquad \mathit{Zero} \qquad : \mathit{True}\\ \qquad\qquad\qquad\qquad\quad\ \mid \mathit{Succ}\ v'''' : \textbf{Repeat f3: case}\ (\mathit{eqnum}\ (\mathit{double}\ y'')\ v'''')\ \textbf{of}\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{True} : \mathit{False}\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad\ \mid \mathit{False} : \mathit{True}\end{array} \tag{18.15}$$

Now, expression (18.15) is an instance of expression (18.12). A repeat node is therefore created at the occurrence of expression (18.15). This results in the partial process tree which is shown in Fig. A.26.

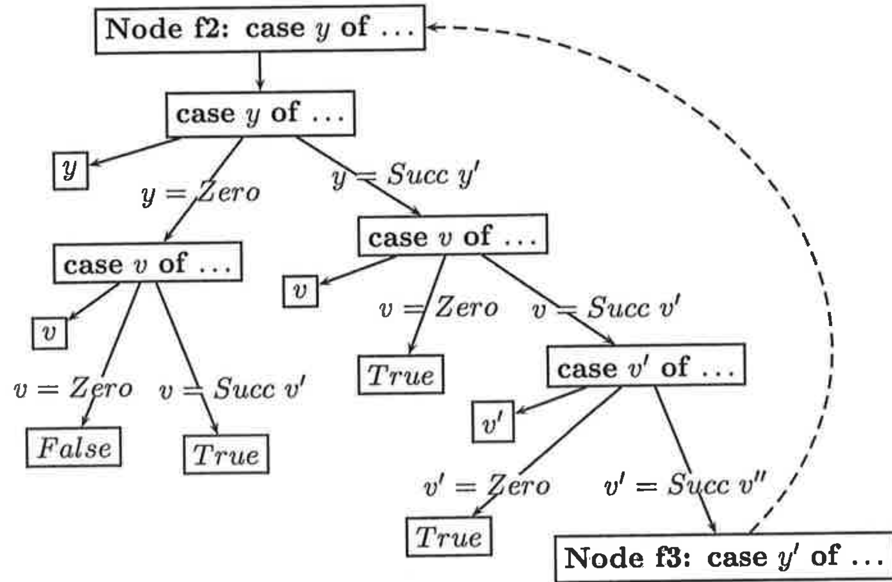

Figure A.26: Partial process tree (6) for $\mathcal{T}[\![iff\ (even\ x)\ (eqnum\ (double\ y)\ x)]\!]$

From the partial process tree shown in Fig. A.26, the residual program given by expression (18.16) is constructed.

$$
\begin{array}{l}
\textbf{letrec } f2 \ = \ \lambda y.\lambda v.\textbf{case } y \textbf{ of} \\
\qquad\qquad\qquad\qquad Zero \quad : \textbf{case } v \textbf{ of} \\
\qquad\qquad\qquad\qquad\qquad\qquad Zero \quad : False \\
\qquad\qquad\qquad\qquad\qquad\quad\ \mid Succ\ v' : True \\
\qquad\qquad\qquad\quad\ \mid Succ\ y' : \textbf{case } v \textbf{ of} \\
\qquad\qquad\qquad\qquad\qquad\qquad Zero \quad : True \\
\qquad\qquad\qquad\qquad\qquad\quad\ \mid Succ\ v' : \textbf{case } v' \textbf{ of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad Zero \quad : True \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad\ \mid Succ\ v'' : f2\ y'\ v'' \\
\textbf{in } f2\ y\ v
\end{array}
\qquad (18.16)
$$

Thus, the above transformation results in expression (18.17).

Node f1: (18.17)

$$
\begin{array}{l}
\textbf{case } x \textbf{ of} \\
\quad \mathit{Zero} \quad : \textbf{letrec} \\
\qquad\qquad f2 = \lambda y.\lambda v.\textbf{case } y \textbf{ of} \\
\qquad\qquad\qquad\qquad \mathit{Zero} \quad : \textbf{case } v \textbf{ of} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{Zero} \quad : \mathit{True} \\
\qquad\qquad\qquad\qquad\qquad\quad\ \mid \mathit{Succ}\ v' : \mathit{False} \\
\qquad\qquad\qquad\qquad \mid \mathit{Succ}\ y' : \textbf{case } v \textbf{ of} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{Zero} \quad : \mathit{False} \\
\qquad\qquad\qquad\qquad\qquad\quad\ \mid \mathit{Succ}\ v' : \textbf{case } v' \textbf{ of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{Zero} \quad : \mathit{False} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathit{Succ}\ v'' : f2\ y'\ v'' \\
\qquad\qquad \textbf{in } f2\ y\ v \\
\quad \mid \mathit{Succ}\ x' : \textbf{case } x' \textbf{ of} \\
\qquad\qquad \mathit{Zero} \quad : \textbf{letrec} \\
\qquad\qquad\qquad f2 = \lambda y.\lambda v.\textbf{case } y \textbf{ of} \\
\qquad\qquad\qquad\qquad\qquad \mathit{Zero} \quad : \textbf{case } v \textbf{ of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{Zero} \quad : \mathit{False} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathit{Succ}\ v' : \mathit{True} \\
\qquad\qquad\qquad\qquad\qquad \mid \mathit{Succ}\ y' : \textbf{case } v \textbf{ of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{Zero} \quad : \mathit{True} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathit{Succ}\ v' : \textbf{case } v' \textbf{ of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{Zero} \quad : \mathit{True} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathit{Succ}\ v'' : f2\ y'\ v'' \\
\qquad\qquad\qquad \textbf{in } f2\ y\ v \\
\qquad\quad \mid \mathit{Succ}\ x'' : \textbf{Repeat f1: case } (\mathit{even}\ x'') \textbf{ of} \\
\qquad\qquad\qquad\qquad\qquad \mathit{True} : \mathit{eqnum}\ (\mathit{double}\ y)\ v \\
\qquad\qquad\qquad\qquad\quad \mid \mathit{False} : \textbf{case } (\mathit{eqnum}\ (\mathit{double}\ y)\ v) \textbf{ of} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{True} : \mathit{False} \\
\qquad\qquad\qquad\qquad\qquad \mid \mathit{False} : \mathit{True}
\end{array}
$$

Expression (18.17) is further transformed. We skip the details of this transformation. The residual program given by expression (18.18) is constructed from this transformation.

$$
\begin{array}{l}
\textbf{letrec} \\
\mathit{f1} \ = \ \lambda x.\,\textbf{case}\ x\ \textbf{of} \\
\qquad \mathit{Zero} \quad : \textbf{letrec} \\
\qquad\qquad \mathit{f2} \ = \ \lambda y.\lambda v.\,\textbf{case}\ y\ \textbf{of} \\
\qquad\qquad\qquad \mathit{Zero} \quad : \textbf{case}\ v\ \textbf{of} \\
\qquad\qquad\qquad\qquad \mathit{Zero} \quad : \mathit{True} \\
\qquad\qquad\qquad\qquad \mid \mathit{Succ}\ v' : \mathit{False} \\
\qquad\qquad\qquad \mid \mathit{Succ}\ y' : \textbf{case}\ v\ \textbf{of} \\
\qquad\qquad\qquad\qquad \mathit{Zero} \quad : \mathit{False} \\
\qquad\qquad\qquad\qquad \mid \mathit{Succ}\ v' : \textbf{case}\ v'\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad \mathit{Zero} \quad : \mathit{False} \\
\qquad\qquad\qquad\qquad\qquad \mid \mathit{Succ}\ v'' : \mathit{f2}\ y'\ v'' \\
\qquad\qquad \textbf{in}\ \mathit{f2}\ y\ v \\
\qquad \mid \mathit{Succ}\ x' : \textbf{case}\ x'\ \textbf{of} \\
\qquad\qquad \mathit{Zero} \quad : \textbf{letrec} \\
\qquad\qquad\qquad \mathit{f2} \ = \ \lambda y.\lambda v.\,\textbf{case}\ y\ \textbf{of} \\
\qquad\qquad\qquad\qquad \mathit{Zero} \quad : \textbf{case}\ v\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad \mathit{Zero} \quad : \mathit{False} \\
\qquad\qquad\qquad\qquad\qquad \mid \mathit{Succ}\ v' : \mathit{True} \\
\qquad\qquad\qquad\qquad \mid \mathit{Succ}\ y' : \textbf{case}\ v\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad \mathit{Zero} \quad : \mathit{True} \\
\qquad\qquad\qquad\qquad\qquad \mid \mathit{Succ}\ v' : \textbf{case}\ v'\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{Zero} \quad : \mathit{True} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathit{Succ}\ v'' : \mathit{f2}\ y'\ v'' \\
\qquad\qquad\qquad \textbf{in}\ \mathit{f2}\ y\ v \\
\qquad\qquad \mid \mathit{Succ}\ x'' : \mathit{f1}\ x'' \\
\textbf{in}\ \mathit{f1}\ x
\end{array}
\qquad (18.18)
$$

By substituting back the extracted variable x for v within expression (18.18), and by transforming the expression resulting from the substitution, the partial process tree shown in Fig. A.27 is obtained.

From this partial process tree, we construct the residual program shown in Fig. A.28.

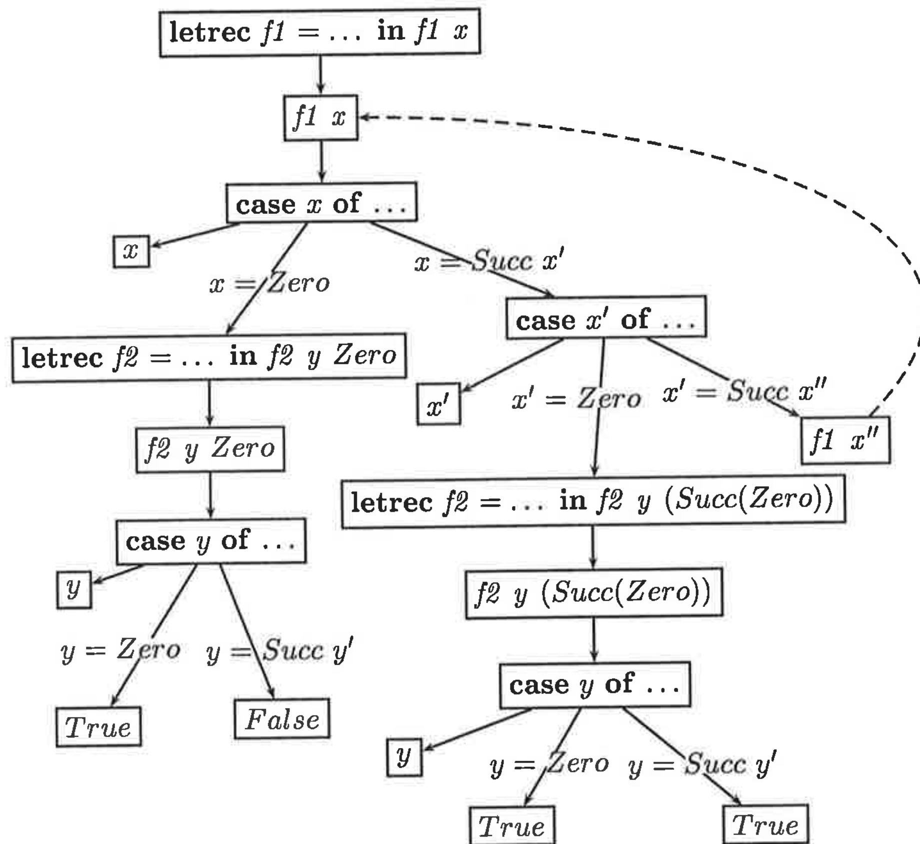

Figure A.27: Partial process tree (7) for $\mathcal{T}[\![\mathit{iff}\ (\mathit{even}\ x)\ (\mathit{eqnum}\ (\mathit{double}\ y)\ x)]\!]$

$$
\begin{array}{l}
\textbf{letrec}\ \mathit{f1} = \lambda x.\textbf{case}\ x\ \textbf{of} \\
\qquad\qquad\qquad \mathit{Zero} \quad : \textbf{case}\ y\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad \mathit{Zero} \quad : \mathit{True} \\
\qquad\qquad\qquad\qquad\qquad \mid \mathit{Succ}\ y' : \mathit{False} \\
\qquad\qquad\qquad \mid \mathit{Succ}\ x' : \textbf{case}\ x'\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad \mathit{Zero} \quad : \textbf{case}\ y\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{Zero} \quad : \mathit{True} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathit{Succ}\ y' : \mathit{True} \\
\qquad\qquad\qquad\qquad\qquad \mid \mathit{Succ}\ x'' : \mathit{f1}\ x'' \\
\textbf{in}\ \mathit{f1}\ x
\end{array}
$$

Figure A.28: Residual program for $\mathcal{T}[\![\mathit{iff}\ (\mathit{even}\ x)\ (\mathit{eqnum}\ (\mathit{double}\ y)\ x)]\!]$

Appendix B

Theorem Proving in Poitín

B.1 Example

Example 19

Consider the following conjecture (19.1), which states that the length of appending two lists is equal to the addition of their individual lengths.

$$\text{ALL } xs.\text{ALL } ys.eqnum\ (length\ (append\ xs\ ys))\ (plus\ (length\ xs)\ (length\ ys)) \tag{19.1}$$

The following definition of the function *length* is used along with the definitions of *append*, *eqnum* and *plus* as defined in Chapter 2.

$$\begin{aligned} length &= \lambda xs.\,\mathbf{case}\ xs\ \mathbf{of} \\ &\qquad\qquad Nil \qquad\quad : Zero \\ &\qquad\qquad \mid Cons\ x\ xs' : Succ\ (length\ xs') \end{aligned}$$

The proof of conjecture (19.1) is guided by distillation rule $(\mathcal{T}2)$ (Fig. 4.4). Rule $(\mathcal{T}2)$ applies distillation to expression (19.2).

$$eqnum\ (length\ (append\ xs\ ys))\ (plus\ (length\ xs)\ (length\ ys)) \tag{19.2}$$

Applying distillation to expression (19.2), we obtain the following distilled expression (19.3).

$$
\begin{array}{l}
\textbf{letrec} \\
\quad f0 \;=\; \lambda xs.\,\textbf{case}\ xs\ \textbf{of} \\
\qquad\qquad Nil \quad : \textbf{case}\ ys\ \textbf{of} \\
\qquad\qquad\qquad\qquad Nil \quad : True \\
\qquad\qquad\qquad\qquad \mid Cons\ y\ ys' : \ \textbf{letrec} \\
\qquad\qquad\qquad\qquad\qquad f1 \;=\; \lambda ys'.\,\textbf{case}\ ys'\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad Nil \quad : True \\
\qquad\qquad\qquad\qquad\qquad\qquad \mid Cons\ y'\ ys'' : f1\ ys'' \\
\qquad\qquad\qquad\qquad\qquad \textbf{in}\ f1\ ys' \\
\qquad\qquad \mid Cons\ x\ xs' : f0\ xs' \\
\textbf{in}\ f0\ xs
\end{array}
\tag{19.3}
$$

The proof of the corresponding proof expression obtained by pre-processing expression (19.3) proceeds as shown below.

$$
\begin{array}{l}
\mathcal{A}[\![\,\textbf{letrec} \\
\quad f0 \;=\; \lambda xs.\lambda ys.\,\textbf{case}\ xs\ \textbf{of} \\
\qquad\qquad Nil \quad : \textbf{case}\ ys\ \textbf{of} \\
\qquad\qquad\qquad\qquad Nil \quad : True \\
\qquad\qquad\qquad\qquad \mid Cons\ y\ ys' : \textbf{letrec} \\
\qquad\qquad\qquad\qquad\qquad f1 \;=\; \lambda ys'.\,\textbf{case}\ ys'\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad Nil \quad : True \\
\qquad\qquad\qquad\qquad\qquad\qquad \mid Cons\ y'\ ys'' : f1\ ys'' \\
\qquad\qquad\qquad\qquad\qquad \textbf{in}\ f1\ ys' \\
\qquad\qquad \mid Cons\ x\ xs' : f0\ xs'\ ys \\
\quad \textbf{in}\ f0\ xs\ ys\,]\!]\ \{\}\ \{xs, ys\}
\end{array}
\qquad \text{(by } \mathcal{T}2\text{)}
$$

$$
\begin{array}{l}
= \mathcal{A}[\![\textbf{case } xs \textbf{ of} \\
\quad \quad \mathit{Nil} \quad : \textbf{case } ys \textbf{ of} \\
\quad \quad \quad \quad \mathit{Nil} \quad : \mathit{True} \\
\quad \quad \quad \quad \mid \mathit{Cons}\ y\ ys' : \ \textbf{letrec } f1 = \lambda ys'.\textbf{case } ys' \textbf{ of} \\
\quad \quad \quad \quad \quad \quad \quad \quad \mathit{Nil} \quad : \mathit{True} \\
\quad \quad \quad \quad \quad \quad \quad \quad \mid \mathit{Cons}\ y'\ ys'' : f1\ ys'' \\
\quad \quad \quad \quad \quad \quad \textbf{in } f1\ ys' \\
\quad \quad \mid \mathit{Cons}\ x\ xs' : f0\ xs'\ ys]\!]\ \{f0\ xs\ ys\}\ \{xs, ys\} \\
\hfill (\text{by } \mathcal{A}6)
\end{array}
$$

$$
\begin{array}{l}
= \mathcal{T}[\![(\mathcal{A}[\![\textbf{case } ys \textbf{ of} \\
\quad \quad \mathit{Nil} \quad : \mathit{True} \\
\quad \quad \mid \mathit{Cons}\ y\ ys' : \ \textbf{letrec } f1 = \lambda ys'.\textbf{case } ys' \textbf{ of} \\
\quad \quad \quad \quad \quad \quad \mathit{Nil} \quad : \mathit{True} \\
\quad \quad \quad \quad \quad \quad \mid \mathit{Cons}\ y'\ ys'' : f1\ ys'' \\
\quad \quad \quad \quad \textbf{in } f1\ ys']\!]\ \{f0\ xs\ ys\}\ \{xs, ys\}) \\
\quad \quad \quad \quad \quad \wedge\ (\mathcal{A}[\![f0\ xs'\ ys]\!]\ \{f0\ xs\ ys\}\ \{xs, ys, x, xs'\})]\!]\ \{\}\ \{\} \\
\hfill (\text{by } \mathcal{A}5)
\end{array}
$$

$$
\begin{array}{l}
= \mathcal{T}[\![(\mathcal{T}[\![(\mathcal{A}[\![\mathit{True}]\!]\ \{f0\ xs\ ys\}\ \{xs, ys\}) \hfill (\text{by } \mathcal{A}5) \\
\quad \quad \quad \quad \wedge\ (\mathcal{A}[\![\ \textbf{letrec } f1 = \lambda ys'.\textbf{case } ys' \textbf{ of} \\
\quad \quad \quad \quad \quad \quad \quad \quad \mathit{Nil} \quad : \mathit{True} \\
\quad \quad \quad \quad \quad \quad \quad \quad \mid \mathit{Cons}\ y'\ ys'' : f1\ ys'' \\
\quad \quad \quad \quad \quad \quad \textbf{in } f1\ ys']\!]\ \{f0\ xs\ ys\}\ \{xs, ys, y, ys'\})]\!]\ \{\}\ \{\}) \\
\quad \quad \quad \quad \wedge\ (\mathcal{A}[\![f0\ xs'\ ys]\!]\ \{f0\ xs\ ys\}\ \{xs, ys, x, xs'\})]\!]\ \{\}\ \{\}
\end{array}
$$

$$
\begin{array}{l}
= \mathcal{T}[\![(\mathcal{T}[\![\mathit{True} \wedge (\mathcal{A}[\![\textbf{case } ys' \textbf{ of} \\
\quad \quad \quad \quad \mathit{Nil} \quad : \mathit{True} \\
\quad \quad \quad \quad \mid \mathit{Cons}\ y'\ ys'' : f1\ ys'']\!]\ \{f0\ xs\ ys, f1\ ys'\}\ \{xs, ys, y, ys'\})]\!]\ \{\}\{\}) \\
\quad \quad \quad \wedge\ (\mathcal{A}[\![f0\ xs'\ ys]\!]\ \{f0\ xs\ ys\}\ \{xs, ys, x, xs'\})]\!]\ \{\}\ \{\} \hfill (\text{by } \mathcal{A}2, \mathcal{A}6)
\end{array}
$$

$$
\begin{array}{l}
= \mathcal{T}[\![(\mathcal{T}[\![\mathit{True}\ \wedge\ (\mathcal{T}[\![(\mathcal{A}[\![\mathit{True}]\!]\ \{f0\ xs\ ys, f1\ ys'\}\ \{xs, ys, y, ys'\}) \\
\quad \quad \wedge\ (\mathcal{A}[\![f1\ ys'']\!]\ \{f0\ xs\ ys, f1\ ys'\}\ \{xs, ys, y, ys', y', ys''\})]\!]\ \{\}\ \{\})]\!]\ \{\}\ \{\}) \\
\quad \quad \wedge\ (\mathcal{A}[\![f0\ xs'\ ys]\!]\ \{f0\ xs\ ys\}\ \{xs, ys, x, xs'\})]\!]\ \{\}\ \{\} \hfill (\text{by } \mathcal{A}5)
\end{array}
$$

$$
\begin{array}{l}
= \mathcal{T}[\![(\mathcal{T}[\![\mathit{True}\ \wedge\ (\mathcal{T}[\![\mathit{True}\ \wedge\ \mathit{True}]\!]\ \{\}\ \{\})]\!]\ \{\}\ \{\}) \\
\quad \quad \quad \wedge\ (\mathcal{A}[\![f0\ xs'\ ys]\!]\ \{f0\ xs\ ys\}\ \{xs, ys, x, xs'\})]\!]\ \{\}\ \{\} \hfill (\text{by } \mathcal{A}2, \mathcal{A}7)
\end{array}
$$

$$
\begin{array}{l}
= \mathcal{T}[\![(\mathcal{T}[\![\mathit{True}\ \wedge\ \mathit{True}]\!]\ \{\}\ \{\})\ \wedge\ (\mathcal{A}[\![f0\ xs'\ ys]\!]\ \{f0\ xs\ ys\}\ \{xs, ys, x, xs'\})]\!]\ \{\}\ \{\} \\
\hfill (\text{by } \mathcal{T}1, \mathcal{P})
\end{array}
$$

$$= \mathcal{T}[\![True \ \wedge \ (\mathcal{A}[\![f0 \ xs' \ ys]\!] \ \{f0 \ xs \ ys\} \ \{xs, ys, x, xs'\})]\!] \ \{\} \ \{\} \qquad \text{(by } \mathcal{T}1,\mathcal{P}\text{)}$$
$$= \mathcal{T}[\![True \ \wedge \ True]\!] \ \{\} \ \{\} \qquad \text{(by } \mathcal{A}7\text{)}$$
$$= True \qquad \text{(by } \mathcal{T}1,\mathcal{P}\text{)}$$

We obtain the truth value *True* as required. This completes the proof of conjecture (1), and demonstrates that it is a theorem.