Proof Planning Coinduction

Louise Dennis

Ph.D.
Department of Artificial Intelligence
University of Edinburgh
1998

Abstract

Coinduction is a proof rule which is the dual of induction. It allows reasoning about non–well–founded sets and is of particular use for reasoning about equivalences.

In this thesis I present an automation of coinductive theorem proving. This automation is based on the ideas of proof planning [Bundy 88]. Proof planning as the name suggests, plans the higher level steps in a proof without performing the formal checking which is also required for a verification. The automation has focused on the use of coinduction to prove the equivalence of programs in a small lazy functional language which is similar to Haskell.

One of the hardest parts in a coinductive proof is the choice of a relation, called a bisimulation. The automation here described makes an initial simplified guess at a bisimulation and then uses critics, revisions based on failure, and generalisation techniques to refine this guess.

The proof plan for coinduction and the critic have been implemented in *CLAM* [Bundy *et al* 90b] with encouraging results. The planner has been successfully tested on a number of theorems. Comparison of the proof planner for coinduction with the proof plan for induction implemented in *CLAM* [Bundy *et al* 91] has highlighted a number of equivalences and dualities in the process of these proofs and has also suggested improvements to both systems.

This work has demonstrated not only the possibility of fully automated theorem provers for coinduction but has also demonstrated the uses of proof planning for comparison of proof techniques.

I declare that this thesis has been composed by myself and that the work described is my own.

Louise A. Dennis

Acknowledgements

First and foremost I should like to thank my supervisors Alan Bundy and Ian Green without whose help at every stage of the process this PhD would never have been completed.

Thanks to the members of the Dream group in Edinburgh and the Languages and Programming group in Nottingham for help, comments and general support. In particular thanks to Colin Taylor for reading through the proofs in chapter 10 and Mark Jones for being a superb mentor during my year at Nottingham.

Thanks also to Andy Gordon, Graham Collins, Marcello Fiore, Jacob Frost and Larry Paulson all of whom have answered various queries I have had. Andy Gordon deserves a special mention here for reading parts of the thesis in advance, commenting on them and giving me a lot of help in understanding operational semantics.

Lastly my family and especially Bill.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Overview

This thesis describes a proof strategy for coinduction. This strategy is based on the idea of *proof planning* [Bundy 88] and determines the high–level steps required in a coinductive proof.

This proposed strategy has been implemented as a program, *CoClAM*, that will automatically generate a plan of the object–level inference steps required by a coinductive proof of the equivalence of two functional programs.

The program has been tested on examples using the operational semantics of a small functional language which is similar to ones proposed by Gordon [Gordon 95a].

The strategy draws upon ideas from theorem proving for induction. It takes the idea of proof planning, together with the associated idea of a proof critic. It also borrows the proof method rippling. It uses a generalisation critic, based on work by Walsh [Walsh 96] for an implicit induction prover, to refine the choice of bisimulation.

The thesis also compares the process of inductive and coinductive proof in the light of the proof strategies developed for each technique.

1.2 Motivation

This thesis attempts a first step towards the provision of a fully automated theorem prover for coinduction.

The past ten years have witnessed an increased interest in techniques for describing and proving the properties of infinite structures and processes. Among the foremost of these are bisimulation relations and bisimilarity (a form of equality on infinite objects). This has promoted interest in the proof principle of coinduction (proving objects to be bisimilar). Coinduction is a dual to induction. Duals are of great interest, particularly in category theory. Other duals related to coinduction are coalgebras (a categorical description of infinite objects) and corecursion (a definition principle for functions on infinite objects).

Infinite processes arise naturally in computer science. They were first investigated seriously in the field of concurrency (e.g. [Park 70]) where looping finite state machines are commonplace. However, they are also potentially present in any looping program and are useful when considering Input/Output issues. For instance, a lazy or call–by–need evaluation procedure only evaluates functions when they are required and may not fully evaluate[1] them, being content to evaluate, for instance, only the first element of a list, leaving the rest unevaluated until it is needed. In this way a potentially infinite process may be present in a program without forcing the entire program to be non–terminating. Programming languages which exploit these ideas are often referred to as lazy languages. Their semantics are generally expressed in an operational style. This thesis concentrates on the use of coinduction with the operational semantics of lazy functional languages.

The interest in what can broadly be described as coalgebraic methods in computer science and specifically in coinduction has lead to the development of various proof tools to aid in their use. These tools are a part of the general growth of theorem proving tools. The need for rigorous proof in formal methods has long been recognised. The problem has always been that such proofs are long, tedious and (when done by hand) particularly susceptible to error (for these reasons). These problems have been partly responsible for the development of a number of proof tools which range from proof checkers (which verify the correctness of each step) to tools which attempt to perform some or all of the required proof steps automatically thus removing much of the tedium for the user.

The first and most widely used of the proof tools for coinduction is the Edinburgh Concurrency Workbench (CWB) [Cleaveland *et al* 89] which was developed for use with CCS [Milner 89], but there has also been an effort to incorporate

[1]This assertion is not strictly accurate since lazy languages use a different notion of a value from strict languages. "Fully evaluate" is used here as it is used in strict languages.

tools for coinduction in several large theorem proving environments including HOL [Gordon & Melham 93], Isabelle [Paulson 94a] and PVS [Owre *et al* 96]. CWB is an automated tool but only deals with ground terms and finite bisimulations. None of the others make any attempt at full automation and rely on user input. At the very least they require the user to supply a bisimulation and they often require the user to guide other steps in the proof as well.

One of the inspirations for this work is the observation that not only is coinduction the theoretical dual of induction but there are observed similarities or dualities in the proof process. This suggests that inspiration can be drawn from inductive theorem provers when attempting to provide heuristics for coinductive proof. There are several theorem provers which attempt to fully automate inductive proof. The first and most well–known of these is the Boyer–Moore theorem prover, [Boyer & Moore 90]. This prover is essentially a black box and the techniques it uses for guiding proof attempts are not visible to the user. It also uses a powerful hints mechanism for many proofs. *CLAM* [Bundy *et al* 90b] is in some ways a successor of the Boyer–Moore theorem prover since it employs a powerful heuristic, called *rippling* [Bundy *et al* 93], which was developed as a rational reconstruction of the techniques used in Boyer–Moore. *CLAM* also represents a departure from previous provers, such as Boyer–Moore, which attempted to combine the rigorous object–level proof steps with the search for a proof. *CLAM* explicitly separates these processes, first of all *planning* the proof using descriptions of the effects of chains of object level rules and subsequently passing the plan to an object–level theorem prover. This process is called *proof planning*. *CLAM* was chosen as the basis for the work presented here since the techniques are clearly described, unlike the techniques employed by Boyer–Moore (and in its descendant NQTHM) which are hidden from the user, and it provides an environment in which new techniques may be developed. The work described is intended to be general enough for use in any proof planning system, however the implementation was all done in *CLAM*.

1.3 Contribution

The main contribution of this work can be found in chapters 5 and 6. These describe a process by which a bisimulation for coinduction can be found and by which the various phases of a coinductive proof can be automatically performed. The central contributions can be summarised as follows:

- The most important part of the work is demonstrating that the choice of bisimulation relation in coinductive proof can, in many cases, be performed automatically; I have provided a model of how to do this and implemented it in *CLAM*3.
- I have developed a number of new heuristics for forming such bisimulations, one of which draws on work on generalisation in the fields of term rewriting and implicit induction and so provides another application for such techniques.

- In the course of developing this model, I have also developed a proof strategy for coinduction which allows many coinductive proofs to be performed fully automatically.

Chapters 5 and 6 also contain some other contributions one of which is theoretical and two others which add support to the proof planning methodology.

- I have provided some new theoretical results about the smallest bisimulation required to prove a given theorem.
- I have shown that the rippling heuristic, which was developed for induction, is also of use for coinduction. I've also used the idea of *difference matching* that underlies rippling in forming bisimulations. This supports the belief that identification and manipulation of differences between expressions is an important part of theorem proving.
- I've shown that the ideas listed as major contributions can be implemented naturally in a proof planning system which provides support for that methodology of automating poof.

In chapter 10 I discuss a further program of work evolving from a comparison of induction and coinduction. This gives rise to a couple of additional contributions which are outside of the main thrust of the thesis.

- I've postulated that not only are the proof principles of induction and coinduction theoretically linked, but that there are also links in the practical process of such proofs. I have attempted to draw out these links in the light of the proposed proof strategies. In the course of this I believe I have contributed to the general understanding of the nature of coinductive proof.
- I have provided theoretical results about the links between the choice of induction scheme and the choice of an appropriate *labelled transition system* when performing induction on programs in lazy functional languages.

1.4 The Organisation of this Thesis

The rest of this thesis is organised as follows:

- In the next chapter, chapter 2, I discuss coinduction and the mathematical theory behind it. I also briefly survey some of the current proof tools available for coinduction.
- Chapter 3 discusses the use of coinduction in the operational semantics of functional languages and presents several examples of coinductive proof in this setting in order to motivate the heuristics proposed in chapters 5 and 6.

- Chapter 4 introduces proof planning.
- Chapters 5 and 6 present a generic proof plan or proof strategy for coinduction.
- Chapter 7 reports the result of experimental testing of this proof strategy.
- Chapter 8 examines related work in both the provision of proof tools for coinduction and in generalisation techniques.
- Chapter 9 discusses proposed modifications and extensions to the proof strategy.
- Chapter 10 compares the new proof strategy for coinduction and the proof strategy for induction and examines the observed similarities between the two proof processes.
- Chapter 11 evaluates the achievements and contribution of the work here reported.
- There are 6 appendices: A glossary of terms; A detailed breakdown of the theorems proved by the system; Transcripts of runs of the implementation; some meta–theorems are proved which are used in the course of coinductive proofs; a discussion of work linking *CoCL^AM* to an object level theorem prover, Isabelle and lastly a comparison of the proofs of several theorems using induction and coinduction.

Chapter 2

Background

2.1 Introduction

The purpose of this chapter is to provide a general survey of work on coinduction.

It starts with a very general discussion of the areas where coinduction is used. This is followed by a discussion of fixedpoints and a statement of the coinduction rule.

It then looks at the three areas of Communicating Systems, Non–Well–Founded Sets and Functional Programming, all of which use the idea of bisimulation, which is central to coinduction. Each section aims to describe the sort of objects coinduction is used to reason about and provide the statement of the coinduction principle for that area. Some of the discussion is illustrated with examples.

§2.5, 2.6 and 2.7 are more general and look at coinduction proof principles, the relationship of coinduction to congruence, recent attempts to unify the various developments of coinductive principles within category theory and coinductive definitions.

Lastly the use of coinduction, and the development of tools to support coinduction in various theorem provers is examined.

2.2 Construction versus Observation

Jacobs and Rutten [Jacobs & Rutten 97] identify a pervasive distinction in computer science between construction and observation. This is similar to the distinction between induction and coinduction.

Inductive datatypes exploit the idea that every element of the type can be constructed from some set of base elements. Induction rules come in many forms, but all of them hinge on the fact that the set of objects under consideration is partially ordered by some *well–founded* order. An order is said to be well–founded if there are no infinite descending chains. Induction is a natural proof principle to use for "constructed" datatypes since the process of construction supplies a well–founded relation on the type[1].

However, it is not the case that all datatypes naturally occurring in computer science conform to this construction style of definition. Consider a ticket machine which has a roll of printed tickets and a button. Pressing the button causes the machine dispense a ticket. If the machine runs out of tickets, it stops (some final state is reached). It is far more natural to discuss this machine in terms of observing what the ticket is, and observing the state of the machine after it has dispensed a ticket, than in terms of how one ticket or state of the machine is formed from the previous ticket and state. The problem with trying to prove any formal properties for the machine described in terms of observation is that the natural order imposed by the description isn't necessarily well–founded. It is theoretically possible to have machines that never reach their final state (for instance, in the example, the roll of tickets might be refilled regularly before the previous roll runs out). In fact, computer programs fail to terminate often enough for it to be a common part of computing. This sort of non–termination need not be "buggy", for instance, it is undesirable behaviour in the ticket machine if it runs out of tickets.

Similarly, in functional languages with lazy evaluation it becomes possible to define functions that won't terminate in a non–catastrophic way since instead of trying to evaluate the whole function the program will only evaluate those parts it actually needs, e.g. the heads of lists.

Coinduction as a proof principle handles non–well–founded datatypes and so has come to be the preferred method of proof in certain domains. It is possible to present coinductive problems in such a way that induction can be used in the process of proof. This is done by providing an order in which one state is less than another by being closer to the start state, rather than the final state (which is how induction is more commonly used, but which, of course, may not exist in

[1] Though the relation imposed by type constructors may not be the one used to justify any particular inductive proof involving the datatype.

a lazy setting). Coinduction, nevertheless, is increasingly felt to be more elegant and simple as a proof principle for certain classes of problem.

2.3 The Theory of Coinduction and Recursive Datatypes

The first approach to modelling the sort of infinite processes described in §2.2 was based on the theory of fixedpoints. The study of fixedpoints grew up out of Tarksi's work [Tarski 55].

Definition 2.1 *A* **fixedpoint** *of a function $\mathcal{F}$ is an element, D, of its domain such that*

$$\mathcal{F}(D) = D$$

Definition 2.2 *A function, f, on sets is* **monotone** *if*

$$\forall A, B. A \subseteq B \Rightarrow f(A) \subseteq f(B)$$

Definition 2.3 *A* **lattice** *is a partially ordered set in which for all pairs of elements, s and t in the lattice, there exist elements sup and inf in the lattice such that $s \leq sup$, $t \leq sup$, $inf \leq s$ and $inf \leq t$. Moreover for any element z in the lattice if $s \leq z$ and $t \leq z$ then $sup \leq z$ and if $z \leq s$ and $z \leq t$ then $z \leq inf$.*

Tarksi showed that a monotone function has a lattice of fixedpoints which has a greatest and least element. The least fixed point is associated with the induction rule, the greatest fixed point with the coinduction rule.

2.3.1 Least Fixed Points and Induction

The least fixed point is defined as [Paulson 93]

$$lfp(\mathcal{F}) = \bigcap \{\mathcal{A} \mid \mathcal{F}(\mathcal{A}) \subseteq \mathcal{A}\} \tag{2.1}$$

Induction acts as an elimination rule for least fixed points. Inductive domains (e.g. the natural numbers) can be regarded as the least fixed points of monotone functions. A general form of the induction rule is [Frost 95]

$$\frac{\mathcal{F}(\mathcal{A}) \subseteq \mathcal{A}}{lfp(\mathcal{F}) \subseteq \mathcal{A}} \tag{2.2}$$

A more usual form of the rule comes from first expressing $\mathcal{A}$ in terms of some property P.

$$\frac{\mathcal{F}(\{x|P(x)\}) \subseteq \{x|P(x)\}}{lfp(\mathcal{F}) \subseteq \{x|\ P(x)\}} \tag{2.3}$$

then expanding the definition of $\mathcal{F}$ in $\mathcal{F}(\{x|P(x)\}) \subseteq \{x|P(x)\}$ and performing some inference on the resulting term.

Example 2.1 *Let* $\mathcal{F}(\mathcal{R}) \stackrel{\text{def}}{=} \{\bot\} \cup \{f(x) | x \in \mathcal{R}\}$.

In this case an arbitrary element of $\mathcal{F}(\{x|P(x)\})$ *is of the form* $\bot$ *or* $f(x), x \in \{x|P(x)\}$. *If* $\mathcal{F}(\{x|P(x)\}) \subseteq \{x|P(x)\}$ *then* $\bot \in \{x|P(x)\}$, *i.e.* $P(\bot)$ *and* $f(x) \in \{x|P(x)\}$ *when* $x \in \{x|P(x)\}$ *(i.e.* $P(x) \Rightarrow P(f(x))$*). This gives the expression* $P(\bot) \wedge P(x) \Rightarrow P(f(x))$ *(the base and step case of an induction).*

To specialise still further, if $\bot \equiv 0$ *and* $f \equiv s$ *then* $\mathcal{F}(\mathcal{S}) \stackrel{\text{def}}{=} \{0\} \cup \{s(x) | x \in \mathcal{S}\}$. *Similarly if* $\bot \equiv nil$ *and* $f \equiv ::$ *then* $\mathcal{F}(\mathcal{S}) \stackrel{\text{def}}{=} \{nil\} \cup \{h :: t | t \in \mathcal{S}\}$ *these give standard induction schemes associated with natural numbers and strict lists.*

2.3.2 Greatest Fixed Points and Coinduction

The greatest fixed point is defined as [Paulson 93]

$$gfp(\mathcal{F}) = \bigcup \{\mathcal{A} \mid \mathcal{A} \subseteq \mathcal{F}(\mathcal{A})\} \tag{2.4}$$

Coinduction acts as an elimination rule for greatest fixed points. Coinductive domains are the greatest fixed points of monotone functions. The general coinduction rule is [Frost 95]

$$\frac{\mathcal{A} \subseteq \mathcal{F}(\mathcal{A})}{\mathcal{A} \subseteq gfp(\mathcal{F})} \tag{2.5}$$

A more usual form is [Paulson 93]

$$\frac{a \in \mathcal{A} \quad \mathcal{A} \subseteq \mathcal{F}(\mathcal{A})}{a \in gfp(\mathcal{F})} \tag{2.6}$$

The premise $\mathcal{A} \subseteq \mathcal{F}(\mathcal{A})$ is the dual of $\mathcal{F}(\mathcal{A}) \subseteq \mathcal{A}$ in the induction rule, which was specialised to give some common induction rules. This process can also be applied here.

Example 2.2 *Take the definition of* $\mathcal{F}$ *from example 2.1. Let* x *be an arbitrary element of* $\mathcal{A}$. $\mathcal{A} \subseteq \mathcal{F}(\mathcal{A})$ *if* x *is also in* $\mathcal{F}(\mathcal{A})$. *So either it is* $\bot$ *or there is some* $y \in \mathcal{A}$ *such that* $x = f^{-1}(y)$ *(always assuming, of course, that* f *has a well–defined inverse).*

Thus the expression $\forall x \in \mathcal{A}. x = \bot \vee \exists y \in \mathcal{A}. x = f^{-1}(y)$ *could replace the premise,* $\mathcal{A} \subseteq \mathcal{F}(\mathcal{A})$, *in the coinduction rule effectively specialising it to one particular* $\mathcal{F}$.

It is unusual to see the coinduction rule tailored in this way (unlike the induction rule) and so I have chosen to keep the format of (2.6) when discussing the coinduction rule simply instantiating $\mathcal{F}$ and $gfp(\mathcal{F})$ where appropriate rather than unpacking the proof conditions for each $\mathcal{F}$ as a variety of coinduction rules.

The term "coinduction" is generally believed to have first been used by Milner and Tofte [Milner & Tofte 91]. They defined a greatest fixedpoint that relates the values and types of a small functional language and used coinduction in part of the proof of the consistency of its static and dynamic semantics. However, before it was so named, the coinduction rule was already being used in a number of guises in mathematics and computer science.

2.4 Background and Development

The eariest appearance of bisimulations, a common type of object that arises out of coinduction, was in the field of automata. These ideas were picked up on by workers in other fields and adapted to their use. The most notable of these were the fields of concurrency and functional programming. Early work was also done in set theory. This section will examine the early development of bisimulations in all three of these fields.

2.4.1 Communicating Systems

Park [Park 80,Park 81] first suggested the use of *bisimulations* to represent an idea of equality between concurrent systems. He defines simulations, $\lesssim$, and bisimulations, $\sim$, as follows. Let $\mathcal{M}$ and $\mathcal{M}'$ be *finite automata* over a set, Σ, represented by structures of the form

$$\mathcal{M} = \langle S, s_0, M, F \rangle, \mathcal{M}' = \langle S', s'_0, M', F' \rangle$$

Where S, S' are the *state sets* of $\mathcal{M}$ and $\mathcal{M}'$, $s_0 \in S$ and $s'_0 \in S'$ the *start states*, $F \subseteq S$ and $F' \subseteq S'$ the *accept states* and $M : S \times (\Sigma \cup \{\lambda\}) \to \mathcal{P}(S)$ and $M' : S' \times (\Sigma \cup \{\lambda\}) \to \mathcal{P}(S')$ are *transition functions*, where $\mathcal{P}(S)$ is the power set of S and λ is the null string.

$\mathcal{M}$ *simulates* $\mathcal{M}'$ via $\mathcal{R}$ (written $\mathcal{M} \lesssim^{\mathcal{R}} \mathcal{M}'$) if $\mathcal{R} \subseteq S \times S'$ and

1. $\langle s_0, s'_0 \rangle \in \mathcal{R}$
2. $s \in F, \langle s, s' \rangle \in \mathcal{R} \Rightarrow s' \in F'$
3. $\sigma \in \Sigma \cup \{\lambda\}, \langle s_1, s'_1 \rangle \in \mathcal{R}, s_2 \in M(s_1, \sigma) \Rightarrow \langle s_2, s'_2 \rangle \in \mathcal{R}$ for some $s'_2 \in M'(s'_1, \sigma)$

$\mathcal{M}$ *bisimulates* $\mathcal{M}'$ via $\mathcal{R}$ (written $\mathcal{M} \sim^{\mathcal{R}} \mathcal{M}'$) if $\mathcal{M} \lesssim^{\mathcal{R}} \mathcal{M}'$ and $\mathcal{M}' \lesssim^{\mathcal{R}} \mathcal{M}$.

In essence one automaton bisimulates another via some $\mathcal{R}$ if for all accept or start states s in $\mathcal{M}$, s is related to some accept or start state s' in $\mathcal{M}'$ and all pairs of states obtained by transitions from related pairs of states in $\mathcal{R}$ are also in $\mathcal{R}$.

CCS and the Bisimulation Proof Method

Milner built on these foundations [Milner 89] to provide a semantics for his Calculus for Communicating Systems (CCS) and used it to prove that various concurrent systems matched their specifications.

Milner's calculus consists of a set of *states* (or *agents*), $\mathcal{Q}$, a set of *labels* (or *actions*), $\mathcal{L}$, and a *transition relation* between states. States are represented by capital letters and labels (or *actions*) by lower case letters. Any label, a, has a *co–label*, $\bar{a} \in \mathcal{L}$, such that $\bar{\bar{a}} = a$.

Example 2.3 *Consider a cell with two ports a and $\bar{c}$. It can hold one data item. When the cell is empty it is in state A and when it holds a data item it is in state A'. Data may be placed in the cell through port a (by action a) and removed*[2] *through port $\bar{c}$ (by action $\bar{c}$). Hence the cell can be represented by the two equations:*

$$A \stackrel{\text{def}}{=} a.A' \tag{2.7}$$

$$A' \stackrel{\text{def}}{=} \bar{c}.A \tag{2.8}$$

The expressions, $a.A'$ and $\bar{c}.A$ are *agent expressions*. An agent expression of the form $a.A$ is interpreted as meaning "perform a then proceed according to the definition of A". The transitions between states are accomplished by actions. The set of labelled arrows $\xrightarrow{\alpha}$ where α is an action form the transition relation. So in example 2.3, $A \xrightarrow{a} A'$.

Definition 2.4 *[Milner 89] If for some action $\alpha \in \mathcal{L}$ and some states $P_1, P_2 \in \mathcal{Q}$ $P_1 \xrightarrow{\alpha} P_2$, then P_2 is said to be a* **derivative** *of P_1.*

In the example A' is a derivative of A and A is a derivative of A'

Definition 2.5 *[Milner 89] The set, $\mathcal{E}$, of agent expressions is the smallest set which includes the agent variables and constants (states) and contains the following expressions, where E, E_i are already in $\mathcal{E}$*

1. *$\alpha.E$, a* **Prefix** *(α is an action)*
2. *$\Sigma_{i \in I} E_i$, a* **Summation** *(I is some indexing set). If $I = \{1, 2\}$ then this is written $E_1 + E_2$. The "inactive agent", capable of no action, is $\Sigma_{i \in \emptyset} E_i$. This is usually represented by the symbol* **0**.
3. *$E_1|E_2$, a* **Composition**
4. *$E \backslash L$, a* **Restriction** *where L is a set of actions*
5. *$E[f]$, a* **Relabelling**

The Summation combinator combines two agents expressions as alternatives. The Composition combinator represents parallelism and is probably best described through an example

[2] "output" ports are, by convention, represented by co–labels of input ports. The purpose of the Calculus is to model communicating systems, hence it is reasonable to assume that an output from one agent will often be matched by an input to another.

Example 2.4 *[Milner 89] Consider linking two agents, A and B together where A and B are defined by*

$$\begin{array}{ll} A \stackrel{\text{def}}{=} a.A' & B \stackrel{\text{def}}{=} c.B' \\ A' \stackrel{\text{def}}{=} \bar{c}.A & B' \stackrel{\text{def}}{=} \bar{b}.B \end{array}$$

These are shown in figure 2–1.

Figure 2–1: A Simple Communicating System

This represents a system of two cells. The first cell is empty and may input a data item (action a) once it is holding a data item it can output it (action $\bar{c}$). Similarly the second cell may also either input a data item (action c) or output one (action $\bar{b}$). The other actions can occur in either cell at any time

This is represented as the composition $A|B$. If A can do an action then it can also do it in $A|B$, leaving B undisturbed (i.e. since $A \xrightarrow{a} A'$ $A|B \xrightarrow{a} A'|B$). Similarly $A|B' \xrightarrow{\bar{b}} A|B$.

Consider joining the actions c and $\bar{c}$ so they can not be performed alone This situation of co–occurring actions may be written as the distinguished action τ. τ actions can not be affected or observed by any user since they occur internally within the system. A τ action is a complementary pair (e.g. $(c, \bar{c})$). A τ transition on a composed pair of states leads to the composed pair of the derivatives of those states. In this case $A'|B \xrightarrow{\tau} A|B'$. τ actions are an important concept in the calculus.

The Restriction combinator internalises a set of ports determined by a set of names. Ports may be joined to more than one other port; if they are restricted, however, they are not available any more. If a port's name or co–name appears in the set then the label for this port is removed in any representation of the system. For instance the description of the system in example 2.4 states that c can only occur if $\bar{c}$ also occurs. This implies that there is nowhere else that the output from the cell can go except as input to the next cell. Hence no more ports can be linked with c otherwise there would be a choice of where the data might go. Hence the system described is $(A|B)\backslash\{c\}$ which is illustrated by figure 2–2. Crucially the system in figure 2–2 performs the pair of actions $(c, \bar{c})$ as a τ action.

Sometimes it may be necessary to change the names of ports in a system. This is represented by the use of the Relabelling combinator which requires the presence of a relabelling function which specifies how the names are to be changed

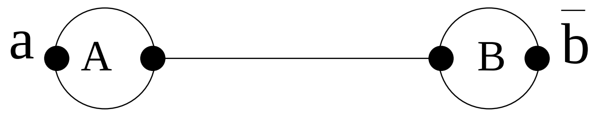

Figure 2–2: $(A|B)\backslash\{c\}$

(e.g. $[a/input]$ is a relabelling function that relabels the actions a in an agent expression as *input*).

Definition 2.6 *[Milner 89] A binary relation $\mathcal{S} \subseteq \mathcal{P} \times \mathcal{P}$ over agents is a* **strong bisimulation** *if $(P, Q) \in \mathcal{S}$ implies, for all $\alpha \in Act$,*

1. Whenever $P \xrightarrow{\alpha} P'$ then, for some $Q', Q \xrightarrow{\alpha} Q'$ and $(P', Q') \in \mathcal{S}$
2. Whenever $Q \xrightarrow{\alpha} Q'$ then, for some $P', P \xrightarrow{\alpha} P'$ and $(P', Q') \in \mathcal{S}$

Definition 2.7 *[Milner 89] P and Q are* **strongly equivalent**, *written $P \sim Q$, if $(P, Q) \in \mathcal{S}$ for some strong bisimulation $\mathcal{S}$.*

Theorem 2.1 *[Milner 89]*

1. $\sim$ is the largest strong bisimulation.

2. $\sim$ is an equivalence relation.

Example 2.5 *[Milner 89] To show an example of a coinductive proof in CCS consider the system comprising two cells A and B each with two ports joined together as the system $(A|B)\backslash\{c\}$ described in figure 2–2. Recall this is represented by the equations.*

$$\begin{array}{ll} A \stackrel{\text{def}}{=} a.A' & B \stackrel{\text{def}}{=} c.B' \\ A' \stackrel{\text{def}}{=} \bar{c}.A & B' \stackrel{\text{def}}{=} \bar{b}.B \end{array}$$

Prove that $(A|B)\backslash\{c\}$ is equivalent to the transition graph in figure 2–3.

This graph can be described by the equations

$$\begin{array}{lcl} C_0 & \stackrel{\text{def}}{=} & \bar{b}.C_1 + a.C_2 \\ C_1 & \stackrel{\text{def}}{=} & a.C_3 \\ C_2 & \stackrel{\text{def}}{=} & \bar{b}.C_3 \\ C_3 & \stackrel{\text{def}}{=} & \tau.C_0 \end{array}$$

Proof: *The proof of this proceeds by finding a strong bisimulation which contains the pair $\langle (A|B)\backslash\{c\},\ C_1 \rangle$. This is generally called the* **bisimulation proof method**.

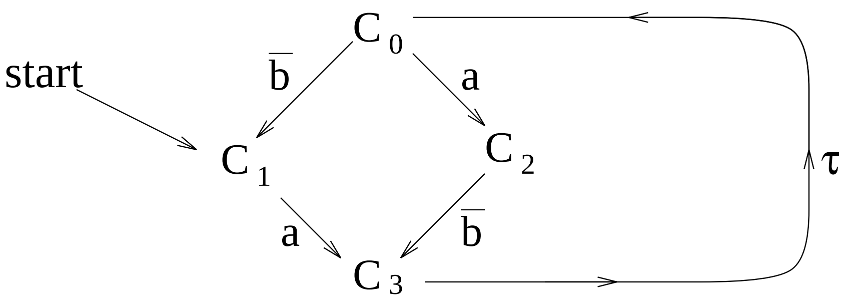

Figure 2–3: The Transition Graph for $A|B$

Let the relation $\mathcal{S}$ be

$$\begin{aligned}\mathcal{S} \;=\; & \{\langle (A|B)\backslash\{c\},\, C_1\rangle, \\ & \langle (A'|B)\backslash\{c\},\, C_3\rangle, \\ & \langle (A|B')\backslash\{c\},\, C_0\rangle, \\ & \langle (A'|B')\backslash\{c\},\, C_2\rangle\}\end{aligned}$$

By inspection it can be seen that the derivatives of each pair of states in $\mathcal{S}$ are also in $\mathcal{S}$, i.e.

$(A\|B)\backslash\{c\}$ C_1	$\xrightarrow{a}$ $\xrightarrow{a}$	$(A'\|B)\backslash\{c\}$ C_3
$(A'\|B)\backslash\{c\}$ C_3	$\xrightarrow{\tau}$ $\xrightarrow{\tau}$	$(A\|B')\backslash\{c\}$ C_0
$(A\|B')\backslash\{c\}$ C_0	$\xrightarrow{\bar{b}}$ $\xrightarrow{\bar{b}}$	$(A\|B)\backslash\{c\}$ C_1
$(A\|B')\backslash\{c\}$ C_0	$\xrightarrow{a}$ $\xrightarrow{a}$	$(A'\|B')\backslash\{c\}$ C_2
$(A'\|B')\backslash\{c\}$ C_2	$\xrightarrow{\bar{b}}$ $\xrightarrow{\bar{b}}$	$(A'\|B)\backslash\{c\}$ C_3

hence that $\mathcal{S}$ is a bisimulation and the two systems are bisimilar.

This proof method is a form of coinduction, though the term was not used by Milner at the time.

Milner uses coinduction to derive a number of laws that can be used to prove bisimilarity. Hence many such proofs don't involve the method directly.

The proof method seems intuitively reasonable since, if you can find a suitable bisimulation it says that if you start at the same "point" in either representation equivalent actions will take you to corresponding points for an arbitrarily large (though) finite sequence of actions.

2.4.2 Non-Well-Founded Sets

The notions of bisimulation originated in concurrency theory, however Aczel lifted the idea and transported it into set theory in order to study non-well-founded sets formulated within a version of *ZF* [Aczel 88]. These are sets containing infinite sequences of nested subsets. For instance the set

$$\Omega \equiv \{\Omega\}$$

These had been classified as extraordinary by mathematicians and excluded from set theory. Aczel developed what he called the Anti-Foundation Axiom (AFA), which asserts the existence of non-well-founded sets.

Aczel's formulation represents sets as graphs. For instance, consider the standard ZF construction of the natural numbers

$$\begin{aligned} 0 &= \emptyset \\ 1 &= \{\emptyset\} \\ 2 &= \{\emptyset,\ \{\emptyset\}\} \\ 3 &= \{\emptyset, \{\emptyset,\ \{\emptyset\}\}\} \\ &\vdots \end{aligned}$$

These are normally represented as trees grown downwards as in figure 2–4.

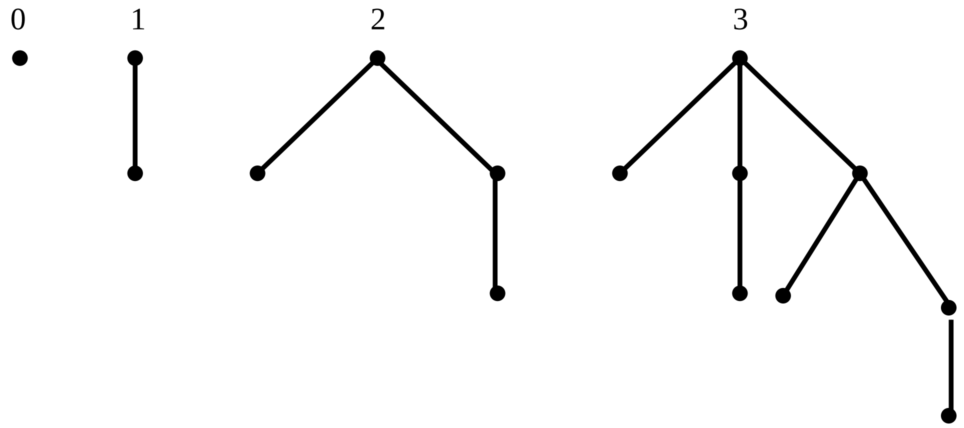

Figure 2–4: The Natural Numbers Represented as Trees

However Aczel advocates representing them as directed graphs or *accessible pointed graphs (apgs)* (figure 2–5). The "top" node of the graphs is the "object" under consideration. Arrows indicate the subset relationship. For instance 2 has 0 and 1 as subsets which is indicated in the graph by arrows from the node labelled "2" to the nodes labelled "1" and "0" ($2 \rightarrow 1$ and $2 \rightarrow 0$). However "1" also contains "0" as a subset so there is an arrow from the node labelled "1" to the node labelled "0" as well ($1 \rightarrow 0$). Hence Ω can be represented with one node and one arrow leading from and to that node as in figure 2–6.

The anti-foundation axiom states:

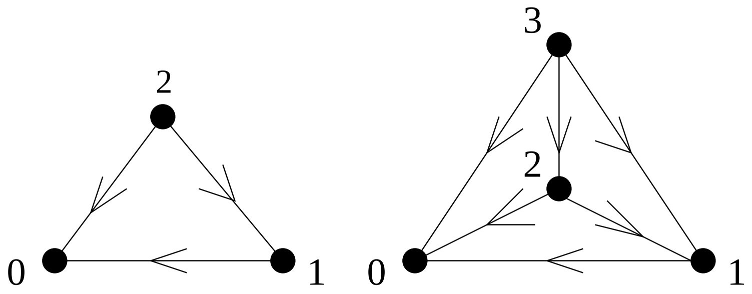

Figure 2–5: 2 and 3 Represented as Graphs

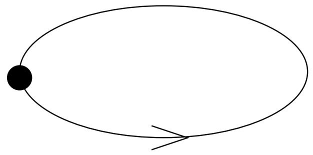

Figure 2–6: Ω represented as a Graph

Every apg has a unique labelling (up to isomorphism).

Aczel proved that ZF + AFA has a model.

Aczel uses bisimulations to prove equality between graphs. He introduces an equivalence $\equiv$ such that for sets a and b, $a \equiv b$ if and only if there is an apg that is a picture of both of them. It is possible to show from this definition that the anti foundation axioms implies

$$a \equiv b \Rightarrow a = b$$

Aczel defines a bisimulation as follows:

Notation: $a_M = \{c \in M : a \to c\}$. e.g. Let $\mathcal{N}$ be the natural numbers, then $0_{\mathcal{N}} = \emptyset$, $1_{\mathcal{N}} = \{0\}$, $2_{\mathcal{N}} = \{0, 1\}$ and $\Omega_{\mathcal{N}} = \Omega$ etc.

Definition 2.8 *[Aczel 88] A binary relation $\mathcal{R}$ on the system M is a* **bisimulation** *on M if $\mathcal{R} \subseteq \mathcal{R}^+$, where $\mathcal{R}^+$is defined for $a, b \in M$ by*

$$\begin{aligned} a\mathcal{R}^+b \quad \Leftrightarrow \quad & (\forall x \in a_M . \exists y \in b_M .\, x\mathcal{R}y)\ \& \\ & (\forall y \in b_M . \exists x \in a_M\ x\mathcal{R}y) \end{aligned} \tag{2.9}$$

Theorem 2.2 *[Aczel 88] If $\mathcal{R}$ is a bisimulation then $a\mathcal{R}b \Rightarrow a \equiv b$*

This is the coinduction principle for non well–founded sets.

2.4.3 The Lazy λ–Calculus

Abramsky [Abramsky 90] noted the difference between the practice and theory of programming languages. The theory was represented by the λ–calculus [Barendregt 84], a central part of which involved the evaluation of expressions to *head normal form*.

Definition 2.9 *[Barendregt 84] A λ-term, M, is a* **head normal form***(HNF) if M is of the form*
$\lambda x_1 \cdots x_n . x M_1 \cdots M_m, n, m \geq 0$. *Where x is a variable (possibly one of the x_i, $i \leq n$ and the M_j are λ–terms (not necessarily in HNF)).*

Definition 2.10 *[Barendregt 84] If M is of the form*

$$M \equiv \lambda x_1 \cdots x_n . (\lambda x . M_0) M_1 \cdots M_m, n \geq 0, m \geq 1$$

then $(\lambda x . M_0) M_1$ is called the **head redex** *of M.*

Informally a λ–term is in head normal form when the term doesn't have a head redex.

The meaning of a λ–term depends upon its head normal form. Abramsky noted, however, that most programming languages did not take evaluation of terms to head normal form, but to *weak head normal form*:

Definition 2.11 *(Adapted from [Peyton Jones 87]) A λ-term M is a* **weak head normal form***(WHNF) if M is of the form*
$\lambda x . M'$ *or* $x M_1 \cdots M_m, n, m \geq 0$.

This mismatch between theory and practice was motivated by efficiency considerations (in particular to avoid problems of name–capture of free variables [Peyton Jones 87]).

Example 2.6 *[Abramsky 90]*

The classic unsolvable term (i.e. a term that has no head normal form) is $\perp \equiv (\lambda x . (xx) x)(\lambda x . (xx) x)$. In the standard theory [Barendregt 84]

$$\lambda x . \perp = \perp$$

since $\lambda x . \perp$ is also unsolvable. However $\lambda x . \perp$ is in weak head normal form and so was distinguished from $\perp$ in many functional languages.

Abramsky's development of the lazy λ–calculus used *applicative bisimulation* to define equivalence between λ–terms.

Definition 2.12 *[Abramsky 90]*

1. *The relation* $M \Downarrow N$ *(M converges to principal weak head normal form N) is defined inductively over closed lambda terms,* Λ^0*, as follows:*
   - $\lambda x.M \Downarrow \lambda x.M$
   - $\frac{M \Downarrow \lambda x.P \quad P[N/x] \Downarrow Q}{MN \Downarrow Q}$
2. M **converges**, $M \Downarrow$, *if* $\exists N.M \Downarrow N$
3. M **diverges**, $M \Uparrow$, *if* $\neg(M \Downarrow)$

This transition system, $(\Lambda^0, \Downarrow)$, can be generalised to the concept of a *quasi–applicative transition system.*

Definition 2.13 *[Abramsky 90] A* **quasi–applicative transition system** *(quasi–ats) is a structure* (A, ev) *where* $ev : A \rightharpoonup (A \rightarrow A)$ *(ev is a partial function from A to functions from A to A).*

Notation:

$$\begin{array}{rcl} a \Downarrow_{qats} b & \equiv & a \in dom(ev) \wedge ev(a) = b \\ a \Downarrow_{qats} & \equiv & a \in dom(ev) \\ a \Uparrow_{qats} & \equiv & a \notin dom(ev) \\ Rel(A) & \equiv & \text{the set of relations on } A \end{array}$$

Definition 2.14 *[Abramsky 90] Let* (A, ev) *be a quasi-ats. Define:*

$$F(\mathcal{R}) \stackrel{\text{def}}{=} \{\langle a,\ b\rangle : \exists f.a \Downarrow_{qats} f \Rightarrow (\exists g.b \Downarrow_{qats} g \wedge \forall c \in A.f(c)\, \mathcal{R}\, g(c))\}$$

$\mathcal{R} \in Rel(A)$ *is an* **applicative bisimulation** *iff* $\mathcal{R} \subseteq F(\mathcal{R})$*.*

Define $\lesssim$ *by:*

$$a \lesssim b \equiv a\mathcal{R}b \text{ for some applicative bisimulation } \mathcal{R}$$

Definition 2.15 *[Abramsky 90]*

$$a \sim b \stackrel{\text{def}}{=} a \lesssim b \wedge b \lesssim a$$

It is a fairly simple matter to prove that $\sim$ is an equivalence relation.

Definition 2.16 *[Abramsky 90] An* **applicative transition system (ats)** *is a quasi-ats* (A, ev) *satisfying*

$$\forall a, b, c \in A.a \Downarrow_{qats} a' \wedge b \lesssim c \Rightarrow a'(b) \lesssim a'(c)$$

Since $\sim$ is an equivalence relation this definition forces it to be a congruence in an applicative transition system.

Going back to the λ–calculus if $l = (\Lambda^0, ev)$ where

$$ev(M) = \begin{cases} P \mapsto N[P/x] & M \Downarrow \lambda x.N \\ \text{undefined} & \text{otherwise} \end{cases}$$

then l can be shown to be an ats and $\Downarrow$ is $\Downarrow_{qats}$. Hence $\sim$ can act as an equality relation between closed λ–terms with $\Downarrow$ acting as an evaluation relation.

Although Abramsky doesn't state a coinduction principle as such, an appropriate one is:

Theorem 2.3 *[Abramsky 90] If (A, ev) is an ats then for $a, b \in A$, $a = b$ if there is an applicative bisimulation $\mathcal{R}$ such that $a\mathcal{R}b$ and $b\mathcal{R}a$.*

2.5 Coinduction Principles

A Coinduction Principle is a statement of the applicability of coinduction. The form the principle takes varies, for instance it depends upon whether bisimulation is a congruence or not. Within the theory of lazy lists developed by Paulson for HOL [Paulson 93], equality between lazy lists is based on work by Bird and Wadler [Bird & Wadler 88] and is defined as

Definition 2.17 *Two lazy lists l_1 and l_2 are equal if for any finite k the first k elements of l_1 and l_2 viewed as a finite list are equal.*

This can be shown to be equivalent to the greatest fixedpoint of the function $llistD\_fun$:

$$llistD\_fun(\mathcal{R}) \stackrel{\text{def}}{=} \{\langle x :: l_1,\ x :: l_2\rangle \mid \langle l_1,\ l_2\rangle \in \mathcal{R}\} \cup \{\langle nil,\ nil\rangle\}$$

Hence the coinduction principle he employs is:

Theorem 2.4 *[Paulson 93] If l_1 and l_2 are lazy lists $l_1 \equiv l_2$ if and only if $\langle l_1, l_2\rangle \in gfp(llistD\_fun)$*

Using the more common terminology of bisimulations the theorem might be reconstructed as

Theorem 2.5 *If l_1 and l_2 are lazy lists $l_1 \equiv l_2$ if and only if there is some bisimulation $\mathcal{R}$ such that $l_1\mathcal{R}l_2$*

Theorems 2.1, 2.2 and 2.3 are all examples of coinduction principles.

Coinduction principles have been developed for recursively defined domains [Pitts 92], datatypes [Fiore 93], I/O in functional languages [Gordon 93], the operational semantics of lazy functional languages [Gordon 95a] and the semantics of object oriented languages [Gordon 96].

2.5.1 Congruence Proofs

The usefulness of bisimulation equivalences is that they can act as congruences. Sometimes they are straightforward congruences e.g. in many functional languages, and sometimes they have to be extended with some additional notion.

For instance Milner has three bisimulation–like structures. His CCS contains both external actions, which are ways external observers may interact with a system, and internal τ actions, which cannot be interfered with from outside. In his calculus there are several forms of bisimulation. *Strong* bisimulations are those which consider τ actions and insist that these must be matched on both sides of the relations. *Weak* bisimulations allow τ actions to be ignored to an extent, but lose congruence as a result. His final relation – *observation congruence* is a modification of weak bisimulation to restore the congruence.

There is always a need for anyone proposing to use coinduction to prove congruence in a given domain to show that bisimilarity will act as a congruence as well as an equivalence. Howe [Howe 89] developed a general procedure for proving the congruence of bisimilarity in functional languages.

2.6 Towards a General Theory

Although I shall not be discussing category theory elsewhere in this thesis, much of the recent theoretical work involving coinduction has used category theory. The theory presented in the following section is not required for an understanding of the work in this thesis, however it helps to explain what all the various coinduction principles have in common. In this way it contributes to an understanding of the extent to which the proof strategy presented is general and to what extent it is specific to the operational semantics of lazy functional languages.

Rutten [Rutten 96] noted that

> "induction principles are well–known and much used. The coinductive definition and proof principles for coalgebras are less well–known by far, and often even not very clearly formulated. . . . many families of systems look rather different from the outside, and so do the corresponding notions of bisimulation."

For this reason he has worked on developing a *Universal Coalgebra* to act as an abstraction of all such systems. In chapter 3, I discuss two coinduction rules. Both of these work on the theory of coinduction as an elimination rule for greatest fixedpoints but otherwise appear dissimilar. This serves as an illustration of the problem outlined by Rutten.

I intend to outline this general theory here in order to provide some idea of how the various presentations relate to each other. It is outside the scope of this thesis to present in full Rutten's work. What is being attempted is to put forward the central definitions and theorems.

2.6.1 Category Theory

Before embarking upon Rutten's work it is necessary to outline a few concepts from category theory. The intention here is to provide only such definitions as are actually needed in the discussion; for a fuller discussion of category theory see [Mac Lane 71].

Informally a category is a collection of objects with arrows between them. These arrows act a bit like functions, in fact the objects that the arrows map between are often referred to as the domain and codomain of the arrow. Arrows may be composed together like functions and each object has an identity arrow. For example, in the category of sets, the objects are sets, and the arrows are functions between sets. Formal definitions follow:

Definition 2.18 *[Aczel 97] A* **category** *consists of* **objects** *and* **arrows**. *An arrow, f, between two objects A and B is written $f : A \to B$. A category also comes with two operations*

1. *If $f : A \to B$ and $g : B \to C$ then (g, f) is called a* **composable pair** *and the* **composite** *arrow $g \circ f : A \to C$. Furthermore for composable pairs (g, f) and (h, g), $(h \circ g) \circ f = h \circ (g \circ f)$.*
2. *An assignment of an arrow id_A to each object A, called the* **identity** *on A, such that for $f : A \to B$, $f \circ id_A = id_B \circ f = f$.*

Definition 2.19 *[Aczel 97] An object, A, is an* **initial object** *in a category if for every other object, B, there is a unique arrow $f : A \to B$.*

For example, in the category of sets the empty set, $\emptyset$, is the unique initial object since for any set, $\mathcal{S}$, the empty function is the unique function $f : \emptyset \mapsto \mathcal{S}$.

Definition 2.20 *[Aczel 97] An object, A, is a* **final object** *in a category if for every other object, B, there is a unique arrow $f : B \to A$.*

In the category of sets all one point sets, $\{a\}$, are final objects, with the constant function as the unique arrow $f : \mathcal{S} \to \{a\}$ where $\forall x \in \mathcal{S}. f(x) = a$.

Categories themselves can be grouped into categories. Among the arrows that may exist between them are a class of arrows called functors which preserve the identity and composition operations.

Definition 2.21 *[Aczel 97] If C_1 and C_2 are categories, a* **functor**, *F, from C_1 to C_2, written $F : C_1 \to C_2$, consists of:*

1. *An assignment of an object $F(A)$ of C_2 to each object A of C_1.*
2. *An assignment of an arrow $F(f) : F(A) \to F(B)$ of C_2 to each arrow $f : A \to B$ of C_1.*

These assignments have the following properties:

- *For each object, A, in C_1, $F(id_A) = id_{F(A)}$.*
- *For each composable pair $(g,\ f)$ in C_1, $(F(g),\ F(f))$ is composable in C_2 and $F(g \circ f) = F(g) \circ F(f)$*

Definition 2.22 *[Aczel 97] A functor from a category C to itself is called an* **endofunctor** *on C.*

Definition 2.23 *[Aczel 97] If F is an endofunctor then an F*-**algebra** *is a pair, $(A,\ \alpha)$, where A is an object of the category and $\alpha : F(A) \to A$.*

Definition 2.24 *[Aczel 97] If F is an endofunctor then an F*-**coalgebra** *is a pair, $(A,\ \alpha)$, where A is an object of the category and $\alpha : A \to F(A)$.*

Definition 2.25 *[Rutten 96] If $(S,\ \alpha_S)$ and $(T,\ \alpha_T)$ are F–coalgebras for some arbitrary endofunctor, F. $f : S \to T$ is an F–***homomorphism** *if $F(f) \circ \alpha_S = \alpha_T \circ f$.*

F–algebras and F–coalgebras themselves can be formed into categories with F–homomorphisms providing the arrows between them and initial and final objects may be found. It has been shown [Smyth & Plotkin 82] that initial F–algebras generalise the idea of the least fixedpoint of F, similarly final F–coalgebras generalise the idea of greatest fixedpoints [Aczel & Mendler 89].

Since the ideas of initial algebras and final coalgebras are more general than those of fixedpoints they provide a wider theory in which to embed programming language semantics that make use of those notions. Initial algebras have traditionally been used to model strict datatypes. Final coalgebras are increasingly being used to model infinite datatypes and certain types of infinite automata etc.

2.6.2 Bisimulations

In this section the notion of a bisimulation which is central to many coinductive proofs is presented formally for the first time. This work is taken from Rutten [Rutten 96]. Rutten is examining coalgebras in general, not just final ones, and showing how they can be used to model various sorts of system.

Definition 2.26 *[Rutten 96] Let Set be the category of sets and let $F : Set \to Set$ be a functor. If (S, α_S) and (T, α_T) are both F–coalgebras (not necessarily final) and $\mathcal{R} \subseteq S \times T$. Then $\mathcal{R}$ is an F–***bisimulation** *if there exists an arrow $\alpha_{\mathcal{R}} : \mathcal{R} \to F(\mathcal{R})$ such that the π_i in the diagram below are F–homomorphisms such that π_S and π_T are the projections from $\mathcal{R}$ onto S and T respectively, and $F(\pi_i) \circ \alpha_{\mathcal{R}} = \alpha_i \circ \pi_i$*

$$
\begin{tikzcd}
S \arrow[d, "\alpha_S"] & R \arrow[l, "\pi_S"'] \arrow[r, "\pi_T"] \arrow[d, "\alpha_R"] & T \arrow[d, "\alpha_T"] \\
F(S) & F(R) \arrow[l, "F(\pi_S)"] \arrow[r, "F(\pi_T)"'] & F(T)
\end{tikzcd}
$$

Definition 2.27 *[Rutten 96]*

A **bisimulation equivalence** *is a bisimulation which is also an equivalence relation.*

The following three theorems are offered without proof; versions of these theorems with proofs can be found in [Rutten 96].

Theorem 2.6 *The diagonal* $\Delta_S \stackrel{\text{def}}{=} \{\langle s,\, s\rangle | s \in \mathcal{S}\}$ *of an* F*–coalgebra is a bisimulation*

Theorem 2.7 $\sim_{\langle S,\,T\rangle} \stackrel{\text{def}}{=} \bigcup\{\mathcal{R}|\mathcal{R}$ *is a bisimulation between* S *and* T $\}$ *is a bisimulation. Moreover* $\sim_{\langle S,\,S\rangle}$ *is a bisimulation equivalence.*

Theorem 2.8 *If* (A, α) *is a final* F*–coalgebra then* Δ_A *is the only bisimulation equivalence on* A *and for every bisimulation* $\mathcal{R}$ *on* A*,* $\mathcal{R} \subseteq \Delta_A$*.*

This last is the central theorem and is a coinduction principle. In fact it is a specialisation of Rutten's theorem, which deals with *simple* systems of which final F–coalgebras are only an example. However, it should be clear that theorem 2.8 is telling us much the same as the coinduction rule in §2.3, specialised to show membership of Δ_A. Bisimilarity is the most common property that is proved using coinduction. It is the relation $\sim_{\langle S,\,S\rangle}$. However, within any particular area it is often presented quite differently.

Example 2.7 *(Based on work in [Rutten 96]) Take lists as an example; two lists,* l_1 *and* l_2*, are said to be bisimilar if there exists some* $\mathcal{R}$ *such that* $\langle l_1,\, l_2\rangle \in \mathcal{R}$ *and for all* l'_1 *and* l'_2 *in* $\mathcal{R}$ $hd(l'_1) = hd(l'_2)$ *and* $\langle tl(l'_1),\, tl(l'_2)\rangle \in \mathcal{R}$ *or* $l'_1 = l'_2 = nil$*.*

Theorem 2.9 *If* $\mathcal{R}$ *is a relation on lazy lists,* A^ω *(streams of elements of* A*), and for all* l'_1 *and* l'_2 *in* $\mathcal{R}$*,* $hd(l'_1) = hd(l'_2)$ *and* $\langle tl(l'_1),\, tl(l'_2)\rangle \in \mathcal{R}$ *or* $l'_1 = l'_2 = nil$ *then* $\mathcal{R}$ *is an* F*–bisimulation for some* F *and hence* $\langle l_1,\, l_2\rangle \in \Delta_{A^\omega}$*.*

Proof. Let $F(S) = 1 + (A \times S)$, such that for any arrow $f : S_1 \to S_2$. $F(f) = id_1 + (id_A \times f)$. F is an endofunctor on the category of sets where $\times$ and $+$ are cartesian product and union respectively and 1 is a 1 element set containing just the distinguished symbol 1.

$(A^\omega,\ 1 + \langle hd,\ tl\rangle)$ is an F–coalgebra, where $1 + \langle hd,\ tl\rangle$ (written α_{A^ω}) is the arrow that takes the stream l to $\langle hd(l),\ tl(l)\rangle$ if $l = h :: t$ or to 1 if $l = nil$.

Lemma 2.1 *[Rutten 96]* $(A^\omega,\ \alpha_{A^\omega})$ *is final.*

Proof. (Based on a sketch of the proof in [Rutten 96]) Let $(\mathcal{S}, 1 + \langle v,\ n\rangle)$ be an arbitrary F–coalgebra where $1 + \langle v,\ n\rangle$ takes some element s of S to $\langle v(s),\ n(s)\rangle$ if $v(s)$ and $n(s)$ exist and to 1 otherwise.

Let $f_\mathcal{S} : \mathcal{S} \to A^\omega$ be the arrow such that for any $s \in \mathcal{S}$ if $v(s)$ and $n(s)$ exist $f_\mathcal{S}(s) = v(s) :: f_\mathcal{S}(n(s))$ and $f_\mathcal{S}(s) = nil$ otherwise.

Let $f \equiv f_\mathcal{S} + \{1 + \langle v,\ n\rangle \mapsto \alpha_{A^\omega}\}$. $f : (\mathcal{S}, 1 + \langle v,\ n\rangle) \to (A^\omega, \alpha_{A^\omega})$.

If $f(s) = h :: t$ for some $s \in \mathcal{S}$ then $hd(f(s)) = v(s)$ and $tl(f(s)) = f_\mathcal{S}(n(s)) = f(n(s))$.

- $(\alpha_{A^\omega} \circ f)(s) = \langle v(s),\ tl(f(s))\rangle = \langle v(s),\ f(n(s))\rangle$.
- $F(f) = id_1 + (id_A \times f)$ by definition, so $(F(f) \circ (1 + \langle v,\ n\rangle))(s) = \langle v(s),\ f(n(s))\rangle$.

So if $f(s) = h :: t$ then $(\alpha_{A^\omega} \circ f)(s) = (F(f) \circ (1 + \langle v,\ n\rangle))(s)$.

If $f(s) = nil$ for some $s \in \mathcal{S}$ then

$$(\alpha_{A^\omega} \circ f)(s) = 1 = (F(f) \circ (1 + \langle v,\ n\rangle))(s)$$

Hence f is an F–homomorphism.

Let g be an arbitrary F–homomorphism from $(\mathcal{S}, 1 + \langle v,\ n\rangle)$ to $(A^\omega,\ \alpha_{A^\omega})$. This means that g must equal $g_\mathcal{S} + \{(1 + \langle v, n\rangle) \mapsto \alpha_{A^\omega}\}$ for some $g_\mathcal{S}$ such that $F(g_\mathcal{S}) \circ (1 + \langle v,\ n\rangle) = \alpha_{A^\omega} \circ g_\mathcal{S}$. So $(\alpha_{A^\omega} \circ g_\mathcal{S})(s) = 1$ or $\langle v(s),\ g_\mathcal{S}(n(s))\rangle = \langle hd(g_\mathcal{S}(s)),\ tl(g_\mathcal{S}(s))\rangle$.

If $(\alpha_{A^\omega} \circ g_\mathcal{S})(s) = 1$ then $g_\mathcal{S}(s) = nil$ (by definition of α_{A^ω}). This means that $(F(g_\mathcal{S}) \circ (1 + \langle v,\ n\rangle)) = 1$ which in turn implies that $v(s)$ and $n(s)$ don't exist (by definition of $1 + \langle v,\ n\rangle$). So if $(\alpha_{A^\omega} \circ g_\mathcal{S})(s) = 1$ then $v(s)$ and $n(s)$ don't exist and $g_\mathcal{S}(s) = nil$.

If $\langle v(s),\ g_\mathcal{S}(n(s))\rangle = \langle hd(g_\mathcal{S}(s)),\ tl(g_\mathcal{S}(s))\rangle$ then $hd(g_\mathcal{S}(s)) = v(s)$ and $tl(g_\mathcal{S}(s)) = g_\mathcal{S}(n(s))$. This means that $v(s)$ and $n(s)$ exist and $g_\mathcal{S}(s) = v(s) :: g_\mathcal{S}(n(s))$.

Hence if $v(s)$ and $n(s)$ exist $g_\mathcal{S}(s) = v(s) :: g_\mathcal{S}(n(s))$, $g_\mathcal{S}(s) = nil$ otherwise. This is the definition of $f_\mathcal{S}$.

Thus f is the unique F–homomorphism from $(\mathcal{S}, 1 + \langle v, n\rangle)$ to $(A^\omega, \alpha_{A^\omega})$ so $(A^\omega, \alpha_{A^\omega})$ is final. □

For $\mathcal{R}$ to be an F–bisimulation on A^ω there must be an arrow $\alpha_{\mathcal{R}} : \mathcal{R} \to F(\mathcal{R})$, such that the π_i are F–homomorphisms (from the definition of F–bisimulation). Note that:

- $\alpha_{\mathcal{R}} : \langle l_1,\ l_2\rangle \to 1 + \langle a,\ \langle l_1',\ l_2'\rangle\rangle$ where $a \in A$ and $\langle l_1',\ l_2'\rangle \in \mathcal{R}$ (by the definition of F)
- π_i is an F–homomorphism iff
  - If $\alpha_{\mathcal{R}}(\langle l_1,\ l_2\rangle) = 1$ then

$$\begin{aligned} F(\pi_i) \circ \alpha_{\mathcal{R}}(\langle l_1,\ l_2\rangle) &= 1 && (2.10)\\ &= 1 + \langle hd,\ tl\rangle \circ \pi_i && (2.11)\end{aligned}$$

  - If $\alpha_{\mathcal{R}}(\langle l_1,\ l_2\rangle) = \langle a,\ \langle l_1',\ l_2'\rangle\rangle$

$$\begin{aligned} F(\pi_i) \circ \alpha_{\mathcal{R}}(\langle l_1,\ l_2\rangle) &= \langle a,\ l_i\rangle && (2.12)\\ &= \langle hd(l_i),\ tl(l_i)\rangle && (2.13)\\ &= 1 + \langle hd,\ tl\rangle \circ \pi_i && (2.14)\end{aligned}$$

We want

$$\alpha_R : \langle l_1,\ l_2\rangle \to \begin{cases} \langle hd(l_1),\ \langle tl(l_1),\ tl(l_2)\rangle\rangle & \text{if } l_1 = h_1 :: t_1 \text{ and } l_2 = h_2 :: t_2 \\ 1 & \text{if } l_1 = l_2 = nil \end{cases} \qquad (2.15)$$

Clearly $\alpha_R : R \to 1 + (A \times R)$. Moreover if for all $\langle l_1,\ l_2\rangle \in \mathcal{R}$, $hd(l_1) = hd(l_2)$ or $l_1 = l_2 = nil$ then $F(\pi_i) \circ \alpha_{\mathcal{R}}(\langle l_1,\ l_2\rangle) = \langle hd(l_i),\ tl(l_i)\rangle \vee 1$ which means that π_i is an F–homomorphism. Hence $\mathcal{R}$ is an F–bisimulation. $\square$

2.7 Coinductive Definitions

The term *coinductive definition* is used to refer to both the definition of datatypes and the definition of functions.

2.7.1 Coinductive Datatypes

It is usual for recursive datatypes to be defined inductively, essentially as the least fixedpoint of a function. However, this will exclude some objects that may be desired, such as $[M, M, \cdots]$, the infinite list of Ms. In this case the dual may be used and the datatype can be defined coinductively as the greatest fixedpoint of the function.

For example, the function $list\_fun$ on sets of lists:

$$list\_fun(\tau, \mathcal{S}) \stackrel{\text{def}}{=} \{h :: t | h : \tau, t \in \mathcal{S}\} \cup \{nil\} \qquad (2.16)$$

gives strict or finite lists as its least fixed point and lazy lists, or streams as its greatest fixedpoint.

Interestingly, if the function had been defined without the $\cup\{nil\}$ then the greatest fixedpoint would have been infinite lists only (without containing any finite lists) and the least fixedpoint would have been the empty set.

2.7.2 Coinductively Defined Functions

An extension of the construction versus observation distinction is to talk about having function definitions that are based on observations rather than constructors. These are also called coinductive or corecursive definitions.

Say $head, tail$ and nil are being treated as observations. Then a coinductive definition of the map function would define the values of $head(map(F, L))$ and $tail(map(F, L))$.

$$\begin{aligned} L = nil \Rightarrow map(F, L) &= nil & (2.17)\\ L \neq nil \Rightarrow head(map(F, L)) &= F(head(L)) & (2.18)\\ L \neq nil \Rightarrow tail(map(F, L)) &= map(F, tail(L)) & (2.19)\end{aligned}$$

The coinductive definitions may assume that the datatype is *purely infinite*, i.e. containing no finite objects, so for infinite streams the coinductive definition of $map(F)$ is:

$$\begin{aligned} head(map(F, L)) &= F(head(L)) & (2.20)\\ tail(map(F, L)) &= map(F, tail(L)) & (2.21)\end{aligned}$$

The first of these definitions of $map(F)$ can be seen to give the same information as the more usual definition of $map(F)$, however, such coinductive definitions are not always directly equivalent. As an example consider the append function, $<>$. The usual definition is:

$$\begin{aligned} nil <> L &= L & (2.22)\\ (H :: T) <> L &= H :: (T <> L) & (2.23)\end{aligned}$$

while the coinductive definition is

$$\begin{aligned} L_1 = nil \wedge L_2 = nil \rightarrow L_1 <> L_2 &= nil & (2.24)\\ L_1 = nil \wedge L_2 = H :: T \rightarrow head(L_1 <> L_2) &= head(L_2) & (2.25)\\ L_1 = nil \wedge L_2 = H :: T \rightarrow tail(L_1 <> L_2) &= nil <> tail(L_2) & (2.26)\\ L_1 = H :: T \rightarrow head(L_1 <> L_2) &= head(L_1) & (2.27)\\ L_1 = H :: T \rightarrow tail(L_1 <> L_2) &= tail(L_1) <> L_2 & (2.28)\end{aligned}$$

This can be written in terms of constructors, by combining the head and tail of equivalent terms as cons cells and rewriting various subterms using the equalities in the conditions. This gives three equations:

$$nil <> nil = nil \tag{2.29}$$
$$nil <> (H :: T) = H :: (nil <> T) \tag{2.30}$$
$$(H :: T) <> L = H :: (T <> L) \tag{2.31}$$

These are not the same as the equations normally used to define $<>$.

Paulson [Paulson 93] formalises this by introducing the function $llist\_corec$ as a dual to the more normal $list\_rec$ (or $fold$) whose introduction rules are:

$$\begin{aligned} list\_rec(nil, c, f) &= c \\ list\_rec(h :: t, c, f) &= f(h, list\_rec(t, c, f)) \end{aligned}$$

in order to define *corecursive* functions. $llist\_corec$ takes an object a and a function f from some set, A, containing a to some set $\{\langle\rangle\} \cup (B \times A)$ where $\langle\rangle$ is the sole value of the nullary product type, `unit`, in Isabelle's Higher Order Logic theory (Isabelle/HOL). $llist\_corec$ is defined as:

$$llist\_corec(a, f) = \begin{cases} nil & \text{if } f(a) = \langle\rangle \\ x :: llist\_corec(b, f) & \text{if } f(a) = \langle x,\ b\rangle \end{cases}$$

$llist\_corec$ is used to define functions and Paulson has shown that for all a and f, functions defined with $llist\_corec$ are lazy lists on B and are unique. From a definition phrased in terms of $llist\_corec$ the more usual introduction rules can be derived, sometimes automatically [Paulson 93].

Example 2.8 *Using $llist\_corec$ $map(F)$ is defined as*

$$map(F, L) \stackrel{\text{def}}{=} llist\_corec(L, \lambda l.list\_case(l, Inr(\langle\rangle), \lambda x.l'.Inr(\langle F(x),\ l''\rangle))) \tag{2.32}$$

where

$$list\_case(nil, c, d) = c \tag{2.33}$$
$$list\_case(H :: T, c, d) = d(H, T) \tag{2.34}$$

Rutten [Rutten 96] is able to generalise this discussion to coinductive definitions within category theory. This is based on the definitions of a transition system specification from [Groote & Vaandrager 92].

V and A are two sets of *variables* and *actions* respectively.

Definition 2.28 *[vanGlabbeek 96] A* **function declaration** *is a pair (f, n) of a* **function symbol** *$f \notin V$ and an arity $n \in nat$. A function declaration $(c, 0)$ is also called a* **constant declaration**. *A* **signature** *is a set of function declarations. The set $\mathcal{T}(\Sigma)$ of terms over a signature Σ is defined recursively by:*

- $V \subseteq \mathcal{T}(\Sigma)$.
- *if* $(f, n) \in \Sigma$ *and* $t_1, \cdots, t_n \in \mathcal{T}(\Sigma)$ *then* $f(t_1, \cdots, t_n) \in \mathcal{T}(\Sigma)$.

Definition 2.29 *[vanGlabbeek 96] Let Σ be a signature. A* **positive Σ–literal** *is an expression $t \xrightarrow{a} t'$ and a* **negative Σ-literal** *an expression $t \stackrel{a}{\not\rightarrow}$ or $t \stackrel{a}{\not\rightarrow} t'$ with $t, t' \in \mathcal{T}(\Sigma)$ and $a \in A$. A* **transition rule** *over Σ is an expression of the form $\frac{H}{\alpha}$ with H a set of Σ–literals and α a Σ–literal. An* **action rule** *is a transition rule with a positive conclusion. A* **transition system specification (TSS)** *is a pair (Σ, R) with Σ a signature and R a set of action rules over Σ.*

The definition of a "meaningful" TSS is an open question of research. [vanGlabbeek 96] considers a number of possible meanings for TSS. The notion of a transition system specification, however, allows Rutten to give the following definition.

Definition 2.30 *[Rutten 96] Let S be a set and (P, π) a final F–coalgebra. Given a transition structure $\alpha : S \to F(S)$ there exists, by the finality of P, a unique homomorphism $f_\alpha : S \to P$. Thus, specifying a transition structure on S uniquely defines a function $f_\alpha : S \to P$ which is consistent with that specification in that it is a homomorphism. f_α is said to be defined by coinduction from (the specification of) α. f_α is sometimes called the* **coinductive extension** *of α.*

Example 2.9 *Let's return to the example of the definitions of map in order to illustrate this definition. S is the domain of the function, i.e. $(\sigma \to \tau) \times A^\omega$. (P, π) is $(A^\omega, \alpha_{A^\omega})$ as discussed in §2.6.2.*

So to define map coinductively it is necessary to specify a transition structure $\alpha_{map} : ((\sigma \to \tau) \times A^\omega) \to (1 + (A \times ((\sigma \to \tau) \times A^\omega)))$

Define the transition structure α_{map} by

$$\langle F,\ nil\rangle \stackrel{\alpha_{map}}{\rightarrow} 1 \tag{2.35}$$

$$\langle F,\ H :: T\rangle \stackrel{\alpha_{map}}{\rightarrow} \langle F(H),\ \langle F,\ T\rangle\rangle \tag{2.36}$$

This gives rise to a unique function f_{map} from $(\sigma \to \tau) \times A^\omega$ to A^ω such that $\alpha_{A^\omega} \circ f_{map} = F(f_{map}) \circ \alpha_{map}$. If $\langle F,\ L\rangle \stackrel{\alpha_{map}}{\rightarrow} 1$ then $L = nil$ and $F(f_{map}) \circ \alpha_{map}(\langle F,\ L\rangle) = 1$ hence $\alpha_{A^\omega} \circ f_{map}(\langle F,\ L\rangle) = 1$ which implies that

$$f_{map}(\langle F,\ nil\rangle) = nil$$

Similarly if $\langle F,\ L\rangle \stackrel{\alpha_{map}}{\rightarrow} \langle a,\ \langle F,\ T\rangle\rangle$ then $a = F(H)$ and and $L = H :: T$, moreover $F(f_{map}) \circ \alpha_{map}(\langle F,\ L\rangle) = \langle F(H),\ f_{map}(\langle F,\ T\rangle)\rangle$ hence $\alpha_{A^\omega} \circ f_{map}(\langle F,\ L\rangle) = \langle F(H),\ f_{map}(\langle F,\ T\rangle)\rangle$ which implies that

$$f_{map}(\langle F,\ H :: T\rangle) = F(H) :: f_{map}(F, T)$$

The Problem with Flatten

Paulson comments that corecursion raises some interesting problems. For instance the function $flatten$ is usually defined using the rules:

$$\begin{aligned} flatten(nil) &= nil && (2.37)\\ flatten(H :: T) &= H <> flatten(T) && (2.38)\end{aligned}$$

An attempt to define something similar coinductively requires a transition structure $\alpha_{flatten} : (A^\omega)^\omega \to 1 + (A \times (A^\omega)^\omega$

$$\begin{aligned} nil &\overset{\alpha_{flatten}}{\to} nil && (2.39)\\ nil :: T &\overset{\alpha_{flatten}}{\to} \; ? && (2.40)\\ (H :: T) :: T' &\overset{\alpha_{flatten}}{\to} \langle H,\, T :: T'\rangle && (2.41)\end{aligned}$$

(2.40) is problematic, in fact it isn't possible to define the observations from $nil :: T$ to $1 + (A \times (A^\omega)^\omega)$ without knowing facts about T, namely whether a non nil element of T exists and if so, what the first non nil element is.

It is easy to grasp that $flatten(nil :: T) = flatten(T)$ is problematic, especially if you consider the implications of trying to flatten an infinite list of empty lists which will result in a non–terminating process. This is formalised by the difficulty in defining an appropriate transition structure. However, this is an informal argument as to why this can't be done; I'm not aware of a formal proof of the impossibility of this definition.

2.7.3 Corecursion and Unfold

Rutten's definition of coinductive extension can be illustrated by the diagram

$$\begin{tikzcd} S \arrow[r, "\alpha"] \arrow[d, "f_\alpha"'] & F(S) \arrow[d, "F(f_\alpha)"] \\ P \arrow[r, "\pi"'] & F(P) \end{tikzcd}$$

Coinductive extension has a dual [Hutton 98] illustrated by the diagram where (P, ρ) is an initial G-algebra, and $\beta : G(S) \to S$ is known as $fold$. $fold$ is a well

$$
\begin{tikzcd}
G(P) \arrow[r, "\rho"] \arrow[d, "G(g_\beta)"'] & P \arrow[d, "g_\beta"] \\
G(S) \arrow[r, "\beta"'] & S
\end{tikzcd}
$$

known function and expresses a generality in recursive definitions, for instance for lists:

$$
\begin{aligned}
fold(op, a, nil) &= a && (2.42)\\
fold(op, a, h :: t) &= op(h, fold(op, a, t)) && (2.43)
\end{aligned}
$$

Similarly α is called $unfold$. The problem with $flatten$ illustrates the fact that not all functions into greatest fixedpoints can be expressed using $unfold$. Some authors have referred to $flatten$ as "having no corecursive definition". I have adopted this convention in the rest of thesis, but readers should be aware that "no corecursive definition" is being used here in a restricted sense meaning that the function can not be expressed using $unfold$.

2.8 Coinduction in Theorem Provers

Several standard theorem provers have capabilities for coinductive proof.

2.8.1 Isabelle

Perhaps the earliest of this work was done in Isabelle [Paulson 94a] for which a special package has been developed for coinductive definitions [Paulson 94b]. This package is designed around fixedpoint theory and allows inductive and coinductive datatypes to be defined together with their elimination rules (i.e. induction and coinduction) from the introduction rules.

Example 2.10 *Take the introduction rules for lists:*

$$
\frac{}{nil \in list(A)} \qquad \frac{a \in A \quad l \in list(A)}{a :: l \in list(A)}
$$

These can be translated into two fixedpoint datatype definitions of the inductive and coinductive datatypes respectively.

$$list(A) = lfp(\lambda X.\{a :: l | a \in A, l \in X\} \cup \{nil\})$$

$$llist(A) = gfp(\lambda X.\{a :: l | a \in A, l \in X\} \cup \{nil\})$$

The inductive definition specifies the least closed sets under the given rules and a coinductive definition specifies the greatest closed set. From these definitions Isabelle can derive the induction and coinduction rules:

$$\frac{\{a :: l |\ a \in A,\ l \in X\} \subseteq X \quad nil \in X}{list(A) \subseteq X}$$

$$\frac{a \in X \quad X \subseteq \{a :: l |\ a \in A,\ l \in X\} \cup \{nil\}}{a \in llist(A)}$$

Milner and Tofte's work has been reproduced in Isabelle [Frost 95]. Their consistency result has been proved both in Isabelle's HOL and ZF theories. The first of these used Paulson's development of coinduction in HOL [Paulson 93] the second used the coinductive definition package for ZF.

2.8.2 HOL

The work on coinduction in the HOL system [Gordon 88], has perhaps been carried furthest with the provision of tactics for coinduction. This is part of a project to provide a proof tool for reasoning about a small, lazy functional language. A theory for the lazy functional semantics has been developed in HOL [Collins 96]. Equivalence is defined by Abramsky's applicative bisimulation and so coinduction has been identified as a central process. A number of tactics have been developed to support reasoning with this language. One applies the coinduction rule, given the provision of a bisimulation by the user. It can even speculate simple bisimulations automatically. There are also a number of tactics for proving relations to be bisimulations. Both this and Isabelle's coinductive definition package are discussed in greater detail in chapter 8.

2.8.3 Coq

Work has also been done in Coq [Paulin-Mohring 95] where streams (lazy lists), a coinductive datatype, have been used in the specifications of a sequential multiplier and the sieve of Eratosthenes. Although coinductive datatypes were used, the coinduction proof rule itself was not employed in the verification of these specifications.

2.8.4 PVS

Hensel and Jacobs [Hensel & Jacobs 97] have axiomatised a theory of (possibly) infinite sequences in PVS [Owre *et al* 92], using final coalgebras. This can be used for coinductive proofs although no special tools exist to aid the proof.

2.8.5 The Concurrency Workbench

As might be expected, given that the use of coinduction originated in concurrency some of the most impressive proof tools appear there. The Concurrency Workbench (CWB)[Cleaveland *et al* 89] allows a user to define systems in the syntax of CCS and analyse their state space, check for equivalences and verify processes within a modal logic. At present the CWB can only deal with finite state processes and it doesn't cope with value–passing CCS, however work is underway in both these areas [Bradfield & Stirling 90], [Bruns 91], [Monroy *et al* 95].

The CWB uses coinduction (as the bisimulation proof method) to check for equivalences.

2.9 Conclusion

This chapter has attempted to give an overview of work in coinduction. It splits roughly into three parts.

The first, in sections §2.3 and §2.4, dealt with the notion of bisimulation and three areas in which it is used, describing its formulation in those areas in some details.

The second section dealt with more theoretical aspects of coinduction in particular the development of coinduction principles (§2.5), the theory of final coalgebras (§2.6) and coinductive definitions (§2.7).

The last section (§2.8) looked at various implementations of coinduction and proof tools for coinduction.

The overall impression should be that a great deal of work has been done on providing in theoretical bases for coinduction in certain specific areas, but that the understanding of any sort of general presentation is only in its infancy. For instance the definition of bisimulation clearly has some common elements wherever it appears: it is the greatest fixedpoint of some function F on relations and some members of $F(\mathcal{R})$ for some relation $\mathcal{R}$ depend upon the values of a pair in $\mathcal{R}$ in some way. However the specifics of that dependence vary greatly: they have derivatives in $\mathcal{R}$ (CCS), their tails are in $\mathcal{R}$ (lazy lists) etc. Thus although some aspects of a coinductive proof are the same wherever coinduction occurs, e.g. the application of some sort of coinduction rule, some aspects vary greatly from domain to domain, e.g. the proof requirements needed to demonstrate bisimilarity.

Furthermore although some attempt has been made to provide tools to aid coinductive proofs, the only area where these are regularly used on a wide range of problems and where coinduction is a standard proof method is in the field of Concurrency. However, this is likely to change in the future with the increased use of formal methods for lazy functional languages and the recent development of coinduction principles for the semantics of object–oriented languages.

Chapter 3

Coinduction Specific to Functional Languages

3.1 Introduction

This chapter aims to give an overview of the operational semantics of functional languages and at the same time to present a number of examples of coinductive proofs in this setting in order to examine similarities and differences across a set of proofs.

3.2 Operational Semantics

Operational semantics describe how programs in some language function on some abstract machine; this is in contrast to denotational semantics that describe what object a program denotes. This object is usually mathematical but it can be a program in another language.

The operational semantics of lazy functional languages are based on Abramsky's work [Abramsky 90] and use his notion of applicative bisimulation to build up an equational theory.

v ranges over values a, b and c range over programs e ranges over expressions

Figure 3–1: Notational Conventions

Recall that in the lazy λ–calculus the "meaning" of a term is its weak head normal form (in the case of the "pure" calculus described in chapter 2 the WHNF is a λ–abstraction). WHNFs are often referred to as *values* and this convention is adopted here. The calculus also contains two relations, *big* and *small* step reduction.

Definition 3.1 *[Gordon 95b]* **Big step evaluation** *is specified inductively by the two rules:*

$$\frac{}{\lambda x.e \Downarrow \lambda x.e} \tag{3.1}$$

$$\frac{a \Downarrow \lambda x.e \quad e[b/x] \Downarrow v}{a\ \ b \Downarrow v} \tag{3.2}$$

Big step evaluation corresponds to the notion of the whole evaluation process of a program to discover its outcome.

An important facet of reasoning about programs is knowing when two programs are interchangeable. This is formalised by *Morris–style Contextual Equivalence* [Morris 68].

Definition 3.2 *[Gordon 95b] Let a* **context** $\mathcal{C}$, *be an expression such that there are no free variables in* $\mathcal{C}$, *except for* $\{-\}$. *This variable is a* **hole** *that will be filled in.*

Definition 3.3 *[Gordon 95b]*

Let $\mathcal{C}$ *be a context, define the relation* $\simeq$, **contextual equivalence**, *as follows:*

$$\mathcal{C}[a] \stackrel{\text{def}}{=} \mathcal{C}[a/-] \tag{3.3}$$

$$a \Downarrow \stackrel{\text{def}}{=} \exists v.(a \Downarrow v) \tag{3.4}$$

$$a \sqsubseteq_{\sim} b \stackrel{\text{def}}{=} \forall \mathcal{C}(\mathcal{C}[a] \Downarrow \Rightarrow \mathcal{C}[b] \Downarrow) \tag{3.5}$$

$$a \simeq b \stackrel{\text{def}}{=} a \sqsubseteq_{\sim} b \& b \sqsubseteq_{\sim} a \tag{3.6}$$

Definition 3.4 *[Gordon 95b] An* **experiment**, $\mathcal{E}$, *is a context of the form* $-a$, *where* a *is a program. If* $\bar{\mathcal{E}} = \mathcal{E}_1, \cdots, \mathcal{E}_n$, $\bar{\mathcal{E}}[b]$ *means* $\mathcal{E}_1[\cdots \mathcal{E}_n[b] \cdots]$.

Notice that the experiments are a special case of contexts.

Lemma 3.1 (Context Lemma) *[Milner 77](and more recently [Berry et al 86])* $a \sqsubseteq_{\sim} b$ *iff* $\forall \bar{\mathcal{E}}(\bar{\mathcal{E}}[a] \Downarrow \Rightarrow \bar{\mathcal{E}}[b] \Downarrow\ /\ (\bar{\mathcal{E}}[a] = \bar{\mathcal{E}}[b]))$.

Definition 3.5 *[Gordon 95b]* **Small step reduction**, $a \overset{\text{red}}{\leadsto} b$, *is specified inductively by the rules:*

$$\frac{}{(\lambda x.e)b \overset{\text{red}}{\leadsto} e[b/x]} \tag{3.7}$$

$$\frac{a \overset{\text{red}}{\leadsto} a'}{\mathcal{E}[a] \overset{\text{red}}{\leadsto} \mathcal{E}[a']} \tag{3.8}$$

Small step reduction corresponds to a notion of one step in a step by step process of program evaluation.

3.2.1 Static Semantics

The lazy λ–calculus is extended to typed languages by extending the syntax and semantics.

Types are defined inductively (although later in this chapter I will discuss defining the types coinductively). The following basic typing rules are added for constants and variables, λ–abstractions and function application (i.e. the elements of the untyped calculus). Γ is an *environment*, a finite map from variables to types, of form $x_1 : A_1, \cdots, x_n : A_n$. $Dom(\Gamma)$, the domain of Γ is the set of variables $x_1, \cdots, x_n$.

$$\frac{}{\Gamma, x : \tau, \Gamma' \vdash x : \tau}$$

$$\frac{\Gamma, x : \sigma \vdash e : \tau \quad x \notin Dom(\Gamma)}{\Gamma \vdash \lambda x : \sigma . e : \sigma \to \tau}$$

$$\frac{\Gamma \vdash e_1 : \sigma \to \tau \quad \Gamma \vdash e_2 : \sigma}{\Gamma \vdash e_1(e_2) : \tau}$$

These are a part of the *static semantics* (semantics used at compile time, e.g. for type checking) of the language which provides rules for building up typed expressions:

Example 3.1 *In the case of the type of lists the static semantics would associate the list constructors with types:*

$$\Gamma \vdash nil : list(\tau) \qquad \frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : list(\tau)}{\Gamma \vdash e_1 :: e_2 : list(\tau)}$$

They would probably also (though not necessarily) provide typing rules for some sort of case analysis function to be used in function definition:

$$\frac{\Gamma \vdash e_1 : list(\tau) \quad \Gamma \vdash e_2 : \sigma \quad \Gamma \vdash e_3 : \tau \to list(\tau) \to \sigma}{\Gamma \vdash list\_case(e_1, e_2, e_3) : \sigma}$$

3.2.2 Dynamic Semantics

The definition of big step evaluation is also extended to a typed language by explicit linking to small step reduction:

Definition 3.6 *[Gordon 95b] Let* $\overset{\text{red}^*}{\leadsto}$ *denote the reflexive transitive closure of* $\overset{\text{red}}{\leadsto}$.

$$a \overset{\text{red}}{\leadsto} \overset{\text{def}}{=} \exists b.(a \overset{\text{red}}{\leadsto} b) \tag{3.9}$$

$$a \Downarrow b \overset{\text{def}}{=} a \overset{\text{red}^*}{\leadsto} b \wedge \neg(b \overset{\text{red}}{\leadsto}) \tag{3.10}$$

$$a \Downarrow \overset{\text{def}}{=} \exists b. a \Downarrow b \tag{3.11}$$

$$a \Uparrow \overset{\text{def}}{=} a \overset{\text{red}^*}{\leadsto} b \Rightarrow b \overset{\text{red}}{\leadsto} \tag{3.12}$$

If $a \Downarrow b$ *then* a **evaluates to** b. *If* $a \Downarrow$ *then* a **converges**. *If* $a \Uparrow$ *then* **a diverges**.

The *dynamic semantics* of the language (semantics used at run time, e.g. for evaluation) extends the set of values (in the untyped calculus consisting only of λ–abstractions), small step reduction rules and experiments.

So in **example 3.1** *the set of values would be extended to include*

$$v ::= nil \mid e_1 :: e_2$$

The small step reduction rules would be extended to include

$$list\_case(nil, b, c) \overset{\text{red}}{\leadsto} b$$

$$list\_case(e_1 :: e_2, b, c) \overset{\text{red}}{\leadsto} c(e_1)(e_2)$$

Lastly the experiments would be extended to include

$$\mathcal{E} ::= list\_case(-, e_1, e_2)$$

3.3 Labelled Transition Systems

The last component of the operational semantics involves the use of labelled transition systems. In its most basic form a labelled transition system is a binary relation on terms or processes indexed by a set of labels. Within the semantics of lazy languages the labels tend to be type destructors and the RHS of the relation is the effect of applying that type destructor to the LHS. Often the labels (or *transitions*) are considered to be things you can "observe" about the program.

In chapter 2 a definition of a transition system specification was given. It is important to note that a transition system specification is not necessarily the same as a labelled transition system, since a TSS requires that the hypotheses of all transition rules are also Σ–literals. However, it is often possible to turn a labelled transition system into a TSS. For instance, most of the systems examined in this thesis have either Σ–literals as hypotheses or a reduction relation (e.g. $a \overset{\text{red}}{\rightsquigarrow} a'$). It is possible to regard $\overset{\text{red}}{\rightsquigarrow}$ as a transition and hence make these systems into TSSs. This approach is not adopted here, however Gordon [Gordon 95a] extends his congruence proofs to apply to his Haskell-like language where $\overset{\text{red}}{\rightsquigarrow}$ is treated as a transition.

The transition relation between two terms a and a' labelled by α is generally written as $a \xrightarrow{\alpha} a'$ (as in CCS) and this notation will be used throughout this thesis. There are some general definitions of bisimilarity which apply to all labelled transition systems.

The intuition behind equivalence in labelled transition systems is that two expressions are equivalent if they always make matching transitions to two more equivalent expressions. This relation is built up first as an order using $[-]$, a function on relations. This function takes one relation and produces another. Every transition from an expression on the left or a pair in the resulting relation is matched by a transition from the expression on the right and the results of these transitions are in the original relation. $[-]$ doesn't guarantee that all transitions from the second member of a pair can be matched by transitions from the first. This required symmetry is obtained by intersecting with various complements $\mathcal{S}^{\text{op}}$[1], where $a\mathcal{S}^{\text{op}}b$ iff $b\mathcal{S}a$.

$$[\mathcal{S}] \overset{\text{def}}{=} \{\langle a,\, b\rangle | \text{For all } \alpha \text{ if } a \xrightarrow{\alpha} a' \text{ there is a } b' \text{ with } b \xrightarrow{\alpha} b' \text{ and } a'\mathcal{S}b'\}$$

$$\langle \mathcal{S} \rangle \overset{\text{def}}{=} ([\mathcal{S}] \cap [\mathcal{S}^{\text{op}}])\text{op}$$

Notice that $\langle - \rangle$ depends upon the underlying transition system. The system presented here will be referred to as $\mathcal{T}$ and so $\langle - \rangle_{\mathcal{T}}$ is the most commonly used

[1] Taking the notation from Gordon

function in this thesis. Where there is no confusion about the transition system under consideration the subscript is dropped.

The greatest fixedpoint of $\langle - \rangle$ is written $\sim$. Two objects, a and b, in a labelled transition system are said to be *bisimilar* iff $a \sim b$.

Theorem 3.1 *[Gordon 95a] $a \sim b$ iff*

1. *For all α if $a \stackrel{\alpha}{\rightarrow} a'$ there is a b' with $b \stackrel{\alpha}{\rightarrow} b'$ and $a' \sim b'$*
2. *For all α if $b \stackrel{\alpha}{\rightarrow} b'$ there is a a' with $a \stackrel{\alpha}{\rightarrow} a'$ and $a' \sim b'$*

This means that any transition one object can perform can be matched by a transition by the other object and the resulting objects are also bisimilar.

Informally, $\sim$ consists of all equivalent "chains" that can be consumed using the language's transitions (e.g. if *tail* and *nil* are transitions then $\sim$ will contain all observationally equivalent finite and infinite lists) and all equivalent abstractions of terms (e.g. $\langle \lambda x.f(x),\ \lambda x.g(x) \rangle$ if $\forall x.f(x) = g(x)$).

Theorem 3.2 *[Gordon 95a] $\sim$ is an equivalence relation.*

Definition 3.7 *[Gordon 95c] A set is F–***dense** *iff $X \subseteq F(X)$*

Definition 3.8 *A* **bisimulation** *is a $\langle - \rangle$-dense relation*

Theorem 3.3 *[Gordon 95a] $\mathcal{R}$ is a bisimulation if $\mathcal{R} \subseteq \langle \mathcal{R} \cup \sim \rangle$.*

This means that

$$\mathcal{R} \subseteq \{\langle a,\ b \rangle |\ \text{whenever } a \stackrel{\alpha}{\rightarrow} a' \text{ and } b \stackrel{\alpha}{\rightarrow} b', a'\mathcal{R}b' \text{ or } a' \sim b' \text{ and vice versa } \}$$

Theorem 3.3 is useful since it allows properties of $\sim$, most notably its reflexivity to be used during the course of a coinductive proof.

The coinduction rule for $\sim$ (based on (2.6)) is

$$\frac{\langle a,\ b \rangle \in \mathcal{R} \quad \mathcal{R} \subseteq \langle \mathcal{R} \rangle}{a \sim b} \tag{3.13}$$

Extended using theorem 3.3 this becomes

$$\frac{\langle a,\ b \rangle \in \mathcal{R} \quad \mathcal{R} \subseteq \langle \mathcal{R} \cup \sim \rangle}{a \sim b} \tag{3.14}$$

this is the standard coinduction rule for a labelled transition system.

So far these definitions are not dependent upon any one given labelled transition system. Generally, however, we want bisimilarity to be a congruence relation on the language that can act as a form of equality. A good deal of work has gone into showing that observational equivalence is observation congruence for various languages and domains [Howe 89], [Pitts 92], [Fiore 93], [Gordon 95a], [Gordon 96]. However this thesis is interested in producing proofs of bisimilarity (the equivalence of two expressions) not congruence (the ability of both expressions to behave interchangeably when replacing a hole in a context). Although the desired final result is probably a proof of congruence since that implies a useful form of equality between two objects, proofs that $\sim$ is a congruence for some particular labelled transition sequence are frequently complicated and none are offered here. If $\sim$ has been established as a congruence then the bisimulation proofs are also proofs of congruence.

3.4 Operational Semantics

The *operational semantics* of the language describes a labelled transition system in which $\sim$ will be equivalent to Abramsky's applicative bisimulation. In this way $a \sim b$ if they both evaluate to the same weak head normal form. This is done by specifying a set of observations (or transitions) which describe something that can be observed about a value. For lists the standard transitions are `nil`, `hd` (head) and `tl` (tail). So the transition rules given in the semantics would be

$$\frac{}{nil \overset{\texttt{nil}}{\rightarrow} \bot} \tag{3.15}$$

$$\frac{a :: b : list(\tau)}{a :: b \overset{\texttt{hd}}{\rightarrow} a} \qquad \frac{a :: b : list(\tau)}{a :: b \overset{\texttt{tl}}{\rightarrow} b} \tag{3.16}$$

$\bot$ is defined to be some arbitrary divergent program (with no transitions).

Lastly there is a lemma about observations which states that if some expression reduces to another then the observations from both expressions are the same. This is important since observations are generally only defined for values, so to determine the observations from an expression it is first necessary evaluate it

$$\frac{a : \tau \quad \tau \not\equiv \tau_1 \rightarrow \tau_2 \quad a \overset{\text{red}}{\rightsquigarrow} b \quad b \overset{\alpha}{\rightarrow} c}{a \overset{\alpha}{\rightarrow} c} \tag{3.17}$$

Note. If $a \overset{\alpha}{\rightarrow} b$ then α is said to *apply to* a. b is said to be the *result* of the transition. The condition $\tau \not\equiv \tau_1 \rightarrow \tau_2$ is to ensure that only reduction which is actually needed is undertaken since transitions from expressions of type $\tau_1 \rightarrow \tau_2$ are unconditional whereas transitions from other types are contingent upon convergence.

Examples of such languages can be found in the literature, for instance in [Gordon 95a] and [Collins 96].

3.5 An Example of a Coinductive Proof in an LTS for a Functional Language

I'm going to provide the static and dynamic semantics together with transition rules for a small functional language. These are in figures 3–2, 3–3 and 3–4. This language is based on Gordon's [Gordon 95a] which, in turn was based on Haskell

Definition 3.9 *The syntax of types, A, and expressions, e, is as follows:*

$$
\begin{array}{rcll}
A & ::= & bool \mid nat \mid A \to A \mid list(A) \mid tree_{bin}(A) \mid tree(A) & \\
e & ::= & x \mid \lambda(x : A)e \mid e(e) \mid rec(f : A \to B, x : A)e & (3.18)\\
 & \mid & 0 \mid s(e) \mid num\_case(e, e, e) & \\
 & \mid & \underline{bv}(bv \in \{true, false\}) \mid \ \ if\ e\ then\ e\ else\ e & \\
 & \mid & nil \mid e :: e \mid list\_case(e, e, e) & (3.19)\\
 & \mid & leaf_{bin}(e) \mid node_{bin}(e, e, e) \mid tree_{bin}\_case(e, e, e) & (3.20)\\
 & \mid & node(e, e) \mid tree\_case(e, e, e) & (3.21)
\end{array}
$$

num_case and $list\_case$ are specialisations of the function $fold$ from chapter 2 for numbers and lists respectively. rec is similar to $fold$, but ignoring the possibility of any kind of base case.

The following definition is useful for discussing coinductive proofs

Definition 3.10 *A set syntactically represented by one related pair of expressions containing free or universally quantified variables is a* **pair scheme**.

If $\mathcal{R}$ is the pair scheme, $\{\langle e_1,\ e_2\rangle\}$, and $e_1 \overset{\text{red}}{\leadsto} e_1'$ then $\langle e_1',\ e_2\rangle \in \mathcal{R}$. Similarly if $e_2 \overset{\text{red}}{\leadsto} e_2'$ then $\langle e_1,\ e_2'\rangle \in \mathcal{R}$,

If x is a universally quantified variable appearing in $\mathcal{R}$ and v a value then if $e_1[v/x] \overset{\text{red}}{\leadsto} e_1'$ then $\langle e_1',\ e_2\rangle \in \mathcal{R}$. Similarly if $e_2[v/x] \overset{\text{red}}{\leadsto} e_2'$ then $\langle e_1,\ e_2'\rangle \in \mathcal{R}$.

For every pair of expressions $\langle e_1, e_2\rangle$ if there exists $e_1{}'$ and $e_2{}'$ such that $e_1 \overset{\text{red}}{\leadsto} e_1{}'$ and $e_2 \overset{\text{red}}{\leadsto} e_2{}'$ and $\langle e_1{}',\ e_2{}'\rangle \in \mathcal{R}$ then $\langle e_1,\ e_2\rangle \in \mathcal{R}$.

Bisimulations in functional languages are invariably the unions of one or more pair schema.

The derived inference rule (3.22) is also needed for most coinductive proofs, its proof can be found in appendix D

$$
\frac{\begin{array}{l}\forall \mathcal{R}.\forall 1 \leq i \leq n.\ \bigwedge_{i=1}^{n}\langle a_i,\ b_i\rangle \in \mathcal{R} \Rightarrow \\ \qquad\qquad \forall \alpha.((a_i \overset{\alpha}{\to} a_i' \vee b_i \overset{\alpha}{\to} b_i') \Rightarrow \\ \qquad\qquad ((a_i \overset{\alpha}{\to} a_i' \wedge b_i \overset{\alpha}{\to} b_i') \wedge \\ \qquad\qquad \langle a_i',\ b_i'\rangle \in \mathcal{R} \cup \sim))\end{array}}{\bigcup_{i=1}^{n}\langle a_i,\ b_i\rangle \subseteq \langle \bigcup_{i=1}^{n}\langle a_i,\ b_i\rangle \cup \sim\rangle} \tag{3.22}
$$

$$\frac{\Gamma,\ f : \sigma \to \tau,\ x : \sigma \vdash e : \tau}{\Gamma \vdash rec(f : \sigma \to \tau, x : \sigma)e : \sigma \to \tau}$$

$$\Gamma \vdash \underline{bv} : bool$$

$$\frac{\Gamma \vdash e_1 : bool \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau}$$

$$\Gamma \vdash 0 : nat \qquad \frac{\Gamma \vdash n : nat}{\Gamma \vdash s(n) : nat}$$

$$\frac{\Gamma \vdash e_1 : nat \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : nat \to \tau}{\Gamma \vdash num\_case(e_1, e_2, e_3) : \tau}$$

$$\Gamma \vdash nil : list(\tau) \qquad \frac{e_1 : \tau \quad \Gamma \vdash e_2 : list(\tau)}{\Gamma \vdash e_1 :: e_2 : list(\tau)}$$

$$\frac{\Gamma \vdash e_1 : list(\tau) \quad \Gamma \vdash e_2 : \sigma \quad \Gamma \vdash e_3 : \tau \to list(\tau) \to \sigma}{\Gamma \vdash list\_case(e_1, e_2, e_3) : \sigma}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash leaf(e) : list(\tau)} \qquad \frac{e_1 : \tau \quad \Gamma \vdash e_2 : tree_{bin}(\tau) \quad \Gamma \vdash e_3 : tree_{bin}(\tau)}{\Gamma \vdash node_{bin}(e_1, e_2, e_3) : tree_{bin}(\tau)}$$

$$\frac{\Gamma \vdash e_1 : tree_{bin}(\tau) \quad \Gamma \vdash e_2 : \tau \to \sigma \quad \Gamma \vdash e_3 : \tau \to tree_{bin}(\tau) \to tree_{bin}(\tau) \to \sigma}{\Gamma \vdash tree_{bin}\_case(e_1, e_2, e_3) : \sigma}$$

$$\frac{e_1 : \tau \quad \Gamma \vdash e_2 : list(tree(\tau))}{\Gamma \vdash node(e_1, e_2) : tree(\tau)}$$

$$\frac{\Gamma \vdash e_1 : tree(\tau) \quad \Gamma \vdash e_2 : \tau \to \sigma \quad \Gamma \vdash e_3 : \tau \to list(tree(\tau)) \to \sigma}{\Gamma \vdash tree\_case(e_1, e_2, e_3) : \sigma}$$

Figure 3–2: Static Semantics

v is a value if it is of the form:

$$\begin{aligned} v \;\; ::= \;\; & \underline{bv} \mid 0 \mid s(n) \mid nil \mid a :: as \mid leaf_{bin}(e) \\ & \mid node_{bin}(e, l, r) \mid node(e, f) \mid \lambda x, e.rec(f, x)e \end{aligned}$$

Reduction Rules:

$$\begin{aligned}
\text{if } \underline{bv} \text{ then } a_{true} \text{ else } a_{false} &\overset{\text{red}}{\rightsquigarrow} a_{bv} \\
num_case(0, b, c) &\overset{\text{red}}{\rightsquigarrow} b \\
num_case(s(n), b, c) &\overset{\text{red}}{\rightsquigarrow} c(n) \\
list_case(nil, b, c) &\overset{\text{red}}{\rightsquigarrow} b \\
list_case(h :: t, b, c) &\overset{\text{red}}{\rightsquigarrow} c(h)(t) \\
tree_{bin}_case(leaf(a), b, c) &\overset{\text{red}}{\rightsquigarrow} b(a) \\
tree_{bin}_case(node(a, l, r), b, c) &\overset{\text{red}}{\rightsquigarrow} c(a)(l)(r) \\
tree_case(node(a, l), b, c) &\overset{\text{red}}{\rightsquigarrow} list_case(l, b(a), c(a)) \\
(\lambda x : \tau.e)(a) &\overset{\text{red}}{\rightsquigarrow} e[a/x] \\
(rec(f, x)e)(a) &\overset{\text{red}}{\rightsquigarrow} e[(rec(f, x)e)/f, a/x]
\end{aligned}$$

$\mathcal{E}$ is an experiment if it is of the form:

$$\begin{aligned} & -(a) \mid \text{if } - \text{ then } a \text{ else } a \mid num\_case(-, a, a) \mid list\_case(-, a, a) \\ & \mid tree_{bin}\_case(-, a, a) \mid tree\_case(, a, a) \end{aligned}$$

Figure 3–3: Dynamic Semantics

$$\frac{a : \tau_1 \to \tau_2 \qquad b : \tau_1}{a \overset{\mathtt{ap}(b)}{\to} a(b)}$$

$$\frac{}{\underline{bv} \overset{bv}{\to} \bot}$$

$$\frac{}{0 \overset{\mathtt{0}}{\to} \bot} \qquad \frac{}{s(n) \overset{\mathtt{p}}{\to} n}$$

$$\frac{}{nil \overset{\mathtt{nil}}{\to} \bot}$$

$$\frac{}{a :: b \overset{\mathtt{hd}}{\to} a} \qquad \frac{}{a :: b \overset{\mathtt{tl}}{\to} b}$$

$$\frac{}{leaf_{bin}(a) \overset{\mathtt{label}}{\to} a}$$

$$\frac{}{node_{bin}(a, l, r) \overset{\mathtt{label}}{\to} a}$$

$$\frac{}{node_{bin}(a, l, r) \overset{\mathtt{left}}{\to} l} \qquad \frac{}{node_{bin}(a, l, r) \overset{\mathtt{right}}{\to} r}$$

$$\frac{}{node(a, f) \overset{\mathtt{label}}{\to} a} \qquad \frac{}{node(a, f) \overset{\mathtt{forest}}{\to} f}$$

$$\frac{a : \tau \qquad \tau \neq \tau_1 \to \tau_2 \qquad a \overset{\text{red}}{\leadsto} b \qquad b \overset{\alpha}{\to} c}{a \overset{\alpha}{\to} c}$$

Figure 3–4: Transition Rules

Example 3.2 is an example of a coinductive proof in this language.

Example 3.2

$$\forall f, x.\ map(f, iterates(f, x)) \sim iterates(f, f(x))$$

where the functions, *map* and *iterates* are defined as

$$map(f, l) \equiv \lambda m.list\_case(l, nil, rec(m, h, t)(f(h) :: m(f, t)))$$
$$iterates(f, x) \equiv \lambda i.rec(i, x)(x :: i(f, f(x)))$$

These give rise to the reduction rules:

$$\begin{array}{rcl} map(F, nil) & \overset{\text{red}}{\rightsquigarrow} & nil \\ map(F, H :: T) & \overset{\text{red}}{\rightsquigarrow} & H :: map(F, T) \\ \\ iterates(F, M) & \overset{\text{red}}{\rightsquigarrow} & M :: iterates(F, F(M)) \end{array}$$

Proof. Let $\mathcal{R}' = \{\langle map(F, iterates(F, X)),\ iterates(F, F(X))\rangle\}$.

Using (3.14) this gives two goals. The first

$$\langle map(F, iterates(F, X)),\ iterates(F, F(X))\rangle \in \mathcal{R}'$$

is trivial leaving

$$\begin{array}{c} \{\langle map(F', iterates(F', X')),\ iterates(F', F'(X'))\rangle\} \subseteq \\ \langle \{\langle map(F, iterates(F, X)),\ iterates(F, F(X))\rangle\} \cup \sim \rangle \end{array} \tag{3.23}$$

To prove this we need to show that any transition, α, that applies to $map(F', iterates(F', X'))$ also applies to $iterates(F', F'(X'))$ and vice versa. Moreover, that the results of these transitions lie in $\mathcal{R}\cup \sim$. These conditions arise out of the definition of $\langle - \rangle$. This is encapsulated by (3.22) which in this case leads to the subgoal:

$$\begin{array}{c} \forall \mathcal{R}.\langle map(F, iterates(F, X)),\ iterates(F, F(X))\rangle \in \mathcal{R} \Rightarrow \\ \forall \alpha.((map(F', iterates(F', X')) \overset{\alpha}{\rightarrow} \phi \vee iterates(F', F'(X')) \overset{\alpha}{\rightarrow} \psi) \Rightarrow \\ ((map(F', iterates(F', X')) \overset{\alpha}{\rightarrow} \phi \wedge iterates(F', F'(X')) \overset{\alpha}{\rightarrow} \psi) \\ \wedge \langle \phi,\ \psi \rangle \in \mathcal{R}\cup \sim)) \end{array} \tag{3.24}$$

This is a bit like a goal in an inductive proof and hence the expression to the left of the first implication is called the *coinduction hypothesis* and the expression to the right is called the *coinduction conclusion*. This distinction makes discussion of the subgoal easier.

Breaking this goal down, the next step is to evaluate $map(F', iterates(F', X'))$ and $iterates(F', F'(X'))$ using the reduction rules for map and $iterates$ to values so that α, ϕ and ψ can be determined.

$$map(F', iterates(F', X')) \Downarrow F'(X') :: map(F', iterates(F', F'(X')))$$
$$iterates(F', F'(X')) \Downarrow F'(X') :: iterates(F', F'(F'(X')))$$

The possible transitions/observations from these two terms are $\stackrel{\mathtt{hd}}{\rightarrow}$ and $\stackrel{\mathtt{tl}}{\rightarrow}$ these transitions apply to both terms:

$$\begin{array}{rcl} F'(X') :: map(F', iterates(F', F'(X'))) & \stackrel{\mathtt{hd}}{\rightarrow} & F'(X') \\ F'(X') :: iterates(F', F'(F'(X'))) & \stackrel{\mathtt{hd}}{\rightarrow} & F'(X') \end{array}$$

$$\begin{array}{rcl} F'(X') :: map(F', iterates(F', F'(X'))) & \stackrel{\mathtt{tl}}{\rightarrow} & map(F', iterates(F', F'(X'))) \\ F'(X') :: iterates(F', F'(F'(X'))) & \stackrel{\mathtt{tl}}{\rightarrow} & iterates(F', F'(F'(X'))) \end{array}$$

Simplifying the goal by instantiating α, ϕ and ψ for each value of α and removing all the conjuncts and disjuncts containing $\stackrel{\alpha}{\rightarrow}$ since these have become trivially true, leaves two new goals:

$$\begin{array}{c} \forall \mathcal{R}. \langle map(F, iterates(F, X)),\ iterates(F, F(X)) \rangle \in \mathcal{R} \Rightarrow \\ \langle F'(X'),\ F'(X') \rangle \in \mathcal{R} \cup \sim \end{array} \tag{3.25}$$

$$\begin{array}{c} \forall \mathcal{R}. \langle map(F, iterates(F, X)),\ iterates(F, F(X)) \rangle \in \mathcal{R} \Rightarrow \\ \langle map(F', iterates(F', F'(X'))),\ iterates(F', F'(F'(X'))) \rangle \in \mathcal{R} \cup \sim \end{array} \tag{3.26}$$

$\langle F'(X'),\ F'(X') \rangle \in \sim$ by reflexivity of $\sim$ and

$$\langle map(F', iterates(F', F'(X'))),\ iterates(F', F'(F'(X'))) \rangle \in \mathcal{R}$$

by appeal to the hypothesis. □

3.5.1 An Aside: The Same Example Using *llistD_fun*

Another function used to supply greatest fixedpoints for lists is $llistD\_fun$, this is not used explicitly in functional languages but proofs involving it are very similar to those involving $\langle - \rangle$.

Equality on lazy lists, $\equiv$, as defined by Paulson [Paulson 93], is a property of the greatest fixedpoint of the function

$$llistD\_fun(\mathcal{R}) \stackrel{\text{def}}{=} \{ \langle h :: t_1,\ h :: t_2 \rangle | \langle t_1,\ t_2 \rangle \in \mathcal{R} \} \cup \{ \langle nil,\ nil \rangle \} \tag{3.27}$$

In this case the coinduction rule (2.6) can be specialised to

$$\frac{\langle a,\, b\rangle \in \mathcal{R} \quad \mathcal{R} \subseteq llistD\_fun(\mathcal{R})}{a \equiv b} \tag{3.28}$$

Paulson show that $\equiv$ acts as an equality for lazy lists, by which he means that for finite k the first k elements of each list are the same. This is based on Bird and Wadler's take lemma [Bird & Wadler 88]. This arises out of his treatment of lazy lists, which makes the equality relation an instance of set equality. This equality isn't really comparable to $\sim$ since it applies only to one type, lists, and is treating them as built upon set theory, while $\sim$ assumes they are defined using some form of operational semantics. However the higher level objects are similar and there are similarities in the form of the proofs, which helps to abstract the general form of a coinductive proof away from the specific domain of the proof, definition of equivalence etc. that is being used.

Redoing the Example Using $llistD\_fun$

Theorem 3.2 becomes

$$\forall f, m.\ map(f, iterates(f, m)) \equiv iterates(f, f(m))$$

As mentioned above equality over lazy lists is a property of $gfp(llistD\_fun)$.

Proof.
Let $\mathcal{R}' \stackrel{\text{def}}{=} \{\langle map(f, iterates(f, m)),\ iterates(f, f(m))\rangle | m : \tau,\ f : \tau \to \tau\}$

According to the premises of (3.28) it is necessary to show that

$$\langle map(f, iterates(f, m)),\ iterates(f, f(m))\rangle \in \mathcal{R}'$$

and $\mathcal{R}' \subseteq llistD\_fun(\mathcal{R}')$.

The first premise is trivially discharged.

The second premise gives the goal:

$$\begin{array}{c} \{\langle map(F, iterates(F, M)),\ iterates(F, F(M))\rangle\} \subseteq \\ llistD\_fun(\{\langle map(F', iterates(F', M')),\ iterates(F', F'(M'))\rangle\}) \end{array} \tag{3.29}$$

To prove this, it is necessary to show that

$$\mathcal{R}' \subseteq \{\langle h :: t_1,\ h :: t_2\rangle \mid \langle t_1,\ t_2\rangle \in \mathcal{R}'\} \cup \{\langle nil,\ nil\rangle$$

That is that any two lists in $\mathcal{R}'$ either have equal heads and tails that are related by $\mathcal{R}'$ or that they are both nil. As in example 3.2 this is encapsulated by a derived rule (3.30) whose proof is in appendix D

$$\frac{\begin{array}{cc} \forall \mathcal{R}. & \forall \mathcal{R}. \\ \forall 1 \leq i \leq n. & \forall 1 \leq i \leq n. \\ \bigwedge_{i=1}^{n} \langle a_i,\ b_i \rangle \in \mathcal{R} \Rightarrow & \bigwedge_{i=1}^{n} \langle a_i,\ b_i \rangle \in \mathcal{R} \Rightarrow \\ hd(a_i) = hd(b_i) & \langle tl(a_i),\ tl(b_i) \rangle \in \mathcal{R} \end{array}}{\bigcup_{i=1}^{n} \langle a_i,\ b_i \rangle \subseteq llistD\_fun(\bigcup_{i=1}^{n} \langle a_i,\ b_i \rangle)} \tag{3.30}$$

(3.30) is used to produce the subgoals (3.31) and (3.32).

$$\begin{array}{rl} \forall \mathcal{R}.\langle map(F, iterates(F, M)),\ iterates(F, F(M)) \rangle \in \mathcal{R} & \Rightarrow \\ \langle tl(map(F', iterates(F', M'))),\ tl(iterates(F', F'(M'))) \rangle \in \mathcal{R} & \end{array} \tag{3.31}$$

Everything before the implication is the *coinduction hypothesis* and everything after the implication the *coinduction conclusion*.

The coinduction hypothesis states the definition of $\mathcal{R}'$ since it is defined precisely to be the set of pairs $\langle map(F, iterates(F, M)),\ iterates(F, F(M)) \rangle$. The coinduction conclusion asks that given the definition of $\mathcal{R}'$ are the tails of an arbitrary pair in $\mathcal{R}'$ also in $\mathcal{R}'$. To do this it is necessary to perform some rewriting on the conclusion to clear away the *tl* symbols and attempt to match with the hypothesis. After rewriting according to the definitions of *map* and *iterates* the goal becomes.

$$\begin{array}{c} \forall \mathcal{R}.\langle map(F, iterates(F, M)),\ iterates(F, F(M)) \rangle \in \mathcal{R} \Rightarrow \\ \langle map(F', iterates(F',\ F'(M'))),\ iterates(F',\ F'(F'(M'))) \rangle \in \mathcal{R} \end{array}$$

The conclusion matches the hypothesis (instantiating F' to F and $F'(M')$ to $F(M)$).

The proof is completed by following a similar rewriting process to show that the heads are equal.

$$\begin{array}{rcl} hd(map(F, iterates(F, M))) & = & hd(iterates(F, F(M))) \\ F(M) & = & F(M) \end{array} \tag{3.32}$$

□

$llistD\_fun$ deals only with lazy lists. It is possible to create similar functions for trees etc., but each has different proof obligations for $\mathcal{R} \subseteq \mathcal{F}(\mathcal{R})$ (i.e. for lists the conditions are that the heads are equal and the tails are in the relation. For trees the labels must be equal (if it is a labelled tree) and the subtrees with the daughters of the top node as top nodes must be in the relation).

The appealing feature of the labelled transition system (LTS) presentation is that it allows you to tell a general story for observational equivalence. Instead of having a number of functions which may have a number of conditions, you have one function $\langle - \rangle$ which requires a search for all observations, meanwhile it is the definition of the language which determines what those observations can be.

However coinduction can be used for both proofs of bisimilarity and type checking (see §3.7 below). A labelled transition system framework has only been developed for the first of these.

3.6 Examples of Coinduction

The purpose of this section is to offer a number of proofs by coinduction in the functional language defined in §3.5. This serves a dual purpose of familiarising the reader with coinductive proofs and allowing similarities and differences between various proofs to be demonstrated.

All these proofs assume the small functional programming language whose syntax and semantics was set out in section 3.5. This means that (3.14) is the coinduction rule.

Example 3.3

$$\forall f, g, l. map(f, map(g, l)) \sim map(f \circ g, l)$$

Proof. Let $\mathcal{R}' = \{\langle map(F, map(G, L)),\ map(F \circ G, L)\rangle\}$

The first premise of (3.14) is trivial. The second premise is, $\mathcal{R}' \subseteq \langle \mathcal{R}' \cup \sim \rangle$. (3.22) introduces the following subgoal:

$$\begin{array}{rl} & \forall \mathcal{R}. \langle map(F, map(G, L)),\ map(F \circ G, L)\rangle \in \mathcal{R} \Rightarrow \\ \forall \alpha. & ((map(F', map(G', L')) \xrightarrow{\alpha} \phi \vee map(F' \circ G', L') \xrightarrow{\alpha} \psi) \quad \Rightarrow \\ & ((map(F', map(G', L')) \xrightarrow{\alpha} \phi \wedge map(F' \circ G', L') \xrightarrow{\alpha} \psi) \quad \wedge \\ & \langle \phi,\ \psi \rangle \in \mathcal{R} \cup \sim)) \end{array} \tag{3.33}$$

To determine the proof of this it is necessary to find all possible values of α, ϕ and ψ. In example 3.2 this was a simple process of reducing the expressions. This case is slightly more complicated since the expressions $map(F', map(G', L'))$ and $map(F' \circ G', L')$ can't be reduced until the value of L' is known. Since L' is a universally quantified variable of list type its value is either nil or $H :: T$ for some H and T.

If $L' = nil$ then $map(F', map(G', L')) \Downarrow nil$ and $map(F' \circ G', L') \Downarrow nil$. Hence $\alpha = \texttt{nil}$ and $\phi = \psi = \bot$ hence $\langle \phi,\ \psi \rangle \in \sim$.

If $L' = H :: T$ then $map(F', map(G', L')) \Downarrow F'(G'(H')) :: map(F', map(G', T))$ and $map(F' \circ G', L') \Downarrow F' \circ G'(H) :: map(F' \circ G', T)$. So α can be `hd` or `tl`. This gives two goals

$$\begin{array}{c} \forall \mathcal{R}. \langle map(F, map(G, L)),\ map(F \circ G, L)\rangle \in \mathcal{R} \Rightarrow \\ \langle F'(G'(H)),\ F' \circ G'(H)\rangle \in \mathcal{R} \cup \sim \end{array} \tag{3.34}$$

which is true by the definition of $\circ$ and the reflexivity of $\sim$.

$$\begin{array}{c} \forall \mathcal{R}. \langle map(F, map(G, L)),\ map(F \circ G, L)\rangle \in \mathcal{R} \Rightarrow \\ \langle map(F', map(G', T)),\ map(F' \circ G', T)\rangle \in \mathcal{R} \cup \sim \end{array} \tag{3.35}$$

which is true by appeal to the hypothesis.

Hence $\mathcal{R}'$ is a bisimulation. $\Box$.

This proof is in many ways similar to that of example 3.2. It followed a process in which a relation $\mathcal{R}'$ was formed from the constituents of the original goal. This relation allowed the first premise of the coinduction rule (3.14) to be trivially discharged, leaving only the second premise to be worked on. This premise, a bisimilarity condition, was phrased in terms of subset inclusion. (3.22) then introduced a new subgoal which was an implication involving a coinduction hypothesis and conclusion which contained various uninstantiated variables for transition labels and the resulting transitions. These variables would need to be instantiated before the proof could go through. This was done using reduction rules and in the example above also casesplitting L. The resulting goals were discharged by appeal to the hypothesis or the reflexivity of $\sim$.

In this next example the choice for $\mathcal{R}'$ is more complex than in the examples previously shown.

Example 3.4

$$\forall a, b. lswap(a, b) \sim merge(lconst(a), lconst(b))$$

where:

$$lswap(A, B) \stackrel{\text{red}}{\rightsquigarrow} A :: lswap(B, A) \tag{3.36}$$
$$lconst(A) \stackrel{\text{red}}{\rightsquigarrow} A :: lconst(A) \tag{3.37}$$
$$merge(nil, L) \stackrel{\text{red}}{\rightsquigarrow} L \tag{3.38}$$
$$merge(L, nil) \stackrel{\text{red}}{\rightsquigarrow} L \tag{3.39}$$
$$merge(H_1 :: T_1, H_2 :: T_2) \stackrel{\text{red}}{\rightsquigarrow} H_1 :: H_2 :: merge(T_1, T_2) \tag{3.40}$$

Proof.

$$\begin{aligned} \text{Let } \mathcal{R}' \;=\; & \{\langle lswap(A, B),\ merge(lconst(A), lconst(B))\rangle\} \cup \\ & \{\langle lswap(B, A),\ B :: merge(lconst(A), lconst(B))\rangle\} \end{aligned}$$

Once again the first premise of (3.14) is trivially discharged leaving the second premise, $\mathcal{R}' \subseteq \langle \mathcal{R}' \cup \sim \rangle$.

(3.22) allows new subgoals to be introduced. The fact that the relation contains two separate pair schema results in two subgoals, (3.41) and (3.42).

$$\begin{gathered} \forall \mathcal{R} \langle lswap(A, B),\ merge(lconst(A), lconst(B))\rangle \in \mathcal{R} \wedge \\ \langle lswap(B', A'),\ B' :: merge(lconst(A'), lconst(B'))\rangle \in \mathcal{R} \Rightarrow \end{gathered}$$

$$\begin{aligned} \forall \alpha.\quad & ((lswap(A'', B'') \xrightarrow{\alpha} \phi \vee merge(lconst(A''), lconst(B'')) \xrightarrow{\alpha} \psi) \;\Rightarrow \\ & ((lswap(A'', B'') \xrightarrow{\alpha} \phi \wedge merge(lconst(A''), lconst(B'')) \xrightarrow{\alpha} \psi) \;\wedge \\ & \qquad \langle \phi,\ \psi\rangle \in \mathcal{R} \cup \sim)) \end{aligned} \tag{3.41}$$

$lswap(A'', B'')$ evaluates to $A'' :: lswap(A'', B'')$. $merge(lconst(A''), lconst(B''))$ evaluates to $A'' :: B'' :: merge(lconst(A''), lconst(B''))$. These are values and the transitions from them are `hd` and `tl`.

$$\begin{array}{rcl} lswap(A'', B'') & \stackrel{\texttt{hd}}{\rightarrow} & A'' \\ merge(lconst(A''), lconst(B'')) & \stackrel{\texttt{hd}}{\rightarrow} & A'' \end{array}$$

$$\begin{array}{rcl} lswap(A'', B'') & \stackrel{\texttt{tl}}{\rightarrow} & lswap(B'', A'') \\ merge(lconst(A''), lconst(B'')) & \stackrel{\texttt{tl}}{\rightarrow} & B'' :: merge(lconst(A''), lconst(B'')) \end{array}$$

$\langle A'',\ A''\rangle \in\sim$ by the reflexivity of $\sim$.

$\langle lswap(B'', A''),\ B'' :: merge(lconst(A''), lconst(B''))\rangle \in \mathcal{R}$ by appeal to the second hypothesis.

$$\begin{array}{c} \langle lswap(A, B),\ merge(lconst(A), lconst(B))\rangle \in \mathcal{R} \wedge \\ \langle lswap(B', A'),\ B' :: merge(lconst(A'), lconst(B'))\rangle \in \mathcal{R} \Rightarrow \end{array}$$

$$\begin{array}{ll} \forall\alpha. & (lswap(B'', A'') \stackrel{\alpha}{\rightarrow} \phi \vee B'' :: merge(lconst(A''), lconst(B'')) \stackrel{\alpha}{\rightarrow} \psi) \quad \Rightarrow \\ & (lswap(B'', A'') \stackrel{\alpha}{\rightarrow} \phi \wedge B'' :: merge(lconst(A''), lconst(B'')) \stackrel{\alpha}{\rightarrow} \psi \quad \wedge \\ & \qquad \langle \phi,\ \psi\rangle \in \mathcal{R} \cup \sim) \end{array} \tag{3.42}$$

$B'' :: merge(lconst(A''), lconst(B''))$ is already a value. $lswap(B'', A'')$ evaluates to $B'' :: lswap(A'', B'')$. The possible instantiations of α are `hd` and `tl`.

$$\begin{array}{rcl} lswap(B'', A'') & \stackrel{\texttt{hd}}{\rightarrow} & B'' \\ B'' :: merge(lconst(A''), lconst(B'')) & \stackrel{\texttt{hd}}{\rightarrow} & B'' \end{array}$$

$$\begin{array}{rcl} lswap(B'', A'') & \stackrel{\texttt{tl}}{\rightarrow} & lswap(A'', B'') \\ B'' :; merge(lconst(A''), lconst(B'')) & \stackrel{\texttt{tl}}{\rightarrow} & merge(lconst(A''), lconst(B'')) \end{array}$$

$\langle B'',\ B''\rangle \in\sim$ by the reflexivity of $\sim$.

$\langle lswap(A'', B''),\ merge(lconst(A''), lconst(B''))\rangle \in \mathcal{R}$ by appeal to the first hypothesis.

Hence $\mathcal{R}'$ is a bisimulation. $\Box$.

In this example a more complicated bisimulation was needed consisting of two pair schema as opposed to one. The tail transitions from each of these schema gave pairs in the other scheme and the process of reduction and instantiation of transitions had to be performed twice, once for each scheme in the bisimulation.

This next example requires rather more complex reasoning than previously used in order to determine instantiations for transitions, variables etc.

Example 3.5 *For all functions, g, which have well–defined inverses*

$$\forall a, l.\ delete(a, map(g, l)) \sim map(g, delete(g^{-1}(a), l))$$

$$delete(A, nil) \overset{\text{red}}{\rightsquigarrow} nil \tag{3.43}$$
$$H = A \to delete(A, H :: T) \overset{\text{red}}{\rightsquigarrow} delete(A, T) \tag{3.44}$$
$$H \neq A \to delete(A, H :: T) \overset{\text{red}}{\rightsquigarrow} H :: delete(A, T) \tag{3.45}$$

Proof. Let $\mathcal{R}' = \{\langle delete(A, map(G, L)),\ map(G, delete(G^{-1}(A), L))\rangle\}$

The first premise of (3.14) is trivially discharged leaving the second premise, $\mathcal{R}' \subseteq \langle \mathcal{R}' \cup \sim\rangle$.

Using (3.22) produces the following subgoal:

$$\begin{array}{rc} & \forall \mathcal{R}.\langle delete(A, map(G, L)),\ map(G, delete(G^{-1}(A), L))\rangle \in \mathcal{R} \Rightarrow \\ \forall \alpha. & ((delete(A', map(G', L')) \overset{\alpha}{\to} \phi \vee map(G', delete(G'^{-1}(A'), L')) \overset{\alpha}{\to} \psi) \ \Rightarrow \\ & ((delete(A', map(G', L')) \overset{\alpha}{\to} \phi \wedge map(G', delete(G'^{-1}(A'), L')) \overset{\alpha}{\to} \psi) \ \wedge \\ & \langle \phi,\ \psi\rangle \in \mathcal{R} \cup \sim)) \end{array} \tag{3.46}$$

As in example 3.3 reduction can't take place until the value of L' is known. Since L' is a universally quantified variable of list type it's value is either nil or $H :: T$ for some H and T. However the situation is more complicated than this.

Consider the following 3 cases

1. There is an element E of L' such that $G'(E) \neq A$. Let H' be the first such element appearing in L'. Let T' be the list of the remaining elements of L' that appear after H' in the order that they appear. In this case $delete(A', map(G', L'))$ evaluates to $G'(H') :: delete(A', map(G', T')$ and $map(G', delete(G'^{-1}(A'), L'))$ evaluates to $G'(H') :: map(G', delete(G'^{-1}(A'), T'))$. These are both values and the transitions are `hd` and `tl`.

$$delete(A', map(G', L')) \overset{\mathtt{hd}}{\to} G'(H')$$
$$delete(G'^{-1}(A'), L')) \overset{\mathtt{hd}}{\to} G'(H')$$

$$delete(A', map(G', L')) \overset{\mathtt{tl}}{\to} delete(A', map(G', T')$$
$$delete(G'^{-1}(A'), L')) \overset{\mathtt{tl}}{\to} map(G', delete(G'^{-1}(A'), T'))$$

$\langle G'(H'),\ G'(H')\rangle \in\sim$ by the reflexivity of $\sim$ and $\langle delete(A', map(G', T')),\ map(G', delete(G'^{-1}(A'), T'))\rangle \in \mathcal{R}$ by appeal to the hypothesis.

2. There is no element E in L' such that $G'(E) \neq A$ and L' is finite. $delete(A', map(G', L'))$ and $map(G', delete(G'^{-1}(A'), L'))$ will both evaluate to nil. The transition from nil is `nil` so $\phi = \psi = \perp$ and $\langle\phi,\ \psi\rangle \in\sim$ by the reflexivity of $\sim$.
3. There is no element E in L' such that $G'(E) \neq A$ and L' is infinite. $delete(A', map(G', L'))$ and $map(G', delete(G'^{-1}(A'), L'))$ have no value. $delete$ will continue to consume the head of L' forever. In effect this is a non–terminating program. In this case there are no possible transitions that apply to either side of the relations and (3.46) is trivially true.

Hence $\mathcal{R}'$ is a bisimulation. $\Box$.

This proof required fairly complex reasoning and relied on some ability to recognise that reduction was non–terminating in the third case. This sort of reasoning is rather ill–defined and proofs which require it have been omitted from the scope of the automation described later in this thesis.

This next proof is fairly straightforward and is included to show coinduction being used with a non–list type.

Example 3.6

$$\forall x, y.x + y \sim y + x$$

$+$ has its usual recursive definition:

$$0 + Y \overset{\text{red}}{\rightsquigarrow} Y \tag{3.47}$$

$$s(X) + Y \overset{\text{red}}{\rightsquigarrow} s(X + Y) \tag{3.48}$$

The following are standard lemmata about $+$ which will be used in the proof:

$$Y + 0 \overset{\text{red}}{\rightsquigarrow} Y \tag{3.49}$$

$$X + s(Y) \overset{\text{red}}{\rightsquigarrow} s(X + Y) \tag{3.50}$$

Proof. Let $\mathcal{R}' = \{\langle X + Y,\ Y + X\rangle\}$

The first premise of (3.14) is trivially discharged leaving the second premise, $\mathcal{R}' \subseteq \langle \mathcal{R}' \cup \sim\rangle$.

Using (3.22) produces the following subgoal:

$$\forall \alpha.\ \begin{array}{c} \forall \mathcal{R}.\langle X + Y,\ Y + X\rangle \in \mathcal{R} \Rightarrow \\ ((X' + Y' \overset{\alpha}{\rightarrow} \phi \vee Y' + X' \overset{\alpha}{\rightarrow} \psi) \ \ \Rightarrow \\ ((X' + Y' \overset{\alpha}{\rightarrow} \phi \wedge Y' + X' \overset{\alpha}{\rightarrow} \psi) \ \ \wedge \\ \langle \phi,\ \psi\rangle \in \mathcal{R} \cup \sim)) \end{array} \tag{3.51}$$

Once again $X' + Y'$ and $Y' + X'$ can't be reduced without knowing the values of X' and Y'. Recall that the values of natural numbers are 0 and $s(N)$. There are three cases:

- If $X' = 0$ and $Y' = 0$ then $\alpha = \mathtt{0}$ and $\phi = \psi = \bot$, hence $\langle \phi,\, \psi \rangle \in\sim$ by the reflexivity of $\sim$.
- If $X' = 0$ and $Y' = s(N)$ then $X' + Y'$ and $Y' + X'$ both evaluate to $s(N)$. $\alpha = \mathtt{p}$ and $\phi = \psi = N$ hence $\langle \phi,\, \psi \rangle \in\sim$.
- If $X' = s(N)$ then $X' + Y'$ evaluates to $s(N + Y')$ and $Y' + X'$ evaluates to $s(Y' + N)$, again $\alpha = \mathtt{p}$ and

$$\begin{aligned} X' + Y' &\xrightarrow{\mathtt{p}} N + Y' \\ Y' + X' &\xrightarrow{\mathtt{p}} Y' + N \end{aligned} \tag{3.52}$$

  $\langle N + Y',\, Y' + N \rangle \in \mathcal{R}$ by appeal to the hypothesis.

Hence $\mathcal{R}'$ is a bisimulation. $\square$.

The last example is one where the bisimulation involves a pair scheme that is a generalisation of the expressions appearing in the statement of the theorem.

Example 3.7

$$\forall f, x. h(f, x) \sim \mathit{iterates}(f, x) \tag{3.53}$$

$$h(F, X) \quad \overset{\text{red}}{\rightsquigarrow} \quad X :: \mathit{map}(F, h(F, X)) \tag{3.54}$$

Proof. Let $\mathcal{R}' = \{\langle \mathit{map}(F)^N(h(F, X)),\, \mathit{iterates}(F, F^N(X)) \rangle\}$

Where $(\cdots)^N$ is defined to be:

$$F^0(X) \quad \overset{\text{red}}{\rightsquigarrow} \quad X \tag{3.55}$$

$$F^{s(N)}(X) \quad \overset{\text{red}}{\rightsquigarrow} \quad F(F^N(X)) \tag{3.56}$$

and the following lemmata about $(\cdots)^N$ are assumed:

$$F^N(F(X)) \quad \overset{\text{red}}{\rightsquigarrow} \quad F(F^N(X)) \tag{3.57}$$

$$(\mathit{map}(F)^N)(H :: T) \quad \overset{\text{red}}{\rightsquigarrow} \quad F^N(H) :: (\mathit{map}(F)^N)(T) \tag{3.58}$$

The first premise of (3.14) applied to (3.53) gives the subgoal

$$\langle h(F, X),\, \mathit{iterates}(F, X) \rangle \in \{\langle (\mathit{map}(F))^N(h(F, X)),\, \mathit{iterates}(F, F^N(X)) \rangle\} \tag{3.59}$$

We know that $\langle (\mathit{map}(F))^0(h(F, X)),\, \mathit{iterates}(F, F^0(X)) \rangle \in \mathcal{R}'$ by the definition of $\mathcal{R}'$.

$$(map(F))^0(h(F,X)) \overset{\text{red}}{\rightsquigarrow} h(F,X) \tag{3.60}$$
$$iterates(F, F^0(X)) \overset{\text{red}}{\rightsquigarrow} iterates(F,X) \tag{3.61}$$

Hence $\langle h(F,X), iterates(F,X)\rangle \in \{\langle (map(F))^N(h(F,X)), iterates(F, F^N(X))\rangle\}$ by definition 3.10.

The second premise of (3.14) is $\mathcal{R}' \subseteq \langle \mathcal{R}' \cup \sim\rangle$.

Using (3.22) produces the following goal subgoal:

$$\begin{array}{ll} & \forall \mathcal{R}.\langle map(F)^N(h(F,X)),\ iterates(F,F^N(X))\rangle \in \mathcal{R} \Rightarrow \\ \forall \alpha. & (map(F')^{N'}(h(F',X')) \xrightarrow{\alpha} \phi \vee iterates(F', F'^{N'}(X')) \xrightarrow{\alpha} \psi) \ \Rightarrow \\ & (map(F')^{N'}(h(F',X')) \xrightarrow{\alpha} \phi \wedge iterates(F', F'^{N'}(X')) \xrightarrow{\alpha} \psi \ \wedge \\ & \langle \phi,\ \psi\rangle \in \mathcal{R} \cup \sim) \end{array} \tag{3.62}$$

$map(F')^{N'}(h(F',X'))$ evaluates to $F'^{N'}(X') :: map(F')^{s(N')}(h(F',X'))$ and $iterates(F', F'^{N'}(X'))$ evaluates to $F'^{N'}(X') :: iterates(F', F'^{s(N')}(X'))$.

The possible transitions are `hd` and `tl`

$$F'^{N'}(X') :: map(F')^{s(N')}(h(F',X')) \xrightarrow{\texttt{hd}} F'^{N'}(X')$$
$$F'^{N'}(X') :: iterates(F', F'^{s(N')}(X')) \xrightarrow{\texttt{hd}} F'^{N'}(X')$$

$$F'^{N'}(X') :: map(F')^{s(N')}(h(F',X')) \xrightarrow{\texttt{tl}} map(F')^{s(N')}(h(F',X'))$$
$$F'^{N'}(X') :: iterates(F', F'^{s(N')}(X')) \xrightarrow{\texttt{tl}} iterates(F', F'^{s(N')}(X'))$$

Hence $\mathcal{R}'$ is a bisimulation. □

NB. Example 3.6 could also have been proved using a generalised bisimulation, $\mathcal{R}' \overset{\text{def}}{=} \{\langle X + s^N(Y),\ Y + s^N(X)\rangle\}\}$, without the need for the additional lemmata.

3.7 Type–Checking

In the final section of this chapter I'm going to examine the use of coinduction for type–checking The only instance I've come across where coinduction is used for this is due to Paulson [Paulson 93] and so isn't framed by the operational semantics of a functional language. However in section 3.7.1 I hope to show how it can be placed in such a framework and provides another LTS. Paulson has a function on sets of lists of type $list(\tau)$ called $list\_fun$

$$list\_fun(\mathcal{S}, U) \stackrel{\text{def}}{=} \{h :: t | h \in U \wedge t \in \mathcal{S}\} \tag{3.63}$$

The least fixedpoint of $list\_fun$ is the set of strict or finite lists. The greatest fixedpoint is the set of lazy lists of elements of some set U, $llist(U)$. The coinduction rule can thus be used to show that some list, l is of type $gfp(list\_fun)$.

Example 3.8

$$M \in U \Rightarrow lconst(M) \in llist(U)$$

Where $lconst(M) \stackrel{\text{red}}{\leadsto} M :: lconst(M)$

Proof.

The coinduction rule is:

$$\frac{a \in \mathcal{S}' \quad \mathcal{S}' \subseteq list\_fun(\mathcal{S}, U)}{a \in llist(U)} \tag{3.64}$$

Let

$$\mathcal{S}' \stackrel{\text{def}}{=} \{lconst(M) \mid M \in U\} \tag{3.65}$$

Clearly $lconst(M) \in \mathcal{S}$ which leaves us with the goal:

$$\{lconst(M) \mid M \in U\} \subseteq list\_fun(\{lconst(M) \mid M \in U\}, U) \tag{3.66}$$

Rewriting this according to (3.63) gives

$$\{lconst(M) \mid M \in U\} \subseteq \{h :: t \mid h \in U \wedge t \in \{lconst(M) \mid M \in U\}\} \tag{3.67}$$

An inference rule like (3.22) is needed here:

$$\frac{\begin{array}{cc} \forall \mathcal{S}.\forall 1 \leq i \leq n. & \forall \mathcal{S}.\forall 1 \leq i \leq n. \\ \bigwedge_{i=1}^{n} a_i \in \mathcal{S} \Rightarrow hd(a_i) \in U & \bigwedge_{i=1}^{n} a_i \in \mathcal{S} \Rightarrow tl(a_i) \in \mathcal{S} \end{array}}{\bigcup_{i=1}^{n}\{a_i\} \subseteq llist\_fun(\bigcup_{i=1}^{n}\{a_i\}, U)} \tag{3.68}$$

(Again this is justified in appendix D).

This produces the subgoals

$$\begin{array}{c}\forall \mathcal{S}.(M \in U \Rightarrow lconst(M) \in \mathcal{S}) \Rightarrow (M' \in U \Rightarrow hd(lconst(M')) \in U) \\ \forall \mathcal{S}.(M \in U \Rightarrow lconst(M) \in \mathcal{S}) \Rightarrow (M' \in U \Rightarrow tl(lconst(M')) \in \mathcal{S})\end{array} \tag{3.69}$$

Rewriting $lconst(M')$ to $M' :: lconst(M')$ and then rewriting the coinduction conclusions using $hd(H :: T) = H$ and $tl(H :: T) = T$ proves the theorem since $M' \in U$ by assumption and $lconst(M') \in \mathcal{S}$ by appeal to the hypothesis. □

Once again the process of this proof was very similar to that of the previous proofs. A set (this time *not* a relation) was introduced by the coinduction rule. Proving this set was a member of the greatest fixedpoint involved reduction and then appeal to the hypothesis.

3.7.1 Adapting Type Checking to Labelled Transition Systems

Since coinduction is usually used with reference to labelled transition systems it is interesting to see whether the above example can be placed in such a system and how that effects the process of proof.

Paulson defines types as greatest fixedpoints; however in most presentations of labelled transition systems types are defined by an inductively given relation[2] – for a typical example see [Gordon 95a].

If we're looking for a "coinductive definition" of types then we have to talk not about how one element of the type is *constructed* out of previous elements of the type, but what the types are of elements reached by *observing* an element of the type. We would be replacing the static semantics of the language (see §3.2.1), by some operational type semantics.

So in a language containing only list types we would have a type semantics that looked something like figure 3–5 where the transition system is extended by *type observations* which act on pairs of expressions and types. $\perp$ is an arbitrary divergent program of some function type, $\pi \to \rho$. There are no observable transitions from $\perp$. This new type relation $:_c$ is defined as:

$$a :_c \tau \stackrel{\text{def}}{=} a \sim \langle a,\ \tau \rangle \tag{3.75}$$

Under this formulation the proof of example 3.8 is as follows.

[2] Andy Gordon, private communication.

$$\frac{b :_c \tau}{\langle e_1,\, \tau \rightarrow \sigma\rangle \xrightarrow{b} \langle e_1(b),\, \sigma\rangle} \tag{3.70}$$

$$\langle nil,\, llist(\tau)\rangle \xrightarrow{\texttt{nil}} \langle \bot,\, \pi \rightarrow \rho\rangle \tag{3.71}$$

$$\langle e_1 :: e_2,\, llist(\tau)\rangle \xrightarrow{\texttt{hd}} \langle e_1,\, \tau\rangle \tag{3.72}$$

$$\langle e_1 :: e_2,\, llist(\tau)\rangle \xrightarrow{\texttt{tl}} \langle e_2,\, llist(\tau)\rangle \tag{3.73}$$

$$\frac{\langle a,\, \tau\rangle \quad a \overset{\text{red}}{\rightsquigarrow} a' \quad \langle a',\, \tau\rangle \xrightarrow{\alpha} \langle b,\, \sigma\rangle}{\langle a,\, \tau\rangle \xrightarrow{\alpha} \langle b,\, \sigma\rangle} \tag{3.74}$$

Figure 3–5: Observations on List Types

Proof. Let $\mathcal{R} \overset{\text{def}}{=} \{\langle lconst(M),\, \langle lconst(M),\, llist(\tau)\rangle\rangle | M :_c \tau\}$. The first premise of the coinduction rule is trivial which leaves the goal.

$$\begin{gathered}\{\langle lconst(M),\, \langle lconst(M),\, llist(\tau)\rangle\rangle | M :_c \tau\} \subseteq \\ \langle\{\langle lconst(M),\, \langle lconst(M),\, llist(\tau)\rangle\rangle | M :_c \tau\} \cup \sim\rangle)\end{gathered} \tag{3.76}$$

(3.22) produces the subgoal:

$$\begin{gathered}M :_c \tau \Rightarrow \langle lconst(M),\, \langle lconst(M),\, llist(\tau)\rangle\rangle \in \mathcal{R} \Rightarrow \\ \forall \alpha . M' :_c \tau \Rightarrow ((lconst(M') \xrightarrow{\alpha} \phi \vee \langle lconst(M'),\, llist(\tau)\rangle \xrightarrow{\alpha} \langle \phi,\, \tau_\phi\rangle) \Rightarrow \\ ((lconst(M') \xrightarrow{\alpha} \phi \wedge \langle lconst(M'),\, llist(\tau)\rangle \xrightarrow{\alpha} \langle \phi,\, \tau_\phi\rangle \wedge) \\ \langle \phi,\, \tau_\phi\rangle \in \mathcal{R} \cup \sim))\end{gathered}$$

Reduction and analysis of the possible transitions gives two goals:

$$\begin{gathered}(M :_c \tau \Rightarrow \langle lconst(M),\, \langle lconst(M),\, llist(\tau)\rangle\rangle \in \mathcal{R}) \Rightarrow \\ (M' :_c \tau \Rightarrow \langle lconst(M'),\, \langle lconst(M'),\, llist(\tau)\rangle\rangle \in \mathcal{R} \cup \sim\})\end{gathered} \tag{3.77}$$

$$\begin{gathered}(M :_c \tau \Rightarrow \langle lconst(M),\, \langle lconst(M),\, llist(\tau)\rangle\rangle \in \mathcal{R}) \Rightarrow \\ (M' :_c \tau \Rightarrow \langle M',\, \langle M',\, \tau\rangle\rangle \in \mathcal{R} \cup \sim)\end{gathered} \tag{3.78}$$

(3.77) is true by appeal to the hypothesis and 3.78 is true since $M' :_c \tau$ is assumed and is equivalent to $M' \sim \langle M',\, \tau\rangle$. □

3.8 Conclusion

The use of labelled transition systems is becoming more widespread and the most common use of coinduction is with list datatypes. As a result this chapter has focused on the operational semantics of lazy functional languages and on examples of coinduction this setting. It also looked briefly at an alternative formation for list examples involving the function $llistD\_fun$.

A general pattern for the mechanics of a coinductive proof has emerged which has involved providing some relation $\mathcal{R}$ in terms of pair schema, analysing the reduction behaviour of these pair schema in order to find the Weak Head Normal Forms of each member of the scheme (if they exist), as a result transition labels are instantiated and the definition of $\mathcal{R}$ or the reflexivity of $\sim$ is appealed to in order to complete the proof. It is this sort of general pattern for a family of proofs that forms the basis for proof plans (discussed in chapter 4) and this particular pattern that forms the basis for the proof strategy for coinduction proposed in chapter 5.

Although the focus of the examples was on one particular language, the use of coinductive proof for type–checking was also examined and a similar pattern emerges here as well.

Chapter 4

Proof Planning

4.1 Introduction

This chapter discusses proof planning.

It uses the proof plan for induction, in particular the Rippling heuristic and associated Wave method, as an example throughout the chapter to illustrate the various ideas associated with proof planning.

It starts with a general discussion of proof planning and then looks in more detail at proof methods and proof critics.

4.2 Proof Planning

Proof plans were first proposed by Bundy [Bundy 88] and have been successfully applied to inductive theorem proving and other domains. Proof plans have two basic components, proof *methods* and proof *tactics*. The tactics are combinations of low-level inference rule applications. Methods characterise the tactics by specifying pre–conditions and outputs of their application. The idea is to make a plan of the tactics needed to conduct a given proof in advance of applying those tactics. A completed proof plan is executed by executing the tactic part of the plan by giving it to a tactic based theorem prover which will provide a formal verification of the theorem. The object is to separate *proof discovery* from *proof checking*. A general observation is that proof methods often embody various heuristics for navigating the search space, whereas tactics do not.

Proof planning has been implemented in *CLAM* [Bundy *et al* 90b] and Omega [Benzmüller *et al* 97]. The discussion in this chapter is based on the implementation in *CLAM* although it is intended to be general.

Coinductive proof discovery, especially the discovery of an appropriate bisimulation, is a significant task and this makes proof planning an attractive option in any attempt to automate or provide proof tools to support coinduction.

4.2.1 Proof Tactics

The proof rules for many logics are very low level. Proofs constructed from just the basic rules of inference in these logics are often long and hard to understand, moreover many stretches of such a proof will appear to be trivial. The idea behind tactics is to combine sequences of the basic inference rules together. In particular to combine sequences which frequently occur and correspond to some high level task. Often tactics require certain parameters to be passed to them by the user, for instance the theorem prover Isabelle [Paulson 94a] has a tactic `res_inst_tac` described in the manual as:

> `res_inst_tac` *insts thm i* instantiates the rule *thm* with the instantiation *insts*, as described above, and then performs resolution on subgoal *i*. Resolution typically causes further instantiations, you need not give explicit instantiations for every variable in the rule.

insts, thm and *i* all have to be provided by the user.

4.2.2 Proof Methods

Proof methods are often described as partial specifications of proof tactics. They consist of 3 slots:

Preconditions	Conditions which must hold for the method to apply
Outputs	Subgoals generated by the method, a list of meta–level sequents
Tactic	The name of the tactic that constructs the piece of object–level proof corresponding to the method

The last slot, the tactic slot, depends upon the object level theorem prover to which the plans are to be passed. In nearly all *CLAM* methods this is *Oyster* [Horn 88] [Horn & Smaill 90]. The work in this thesis hasn't been linked to an object level prover although linking it to Isabelle has been investigated (appendix E).

4.2.3 The Planning Mechanism

A proof planning system links together methods by matching the pre–conditions of one method to the outputs of another. This sequence of tactics is called the *proof plan* for that theorem. It not only specifies which tactics to apply, but where appropriate it should also supply any parameters the tactics require.

The process of linking methods in *CLAM* is generally performed in a simple depth first manner. If more than one method applies to a given goal *CLAM* will apply the first in some (possibly user defined) list, only trying the others if it fails to find a proof plan and is forced to backtrack. It is also possible to plan proofs breadth first or with iterative deepening. The relative merits of these various search strategies have been extensively documented in Artificial Intelligence texts (such as [Luger & Stubblefield 93]). More sophisticated search techniques have also been proposed, such as best first search and depth–bounded discrepancy search [Walsh 97], and in some cases implemented [Manning *et al* 93]. In this thesis it is assumed that a depth first strategy is being employed.

4.2.4 Proof Critics

The proof strategy provides a guide as to which proof methods should be chosen at any given stage of the proof. Knowing which method is expected to apply gives additional information should the system generating the plan fail to apply it. Since heuristics are employed in the generation of proof plans it is possible for a proof planning system to fail to find a plan even when one exists. To this end *proof critics* can be employed to analyse the reasons for failure and propose alternative choices or even suggest finding new information in order to complete the proof plan.

4.3 Proof Planning Induction

The most widely explored application of proof planning has been proof by induction. This has the benefit of being more generally familiar than coinduction, so I intend to use this as an illustration of the proof planning process.

Most people who have performed an inductive proof will have some sort of proof plan in their head. Figure 4–1 shows such a proof plan. To perform induction you start out by choosing an *induction scheme* or rule, for instance:

$$\frac{P(0) \quad P(n) \vdash P(s(n))}{\forall n. P(n)} \tag{4.1}$$

This splits the proof into one or more *base cases* and *step cases*. These then have to be proved in turn. The base case may be trivial, or require further induction. The step case proof revolves around rewriting the *induction conclusion* so that the *induction hypothesis* may be used to complete the proof (this is called *fertilization*).

Figure 4–1 is a general proof plan for induction; however any given inductive proof will have a specific proof plan, which will include details of the induction scheme to be used and the specific rewrite rules required. Following Richardson [Richardson 95] I shall call the general proof plan the *proof strategy* and reserve the term proof plan for specific instances of proof strategies.

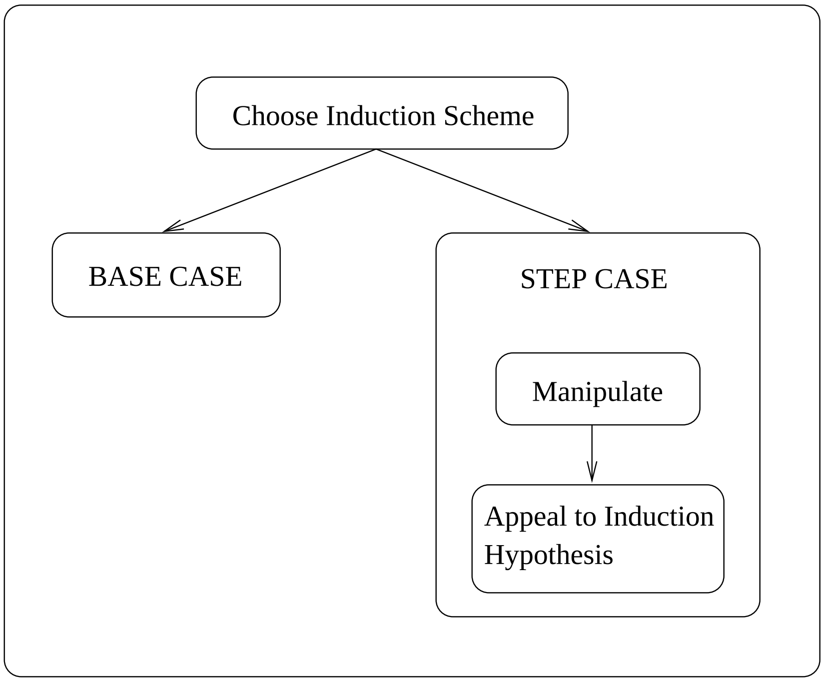

Figure 4–1: A Proof Plan for Mathematical Induction

4.4 The Wave Method

I'm going to use the Wave method as an example of a proof method. It is an important proof method for induction and controls the rewriting of the step case goals. It relies on a number of formal definitions which are stated here, however it is only necessary for the reader to grasp the intuition behind rippling. The formal definitions are not required in the rest of this thesis and can be omitted if so wished.

Definition 4.1 *[Thomas & Jantke 89] A* **rewrite rule** *is an ordered pair of terms of the form $l \leadsto r$.*

Definition 4.2 *[Thomas & Jantke 89] A* **term rewriting system** *consists of a set of rewrite rules.*

This means that $\leadsto$ forms a relation in a term rewriting system.

Rewrite rules are applied by the *rewrite rule of inference*, (4.2) [Bundy 83]:

$$\frac{exp[sub] \quad lhs \leadsto rhs}{exp[rhs\,\phi]} \tag{4.2}$$

where ϕ is a most general substitution such that $lhs\,\phi \equiv sub$.

Let $\rightsquigarrow^*$ denote the reflexive transitive closure of $\rightsquigarrow$. Term rewriting systems are used to implement *equational theories* (among other things). The intention is to have a decision procedure for equality: terms u, v are equal in the equational theory if there is some w such that $u \rightsquigarrow^* w$ and $v \rightsquigarrow^* w$. In general rewrite rules embody some concept of similarity or equality between expressions.

Definition 4.3 *[Bundy 83] If, by ignoring the order of the pairs and allowing rewriting with $rhs \rightsquigarrow lhs$ as well as $lhs \rightsquigarrow rhs$, one expression can be rewritten into another then the two expression are said to be* **similar** *with respect to the set of rules.*

Since rewrite rules tend to embody some notion of equality it would be natural to expect reflexivity, symmetry, transitivity and congruence closure to be important properties connected with them. However symmetry, in particular, introduces termination problems for the use of the rule of inference.

One response to this is, while allowing $\overset{\text{red}}{\rightsquigarrow}$ to be an equivalence relation, to disallow unrestricted application of (4.2) and guide it instead by heuristics. Another approach is to exploit ideas of normal forms and confluence (rewriting to identical normal forms) [Huet & Oppen 80].

Rippling is a heuristic for guiding rewriting to prevent non–termination and it is embodied in the Wave method in *CL^AM*. It is a heuristic that is specifically tailored for use with induction. It guides the application of (4.2) by a process of meta–level annotation of the object–level terms. Rippling requires a match between the annotations on the expression and the LHS of the rule as well as between the expression and the LHS themselves. This effectively restricts a (possibly symmetric) relation, $\rightsquigarrow$, to a well–founded order. I shall use $\rightsquigarrow$ for both orders (i.e. the order on unannotated terms and the order on annotated terms). In any situation it should be clear from the context which is being used.

The Wave method is motivated by the intuition that the step case of an inductive proof involves an *induction hypothesis* which is embedded within the *induction conclusion*. Rippling aims to eliminate the difference between the conclusion and the hypothesis. The meta–level annotations that control the rewriting make these differences explicit.

4.4.1 Annotations and Difference Matching

The annotations are described as *meta–level* because they do not add any information on the object level, instead they are intended to provide information about where the rewriting process is heading. The annotations are determined by *difference matching* [Basin & Walsh 92] the induction hypothesis and conclusion. Much of the following discussion is based on [Basin & Walsh 96].

Definition 4.4 *[Walsh 96] A* **wave annotation** *consists of a* **wave front***, a box, with one or more* **wave holes***, distinct proper subterms of the wave front which are underlined.*

The intention is that the wave front (excluding the wave hole) is the difference between the induction hypothesis and conclusion. This gives some sort of special status to everything in the expression that is not in the wave front (excluding the wave hole). This is called the *skeleton*.

Definition 4.5 *[Basin & Walsh 96] The* **skeleton** *of an expression, exp, is defined inductively as the set, skel(exp).*

1. *If exp is a variable then* $skel(exp) = \{exp\}$.
2. *If* $exp = \boxed{f(t_1, \cdots, t_n)}$ *then the* $skel(exp) = \{s \mid \exists i. t_i = \underline{t'_i} \wedge s \in skel(t'_i)\}$
3. *If* $exp = f(t_1, \cdots, t_n)$ *then* $skel(exp) = \{f(s_1, \cdots, s_n) \mid \forall i. s_i \in skel(t_i)\}$

Definition 4.6 *[Basin & Walsh 96] The* **erasure** *is the unannotated term corresponding to the annotated one. It is also defined inductively using the function erase.*

1. *If exp is a variable then* $erase(exp) = exp$;
2. *If* $exp = \boxed{f(t_1, \cdots, t_n)}$ *then* $erase(exp) = f(s_1, \cdots, s_n)$ *where if* $t_i = \underline{t'_i}$ *then* $s_i = erase(t'_i)$ *else* $s_i = erase(t_i)$;
3. *If* $exp = f(t_1, \cdots, t_n)$ *then* $erase(exp) = f(s_1, \cdots, s_n)$ *where* $s_i = erase(t_i)$.

Definition 4.7 *[Walsh 96]* **Difference matching** *is a process which annotates a term s with respect to a term, t such that s′ is a* **difference match** *of s and t iff* $skeleton(s') = t$ *and* $erase(s') = s$.

Example 4.1 (An Example of A Meta–Level Annotation) *Consider the theorem*

$$\forall f : \rho \to \sigma . g : \tau \to \sigma . l : list(\tau).\ map(f \circ g, l) = map(f, map(g, l))$$

An inductive proof of this theorem will lead to the step case goal:

$$\begin{array}{c} map(F \circ G, l) = map(F, map(G, l)) \qquad \Rightarrow \\ map(F' \circ G', h :: l) = map(F', map(G', h :: l)) \end{array} \tag{4.3}$$

Difference matching the induction conclusion and hypothesis gives the annotated goal

$$\begin{array}{c} map(F \circ G, l) = map(F, map(G, l)) \Rightarrow \\ map(F' \circ G', \boxed{h :: \underline{l}}) = map(F', map(G', \boxed{h :: \underline{l}})) \end{array} \tag{4.4}$$

Basin and Walsh present an algorithm to perform difference matching [Basin & Walsh 92].

4.4.2 Annotated Rewrite Rules

Rewrite rules are annotated for rippling. They are not annotated by difference matching, but by nominating a skeleton on each side of the rewrite. A rewrite rule may be annotated in different ways for a number of different skeletons, and so give rise to several annotated rules.

Consider the rewrite rule

$$map(F, H :: T) \rightsquigarrow F(H) :: map(F, T) \tag{4.5}$$

Annotating this with $\{map(F, T)\}$ as the skeleton on both sides gives the annotated rewrite rule

$$map(F, \boxed{H :: \underline{T}}) \rightsquigarrow \boxed{F(H) :: \underline{map(F, T)}} \tag{4.6}$$

Definition 4.8 (Annotated Rewrite Rule Application) *An annotated rewrite rule, $lhs \rightsquigarrow rhs$, may be applied to an annotated term, T, to yield an annotated term, T' if:*

1. *S is a subterm of T,*
2. *there is a substitution σ such that:*
   (a) *$erase(S) = erase(lhs)\sigma$*
   (b) *$skeleton(S) = skeleton(lhs)\sigma$*
   (c) *$T' = T[rhs/S]\sigma$*

Wave rules are a special case of annotated rules. They are formally defined in definition 4.18. Informally they are annotated rewrite rules which are *skeleton preserving* and *measure decreasing* under an appropriate ordering on annotated terms.

Definition 4.9 *[Basin & Walsh 96] An annotated rule is* **skeleton preserving** *if some of the skeletons on the LHS also appear on the RHS and no new skeletons are introduced, i.e. $skel(LHS) \supseteq skel(RHS)$*

4.4.3 Rippling

The following discussion of *rippling in* and *rippling out* is intended to motivate the definition of the measure on annotated terms in §4.4.4. Annotations are extended with an *orientation* indicated by either $\uparrow$ for an *outward wave front* or $\downarrow$ for an *inward wave front*. The previous definitions are extended in the obvious way to include these extra annotations. These oriented wave fronts are used to either *ripple out* or to *ripple in*.

Rippling Out

The most simple form of rippling, *rippling out*, constrains wave front to move "outwards" in the term structure. Informally, a wave rule is an outward wave rule if more of the skeleton appears in the wave holes on the RHS than appears in them on the LHS. Hence (4.6) is an outward wave rule and so is annotated with outward wave fronts as shown in (4.7).

$$map(F, \boxed{H :: \underline{T}}^{\uparrow}) \leadsto \boxed{F(H) :: \underline{map(F,T)}}^{\uparrow} \tag{4.7}$$

The purpose of the heuristic is to drive the differences between the induction hypothesis and conclusion right to the outside of the term structure. It is the experience of people doing inductive proof that this process tends to allow the differences to be canceled away, or for the induction hypothesis to be applied as a rewrite rule to the wave hole. The process of applying wave rules is called rippling, since, in this outward form, the wave rules "ripple" out like waves in a pool into which something has been dropped.

In example 4.1, (4.7) is applied to the induction conclusion as follows:

$$map(F \circ G, \boxed{h :: \underline{l}}^{\uparrow}) = map(F, map(G, \boxed{h :: \underline{l}}^{\uparrow})) \tag{4.8}$$

$$\boxed{F \circ G(h) :: \underline{map(F \circ G, l)}}^{\uparrow} = map(F, \boxed{G(h) :: \underline{map(G,l)}}^{\uparrow}) \tag{4.9}$$

$$\boxed{F \circ G(h) :: \underline{map(F \circ G, l)}}^{\uparrow} = \boxed{F(G(h)) :: \underline{map(F, map(G,l))}}^{\uparrow} \tag{4.10}$$

At this point the induction hypothesis can be used to rewrite one side of the equation to produce an expression that is identical to that on the other side of the equation, providing $F \circ G(h)$ is defined as $F(G(h))$. As a result the goal is trivially true.

Rippling In

It is also possible to *ripple in*. An inward wave rule is the opposite of an outward wave rule. Less of the skeleton appears in wave holes on the RHS of an inward wave rule than appears inside the wave fronts on the LHS.

Thus the object–level rewrite rule system may be symmetric, but the meta–level system won't be because the orientation of the wave fronts will be different. For instance the following pair of rules would form a non–terminating rewrite rule system at the object level, but the meta–level annotations don't match so no looping occurs.

$$map(F, \boxed{H :: \underline{T}}^{\uparrow}) \leadsto \boxed{F(H) :: \underline{map(F,T)}}^{\uparrow} \tag{4.11}$$

$$\boxed{F(H) :: \underline{map(F,T)}}^{\downarrow} \leadsto map(F, \boxed{H :: \underline{T}}^{\downarrow}) \tag{4.12}$$

An outward wave front on the left of a wave rule may "become"[1] an inward wave front on the right, but an inward wave front may not become an outward one. This is the central intuition in the wave rule measure. This imposes a heuristic of rippling out then rippling in.

The intention of rippling out was to move differences to the top of the term tree. If we start rippling in and moreover forbid inward wave fronts to become outward ones we preserve the differences between the induction hypothesis and conclusion somewhere in the middle of the term structure. Rippling in is of use in theorems where universally quantified variables appear in the induction conclusion. These variables can absorb the differences, provided the same difference structure is absorbed wherever the variable occurs (see example 4.2).

Definition 4.10 *[Bundy et al 93] A* **sink** *is a subterm of an annotated term that corresponds to a universally quantified variable in the hypothesis.*

Definition 4.11 *A wave front is said to be* **sinkable** *if its skeleton contains a sink.*

Wave fronts may only be rippled in if they are sinkable[2].

Sometimes the wave–fronts simply can't be annotated inwards or outwards, since the differences are moving "sideways" . Take the definition of $qrev$ as an example.

$$\begin{aligned} qrev(nil, L) &\rightsquigarrow L & (4.13)\\ qrev(H :: T, L) &\rightsquigarrow qrev(T, H :: L) & (4.14)\end{aligned}$$

If $\{qrev(T, L)\}$ is treated as the skeleton in (4.14) then it can be preserved on both sides of the rewrite. On the left T will appear inside the wave front while on the right L will. In this case the rule is annotated outwards on the left and inwards on the right.

$$qrev(\boxed{H :: \underline{T}}^{\uparrow}, M) \rightsquigarrow qrev(T, \boxed{H :: \underline{M}}^{\downarrow}) \qquad (4.15)$$

[1]Wave fronts are not actually paired off explicitly on either side of a wave rule so it is a bit misleading to talk about a wave front on the left becoming a wave front on the right, however it serves to convey the intuition behind the heuristic.

[2]The common annotation for a sink is $\lfloor \cdots \rfloor$ placed around a skolem constant. In induction it is unusual to have more than one sink in the induction conclusion; however in coinduction there are usually a great many and I felt that capitalisation provided a less fussy annotation, this difference is purely syntactic. $\lfloor \cdots \rfloor$ can be expanded to indicate that term structure is being absorbed by the sink.

Example 4.2 (Rippling In) *Consider the theorem*

$$\forall l, m : list(\tau).\ rev(l) <> m = qrev(l, m) \tag{4.16}$$

The step case goal is

$$\begin{gathered} rev(l) <> M = qrev(l, M) \Rightarrow \\ rev(\boxed{h :: \underline{l}}^{\uparrow}) <> M' = qrev(\boxed{h :: \underline{l}}^{\uparrow}, M') \end{gathered} \tag{4.17}$$

(4.15) is available as an annotated rule as are the following:

$$rev(\boxed{X :: \underline{Y}}^{\uparrow}) \rightsquigarrow \boxed{\underline{rev(Y)} <> X :: nil}^{\uparrow} \tag{4.18}$$

$$(\boxed{\underline{U} <> V}^{\uparrow}) <> W \rightsquigarrow U <> (\boxed{V <> \underline{W}}^{\downarrow}) \tag{4.19}$$

Using these the conclusion can be rippled as follows:

$$\begin{aligned} \boxed{\underline{rev(t)} <> h :: nil}^{\uparrow} <> M' &= qrev(\boxed{h :: \underline{t}}^{\uparrow}, M') && (4.20) \\ \boxed{\underline{rev(t)} <> h :: nil}^{\uparrow} <> M' &= qrev(t, \boxed{h :: \underline{M'}}^{\downarrow}) && (4.21) \\ rev(t) <> (\boxed{h :: nil <> \underline{M'}}^{\downarrow}) &= qrev(t, \boxed{h :: \underline{M'}}^{\downarrow}) && (4.22) \end{aligned}$$

$h :: nil <> M'$ can be simplified to $h :: M'$ making the conclusion

$$rev(t) <> (\boxed{h :: \underline{M'}}^{\downarrow}) = qrev(t, \boxed{h :: \underline{M'}}^{\downarrow}) \tag{4.23}$$

$h :: M'$ is a sink and the goal can be proved by appeal to the induction hypothesis.

The last simplification step is not a rippling step since $h :: nil$ and h are both unannotated subterms of the annotated terms. At this point rippling is said to be *blocked*, that is no further rippling can occur, but fertilization isn't possible. There are a number of normalisation techniques which can perform the necessary simplification to unblock rippling. A number of these have been transformed into proof methods. These techniques vary from one version of $CL^{A}M$ to another.

4.4.4 The Wave Rule Measure

Rippling out and rippling in, as discussed above, form the basis of the rippling heuristic. Termination of rippling has been proved via the provision of a *wave rule measure* [Basin & Walsh 96] which embodies the constraints outlined above.

Definition 4.12 *[Basin & Walsh 96] The* **position** *of a subterm of an expression is a path address represented by a string (concatenated by .). The subterm of a term t at position p is written t/p where:*

$$\begin{aligned} t/nil &= t \\ t/i.p &= s_i/p \end{aligned}$$

If s is a subterm of t at position p, its **depth** *is the length of p. The* **height** *of t, written $|t|$, is the maximal depth of any subterm of t.*

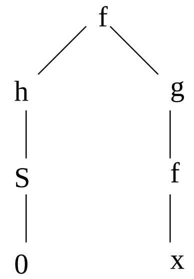

Figure 4–2: $f(h(s(0), g(f(x))))$ Represented as a Tree

For instance, in the figure, $h(s(0))$ is at position 1, $f(x)$ is at position 2.1.

Annotated terms are regarded as decorated trees where the tree is the skeleton and the wave fronts are boxes decorating the nodes. The function symbols in the skeleton can be abstracted away (since wave rules are skeleton preserving). The function symbols in wave fronts can also be ignored and replaced by their *width*: The number of nested function symbols between the root of the wave front and the wave hole. This is the *weight* of the wave front.

If an annotated expression has a wave front with more than one wave hole then it is *weakened* to a set of expressions whose wave fronts contain only a single hole. This is done by erasing all but one wave hole. A wave hole $\underline{t_i}$ is erased by removing the underline and erasing any further annotation in t_i. A wave front is *maximally weak* when it has exactly one wave hole. A term is *maximally weak* when all its wave fronts are maximally weak. $weakenings(s)$ is the set of maximal weakenings of a term s.

Definition 4.13 *[Basin & Walsh 96] The* **out–measure**, $MO(t)$, *of a maximally weak annotated term t is a list of length $|skel(t)|+1$ whose i–th element is the sum of the weights of all outward wave fronts at depth i. The* **in–measure**, $MI(t)$, *is a list whose i–th element is the sum of the weights of all inward wave fronts at depth i. The* **measure** *of an annotated term, $M(t)$ is the pair of out and in–measures, $\langle MO(t),\ MI(t)\rangle$.*

Definition 4.14 *Let $>_{lex}$ be the* **lexicographic order** *on lists of naturals defined by:*

$$h_1 > h_2 \Rightarrow h_1 :: t_1 \quad >_{lex} \quad h_2 :: t_2 \tag{4.24}$$
$$t_1 >_{lex} t_2 \Rightarrow h :: t_1 \quad >_{lex} \quad h :: t_2 \tag{4.25}$$

$>_{revlex}$ is the **reversed lexicographic order** *on list and $l_1 >_{revlex} l_2$ if $rev(l_1) >_{lex} rev(l_2)$.*

Definition 4.15 *[Basin & Walsh 96] The order on maximally weak annotated terms, $\succ$, is defined as: $t \succ s$ iff $skel(s) = skel(t)$ and either $MO(t) >_{revlex} MO(s)$ or $MO(t) = MO(s)$ and $MI(t) >_{lex} MI(s)$.*

Definition 4.16 *[Basin & Walsh 96] A* **multi–set ordering** *$>>$ is induced from a given ordering $>$ whereby $M >> N$ iff N can be obtained from M by replacing one or more elements in M by any finite number of elements each of which is smaller (under $>$) than one of the replaced elements.*

Definition 4.17 *[Basin & Walsh 96] For l and r annotated terms, $l \succ^* r$ iff $weakenings(l) \succ\!\succ weakenings(r)$ where $\succ\!\succ$ is the multi–set extension of the order for maximally weak terms.*

No proof of the well–foundedness of this measure is offered, the interested reader is referred to [Basin & Walsh 96].

4.4.5 Wave Rules

From this work we are now in a position to offer a formal definition of a *wave rule*.

Definition 4.18 *[Basin & Walsh 96] An annotated rewrite rule, $l \rightsquigarrow r$, is a* **wave rule** *if it is skeleton preserving and $l \succ^* r$.*

CLAM embodies rippling as the Wave method which is presented in figure 4–3.

Preconditions

1. There is an annotated term at *Position* in the Goal.
2. There is a wave rule that applies to that term and rewrites it to *Newterm*.
3. Any conditions on the application of the rule are satisfied

Outputs

Replace the term at *Position* with *Newterm*.

Figure 4–3: The *Wave* Method

4.5 Proof Critics

No claims for completeness are made for the proof methods used by *CLAM*. An acknowledged step in the theorem proving process is the conjecture of new information (for instance witnesses for existential quantifiers), this process has often been referred to as a *eureka step*. Two processes for handling eureka steps automatically have been implemented in *CLAM*. These are *proof critics* and *middle–out reasoning*, both of which are discussed below.

Ireland [Ireland 92] observed that failed proof attempts often provided important information that could be exploited to lead to a complete proof. In particular such failed attempts can be used to refine "guesses" previously made at unknown information. They can also be used to identify the fact that some information (e.g. a lemma) was missing. Proof critics exploit failed proof attempts in this way.

Critics are expressed in terms of preconditions and patches. The preconditions examine the reasons why the method has failed to apply; ideally they do this with reference to the preconditions of the proof method (in the implementation of *CLAM*3 some simple critics which deal with merging or splitting wave fronts do not refer to the methods' preconditions explicitly) although they may also include extra information. The proposed patch suggests some change to the proof plan. It may choose to propagate this change back through the plan and then continue from the current point, or it may choose to jump back to a previous point in the proof plan.

Typically, if a proof method fails to apply, a critic will analyse that failure to see if specific conditions could have been altered which would have allowed the method to apply. It is not necessary to have critics attached to every method, in fact, the proof plan for induction only attaches critics to the Wave method. The critics patch the proof in different ways depending upon which of the method's preconditions failed.

I am going to discuss one such critic, the Induction Revision critic (§4.5.2), as an example. The induction method uses heuristics to choose an induction scheme, hence it is possible that the scheme chosen may be inappropriate. The incorrect choice of scheme won't show itself until some point in the rewriting process when the Wave method will fail to apply but fertilization can't occur (as it should according to the proof strategy).

4.5.1 Middle–out Reasoning

The Induction Revision critic also utilises middle–out reasoning. Middle–out reasoning was first described by Bundy et al [Bundy *et al* 90a]. Middle–out reasoning postpones making eureka steps for as long as possible in the proof process, adapting methods to cope with partial information where necessary. In this way more information about the nature of the eureka step can be determined.

A particularly important tool in middle–out reasoning is the use of meta–variables to mark partially instantiated terms. This is potentially very explosive since there may be numerous potential instantiations for the meta–variable. Middle–out reasoning often employs a strategy of gradual instantiation of meta–variables which will probably need higher order unification. Higher order unification is undecidable and so techniques for controlling the instantiation of the meta–variables are vital for middle–out reasoning.

Hesketh [Hesketh 91] advocates the use of a unifiability algorithm based on that of Huet [Huet 75]. This algorithm, as well as testing for unifiability, generates substitution sets which in many cases are complete. She further suggests that the meta–level control information represented by the wave fronts can be used to further restrict the unification process to a matching process. Ireland extended this idea further to exploit the directionality of the wave fronts as well [Ireland & Bundy 96].

The meta–level language is extended in the light of this to include *potential wave fronts*, $\boxed{\cdots}$. These indicate positions at which wave fronts might appear, the contents of the wave front is indicated by a higher order meta–variable. For instance the expressions $\boxed{F_1(\underline{x})}^{\uparrow}$ indicates that there could be an outward wave front around x. These potential wave fronts can be matched against annotated terms, so $\boxed{F_1(\underline{x})}^{\uparrow}$ could be matched by x (no wave front), $\boxed{s(\underline{x})}^{\uparrow}$, $\boxed{z + \underline{x}}^{\uparrow}$ and many other expresions. In particular, expressions with potential wave fronts are often matched against the LHS of wave rules.

4.5.2 The Induction Revision Critic

Example 4.3

$$\forall l_1, l_2 : list(\tau).even(length(l_1 <> l_2)) \Leftrightarrow even(length(l_2 <> l_1)) \qquad (4.26)$$

The available wave rules from the definitions of *even*, <> and *length* include

$$even(\boxed{s(s(\underline{X}))}^{\uparrow}) \rightsquigarrow even(X) \tag{4.27}$$

$$length(\boxed{X :: \underline{Y}}^{\uparrow}) \rightsquigarrow \boxed{s(\underline{length(Y)})}^{\uparrow} \tag{4.28}$$

$$\boxed{X :: \underline{Y}}^{\uparrow} <> Z \rightsquigarrow \boxed{X :: (\underline{Y <> Z})}^{\uparrow} \tag{4.29}$$

The system also has access to the following lemma:

$$length(X <> \boxed{Y :: \underline{Z}}^{\uparrow}) \rightsquigarrow \boxed{s(\underline{length(X <> Z)})}^{\uparrow} \tag{4.30}$$

The heuristics for suggesting an induction scheme have led to the system choosing the induction rule

$$\frac{P(nil) \quad \forall h, t.P(t) \Rightarrow P(h :: t)}{\forall l.P(l)} \tag{4.31}$$

l_1 in (4.26) has also been chosen as a suitable variable to apply this rule to which makes $P(t)$ the equation $\forall l.even(length(t <> l)) \Leftrightarrow even(length(l <> t))$, hence the subgoal takes the form:

$$\begin{gathered}\forall h, t, (\forall l.even(length(t <> l)) \Leftrightarrow even(length(l <> t))) \Rightarrow \\ (\forall l'.even(length(\boxed{h :: \underline{t}}^{\uparrow} <> l')) \Leftrightarrow even(length(l' <> \boxed{h :: \underline{t}}^{\uparrow})))\end{gathered} \tag{4.32}$$

The universal $\forall l$ has been standardized apart in the induction hypothesis and conclusion. For the rest of this discussion the various quantifiers will be omitted, h and t are universal variables whose scope is the induction hypothesis *and* conclusion. The scope of l is the induction hypothesis and the scope of l' the induction conclusion – this makes l' a sink.

Using the wave rules (4.29), (4.27) and (4.28) the conclusion rewrites to

$$even(\boxed{s(\underline{length(t <> L')})}^{\uparrow}) \Leftrightarrow even(\boxed{s(\underline{length(L' <> t)})}^{\uparrow}) \tag{4.33}$$

Rippling on both sides of the arrow is blocked. This corresponds to failure of the second precondition of the wave method. Fertilization also doesn't apply, so this is a failed proof attempt.[3]

[3] The failure of fertilization is not a precondition of the Wave method but, in *CLAM*3, it is implicit in its failure (since *CLAM*3 orders the methods and tries each in order. If *CLAM*3 is attempting to use the Wave method then fertilization has already failed). Since it is an important assumption for the use of a Wave critic that fertilization has failed, it is conceivable that in a planner that used some other strategy for choosing methods the Wave method or critic would have to be extended.

There are several possible reasons why the proof is failing at this point: it might be a non–theorem; the choice of induction scheme might be wrong or there might be a missing lemma or both. Heuristics are provided to distinguish between these situations. It is only profitable to look for a new induction scheme if there is some wave rule that would apply if this scheme had been in place, in particular a wave rule that would have applied if there had been some extra structure in the wave fronts. This extra structure is indicated by inserting meta variables wrapped around each wave hole in the blocked term and then using middle–out reasoning to search for an appropriate instantiation. Potential wave fronts show where wave annotations might appear given this new structure. The system then attempts to unify this expression with the LHS of a wave rule. In the example this involves looking for the LHS of a wave rule that will unify with

$$even(\boxed{s(\boxed{F_1(\underline{length(Y <> \boxed{F_2(\underline{Z})}^{\uparrow})})}^{\uparrow})}^{\uparrow}) \leadsto \ldots \tag{4.34}$$

where F_1 and F_2 are second order meta variables. The LHS of (4.27) unifies with (4.34) using a higher order unification algorithm, instantiating F_1 to $\lambda x.s(x)$ and F_2 to $\lambda x.x$. If such a *partial wave rule match* is found then the extra structure needed can be rippled back in (temporarily treating the induction variable as the only sink present) to determine the form of the revised induction scheme. In the example this suggests the need for an additional wave front of the form $\boxed{s(\underline{\ldots})}^{\uparrow}$ which, in turn, suggests a two step induction using the scheme (4.35)

$$\frac{P(nil) \quad P(h :: nil) \quad P(t) \Rightarrow P(h_1 :: h_2 :: t)}{P(l)} \tag{4.35}$$

This scheme successfully leads to a proof plan. The Induction Revision critic just described is shown in figure 4–4.

Induction critics have been developed to perform induction revision, lemma discovery, generalisation and case–splitting. These are discussed in [Ireland & Bundy 96] and the above example is taken from there. That paper also contains discussion of the higher order unification required by the critics. Critics have also been used elsewhere by Walsh [Walsh 96] in an implicit induction prover (this is discussed in chapters 6 and 8).

Preconditions:

1. Precondition 1 of the wave method holds. (i.e. there are wave annotations present)
2. Precondition 2 of the wave method fails (i.e. there is no applicable wave rule)
3. There is a wave rule, that partially matches, i.e. unifies with the current term when extra meta variables for functions and potential wave fronts are inserted into the wave front.

Patch

1. Revise the induction scheme by rippling in the new structure.
2. Propagate the new scheme throughout the plan generated so far (e.g. add the extra base case goal)

Figure 4–4: Wave Critic: *Induction Revision*

4.6 Conclusion

This chapter outlined the major concepts of proof planning, which is the proposed technique for the automation of coinduction. These concepts were illustrated with two examples. One was the Wave proof method and this involved a discussion of rippling, the main ideas in rippling were explained but some of the details were omitted, these are covered in full in [Bundy *et al* 93] and [Basin & Walsh 96]. The other was the Induction Revision Critic.

Since coinduction is a dual to induction it might be appropriate to attempt to apply proof planning to it. The fact that all coinduction proofs involve a eureka step, the choice of bisimulation, suggests that proof critics or middle–out reasoning may be particularly important in the development of such a proof strategy.

This discussion was intended to cover the concepts that will be needed to understand the proof strategy for coinduction presented in chapters 5 and 6.

Chapter 5

A Proof Strategy for Coinduction

5.1 Introduction

This chapter is going to describe in detail the proof methods developed for coinduction. These methods have been implemented in `clam.v3.2` using Prolog to form a system called *CoCLAM*. However since proof plans are intended to be general descriptions of the higher level steps in the proof process the tactic and method specifications have been described in a natural language format, as in chapter 4, which should stand alone from any particular implementation of proof planning.

The chapter starts with two worked examples of coinduction (one using it to determine bisimilarity and the other using it for type–checking). They are analysed in some detail in order to distinguish the individual steps in a coinductive proof which will correspond to proof methods in the final proof strategy.

In the light of this analysis a proof strategy is proposed along with several proof methods designed specifically for use with coinduction and labelled transition systems.

The use of rewriting within coinduction is also considered and methods for performing it are discussed.

5.2 The Scope of the Proof Strategy

For practical reasons it proved desirable to limit the scope of the problems for which the proof strategy was devised. It is assumed in everything that follows that the proof is taking place within some pre–defined *deterministic* labelled transition system.

Definition 5.1 *A labelled transition system $\mathcal{T}$ is* **deterministic** *if for all a and for all transitions α in $\mathcal{T}$ if $a \xrightarrow{\alpha} a'$ and $a \xrightarrow{\alpha} a''$ then $a' = a''$.*

There are several reasons for this choice. Firstly, the bulk of the work using coinduction outside of CCS is presented in this format and it was felt a proof strategy for this sort of problem would have the widest applicability. Secondly work by Rutten [Rutten 96] suggests that many coinductive datatypes may, at some level, be treated as restrictions of labelled transitions systems and so it was felt that this approach had some measure of generality.

A number of specific assumptions are made about the relationship of transitions and values, where the values are some set of distinguished expressions in the language (e.g. the end results of reduction). It is assumed that the transition rules for the language apply transitions only to values and that all values have transitions. It is also assumed that expressions can be reduced to values by the use of an order, $\overset{\text{red}}{\rightsquigarrow}$, and that the only ground expressions which can not be reduced are values.

Lastly a couple of assumptions are made about the form of the transition rules specified by the language. They are assumed to be of the form $a \overset{\alpha}{\rightarrow} b$ without the appearance of any conditional statements. There is one exception to this:

$$\frac{a \overset{\text{red}}{\rightsquigarrow}^{+} a' \quad a' \overset{\alpha}{\rightarrow} b}{a \overset{\alpha}{\rightarrow} b} \tag{5.1}$$

where a' is a value.

These assumptions are all met by standard transition systems representing the operational semantics of functional languages (as described in chapter 3). The assumptions about transitions, values and reduction are the most restrictive assumptions that are made and are needed by the Evaluate method (§5.5.3). Chapter 9 discusses how these assumptions could be broadened.

The second choice made was not to pursue problems which contained terms with no weak head normal form. These problems are characterised by example 3.5 in chapter 3 in which additional analysis was required to show that evaluating the head of each side of the relation would either diverge or terminate with the same value. No criteria are presented for determining which theorems may involve this sort of divergence.

Lastly only proofs involving bisimilarity of some sort are considered, again this choice was motivated by the observation that nearly all the literature is confined to proofs in this category. Hence the proof strategy assumes that a proof of equivalence (of some sort) between two objects is being sought, not a proof of membership of any other greatest fixedpoint.

5.3 Worked Examples of Coinduction

I intend to work through two examples of coinduction, carefully breaking down and discussing each step, since it is this level of understanding that will allow proof methods to be discussed. The discussion is intended to draw out the common features of coinductive proofs. These features should seem reasonable in the light of the examples in chapter 3. In particular, they are intended to illustrate the principal steps involved in a coinductive proof. These steps are highlighted by the individual section headings in the example. The aim is to translate these steps into nodes in the proof strategy, i.e. proof methods. The two proofs in question are examples 3.2 and 3.8 from chapter 3. Justification of the choice of these nodes is offered on a less anecdotal level in §5.4. The main purpose of these worked examples is to refresh the reader's memory of the form of a coinductive proof.

5.3.1 Bisimilarity

This is example 3.2 from chapter 3.

Example 5.1

$$\forall f : \tau_1 \to \tau_2,\ m : \tau_1.\ map(f, iterates(f, m)) \sim iterates(f, f(m))$$

Reduction Rules

$$map(F, nil) \overset{\text{red}}{\leadsto} nil \tag{5.2}$$
$$map(F, H :: T) \overset{\text{red}}{\leadsto} F(H) :: map(F, T) \tag{5.3}$$

$$iterates(F, M) \overset{\text{red}}{\leadsto} M :: iterates(F, F(M)) \tag{5.4}$$

Apply Coinduction Rule

The first step is to use the coinduction rule:

$$\frac{\langle a,\ b\rangle \in \mathcal{R} \quad \mathcal{R} \subseteq \langle \mathcal{R} \cup \sim\rangle}{a \sim b} \tag{5.5}$$

in order to proceed it is necessary to choose a suitable candidate for the bisimulation, $\mathcal{R}$.

Let $\mathcal{R} = \{\langle map(F, iterates(F, M)),\ iterates(F, F(M))\rangle\}$.

Clearly $\langle map(F, iterates(F, M)), iterates(F, F(M))\rangle \in \mathcal{R}$ which leaves us with one remaining goal $\mathcal{R} \subseteq \langle \mathcal{R} \cup \sim\rangle$

Show $\mathcal{R}$ is a Bisimulation.

Using the definitions of $\sim$ from chapter 3 we need to show that every transition, α, that applies to either $map(F, iterates(F, M))$ or $iterates(F, F(M))$ applies to both and that the results are in $\mathcal{R} \cup \sim$. This is expressed by the inference rule 3.22 which gives us the subgoal:

$$
\begin{gathered}
\forall \mathcal{R}. \langle map(F, iterates(F, M)),\ iterates(F, F(M)) \rangle \in \mathcal{R} \Rightarrow \\
\forall \alpha. ((map(F', iterates(F', M')) \xrightarrow{\alpha} \phi \vee iterates(F', F'(M')) \xrightarrow{\alpha} \psi) \Rightarrow \\
((map(F', iterates(F', M')) \xrightarrow{\alpha} \phi \wedge iterates(F', F'(M')) \xrightarrow{\alpha} \psi) \wedge \\
\langle \phi,\ \psi \rangle \in \mathcal{R} \cup \sim))
\end{gathered}
\tag{5.6}
$$

In what follows the LHS of the first $\Rightarrow$ will be referred to as the *coinduction hypothesis* and the RHS as the *coinduction conclusion*.

The variables are named differently in the hypothesis and conclusion because each is separately universally quantified (since the hypothesis is essentially trying to describe all members of a relation, and the conclusion is then trying to prove something about all the members). The universal quantifiers have been dropped, as in chapter 3.

Rewrite

In order to prove goal, (5.6), we are first going to have determine all the possible instantiations of α, ϕ and ψ. To do this we will perform reduction. Specifically we want to reduce the expressions to values (as defined in chapter 3 – so values include expressions such as $H :: T$):

$$
\begin{array}{rcl}
map(F', iterates(F', M')) & \overset{\text{red}}{\rightsquigarrow} & map(F', M' :: iterates(F', F'(M'))) \\
\ldots & \overset{\text{red}}{\rightsquigarrow} & F'(M') :: map(F', iterates(F', F'(M'))) \\
 & & \\
iterates(F', F'(M')) & \overset{\text{red}}{\rightsquigarrow} & F'(M') :: iterates(F', F'(F'(M')))
\end{array}
$$

Take the Transitions

At this point we can determine the appropriate transitions and instantiate the variables ϕ and ψ in the goal. This means setting up several new goals, one for each possible transition. The transitions are found by inspecting the transition rules associated with the value under consideration. In this case the possible transitions are `hd` and `tl`. So there are two new goals.

$$\forall \mathcal{R}.\ \langle map(F, iterates(F, M)),\ iterates(F, F(M))\rangle \in \mathcal{R} \Rightarrow \\ \langle F'(M'),\ F'(M')\rangle \in \mathcal{R} \cup \sim$$

$$\forall \mathcal{R}.\ \langle map(F, iterates(F, M)),\ iterates(F, F(M))\rangle \in \mathcal{R} \Rightarrow \\ \langle map(F', iterates(F', F'(M'))),\ iterates(F', F'(F'(M')))\rangle \in \mathcal{R} \cup \sim$$

More Rewriting and Fertilization

The first of these new goals is true since $\langle F'(M'),\ F'(M')\rangle \in \sim$ by the reflexivity of $\sim$. The second is true since the coinduction hypothesis implies the conclusion. Appealing to the hypothesis is called fertilization and is a method also used in induction (chapter 4).

Although, in this example, fertilization and appeal to the reflexivity of $\sim$ were sufficient to prove the final goals, it is possible that further rewriting might have been required to enable it to take place, e.g. example 3.7 in chapter 3.

5.3.2 Type Checking

The second example (example 3.8 from chapter 3) is for type checking.

Example 5.2

$$\forall m :_c \tau \Rightarrow\ lconst(m) :_c llist(\tau)$$

Reduction Rule

$$lconst(M) \overset{\text{red}}{\rightsquigarrow} M :: lconst(M)$$

Applying the Coinduction Rule

The first step is to use the coinduction rule (5.7) from chapter 3:

$$\frac{\langle l,\ \langle l,\ \tau\rangle\rangle \in \mathcal{R} \quad \mathcal{R} \subseteq \langle \mathcal{R} \cup \sim\rangle}{l :_c \tau} \tag{5.7}$$

To proceed we need to choose a suitable candidate for the relation, $\mathcal{R}$. Let $\mathcal{R} = \{\langle lconst(M),\ \langle lconst(M),\ llist(\tau)\rangle\rangle\}$. Clearly $\langle lconst(M),\ \langle lconst(M),\ llist(\tau)\rangle\rangle \in \mathcal{R}$ which leaves us with the new goal $\mathcal{R} \subseteq \langle \mathcal{R} \cup \sim\rangle$

Show $\mathcal{R}$ is a Bisimulation

Using the definition of $type\_fun$ from chapter 3 we can express the goal as

$$\mathcal{R} \subseteq \{\langle l, \langle l, \tau\rangle\rangle | \forall \alpha. l \xrightarrow{\alpha} l', \langle l, \tau\rangle \xrightarrow{\alpha} \langle l', \tau'\rangle \langle l', \langle l', \tau'\rangle\rangle \in \mathcal{R} \cup \sim\}$$

Using rule (3.22) from chapter 3 we get the subgoal:

$$\begin{gathered} \forall \mathcal{R}. (M :_c \tau \Rightarrow \langle lconst(M), \langle lconst(M), llist(\tau)\rangle\rangle \in \mathcal{R}) \Rightarrow \\ \forall \alpha.((((M' :_c \tau \Rightarrow lconst(M') \xrightarrow{\alpha} \phi) \vee (M' :_c \tau \Rightarrow \langle lconst(M'), llist(\tau)\rangle \xrightarrow{\alpha} \psi)) \\ (((M' :_c \tau \Rightarrow lconst(M') \xrightarrow{\alpha} \phi) \wedge (M' :_c \tau \Rightarrow \langle lconst(M'), llist(\tau)\rangle \xrightarrow{\alpha} \psi))\wedge \\ \langle \phi, \psi\rangle \in \mathcal{R}\cup \sim))) \end{gathered}$$

Rewrite

In order to prove this next goal we are first going to have instantiate α and ϕ. To do this we need to evaluate $lconst(M')$ to a value.

$$lconst(M') \overset{\text{red}}{\leadsto} M' :: lconst(M')$$

Take the Transitions

At this point we can determine the appropriate transitions and set up several new goals, one for each possible transition. In this case the transitions are `hd` and `tl`. So there are two new goals.

$$\begin{gathered} \forall \mathcal{R}. (M :_c \tau \Rightarrow \langle lconst(M), \langle lconst(M), llist(\tau)\rangle\rangle \in \mathcal{R}) \Rightarrow \\ (M' :_c \tau \Rightarrow \langle M', \langle M', \tau\rangle\rangle \in \mathcal{R}\cup \sim) \end{gathered}$$

$$\begin{gathered} \forall \mathcal{R}. (M :_c \tau \Rightarrow \langle lconst(M), \langle lconst(M), llist(\tau)\rangle\rangle \in \mathcal{R}) \Rightarrow \\ (M' :_c \tau \Rightarrow \langle lconst(M'), \langle lconst(M'), llist(\tau)\rangle\rangle \in \mathcal{R}\cup \sim) \end{gathered}$$

More Rewriting and Fertilization

The first of these new goals is true since $M' :_c \tau \Rightarrow \langle M', \langle M', \tau\rangle\rangle \in \sim$. The second is true since the coinduction hypothesis implies the conclusion.

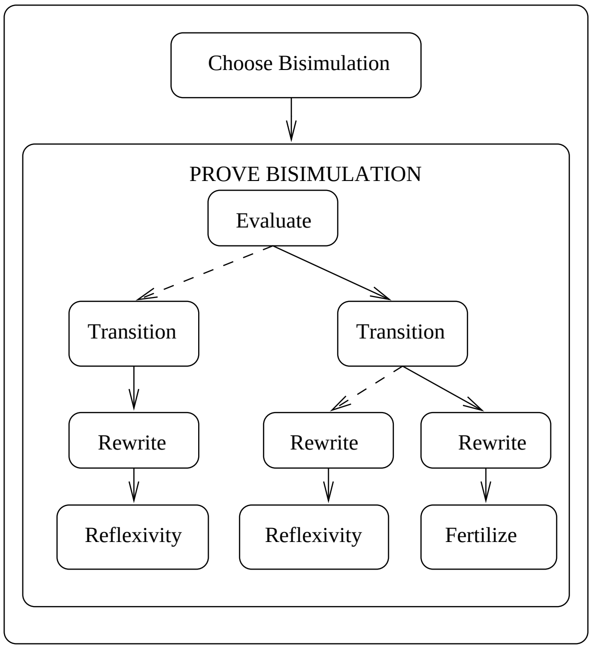

Figure 5–1: A Proof Strategy for Coinduction

5.4 A Proof Strategy for Coinduction

The various examples suggested the strategy outlined in figure 5–1 for coinduction. The strategy embodies the following analysis of the process of coinductive proof within the limits specified in §5.2.

1. A coinductive proof starts with the application of the coinduction rule which produces two subgoals. To produce these subgoals some relation has to be introduced. The relation is chosen so that the first of these new subgoals should be trivial.
2. The second subgoal is more complicated. First it is converted into some goal of the form *coinduction hypothesis* $\Rightarrow$ *coinduction conclusion*. This transformation is based on a derived inference rule that depends heavily on the particular greatest fixedpoint under consideration, a few such inference rules were discussed in chapter 3 and justifications for them appear in Appendix D. Since this investigation is restricted purely to proofs within labelled transition systems only one such inference rule is considered here, that associated with $\langle \cdots \rangle$.
3. The subgoals produced by these inference rules can only be discharged by determining transitions from one or more terms. This requires the evaluation of those terms to a value.

4. Transitions can then be determined by reference to the transition rules of the system.
5. After that further rewriting should lead to trivial goals. e.g. example 3.7 in chapter 3.

Once the general strategy has been determined it is necessary to provide method and tactic descriptions. The tactics will not be described since they have not been implemented and will depend upon the particular object logic.

5.5 Proof Methods

This section will discuss the proof methods required by the proof strategy for coinduction.

5.5.1 Coinduction

This method starts out a coinductive proof by applying the coinduction rule. By inspection of the general rule (2.6) the coinduction method applies if the goal is of the form $a \in gfp(\mathcal{F})$. Given the domain to which the proof plans have been restricted this corresponds to $a \in \sim$. The method's outputs, or the new goals, will be $a \in \mathcal{R}$ and $\mathcal{R} \subseteq \langle \mathcal{R} \cup \sim \rangle$, for some $\mathcal{R}$. These conditions are all legal rather than heuristic in nature. They are derived from the statement of the coinduction rule.

The hard part of any coinductive proof is the choice of $\mathcal{R}$. The proof method proposed uses a heuristic to construct this if it isn't supplied in some other way.

If $\mathcal{R}$ isn't supplied then the obvious heuristic for choosing a set is the smallest possible set that discharges the first precondition of (2.6). If we're trying to prove that $f(\bar{X}) \sim g(\bar{X})$ then the first precondition is $\langle f(\bar{X}),\ g(\bar{X}) \rangle \in \mathcal{R}$ hence the smallest set that discharges this is $\{\langle f(\bar{X}),\ g(\bar{X}) \rangle\}$. Since a heuristic has been employed to make this choice (and a fairly simple heuristic at that) there is a possibility that $\mathcal{R}$ may need to be revised. The choice of $\mathcal{R}$ is defeasible in that its definition can be changed in the event of the proof failing to go through: we use proof critics to identify such situations. The critics for revising the choice are discussed in chapter 6.

The coinduction method is described in figure 5–2.

5.5.2 Gfp Membership

The second stage of a coinductive proof involves showing that $\mathcal{R}$ is a member of the greatest fixedpoint. The Gfp Membership method performs some inference on the goal $\mathcal{R} \subseteq \langle \mathcal{R} \cup \sim \rangle$, expanding the definition of $\sim$ using an inference rule, (3.22), discussed in chapter 3. It is described in figure 5–3.

Precondition

The current goal is $f(\bar{X}) \sim g(\bar{X})$ (where, in the case of type–checking, $g(\bar{X})$ may be a pair of an expression and a type).

Output

1. If no candidate for $\mathcal{R}$ has been supplied the output is:

$$\{\langle f(\bar{X}),\ g(\bar{X})\rangle\} \subseteq \langle \{\langle f(\bar{X}),\ g(\bar{X})\rangle\} \cup \sim\rangle$$

2. If a candidate relation, $\mathcal{R} = \bigcup_{i=1}^{n}\{\langle f_i(\bar{X}_i),\ g_i(\bar{X}_i)\rangle\}$, has been supplied then the output is:

$$\bigcup_{i=1}^{n}\{\langle f_i(\bar{X}_i),\ g_i(\bar{X}_i)\rangle\} \subseteq \langle \bigcup_{i=1}^{n}\{\langle f_i(\bar{X}_i),\ g_i(\bar{X}_i)\rangle\} \cup \sim\rangle$$

Figure 5–2: The Coinduction($\mathcal{R}$) Method

Precondition

1. The current goal is: $\mathcal{R} \subseteq \langle \mathcal{R} \cup \sim\rangle$
2. $\mathcal{R} = \bigcup_{i=1}^{n}\{\langle f_i(\bar{X}_i),\ g_i(\bar{X}_i)\rangle\}$

Outputs

For each i form the goal:

$$\begin{gathered}\forall \mathcal{R}, \bigwedge_{i=1}^{n}\langle f_i(\bar{X}_i),\ g_i(\bar{X}_i)\rangle \in \mathcal{R} \Rightarrow \\ \forall \alpha.((f_i(\bar{X}_i') \xrightarrow{\alpha} \phi \vee g_i(\bar{X}_i') \xrightarrow{\alpha} \psi) \Rightarrow \\ ((f_i(\bar{X}_i') \xrightarrow{\alpha} \phi \wedge g_i(\bar{X}_i') \xrightarrow{\alpha} \psi) \wedge \langle \phi,\ \psi\rangle \in \mathcal{R} \cup \sim))\end{gathered}$$

Figure 5–3: The Gfp Membership($\langle\cdots\rangle$) Method

5.5.3 Evaluation

Rewriting occurs twice in the coinduction proof strategy. Each time it occurs it is used to achieve a different end result. The first time it is used to find instantiations for the transitions. The second time it is used to allow fertilization. Different methods are employed to achieve these two different ends. In this section only the first occurrence of rewriting will be discussed, the second is discussed in §5.5.5.

In chapter 4, rippling, a heuristic for guiding rewriting in induction was discussed. Rippling was considered as a method to apply at those stages in the proof process where some form of rewriting is required. The objective at this stage is to use reduction to find a value, rippling has a wider scope than this (for instance there would be a risk, with rippling, that constructors would be rippled into sinks, rather than rippled out to provide transitions), although it is conceivable that it could be limited to perform reduction alone.

Non–Strict(Lazy) Evaluation

The assumption of a reduction order with values as the least elements of descending chains (if least elements exist) suggests that various well known reduction strategies might be appropriate. These are all non–terminating in various situations, which is undesirable in any automated process. It was not the purpose of this research, however, to improve upon reduction techniques and given the specification of the domain of inquiry, adopting an existing reduction strategy appeared appropriate. The most obvious choice was non–strict evaluation, where terms are only reduced if needed. This is often referred to as lazy evaluation, however lazy evaluation generally also implies that any expression is only evaluated once, which is not done here.

Definition 5.2 *A term, M, is a* **redex** *(reducible expression) if it matches the LHS of a reduction rule.*

Definition 5.3 **Non–strict evaluation** *proceeds by always reducing a redex that is contained in no other redex, until the entire term is a value.*

Non–strict evaluation was chosen because in all functional languages values are defined as weak head normal forms and non–strict evaluation always terminates in a weak head normal form if one exists. In a domain where values are not weak head normal forms then a different strategy might have to be employed.

Extending Non–Strict (Lazy) Evaluation

Unfortunately non–strict evaluation alone is not sufficient. Recall that the first rewriting method is applied to goals of the form

$$\begin{gathered} HYP \Rightarrow \\ \forall \alpha.((l_1 \overset{\alpha}{\rightarrow} \phi \vee l_2 \overset{\alpha}{\rightarrow} \psi) \Rightarrow \\ ((l_1 \overset{\alpha}{\rightarrow} \phi \wedge l_2 \overset{\alpha}{\rightarrow} \psi) \wedge \\ \langle \phi,\ \psi \rangle \in \mathcal{R} \cup gfp(\mathcal{F}))) \end{gathered} \tag{5.8}$$

The expressions, l_1 and l_2 are the ones that have to be rewritten. They are of the form $\forall \bar{x}.exp(\bar{x})$ where the $\bar{x}$ are variables occurring free in exp. Reduction is applied to $exp(\bar{x})$. This is because we want to know the possible values of $exp(\bar{x})$ given arbitrary values of $\bar{x}$.

The problem is that reduction is only guaranteed to terminate in a value if the original expression contains no free variables. Hence $exp(\bar{x})$ may be irreducible even though it is not a value.

If $exp(\bar{x})$ is not a value then it may be possible to reduce it by substituting values for the free variables (since we want to know the value of $exp(\bar{x})$ on all values of its arguments). This suggests an extension to the non–strict evaluation strategy which replaces the variables in $exp(\bar{x})$ with values. Since most types will have more than one value associated with them (e.g. lists have nil and $H :: T$ as values) a case split will have to be performed on the goal in order to ensure that all possible values have been investigated. Notice that this process may well introduce new free variables (e.g. H and T in $H :: T$). This is because values are not constrained to be ground expressions.

Case Splitting

We extend non–strict evaluation with controlled case splitting of free variables. The intuition is that this will allow further reduction which will terminate in values if they exist.

Unfortunately, this affects the termination of the evaluation. This is illustrated in example 5.3.

Example 5.3 *Consider the term*

$$map(F)^N(L) \tag{5.9}$$

where $(\cdots)^n$ *is defined by:*

$$F^0(X) \overset{\text{red}}{\rightsquigarrow} X \tag{5.10}$$

$$F^{s(N)}(X) \overset{\text{red}}{\rightsquigarrow} F(F^N(X)) \tag{5.11}$$

Non–strict evaluation with case–splitting where appropriate reduces $map(F)^N(L)$ *to* L *(if* $N = 0$*) and* $map(F, map(F)^{N_1}(L))$ *(if* $N = s(N_1)$*), neither of these are values.* L *can be made a value by casesplitting but* $map(F, map(F)^{N_1}(L))$ *can not. If* N_1 *is casesplit further then* $map(F, map(F)^{N_1}(L))$ *can be reduced to* $map(F, L)$ *if (*$N_1 = 0$*) and* $map(F, map(F, map(F^{N_2}(L))))$ *if (*$N_1 = s(N_2)$*). Once again* $map(F, L)$ *can be reduced to a value by now casesplitting* L *but* $map(F, map(F, map(F^{N_2}(L))))$ *can not. Further evaluation of the term in this fashion produces a potentially infinite sequence as* N_2 *is casesplit and then* N_3 *and so on. If* N *were a value, rather than a universally quantified variable this process would bottom out at some point with* $N = 0$*, but as it stands it fails to terminate.*

The lemmata (5.12) and (5.13)

$$map(F)^N(nil) \overset{\text{red}}{\rightsquigarrow} nil \tag{5.12}$$
$$map(F)^N(H :: T) \overset{\text{red}}{\rightsquigarrow} F^N(H) :: map(F)^N(T) \tag{5.13}$$

allow $map(F)^N(L)$ *to be evaluated to weak head normal forms nil and* $F^N(H) :: map(F)^N(T)$ *(by case–splitting L).*

(5.12) and (5.13) have to be supplied as lemmata to the system. This is equivalent to supplying a corecursive definition of the function $map(F)^N$ (see chapter 2). This observation suggests strongly that the existence of corecursive definitions of functions is somehow closely linked to issues of termination.

A Reduction Strategy

The reduction strategy on a term is

1. Perform non–strict evaluation (without case–splitting) on term until it terminates (this is guaranteed) in some new term, t.

2. If t is a value we are done.

3. If t is not a value, case–split a free variable, v, appearing in t, replacing it with each possible value in the type of v and restart the process.

There are often a number of possible variables to case–split as was illustrated by the example. A breadth first search of all the variables is performed to find the appropriate ones. The resulting method, the Evaluate Method, is shown in figure 5–4.

An alternative suggestion, given that breadth first search is computationally expensive, is to perform some analysis on potential reduction rules and lemmata and prefer case splits which lead to the application of a reduction rule or lemma that rewrites to a value. This is based on the observation that the patterns required for small–step reduction rules are often values. So such a case split is likely to enable further reduction.

5.5.4 Transitions

The Transition Method is used if there are terms of the form $a \xrightarrow{\alpha} \phi$ in the goal where α and ϕ are uninstantiated, but where a is a value, so α and ϕ can now be instantiated by inspecting the transition rules. This will arise when the expressions on the LHS of transitions that appeared in the output of the Gfp Membership method have been reduced to values. The object is to remove all mention of transitions and get to the point of proving something about the result of the transitions (which is the central proof obligation to show that the chosen set is a bisimulation).

Pre–Conditions

1. The current goal is of the form

$$\begin{array}{c} HYP \Rightarrow \\ \forall\alpha.((l_1 \stackrel{\alpha}{\rightarrow} \phi \vee l_2 \stackrel{\alpha}{\rightarrow} \psi) \Rightarrow \\ ((l_1 \stackrel{\alpha}{\rightarrow} \phi \wedge l_2 \stackrel{\alpha}{\rightarrow} \psi)\wedge \\ \langle \phi,\ \psi\rangle \in \mathcal{R} \cup gfp(\mathcal{F}))) \end{array}$$

2. For some i, l_i $(i = 1, 2)$ rewrites to v_i, a value, using the reduction strategy with the reduction rules, *rules*, and case–splitting some set of variables $\mathcal{V}$.

   Note that, the expression v_i depends upon some particular combination, V, of cases of the variables in $\mathcal{V}$.

Outputs

- Form goals for every combination of casesplits in $\mathcal{V}$ by replacing the variables in $\mathcal{V}$ with one of their cases everywhere they appear in the goal.
- For the combination of cases V also replace l_i with v_i.

Figure 5–4: The Evaluate Method

This situation is slightly complicated in proofs of bisimilarity by the requirement that if a transition applies to one side of a relation it applies to both. However the method remains a fairly straightforward process of applying the transition rules for the language and tidying up the expressions $a \stackrel{\alpha}{\rightarrow} \phi$ which have become trivially true.

Lastly difference matching is performed to allow the rippling used in the next part of the proof strategy. Sometimes more than one difference match may be possible, however the proof strategy doesn't explicitly prefer any one to another.

This method is shown in figure 5–5

5.5.5 Rippling

Recall example 3.7 from chapter 3. The candidate bisimulation is $\{\langle map(F)^N(h(F, X)),\ iterates(F, F^N(X))\rangle\}$

The expressions $map(F')^{N'}(h(F', X'))$ and $iterates(F', F'^{N'}(X'))$ reduce to $F^{N'}(X') :: map(F')^{N'}(map(F', h(F', X')))$ and $F'^{N'}(X') :: iterates(F', F'(F'^{N'}(X')))$ – these are values allowing transitions to be taken. The tail transitions leave the goal:

Preconditions

1. The expression $a \stackrel{\alpha}{\to} \phi$ appears in the goal.
2. The set of transitions that apply to a is *transitions*
3. *transitions* is non–empty and contains all possible transitions from the set of transition rules that apply to the value, a.
4. If the conjunction $a \stackrel{\alpha}{\to} \phi \wedge b \stackrel{\alpha}{\to} \psi$ appears in the goal then all the transitions in *transitions* also apply to b and there is no transition that applies to b that isn't in *transitions*.

Outputs

Form a new goal for each $\alpha_k \in$ *transitions*, by the following process:

1. Replace $a \stackrel{\alpha}{\to} \phi$ with *true* wherever it appears in the goal.
2. Replace all remaining occurrences of ϕ with a' where $a \stackrel{\alpha_k}{\to} a'$.
3. If the expression $b \stackrel{\alpha}{\to} \psi$ appears in the goal also replace this with *true* and instantiate any remaining occurrence of ψ with b' where $b \stackrel{\alpha_k}{\to} b'$.
4. Remove all the occurrences of *true* using the rules
$$(true \Rightarrow P) \rightsquigarrow P \qquad (true \wedge P) \rightsquigarrow P \qquad (true \vee P) \rightsquigarrow true$$
5. If the resulting expression differences matches against one of the hypotheses then annotate the expression accordingly.

Figure 5–5: The Transition(*transitions*) Method

$$\forall \mathcal{R} \langle map(F)^N(h(F,X)),\ iterates(F, F^N(X))\rangle \in \mathcal{R} \Rightarrow$$
$$\langle map(F')^{N'}(map(F', h(F', X'))),\ iterates(F', F'(F'^{N'}(X')))\rangle \in \mathcal{R}$$

This won't fertilize immediately because of the $map(F')$ around $h(F', X')$ and the extra F' around $F'^{N'}(X')$. However, it is clear that the expression is equivalent to $\langle map(F')^{s(N')}(h(F', X')),\ iterates(F', F'^{s(N')}(X'))\rangle$ using the rewrite rules $F(F^N(X)) \rightsquigarrow F^{s(N)}(X)$ and $F^N(F(X)) \rightsquigarrow F(F^N(X))$. We can then fertilize since N' is a sink. However the first of these rewrite rules is the reverse of the reduction rule we have already used to evaluate the expression. So we can't use reduction for this step.

This is much more like the purpose for which rippling is designed, in fact it corresponds to "rippling into a sink" since the aim is to sink the differences between the hypothesis and conclusion. It is still necessary to provide annotations from somewhere. Since the aim is to fertilize using one of the coinduction hypotheses the obvious approach is to difference match against a hypothesis. This difference matching is performed by the Transition Method when forming the output in the anticipation, based on the proof strategy, that rippling is the next method to be applied.

5.5.6 Fertilization and Other Methods

The fertilization method had to be extended slightly for use with coinduction (this is discussed in chapter 7). The only other method required is described as "Reflexivity of $\sim$" in the proof strategy. This is shown in figure 5–6 and is a basic simplification method.

Precondition
The current goal is $\langle X,\, X\rangle \in \mathcal{R} \cup \sim$ or
$X :_c \tau \Rightarrow \langle X,\, \langle X,\, \tau\rangle\rangle \in \mathcal{R} \cup \sim$.

Output
No new goals.

Figure 5–6: The Reflexivity of $\sim$ Method

In several of the proof plans another simplification method available with *CLAM*3 was used. This was Eval_def, *evaluation by definition*. Eval_def unfolds definition equations in unannotated terms and wave fronts. In *CoCLAM* Eval_def interacted with Evaluate to take some of the burden for reduction. This happened because Evaluate deals with one side of the relation at a time - this frequently introduced sufficient case splits for the other side of the relation to be reduced to a value. Eval_def is more efficient than Evaluate since it doesn't have to search for appropriate case splits etc. However it effectively functioned as a restricted form of the Evaluate method.

Simplification steps take a number of forms in numerous difference theorem provers and proof planners. Fertilization, appeal to the reflexivity of $\sim$ (and definition of $:_c$) and the simplified rewriting supplied by Eval_def proved sufficient to handle the various trivial steps required by coinduction. However there is no reason to suppose that these are the only simplification methods that could have been used.

5.6 Conclusion

This chapter has dealt in some detail with the proof methods for coinduction. It has tried to discuss them in a fairly general way using English in the hope that this will aid comprehension and in the belief that the methods themselves shouldn't be tied to one particular implementation or one particular language.

The Coinduction, Gfp Membership and Transition methods were all introduced for the first time and described in some detail.

The rewriting involved in coinductive proofs was also discussed and non–strict evaluation was extended by provision for case splitting universal variables to cope with a stage in the proof strategy for which the standard ripple method appeared inappropriate.

The central coinduction methods, i.e. the Coinduction Method and Gfp Membership Method, proved relatively simple to specify and present. Provision of a selection of methods to control the rewriting process in order to enable the determination of transitions turned out to be a more complex. The Evaluate Method is a first step towards this, but it remains inefficient and unwieldy. It also is heavily dependent on a set of assumptions about the relationship of reduction, transitions and values in the domain under consideration.

In general coinduction fits well into the proof planning framework. It presents fairly obvious steps in a proof process, the choice of a bisimulation, taking transitions and fertilization which are relatively easy to describe. These are linked in the proof strategy by Evaluation and rippling which appear to be more closely associated with ideas from functional programming settings. Rippling is a standard proof planning method and has been extensively investigated. However, the Evaluate method is new.

Chapter 6

Critics For Coinduction

6.1 Introduction

This chapter introduces the Revise Bisimulation Critic. In order to do this the requirements for a relation containing some pair $\langle E,\ F\rangle$ to be a bisimulation are examined. This involves the introduction of the idea of a transition sequence and some theoretical results are proved about the nature of the space of transition sequences.

A strategy is proposed for exploring the space of transitions sequences and in the light of this strategy the Revise Bisimulation critic for the Wave method is proposed to implement it. Some deficiencies of the proposed critic in embodying the strategy are then discussed.

6.2 The Trial Bisimulation

As mentioned in chapter 5 the Coinduction Method uses a heuristic to choose a candidate bisimulation. It was noted that this heuristic was fairly simple and that it was possible for the chosen relation not to be a bisimulation. This is illustrated by example 6.1.

Example 6.1 *Consider example 3.7 from chapter 3.*

$$\forall x, f.\ h(f,x) \sim x :: map(f, h(f,x)) \Rightarrow \forall x, f.\ h(f,x) \sim iterates(f,x)$$

The Coinduction method chooses the trial bisimulation

$$\{\langle h(F,X),\ iterates(F,X)\rangle\} \tag{6.1}$$

The Gfp Membership and Evaluate Methods will provide the subgoal

$$\begin{array}{c} \forall \mathcal{R} \langle h(F,X),\ iterates(F,X)\rangle \in \mathcal{R} \Rightarrow \\ (X' :: map(F', h(F',X')) \xrightarrow{\alpha} \phi \vee X' :: iterates(F'), F'(X') \xrightarrow{\alpha} \psi) \Rightarrow \\ X' :: map(F', h(F',X')) \xrightarrow{\alpha} \phi \wedge X' :: iterates(F'), F'(X') \xrightarrow{\alpha} \psi \wedge \\ \langle \phi,\ \psi \rangle \in \mathcal{R} \cup \sim \end{array} \tag{6.2}$$

There are two possible values for α, `hd` and `tl`. I am only going to consider the subgoal produced using `tl` here since the `hd` transition subgoal is trivial by appeal to the reflexivity of $\sim$.

The `tl` transition produces the subgoal

$$\begin{array}{c} \forall \mathcal{R} \langle h(F,X),\ iterates(F,X)\rangle \in \mathcal{R} \Rightarrow \\ \langle \boxed{map(F', \underline{h(F',X')})}^{\uparrow},\ iterates(F', \boxed{F'(\underline{X'})}^{\uparrow})\rangle \in \mathcal{R} \end{array} \tag{6.3}$$

The proof attempt fails because $\langle map(F', h(F',X')),\ iterates(F', F'(X'))\rangle \notin \mathcal{R}$.

There are two possible solutions to this problem; either the heuristic is improved (for instance with the use of middle–out reasoning) or proof critics (as described in chapter 4) are employed to modify the choice.

The trial bisimulation chosen by the Coinduction method was the smallest possible relation that discharged the first condition of the coinduction rule. If this relation is not a bisimulation but the problem under consideration is genuinely a theorem, then the relation was not large enough. The proof attempt will fail at the fertilization stage because the pair of expressions provided by the Transition method are not in the trial bisimulation. This failure provides useful extra information (the new pair of expressions) to guide an extension of the relation. This is an ideal situation for the application of a proof critic, since the failure has provided more information than was available when the earlier decision was made.

6.3 Transition Sequences

In order to understand better the process undertaken by the Revise Bisimulation critic it is important to introduce the notion of the smallest possible bisimulation that contains a pair of objects and establish some facts about it. The central theorem (6.2) in this section shows that this smallest bisimulation is the least set generated by exploring all possible results of applying transitions to the original pair.

First we need to establish what is meant by “exploring all possible results of transitions to the original pair”. To do this we establish the existence of chains of expressions linked by transitions $a_1 \xrightarrow{\alpha_1} a_2 \xrightarrow{\alpha_2} \cdots$. Furthermore since coinduction requires the same transitions to hold from two expressions we establish a definition for “matched” sequences of transitions.

Definition 6.1 *Let E be some expression in a labelled transition system. If $E \stackrel{\alpha}{\rightarrow} E'$ for some transition α then E' is a* **successor state** *of E.*

Let $\langle E,\ F\rangle$ be a pair of expressions. If E' is a successor state of E and F' is a successor state for F then $\langle E',\ F'\rangle$ is a **successor state** *of $\langle E,\ F\rangle$.*

If E' is a successor state of E and F has no successor states then $\langle E',\ F\rangle$ is a **successor state** *for $\langle E,\ F\rangle$.*

If E has no successor state and F' is an successor state for F then $\langle E,\ F'\rangle$ is a **successor state** *for $\langle E,\ F\rangle$.*

NB. Successor states are not necessarily unique.

Definition 6.2 *If $\langle E',\ F'\rangle$ is a successor state of $\langle E,\ F\rangle$ and moreover there is some transition $\stackrel{\alpha}{\rightarrow}$ such that $E \stackrel{\alpha}{\rightarrow} E'$ and $F \stackrel{\alpha}{\rightarrow} F'$ then $\langle E',\ F'\rangle$ is a* **matched successor state** *of $\langle E,\ F\rangle$.*

Definition 6.3 *If $\{E_1, E_2, \cdots\}$ is some sequence of expressions or pairs of expressions in a deterministic labelled transition system (possibly infinite) such that E_{i+1} is a successor state of E_i then $\{E_1, E_2, \cdots\}$ is a* **transition sequence** *for E_1.*

Definition 6.4 *If $\{E_1, E_2, \cdots\}$ is a transition sequence of pairs of expressions such that for all i E_{i+1} is a matched successor state of E_i then $\{E_1, E_2, \cdots\}$ is a* **matched transition sequence** *for E_1.*

If $E \sim F$ then we would expect to be able to find a matched transition sequence from $\langle E,\ F\rangle$. In fact we would expect to be able to find a number of matched transition sequences if there is more than one possible transition from E and F. We want to select some set of "interesting" transition sequences, in particular we want the set that describes all the expressions visited by the transition sequence before bisimilarity can be established. We are, therefore, not interested in sequences after they have returned to the original pair of expressions, or started to loop in some other way. Therefore we only want to consider sequences up to the first duplication. We are also not interested in syntactically identical pairs of expressions since they are bisimilar by the reflexivity of $\sim$.

Definition 6.5 *A* **chopped transition sequence (CTS)** *for E is any one of the following:*

1. *A finite transition sequence for E which contains no duplicate elements except for its last element which is a duplicate of a previous one. Moreover it contains no pairs which are syntactically identical.*
2. *A finite transition sequence for E which contains no duplicate elements and no syntactically identical pairs, except for its last pair which are syntactically identical.*

3. *A finite transition sequence for E which contains no duplicate elements and no syntactically identical pairs and there are no successor states for its last element.*

4. *An infinite transition sequence for E which contains no duplicates and no syntactically identical pairs.*

1. 2. and 3. are called **Finite Chopped Transition Sequences**. *4. is called an* **Infinite Chopped Transition Sequence**

The main theorem (6.2) in what follows is that the union of chopped transition sequences from some pair $\langle E,\ F\rangle$ in a deterministic transition sequence is the smallest bisimulation that contains $\langle E,\ F\rangle$.

To establish this it is first shown (theorem 6.1) that if some pair $\langle E',\ F'\rangle$ is in a matched transition sequence for $\langle E,\ F\rangle$ then they are in a matched CTS for $\langle E,\ F\rangle$. This fact is used to show that the set of matched CTS's is a bisimulation, by showing that every matched successor of some pair is in a matched CTS. All that is then required is to show that any other bisimulation containing $\langle E,\ F\rangle$ also contains the set of matched CTS's, this is proved by induction on the number of matched successor states.

Lemma 6.1 *If E and F are syntactically identical then all matched successor states of E and F in a deterministic labelled transition system are syntactically identical.*

> **Proof** The proof follows from the determinacy of the labelled transition system. □

Theorem 6.1 *If $\langle E',\ F'\rangle$ is a member of some matched transition sequence for $\langle E,\ F\rangle$ in some deterministic LTS such that E' and F' are not syntactically identical then $\langle E',\ F'\rangle$ is a member of some chopped transition sequence for $\langle E,\ F\rangle$.*

Proof by Induction on i, the length of some matched transition sequence from $\langle E,\ F\rangle$ to $\langle E',\ F'\rangle$

Base Case $\langle E,\ F\rangle$ is the 1st member of all transition sequences from $\langle E,\ F\rangle$, so $\langle E',\ F'\rangle = \langle E,\ F\rangle$. Clearly $\langle E,\ F\rangle$ is also a member of all CTS's for $\langle E,\ F\rangle$.

Step Case Assume that for all $n \leq i$ if $\langle E'',\ F''\rangle$ is the nth member of some matched transition sequence for $\langle E,\ F\rangle$ then it is the member of some CTS for $\langle E,\ F\rangle$ (This is the induction hypothesis and is used to prove case 3(a) below – the other cases don't require it). Let $\langle E',\ F'\rangle$ be the $i+1$th member of some matched transition sequence, T, for $\langle E,\ F\rangle$.

Let T_{i+1} be the sequence of the first $i+1$ elements of T then:

1. If T_{i+1} is chopped then $\langle E',\ F'\rangle$ is in a CTS for $\langle E,\ F\rangle$.

2. If T_{i+1} is not chopped but it can be extended to a CTS by adding some (possibly infinite) sequence of successor states to $\langle E',\ F'\rangle$ then $\langle E',\ F'\rangle$ is in a CTS for $\langle E,\ F\rangle$.
3. If T_{i+1} is not chopped and can't be extended then there is at least one pair $\langle E_j,\ F_j\rangle$ such that either E_j and F_j are syntactically identical or $\langle E_j,\ F_j\rangle$ appears in both the jth and kth positions of T_{i+1} such that $j < k \leq i+1$.
   (a) If there is a pair $\langle E_j,\ F_j\rangle$ that appears in both the jth and kth positions of T_{i+1} such that $j < k \leq i+1$ then the sequence $\langle E_1,\ F_1\rangle, \cdots, \langle E_j,\ F_j\rangle, \langle E_{k+1},\ F_{k+1}\rangle, \cdots, \langle E',\ F'\rangle$ is also a matched transition sequence for $\langle E,\ F\rangle$. In this sequence $\langle E',\ F'\rangle$ is the nth element for some $n \leq i$ (since at least one element has been removed from the sequence) hence by the induction hypothesis $\langle E',\ F'\rangle$ is in some CTS for $\langle E,\ F\rangle$.
   (b) If E_j and F_j are syntactically identical then (since T_{i+1} is matched) all their successor states are syntactically identical (by lemma 6.1) hence E' and F' are syntactically identical which contradicts the initial assumption.

□

We now come to the main theorem.

Theorem 6.2 *The union of matched chopped transition sequences for a bisimilar pair, $\langle E,\ F\rangle$ in a deterministic LTS, (written $MCTS(\langle E,\ F\rangle)$) is a bisimulation and is contained in all other bisimulations, which contain $\langle E,\ F\rangle$. i.e. it is the smallest bisimulation that contains $\langle E,\ F\rangle$.*

Proof. Since $E \sim F$ there exists (at least one) bisimulation $\mathcal{R}$ such that $\langle E,\ F\rangle \in \mathcal{R}$ and $\mathcal{R} \subseteq \langle \mathcal{R} \cup \sim\rangle$.

The proof will proceed in two parts, first showing that $MCTS(\langle E,\ F\rangle)$ is a bisimulation and then that for any bisimulation, $\mathcal{R}$, containing $\langle E,\ F\rangle$, $MCTS(\langle E,\ F\rangle) \subseteq \mathcal{R}$ hence that $MCTS(\langle E,\ F\rangle)$ is the smallest bisimulation containing $\langle E,\ F\rangle$.

1. $MCTS(\langle E,\ F\rangle)$ **is a bisimulation**.

   Let $\langle E',\ F'\rangle$ be an arbitrary member of $MCTS(\langle E,\ F\rangle)$. Then $\langle E',\ F'\rangle$ is the ith member of some matched CTS. To show that $MCTS(\langle E,\ F\rangle)$ is a bisimulation then it is necessary to show that for all α, $E' \xrightarrow{\alpha} E'^{\alpha}$ iff $F' \xrightarrow{\alpha} F'^{\alpha}$ and $\langle E'^{\alpha},\ F'^{\alpha}\rangle \in MCTS(\langle E,\ F\rangle)$. $\langle E'^{\alpha},\ F'^{\alpha}\rangle$ is a matched successor state for $\langle E',\ F'\rangle$ so $\langle E'^{\alpha},\ F'^{\alpha}\rangle$ must be in a matched transition sequence for $\langle E,\ F\rangle$ so by theorem 6.1 $\langle E'^{\alpha},\ F'^{\alpha}\rangle$ must be in a matched CTS for $\langle E,\ F\rangle$.

   Hence $MCTS(\langle E, F\rangle)$ is a $\langle - \rangle$–dense relation and therefore $MCTS(\langle E, F\rangle)$ is a bisimulation.

2. **If $\mathcal{R}$ is a bisimulation containing $\langle E, F\rangle$, $MCTS(\langle E, F\rangle) \subseteq \mathcal{R}$**

   Let $\langle E_i, F_i\rangle$ be the ith member of some matched CTS, T.

   Proof that $\langle E_i, F_i\rangle \in \mathcal{R}$ by induction on i.

   Base Case $\langle E, F\rangle \in \mathcal{R}$ by hypothesis.

   Step Case Assume that $\langle E_n, F_n\rangle \in \mathcal{R}$. Since $\mathcal{R}$ is a bisimulation all the matched successor states of $\langle E_n, F_n\rangle$ are in $\mathcal{R}$. So if $\langle E_{n+1}, F_{n+1}\rangle$ is an arbitrary matched successor state for $\langle E_n, F_n\rangle$ then $\langle E_{n+1}, F_{n+1}\rangle \in \mathcal{R}$.

   Hence $MCTS(\langle E, F\rangle) \subseteq \mathcal{R}$.

$\square$.

This means that analysis of the matched chopped transition sequences of any pair of objects is central to determining their bisimilarity.

6.4 The Coinduction Method Heuristic

The preceding analysis might suggest that a sensible choice of trial bisimulation for some pair $\langle E, F\rangle$ would be

$$\mathcal{B} \stackrel{\text{def}}{=} \{\langle E', F'\rangle | \exists S.S \text{ is a matched CTS for } \langle E, F\rangle, \langle E', F'\rangle \in S\} \tag{6.4}$$

If this set is to qualify as a bisimulation then for any pair in the bisimulation $\langle E', F'\rangle$ and any transition α, $E' \stackrel{\alpha}{\to} E'' \Leftrightarrow F' \stackrel{\alpha}{\to} F''$. To prove this it has to be possible to determine the transitions for an arbitrary pair in the relation and the description of $\mathcal{B}$ in (6.4) simply fails to supply that information.

One method for determining the transitions from arbitrary members of $\mathcal{B}$ is to explore the space of the transitions from $\langle E, F\rangle$. The suggestion here is that a proof critic be used to conduct that exploration.

The heuristic employed by the coinduction method assumes that all chopped transition sequences are of length no more than two.

If the choice of $\{\langle E, F\rangle\}$ made by the Coinduction method is incorrect then it has to be extended somehow because for some transition α, $E \stackrel{\alpha}{\to} E'$ and $F \stackrel{\alpha}{\to} F'$ $\langle E', F'\rangle \notin \{\langle E, F\rangle\}$. If the proof is to go through, the trial bisimulation has to be extended so that *at the very least* it includes $\langle E', F'\rangle$.

We know the transition space can be explored by exploring all chopped transition sequences. If these transition sequences are finite then this is a deterministic process. However some chopped transition sequences are infinite. In this case a finite description for $\mathcal{B}$ may still be possible if there is some sort of identifiable pattern in the sequence (more of this in §6.5.1).

6.5 The Revise Bisimulation Critic

A critic is added to the proof strategy for coinduction which extends the current trial bisimulation by adding in any new successor states that are reached that are not already in the bisimulation. At the same time it seeks for patterns that will allow infinite chopped transition sequences to be finitely described. The critic is shown in figure 6–1.

Although this is a very general description, it should be clear that the coinduction proof strategy together with this critic provides a method for exploring all the chopped transition sequences in an attempt to discover $\mathcal{B}$.

Preconditions

1. The coinduction method has been used in this branch of the proof.
2. The current goal is $\forall R \langle f_1,\ g_1\rangle \in \mathcal{R} \wedge \ldots \wedge \langle f_n,\ g_n\rangle \in \mathcal{R} \Rightarrow \langle \phi,\ \psi\rangle \in \mathcal{R}$. Where ϕ and ψ are both instantiated and $\langle \phi,\ \psi\rangle$ are not already known to be in $\mathcal{R}$.

Patch

If

A pattern has been spotted (see later for details) in the some subsequence of the pairs used to describe $\mathcal{R}$ (i.e. the trial bisimulation) and $\langle \phi,\ \psi\rangle$. A new description has been proposed replacing this subsequence with $\langle \phi_1,\ \psi_1\rangle \cup \cdots \cup \langle \phi_n,\ \psi_n\rangle$ Start the proof again from the most recent call of the coinduction method, supplying the revised relation as the new trial bisimulation.

Else

Change the trial bisimulation by adding the set $\{\langle \phi, \psi\rangle\}$ to it and start the proof again from the most recent call of the coinduction method, supplying the revised relation as the new trial bisimulation.

Figure 6–1: The Revise Bisimulation Critic

As an example of the critic in action consider example 3.4 from chapter 3.

Example 6.2

$$\forall a, b.\ lswap(a, b) \sim merge(lconst(a), lconst(b))$$

where:

$$\begin{aligned} lswap(A,B) &\overset{\text{red}}{\rightsquigarrow} A :: lswap(B,A) && (6.5)\\ lconst(A) &\overset{\text{red}}{\rightsquigarrow} A :: lconst(A) && (6.6)\\ merge(nil,L) &\overset{\text{red}}{\rightsquigarrow} L && (6.7)\\ merge(L,nil) &\overset{\text{red}}{\rightsquigarrow} L && (6.8)\\ merge(H_1 :: T_1, H_2 :: T_2) &\overset{\text{red}}{\rightsquigarrow} H_1 :: H_2 :: merge(T_1,T_2) && (6.9) \end{aligned}$$

The Coinduction method chooses the trial bisimulation

$$\{\langle lswap(A,B),\ merge(lconst(A), lconst(B))\rangle\}$$

The Gfp Membership method provides the subgoal

$$\begin{gathered} \forall \mathcal{R}.\ \langle lswap(A,B),\ merge(lconst(A), lconst(B))\rangle \in \mathcal{R} \Rightarrow \\ ((lswap(A',B') \xrightarrow{\alpha} \phi \vee merge(lconst(A'), lconst(B')) \xrightarrow{\alpha} \psi) \Rightarrow \\ lswap(A',B') \xrightarrow{\alpha} \phi \wedge merge(lconst(A'), lconst(B')) \xrightarrow{\alpha} \psi \wedge \\ \langle \phi,\ \psi\rangle \in \mathcal{R} \cup \sim) \end{gathered} \tag{6.10}$$

and Evaluate method makes this

$$\begin{gathered} \forall \mathcal{R}.\ \langle lswap(A,B),\ merge(lconst(A), lconst(B))\rangle \in \mathcal{R} \Rightarrow \\ ((A' :: lswap(B',A') \xrightarrow{\alpha} \phi \vee A' :: B' :: merge(lconst(A'), lconst(B')) \xrightarrow{\alpha} \psi) \Rightarrow \\ A' :: lswap(B',A') \xrightarrow{\alpha} \phi \wedge A' :: B' :: merge(lconst(A'), lconst(B')) \xrightarrow{\alpha} \psi \wedge \\ \langle \phi,\ \psi\rangle \in \mathcal{R} \cup \sim) \end{gathered} \tag{6.11}$$

There are two possible values for α, `hd` and `tl`. I am only going to consider the subgoal produced using `tl` here since the `hd` transition subgoal is trivial by appeal to the reflexivity of $\sim$.

The `tl` transition produces the subgoal

$$\begin{gathered} \forall \mathcal{R}.\ \langle lswap(A,B),\ merge(lconst(A), lconst(B))\rangle \in \mathcal{R} \Rightarrow \\ \langle lswap(B',A'),\ B' :: merge(lconst(A'), lconst(B'))\rangle \in \mathcal{R} \end{gathered}$$

The proof attempt fails at this point, because it isn't possible to match $\langle lswap(B',A'),\ B' :: merge(lconst(A'), lconst(B'))\rangle$ and $\langle lswap(A,B),\ merge(lconst(A), lconst(B))\rangle$.

Assuming that no pattern indicating an infinite sequence has been detected between this new pair and the hypothesis (more of this in §6.5.1) the critic suggests adding the set $\{\langle lswap(B',A'),\ B' :: merge(lconst(A'), lconst(B'))\rangle\}$ to the trial bisimulation and starting again.

The process of forming the proof plans proceeds once more with the Gfp Membership method producing two subgoals each with two hypotheses instead of one:

$$\begin{array}{c}\forall \mathcal{R}.\ \langle lswap(A,B),\ merge(lconst(A), lconst(B))\rangle \in \mathcal{R} \wedge \\ \langle lswap(B',A'),\ B' :: merge(lconst(A'), lconst(B'))\rangle \in \mathcal{R} \Rightarrow \\ ((lswap(A'',B'') \stackrel{\alpha}{\rightarrow} \phi \vee merge(lconst(A''), lconst(B'')) \stackrel{\alpha}{\rightarrow} \psi) \Rightarrow \\ lswap(A'',B'') \stackrel{\alpha}{\rightarrow} \phi \wedge merge(lconst(A''), lconst(B'')) \stackrel{\alpha}{\rightarrow} \psi \wedge \\ \langle \phi,\ \psi\rangle \in \mathcal{R} \cup \sim)\end{array} \tag{6.12}$$

$$\begin{array}{c}\forall \mathcal{R} \langle lswap(A,B),\ merge(lconst(A), lconst(B))\rangle \in \mathcal{R} \wedge \\ \langle lswap(B',A'),\ B' :: merge(lconst(A'), lconst(B'))\rangle \in \mathcal{R} \Rightarrow \\ ((lswap(B'',A'') \stackrel{\alpha}{\rightarrow} \phi \vee B'' :: merge(lconst(A''), lconst(B'')) \stackrel{\alpha}{\rightarrow} \psi) \Rightarrow \\ lswap(B'',A'') \stackrel{\alpha}{\rightarrow} \phi \wedge B'' :: merge(lconst(A''), lconst(B'')) \stackrel{\alpha}{\rightarrow} \psi \wedge \\ \langle \phi,\ \psi\rangle \in \mathcal{R} \cup \sim)\end{array} \tag{6.13}$$

after Evaluation and the taking of transitions these are both solved: the subgoals resulting from the `hd` transitions by appeal to the reflexivity of $\sim$ and those from the `tl` transitions by appeal to one of the two hypotheses. This final proof plan is very similar to the proof shown in chapter 3.

6.5.1 The Divergence Check

Up until now I have been vague about the process of detecting infinite transition sequences which is the final part of the Revise Bisimulation critic.

The "spotting of patterns" mentioned in figure 6–1 is performed using a *divergence check* based on work by Walsh [Walsh 96]. The check attempts to find some term structure introduced by the revisions which is accumulating in the sequence of equations which describe the trial bisimulation. It is this structure which is preventing fertilization solving the goals (i.e. proving the pair is an element of the trial bisimulation). The critic identifies the accumulating structure using difference matching [Basin & Walsh 92]. It is described in figure 6–2.

The use of the divergence check can be seen more clearly with an example.

Example 6.3 *Consider example 3.7 from chapter 3.*

$$\forall x, f.\ h(f,x) \sim x :: map(f, h(f,x)) \Rightarrow \forall x, f.\ h(f,x) \sim iterates(f,x)$$

The Coinduction method chooses the trial bisimulation

$$\{\langle h(F,X),\ iterates(F,X)\rangle)\} \tag{6.14}$$

The Gfp Membership and Evaluate Methods will provide the subgoal

Conditions

1. There is a sequence of sets within the trial bisimulation, $\{\langle l_i, r_i\rangle\}$, which have been generated by the revise bisimulation critic.
2. There exist G_l, G_r, H_l, H_r (at least one H_i $(i = l, r)$ non trivial) such that for each j difference matching gives:

$$\{\langle l_j,\ r_j\rangle\} = \{\langle G_l(U_{l_j}),\ G_r(U_{r_j})\rangle\},$$
$$\{\langle l_{j+1},\ r_{j+1}\rangle\} = \{\langle G_l(\boxed{H_l(\underline{U_{l_j}})}),\ G_r(\boxed{H_r(\underline{U_{r_j}})})\rangle\}$$

New Trial Bisimulation

$$\langle G_l((H_l)^n(U_{l_0})),\ G_r((H_r)^n(U_{r_0}))\rangle$$

Figure 6–2: The Divergence Check

$$\begin{array}{c}\forall \mathcal{R}\langle h(F,X),\ iterates(F,X)\rangle \in \mathcal{R} \Rightarrow \\ ((X' :: map(F', h(F',X')) \xrightarrow{\alpha} \phi \vee X' :: iterates(F'), F'(X') \xrightarrow{\alpha} \psi) \wedge \\ X' :: map(F', h(F',X')) \xrightarrow{\alpha} \phi \wedge X' :: iterates(F'), F'(X') \xrightarrow{\alpha} \psi \wedge \\ \langle \phi,\ \psi\rangle \in \mathcal{R} \cup \sim)\end{array} \tag{6.15}$$

There are two possible values for α, `hd` and `tl`. I am only going to consider the subgoal produced using `tl` here since the `hd` transition subgoal is trivial by appeal to the reflexivity of $\sim$.

The `tl` transition produces the subgoal

$$\begin{array}{c}\forall R\langle h(F,X),\ iterates(F,X)\rangle \in \mathcal{R} \Rightarrow \\ \langle \boxed{map(F', \underline{h(F',X')})}^{\uparrow},\ iterates(F', \boxed{F'(\underline{X'})}^{\uparrow})\rangle \in \mathcal{R}\end{array} \tag{6.16}$$

The proof attempt fails because $\langle map(F', h(F',X')),\ iterates(F', F'(X'))\rangle \notin \mathcal{R}$.

According to the critic (figure 6–1) this new pair is added into the trial bisimulation.

The process of forming the proof plans proceeds once more producing two subgoals. The first subgoal is similar to (6.15) but with an additional hypothesis. This extra hypothesis can be used to fertilize at the point where the proof attempt from (6.15) became blocked. The second new subgoal is[1]:

$$\begin{array}{c}
\forall \mathcal{R} \langle h(F,X),\ iterates(F,X)\rangle \in \mathcal{R} \wedge \\
\langle map(F', h(F',X')),\ iterates(F', F'(X'))\rangle \in \mathcal{R} \Rightarrow \\
((F''(X'') :: map(F'', map(F'', h(F'',X''))) \xrightarrow{\alpha} \phi \vee \\
F''(X'') :: iterates(F'', F''(F''(X''))) \xrightarrow{\alpha} \psi) \Rightarrow \\
F''(X'') :: map(F'', map(F'', h(F'',X''))) \xrightarrow{\alpha} \phi \wedge \\
F''(X'') :: iterates(F'', F''(F''(X''))) \xrightarrow{\alpha} \psi \wedge \\
\langle \phi,\ \psi \rangle \in \mathcal{R} \cup \sim)
\end{array} \tag{6.17}$$

Taking `tl` transitions leads to the subgoal

$$\begin{array}{c}
\langle h(F,X),\ iterates(F,X)\rangle \in \mathcal{R} \wedge \\
\langle map(F', h(F',X')),\ iterates(F', F'(X'))\rangle \in \mathcal{R} \Rightarrow \\
\langle map(F'', \boxed{map(F'', \underline{h(F'',X'')})}^{\uparrow}),\ iterates(F'', \boxed{F''(\underline{F''(X'')})}^{\uparrow})\rangle \in \mathcal{R}
\end{array} \tag{6.18}$$

Once again the Revise Bisimulation critic will intervene and suggest adding $\{\langle map(F'', map(F'', h(F'',X''))),\ iterates(F'', F''(F''(X'')))\rangle\}$ to $\mathcal{R}$.

A human can tell that this process can be infinitely repeated, i.e. the transition sequence of `tl`–successors is infinite. At this point difference matching can be used to try and spot a pattern.

The first three elements of the sequence are:

$$\begin{array}{c}
s_0 = \langle h(F,X),\ iterates(F,X)\rangle \in \mathcal{R} \\
s_1 = \langle map(F', h(F',X')),\ iterates(F', F'(X'))\rangle \in \mathcal{R} \\
s_2 = \langle map(F'', map(F'', h(F'',X''))),\ iterates(F'', F''(F''(X'')))\rangle \in \mathcal{R}
\end{array}$$

Difference matching successive pairs of these[2], produces the sequence:

$$\begin{array}{c}
s'_0 = \langle h(F,X),\ iterates(F,X)\rangle \in \mathcal{R} \\
s'_1 = \langle \boxed{map(F', \underline{h(F',X')})},\ iterates(F', \boxed{F'(\underline{X'})})\rangle \in \mathcal{R} \\
s'_2 = \langle \boxed{map(F'', \underline{map(F'', h(F'',X''))})},\ iterates(F'', \boxed{F''(\underline{F''(X'')})})\rangle \in \mathcal{R}
\end{array}$$

It should be clear from viewing the above sequence that the accumulating term structure is being marked out by the wave fronts. This shouldn't be surprising

[1]Clearly it would be more efficient to proceed without repeating the transition analysis for $\langle h(F,X),\ iterates(F,X)\rangle \in \mathcal{R}$ and move straight to the analysis of $\langle map(F', h(F',X')),\ iterates(F', F'(X'))\rangle \in \mathcal{R}$ however this hasn't been implemented.

[2]This difference matching should not be confused with the difference matching after the Transition method which resulted in the annotations on goals 6.16 and 6.18 which was intended to motivate rippling if possible. Here the difference matching is being used to identify a common pattern of differences

since the difference matching singles out differences between two equations and it is precisely these differences which are preventing fertilization occurring between them.

Once divergence is spotted it is necessary to find an appropriate patch. Walsh's divergence critic patched the proofs he was attempting by speculating and proving additional lemmata. In coinductive proofs a generalisation, using $(\cdots)^n$, replaces the subsequence of the trial bisimulation since the divergence is being caused by the repeated addition of H^i (as defined by the divergence check) every time the tail of the latest addition to the trial bisimulation is examined.

The critic instantiates G_l, G_r, H_l, H_r and U_{l_0} and U_{r_0} (from figure 6–2) as *id* (the identity function), $\mathit{iterates}(F)$, $\mathit{map}(F)$, F, $h(F,X)$ and X respectively so $\mathcal{R} = \{\langle \mathit{map}(F)^N(h(F,X)),\ \mathit{iterates}(F, F^N(X))\rangle\}$. This new relation is given to the coinduction method which produces the goal:

$$\begin{array}{c}\{\langle \mathit{map}(F)^N(h(F,X)),\ \mathit{iterates}(F, F^N(X))\rangle\} \subseteq \\ \langle\{\langle \mathit{map}(F')^{N'}(h(F',X')),\ \mathit{iterates}(F', F'^{N'}(X'))\rangle\} \cup \sim\rangle\end{array} \tag{6.19}$$

The Gfp Membership method's preconditions specify that the current goal is a subset goal of this form. In conjunction with Evaluation this produces the goal

$$\begin{array}{c}\forall R \langle \mathit{map}(F)^N(h(F,X)),\ \mathit{iterates}(F, F^N(X))\rangle \in \mathcal{R} \Rightarrow \\ ((F'^{N'}(X') :: \mathit{map}(F')^{N'}(\mathit{map}(F', h(F',X'))) \xrightarrow{\alpha} \phi \vee \\ F'^{N'}(X') :: \mathit{iterates}(F', F'(F'^{N'}(X'))) \xrightarrow{\alpha} \psi) \Rightarrow \\ F'^{N'}(X') :: \mathit{map}(F')^{N'}(\mathit{map}(F', h(F',X'))) \xrightarrow{\alpha} \phi \wedge \\ F'^{N'}(X') :: \mathit{iterates}(F', F'(F'^{N'}(X'))) \xrightarrow{\alpha} \psi \wedge \\ \langle \phi,\ \psi\rangle \in \mathcal{R} \cup \sim)\end{array} \tag{6.20}$$

The Transition method then produces the subgoal:

$$\cdots \Rightarrow \langle \mathit{map}(F')^{N'}(\boxed{\mathit{map}(F', \underline{h(F',X')})}^{\uparrow}),\ \mathit{iterates}(F', \boxed{F'(\underline{F'^{N'}(X')})}^{\uparrow})\rangle \in \mathcal{R} \cup \sim \tag{6.21}$$

which ripples to:

$$\cdots \Rightarrow \langle \mathit{map}(F')^{\boxed{s(\underline{N'})}^{\downarrow}} h(F',X')),\ \mathit{iterates}(F', F'^{\boxed{s(\underline{N'})}^{\downarrow}}(X'))\rangle \in \mathcal{R} \tag{6.22}$$

which can be solved by fertilization[3].

[3]Since all the variables, including N', are sinks

6.6 Limitations of the Divergence Check

Ideally a divergence check for the Revise Bisimulation critic:

1. Fires if the proof strategy was in the process of exploring an infinite chopped transition sequence;

2. Only extends the trial bisimulation by elements in $\mathcal{B}$. In particular it only fires if the proof strategy was in the process of exploring an infinite chopped transition sequence.

Neither of these is the case with the divergence check proposed here (in fact since the problem is probably equivalent to the halting problem, no critic is going to be perfect).

6.6.1 The Check doesn't Fire even though the Sequence is Infinite

Recall the conditions of the divergence check. It will fail to fire if it fails to find a sequence of terms that difference match to produce identical wave fronts.

Difference Matching Fails

This will happen if one member of the sequence is not embedded in some way in the next member.

Example 6.4 *Consider divergence that is hidden within function definitions:*

$$f_1(nil) \overset{\text{red}}{\leadsto} nil \tag{6.23}$$
$$f_1(H :: T) \overset{\text{red}}{\leadsto} (H+1) :: f_2(T) \tag{6.24}$$

$$f_2(nil) \overset{\text{red}}{\leadsto} nil \tag{6.25}$$
$$f_2(H :: T) \overset{\text{red}}{\leadsto} (H+2) :: f_3(T) \tag{6.26}$$

$$f_3(nil) \overset{\text{red}}{\leadsto} nil \tag{6.27}$$
$$f_3(H :: T) \overset{\text{red}}{\leadsto} (H+3) :: f_4(T) \tag{6.28}$$

$$\vdots$$

This transition sequence has no finite description unless the function definitions themselves follow a pattern. In either case the divergence critic is incapable of *spotting that it may be exploring some kind of infinite chopped transition sequence whose ith member is the term $f_i(L)$ where*

$$
\begin{aligned}
f_i(nil) &\overset{\text{red}}{\rightsquigarrow} nil & (6.29)\\
f_i(H :: T) &\overset{\text{red}}{\rightsquigarrow} (H + i) :: f_{i+1}(T) & (6.30)
\end{aligned}
$$

These functions might be generalised by

$$
\begin{aligned}
f(N, nil) &\overset{\text{red}}{\rightsquigarrow} nil & (6.31)\\
f(N, H :: T) &\overset{\text{red}}{\rightsquigarrow} (H + N) :: f(s(N), T) & (6.32)
\end{aligned}
$$

but without information on how they are being generated it is impossible to be certain that this is valid.

Wave Fronts exist but are not Identical

This particular example was suggested by Collins[4].

Example 6.5 *Take the function definitions:*

$$
\begin{aligned}
g(N, nil) &\overset{\text{red}}{\rightsquigarrow} nil & (6.33)\\
g(N, H :: T) &\overset{\text{red}}{\rightsquigarrow} H :: g(s(N), g(N, T)) & (6.34)\\
\\
lconst(M) &\overset{\text{red}}{\rightsquigarrow} M :: lconst(M) & (6.35)
\end{aligned}
$$

Note that g itself is clearly a version of id (the identity function). The following is the chopped transition sequence from $\langle lconst(M),\ g(N, lconst(M))\rangle$ generated by exploring successive $\overset{\texttt{tl}}{\rightarrow}$ transitions.

$$
\begin{gathered}
\langle lconst(M),\ g(N, lconst(M))\rangle \in \mathcal{R}\\
\langle lconst(M),\ g(s(N), g(N, lconst(M)))\rangle \in \mathcal{R}\\
\langle lconst(M),\ g(s(s(N)), g(s(N), g(s(N), g(N, lconst(M)))))\rangle \in \mathcal{R}\\
\vdots
\end{gathered}
\tag{6.36}
$$

There are a number of different ways to difference match this sequence, some of which are shown below:

[4]Private Communication

$$\begin{gathered}
\langle lconst(M),\ g(N, lconst(M))\rangle \in \mathcal{R} \\
\langle lconst(M),\ g(\boxed{s(\underline{N})}, \boxed{g(N, \underline{lconst(M)})})\rangle \in \mathcal{R} \\
\langle lconst(M),\ g(\boxed{s(\underline{s(N)})}, \boxed{g(s(N), g(s(N), \underline{g(N, lconst(M))}))})\rangle \in \mathcal{R} \\
\vdots
\end{gathered} \tag{6.37}$$

$$\begin{gathered}
\langle lconst(M),\ g(N, lconst(M))\rangle \in \mathcal{R} \\
\langle lconst(M),\ \boxed{g(s(N), \underline{g(N, lconst(M))})}\rangle \in \mathcal{R} \\
\langle lconst(M),\ g(\boxed{s(\underline{s(N)})}, \boxed{g(s(N), g(s(N), \underline{g(N, lconst(M))}))})\rangle \in \mathcal{R} \\
\vdots
\end{gathered} \tag{6.38}$$

$$\begin{gathered}
\langle lconst(M),\ g(N, lconst(M))\rangle \in \mathcal{R} \\
\langle lconst(M),\ g(\boxed{s(\underline{N})}, \boxed{g(N, \underline{lconst(M)})})\rangle \in \mathcal{R} \\
\langle lconst(M),\ \boxed{g(s(s(N)), g(s(N), \underline{g(s(N), g(N, lconst(M)))}))}\rangle \in \mathcal{R} \\
\vdots
\end{gathered} \tag{6.39}$$

$$\begin{gathered}
\langle lconst(M),\ g(N, lconst(M))\rangle \in \mathcal{R} \\
\langle lconst(M),\ \boxed{g(s(N), \underline{g(N, lconst(M))})}\rangle \in \mathcal{R} \\
\langle lconst(M),\ \boxed{g(s(s(N)), g(s(N), \underline{g(s(N), g(N, lconst(M)))}))}\rangle \in \mathcal{R} \\
\vdots
\end{gathered} \tag{6.40}$$

There is clearly divergence going on here, but it won't be spotted by the divergence critic since the wave fronts in all possible difference matches change from difference match to difference match. Moreover, even applying difference matching to the sequence of wave–fronts still fails to find a sequence of matching wave fronts.

The bisimulation needed here is:

$$\{\langle lconst(M),\ h^N(lconst(M))\rangle\}$$

Where h is defined by (6.41) and (6.42) and the lemma (6.43) is known.

$$h^0(X) \overset{\text{red}}{\rightsquigarrow} X \tag{6.41}$$

$$h^{s(N)}(X) \overset{\text{red}}{\rightsquigarrow} g(s(N), g(N, h^N(X))) \tag{6.42}$$

$$h^N(H :: T) \overset{\text{red}}{\rightsquigarrow} H :: h^{s(s(N))}(T) \tag{6.43}$$

It is clearly beyond the scope of the critic to either spot this divergence or find the appropriate patch.

This situation is distinguished from the previous one, by the fact that wave fronts are generated, even though they do not match each other. This suggests that if some member of a transition sequence is embedded in a subsequent one (i.e. it is possible to difference match the two expressions) then there is a risk of divergence.

This suggests a number of possible extensions to the critic, including more sophisticated analysis of the wave fronts or the addition of an interactive aspect which, on detection of a difference match indicative of the presence of some sort of pattern, appeals to the user to supply a generalisation.

6.6.2 The Critic Extends the Bisimulation by elements not in $\mathcal{B}$

This will occur if the generalisation proposed by the critic lies outside $\mathcal{B}$.

A Reset is Lost

The obvious situation where this occurs is when the proposed generalisation is $f^N(K)$ when $\mathcal{B}$ only contains $f^M(K), M \in \mathcal{S}$ where $\mathcal{S}$ is some subset of the natural numbers. This will happen if $\mathcal{B}$ may contain $c_1, c_2, c_3, \cdots$, but not c_n for some natural number, n.

For instance suppose I have two functions: $s_{10}(X) \stackrel{\text{def}}{=} s(X) \bmod 10$ and s':

$$N \neq 9 \;\rightarrow\; s'(N, X) \stackrel{\text{red}}{\rightsquigarrow} s(X) \tag{6.44}$$

$$N = 9 \;\rightarrow\; s'(N, X) \stackrel{\text{red}}{\rightsquigarrow} 0 \tag{6.45}$$

NB. For $X \leq 10$ $s'(X, X) = s_{10}(X)$.

$iterates(s_{10}, 0)$ ($iterates$ is defined as in equation (5.4)) and $iterates(\lambda k.s'(k,k), 0)$ are bisimilar. They have the following transition sequence of tail transitions.

$$\begin{array}{c} \langle iterates(s_{10}, 0),\ iterates(\lambda k.s'(k,k), 0)\rangle \in \mathcal{R} \\ \langle iterates(s_{10}, 1),\ iterates(\lambda k.s'(k,k), 1)\rangle \in \mathcal{R} \\ \langle iterates(s_{10}, 2),\ iterates(\lambda k.s'(k,k), 2)\rangle \in \mathcal{R} \\ \vdots \end{array} \tag{6.46}$$

This chopped transition sequence is finite since $s_{10}(9) = s'(9,9) = 0$. Unfortunately the proposed Revise Bisimulation critic will generalise to $\langle iterates(s_{10}, s_{10}^n(0)),\ iterates(\lambda k.s'(k,k), s^n(0)\rangle \in \mathcal{R}$ where the required generalisation is either $\langle iterates(s_{10}, s_{10}^n(0)),\ iterates(\lambda k.s'(k,k), s^n(0)\ mod\ 10)\rangle \in \mathcal{R}$ or the set containing $\langle iterates(s_{10}, m),\ iterates(\lambda k.s'(k,k), m)\rangle \in \mathcal{R}$ for $m \in \{1, \cdots, 9\}$, in order to preserve the finite transition sequence.

Trivially this could have been avoided by extending the sequence far enough before looking for a pattern, however given any arbitrary cut–off of sequence length it would be possible to devise an example like (6.46) which wouldn't be captured by that cut–off.

A more sophisticated fix would involve recognising that one of the functions employed a reset (e.g. s').

6.6.3 Termination

One other problem with the Revise Bisimulation critic is that it always applies if rippling and fertilization are both blocked (its preconditions simply require fertilization to be impossible). This means that the revising process is potentially non–terminating.

As a result, an attempt at a coinductive proof fails only if the analysis of transitions reveals an inconsistency. Thus in those situations where the critic fails to recognise that it is exploring an infinite chopped transition sequence the process of extending the trial bisimulation is non–terminating.

6.7 Conclusion

This chapter has looked at critics for coinductive proofs.

Most importantly it has introduced the Revise Bisimulation critic which extends the proof methods to deal with a much large class of coinductive problems.

This chapter, together with chapter 5 forms the core discussion on proof planning for coinduction. Without focusing on the specifics of implementation they are intended as a description and discussion of the methods and critics required to proof plan coinductive proof.

The important point is that the most obvious strategy for attempting to discover $\mathcal{B}$ for some pair of expressions or some bisimulation containing this pair is via an exploration of the transition sequences from that pair. This can be achieved by the incremental approach suggested here. Although this was discussed within an explicit proof planning framework it seems clear that any implementation of this strategy would employ a "look ahead" process and a "revise" process. This "revise" is clearly an example of the use of a proof critic.

Although a look ahead could be implemented as part of an object–level proof tactic, a higher level (e.g. proof method) view will often be more conceptually efficient since it will allow the potential problem to be identified without having to work through much of the trivial language specific detail of the proof. Any implementation of this sort of strategy will employ elements associated with proof planning. A revision process is central to the strategy and hence a proof method process is desirable from an efficiency point of view. Even if not adopted universally in a system, the proof planning (and particularly the proof critic) paradigm has a clear contribution to make to the discovery of bisimulations for coinductive proof.

Chapter 7

Experimental Results and Evaluation

7.1 Introduction

This chapter reports the results of empirical testing of the proof strategy on two labelled transition systems. It starts with a description of the aims and design of the experiment and a brief report of the results. Complete listings of the results appear in appendix B.

It then examines the reasons for failed proof attempts in more detail and the implications of these failures and other observations arising out of the testing of the proof strategy.

It also considers the representativeness of the chosen set of the test theorems.

7.2 Aim

The proof methods and critics outlined in chapters 5 and 6 are heuristics. Theoretical evaluation of their effectiveness is difficult. They are not guaranteed to find a proof plan in all cases where one exists. The assertion instead is that they are *useful* heuristics. The definition of a *useful* heuristic is, itself, vague. The contention here is that heuristics will lead to proof plans for most *common* problems.

The aim of the experiments reported here, was to attempt to proof plan a representative selection of theorems using the methods and critics already described and determine how many theorems they successfully proof planned.

This serves a double purpose of evaluating the effectiveness of the proposed strategy and highlighting areas of the strategy that need improvement.

7.3 Source of Examples

The proof plans make no claim to deal with cases where divergence analysis of the sort demonstrated in example 3.5 in chapter 3 is required and so theorems which involved such divergence were excluded from consideration. These theorems were identified by inspection.

Examples of coinduction were drawn from the literature in particular [Paulson 93], [Fiore 93], [Rutten 96] and [Jacobs & Rutten 97].

CLAM itself has a corpus of theorems used for testing inductive proof plans. Those that were not excluded because of divergence problems were also used. Many of these were over natural numbers and so a labelled transition system involving $\stackrel{\mathtt{p}}{\rightarrow}$ as a transition was employed. Treating predecessor as a transition is probably not particularly useful for functional semantics, but it allowed the strategy to be tried on a wider class of theorems.

The theorems were planned for the small functional language described in some detail in chapter 3 (its transition rules are reproduced here for ease of reference (Figure 7–1)). However *CoCLAM* was designed with the intention that a number of differing labelled transitions could be supplied to allow a user to "plug and play". Limited testing of this was undertaken by performing some type checking proofs, the results of which are discussed in §7.10.

Examples of coinduction are relatively rare, the same one or two theorems appearing in almost every paper on the subject. Recent research is beginning to produce a larger corpus but this has tended to concentrate on problems outside the domain chosen for consideration (i.e. lazy functional programs). To counteract this, the proposed corpus was extended by a number of theorems taken from the *CLAM* corpus and some designed by myself, in order to provide a reasonably sized database.

It was necessary to use some theorems to test *CoCLAM* as it was being developed. This raised a concern that the proof strategy would be tailored specifically to prove this one set of theorems. In order to offset this the theorems were divided into two distinct groups, a *development set* and a *test set*. The test set was not used until the system was deemed finished.

$$\frac{a : \tau_1 \rightarrow \tau_2 \quad b : \tau_1}{a \xrightarrow{\texttt{ap}(b)} a(b)} \tag{7.1}$$

$$\frac{}{\underline{bv} \xrightarrow{bv} \bot} \tag{7.2}$$

$$\frac{}{0 \xrightarrow{\texttt{0}} \bot} \qquad \frac{}{s(n) \xrightarrow{\texttt{p}} n} \tag{7.3}$$

$$\frac{}{nil \xrightarrow{\texttt{nil}} \bot} \tag{7.4}$$

$$\frac{}{a :: b \xrightarrow{\texttt{hd}} a} \qquad \frac{}{a :: b \xrightarrow{\texttt{tl}} b} \tag{7.5}$$

$$\frac{}{leaf_{bin}(a) \xrightarrow{\texttt{label}} a} \tag{7.6}$$

$$\frac{}{node_{bin}(a, l, r) \xrightarrow{\texttt{label}} a} \tag{7.7}$$

$$\frac{}{node_{bin}(a, l, r) \xrightarrow{\texttt{left}} l} \qquad \frac{}{node_{bin}(a, l, r) \xrightarrow{\texttt{right}} r} \tag{7.8}$$

$$\frac{}{node(a, f) \xrightarrow{\texttt{label}} a} \qquad \frac{}{node(a, f) \xrightarrow{\texttt{forest}} f} \tag{7.9}$$

$$\frac{a : \tau \quad \tau \neq (\tau_1 \rightarrow \tau_2) \quad a \overset{\text{red}}{\rightsquigarrow} b \quad b \xrightarrow{\alpha} c}{a \xrightarrow{\alpha} c} \tag{7.10}$$

Figure 7–1: Transition Rules

7.4 Results

Full tables listing the theorems in each set and detailing which were proved and which were not can be found in appendix B.

The following table is a summary. It details the number of theorems in each set alongside the number that were proved. There is a third column, "implementation failures". This is the number of theorems that failed because of perceived problems with the implementation, as opposed to the proof strategy.

	Number of Theorems	Theorems Proved	Implementation Failures
Development Set	56	48	1
Test Set	55	42	5

7.5 Analysis of Results

It is reassuring to note that the similarity between the figures indicates that the methods and critic were not developed in such a way that they were specifically tuned to the development set, but they had a general applicability across the theorems and were capable of providing proof plans for somewhere around 80% of problems. This result also broadly supports the assertion that the heuristics employed were sufficient to proof plan common problems.

The sections that follow examine the various causes of failure in detail, but they are also listed here, for reference, together with the number of theorems that failed as a result of them. These are split into two groups, those failures that are thought to have arisen out of shortcomings in the proof strategy and those out of shortcomings in the implementation. For ease of reference these are termed *proof strategy errors* and *implementation errors*.

Error Type	Error	Development Set	Test Set
Proof Strategy Errors	Bisimulation Explosion	4	6
	False Hypothesis not Recognised		1
	Memory Error	1	1
	Matching of $(\cdots)^N$	2	
Implementation Errors	Incorrect Generalisation	1	2
	Initial Definition of Bisimulation		1
	Substitution Error		1
	Error in finding Transitions		1

Attempts were made to correct the Implementation Errors after the experiment, and subsequently 4 of the theorems in the test set that had failed because of implementational errors were planned. The fifth ran into the problem of bisimulation explosion. The problem causing incorrect generalisation for the theorem in the development set was not resolved.

7.6 Failure Analysis

7.6.1 Proof Strategy Errors

Discussion of the proof strategy errors follows:

Bisimulation Explosion

5 of the theorems in the development set and 6 in the test set failed because of a phenomenon we will call *bisimulation explosion*. Bisimulation explosion occurs when the number of pair schema involved in the bisimulation slow the proof planning process down unacceptably (in the case of the experiments reported here, no proof plan was found in 12 hours).

Consider example 10 in chapter 3 (the commutativity of plus). This was proved with the aid of two lemmata about plus:

$$X + 0 = X \tag{7.11}$$
$$X + s(Y) = s(X + Y) \tag{7.12}$$

At the time it was commented that an alternative to employing these lemmata was to use $\{\langle X + s^N(Y),\ Y + s^N(X)\rangle\}$ as the bisimulation. This bisimulation gives rise to the goal:

$$\begin{array}{c}\forall \mathcal{R}\langle X + s^N(Y),\ Y + s^N(X)\rangle \in \mathcal{R} \Rightarrow \\ (\forall \alpha.((X + s^N(Y) \xrightarrow{\alpha} \phi \vee Y + s^N(X) \xrightarrow{\alpha} \psi) \Rightarrow \\ ((X + s^N(Y) \xrightarrow{\alpha} \phi \wedge Y + s^N(X) \xrightarrow{\alpha} \psi) \wedge \\ \langle \phi,\ \psi\rangle \in \mathcal{R} \cup \sim)))\end{array}$$

In order to find the transitions from $X + s^N(Y)$ and $Y + s^N(X)$ several cases for various combinations of values for X, Y and N have to be considered, as discussed in §5.5.3, where the Evaluate method was first presented.

Take the case where $X = N = 0$ and $Y = s(Y_1)$, $X + s^N(Y)$ evaluates to $s(Y_1)$ to which the transition $\mathtt{p}$ applies resulting in Y_1 while $s(Y_1 + 0) \xrightarrow{\mathtt{p}} Y_1 + 0$, in the absence of the lemma (7.11) the expression

$$\langle X + s^N(Y),\ Y + s^N(X)\rangle \in \mathcal{R} \Rightarrow \langle Y_1,\ Y_1 + 0\rangle \in \mathcal{R}$$

is unprovable and the bisimulation has to be extended to include $\langle Y_1,\ Y_1 + 0\rangle$.

The final bisimulation (assuming that no additional lemmata about plus have been provided) used by this proof is:

$$\begin{array}{c}\{\langle V_0 + s^N(V_1),\ V_1 + s^N(V_0)\rangle\} \cup \\ \{\langle V_0 + 0,\ V_0\rangle\} \cup \\ \{\langle V_0,\ V_0 + 0\rangle\} \cup \\ \{\langle s^{s(N)}(V_1),\ V_1 + s^{s(N)}(0)\rangle\} \cup \\ \{\langle V_0 + s^{s(N)}(0),\ s^{s(N)}(V_0)\rangle\}\end{array}$$

This bisimulation appears large and inelegant. However without the inclusion of additional lemmata there does not appear to be a smaller one. Although *CoCLAM* found this bisimulation, for more complicated theorems such as,

$(z \times x) \times y \sim (z \times y) \times x$ the size of the bisimulation became so large that the system slowed down unacceptably. This slow down appears to be exponential and is particularly severe because of inefficiencies in the implementation of *CLAM*3's Wave method (other versions of *CLAM* have more efficient algorithms for Rippling, but do not support critics). At present the inefficiencies in the Wave method obscure any other inefficiencies that may exist in the rest of the system. However for large $\mathcal{R}$ it is not unreasonable to suppose that goals such as

$$\mathcal{R} \subseteq \langle \mathcal{R} \cup \sim \rangle$$

could remain a problem even if more efficient implementations of Rippling were used.

There is some further work that is needed to investigate this problem. A more efficient implementation of Rippling needs to be employed to see whether the slow down continues to be exponential. If this is the case there are several approaches that can be taken. One would be to investigate lemma speculation critics in order to find the additional lemmata that would reduce the bisimulation and this is discussed in chapter 10. Another would be to investigate more "efficient" representations of $\mathcal{R}$.

False Hypothesis not Recognised

No additional methods were supplied to *CoCLAM* to enable it to discharge hypotheses. These were regarded as general proof methods, not methods that were peculiar to coinduction in any way.

In the course of the investigation it transpired that *CLAM*3's and hence *CoCLAM*'s ability to evaluate hypotheses was very limited.

The theorem in which this limitation caused failure was

$$X = 0 \vee Y = 0 \Rightarrow X \times Y \sim 0$$

In planning this theorem *CoCLAM* examined a number of possible case splits on X and Y. If X and Y are both non zero then the transitions from $X \times Y$ and 0 are different. However this situation is disallowed since the hypothesis is false if X and Y are both non zero. *CoCLAM* failed to recognise the falsity of the hypothesis.

Memory Error During Evaluation

In two theorems the Evaluate method failed with a memory error. For both these theorems the search space for transitions was large, requiring in one case two separate lists to be split twice and in the second four times. This meant that solutions appeared relatively deep in the search tree (i.e. at depth 4 or 8 respectively). To illustrate this consider the theorem:

$$merge(merge(odd\_list(A), odd\_list(B)), merge(even\_list(A), even\_list(B))) = zig\_zag(nil, A :: nil, B :: nil) \tag{7.13}$$

where

$$\begin{aligned}
merge(nil, A) &\leadsto A && (7.14)\\
merge(A, nil) &\leadsto A && (7.15)\\
merge(H_1 :: T_1, H_2 :: T_2) &\leadsto H_1 :: H_2 :: merge(T_1, T_2) && (7.16)
\end{aligned}$$

$$\begin{aligned}
odd_list(nil) &\leadsto nil && (7.17)\\
odd_list(H :: nil) &\leadsto H :: nil && (7.18)\\
odd_list(H_1 :: H_2 :: T) &\leadsto H_1 :: odd_list(T) && (7.19)
\end{aligned}$$

$$\begin{aligned}
even_list(nil) &\leadsto nil && (7.20)\\
even_list(H :: nil) &\leadsto nil && (7.21)\\
even_list(H_1 :: H_2 :: T) &\leadsto H_2 :: even_list(T) && (7.22)
\end{aligned}$$

$$\begin{aligned}
zig_zag(nil, nil, nil) &\leadsto nil && (7.23)\\
zig_zag(nil, nil, H :: T) &\leadsto zig_zag(nil, H :: T, nil) && (7.24)\\
zig_zag(nil :: T, nil, L) &\leadsto zig_zag(T, L, nil) && (7.25)\\
zig_zag((H_1 :: T_1) :: T, nil, L) &\leadsto H_1 :: zig_zag(T, L, T_1 :: nil) && (7.26)\\
zig_zag(L_1, nil :: T, L_2) &\leadsto zig_zag(L_1, T, L_2) && (7.27)\\
zig_zag(L_1, (H_1 :: T_1) :: T, L_2) &\leadsto H_1 :: zig_zag(L_1, T, T_1 :: L_2) && (7.28)
\end{aligned}$$

In order to find a value for

$$merge(merge(odd\_list(A), odd\_list(B)), merge(even\_list(A), even\_list(B)))$$

It is necessary to split both A and B into $H_{1A} :: H_{2A} :: T_A$ and $H_{1B} :: H_{2B} :: T_B$. This solution is going to appear at depth 4 in any search tree of possible combinations of casesplits on the variables which, with the breadth–first method that was being used, proved too computationally expensive.

The memory failure indicated inefficiency in the implementation of the Evaluate method (for instance a depth–first with iterative deepening strategy would have been more robust). But also suggested that the sort of brute force search through all possible case splits on all variables that is used by the Evaluate method may always cause problems where a large number of these is required. This would suggest that for larger problems it is foreseeable that heuristics would be necessary to guide this search.

Matching of $(\cdots)^N$

[Thomas & Watson 93] observed that, in our setting, $(\cdots)^N$ is somehow different in character from many other functions. This is also highlighted by [Chen *et al* 90]'s need to provide a special matching algorithm to match "normal" terms with their *recurrence terms*, expressions representing repeated function application.

In the proof method for coinduction this task is undertaken by the Fertilize method. In its original state (as for induction) the Fertilize method looks for a uniform instantiation of variables in the induction hypothesis and the conclusion. The addition of expressions involving $(\cdots)^N$ complicates this since an instance of such an expression may not itself explicitly contain $(\cdots)^N$. The Fertilize method was extended during the developmental phase so that it could recognise $f(x)$ as an instance of $f^n(x)$. However it proved incapable of recognising that $s(s(y))$, for instance, was an instance of $s^{s^n(x)}y$. In a couple of proofs this failure led to unnecessary revisions of the trial bisimulation which in turn led to bisimulation explosion. This lack of generality in the method clearly needs to be addressed. The Fertilize method needs to be expanded in a systematic way to cope with the matching of terms involving repeated function application, or a new method needs to be developed to do this task. The first of these options is preferable since the object of the matching is still to exploit the coinduction hypothesis.

7.6.2 Implementational Errors

The remaining errors arose out of deficiencies in the implementation. They are detailed briefly here for the sake of completeness.

Incorrect Generalisation *CoCLAM* incorrectly generalised $H_1 :: H_2 :: T$ to $(:: (H_2 :: T))^N(H_1)$ instead of $(:: H)^N(T)$. This occurred since it failed to keep track of the types of variables in the generalisation step.

A similar problem occurred in one of the development theorems but wasn't originally recognised as such, it being thought that the theorem failed because the bisimulation required would need the synthesis of a new function as in example 6.5 in chapter 6.

The third proof attempt which generalised incorrectly produced the trial bisimulation

$$\{\langle M + s^X(N),\ N + s^X(N)\rangle\}$$

where N appears twice on the RHS of the relation instead of

$$\{\langle M + s^X(N),\ N + s^X(M)\rangle\}$$

when the commutativity of plus appeared as a subgoal of the proof.

This occurred because the variables in the various pair schema describing the trial bisimulation were not standardised apart and this caused substitution to occur incorrectly in the Revise Bisimulation Critic.

Initial Definition of Bisimulation Consider the theorem

$$even(N) \Rightarrow parity(true, L) \sim numparity(N, L)^1$$

The trial bisimulation should be

$$\{\langle parity(true, L),\ numparity(N, L)\rangle \mid even(N)\}$$

The original implementation of the Coinduction method didn't allow for this style of representation with anything except typing conditions appearing after $\mid$ as qualifications on pair schema. It was extended hurriedly towards the end of the development phase in an unsatisfactory way that represented such a set as

$$even(N) \Rightarrow \{\langle parity(true, L),\ numparity(N, L)\rangle\}$$

Although this was sufficient for simple theorems, in this case it interfered with the representation of types and lost the information concerning the types of L.

Substitution Error Consider the theorem

$$tick = map(flip01, tock)$$

where

$$tick \stackrel{\text{red}}{\rightsquigarrow} s(0) :: tock \tag{7.29}$$

$$tock \stackrel{\text{red}}{\rightsquigarrow} 0 :: tick \tag{7.30}$$

$$flip01(s(0)) \stackrel{\text{red}}{\rightsquigarrow} 0 \tag{7.31}$$

$$flip01(0) \stackrel{\text{red}}{\rightsquigarrow} s(0) \tag{7.32}$$

During proof planning *CoCLAM* generated the subgoal

$$\begin{gathered}\forall \mathcal{R} \langle tick,\ map(flip01, tock)\rangle \in \mathcal{R} \Rightarrow \\ \langle tock,\ map(flip01, tick)\rangle \in \mathcal{R}\end{gathered} \tag{7.33}$$

Unfortunately the Revise Bisimulation Critic treated *tick* and *tock* as variables, rather than functions. It found a trivial difference match and proposed a new trial bisimulation

$$\{\langle tick,\ map(flip01, tock)\rangle\}$$

This meant that the proof plan diverged alternating between the trial bisimulations $\{\langle tock,\ map(flip01, tick)\rangle\}$ and $\{\langle tick,\ map(flip01, tock)\rangle\}$.

[1]The definitions of *parity* and *numparity* are unimportant to this discussion but can be found in appendix B

Error in finding Transitions Consider the theorem

$$repl(A, B, lswap(A, B)) \sim lconst(A)$$

where

$$
\begin{aligned}
repl(A, B, nil) &\overset{\text{red}}{\leadsto} nil && (7.34)\\
B \neq H \Rightarrow repl(A, B, H :: T) &\overset{\text{red}}{\leadsto} H :: repl(A, B, T) && (7.35)\\
B = H \Rightarrow repl(A, B, H :: T) &\overset{\text{red}}{\leadsto} A :: repl(A, B, T) && (7.36)
\end{aligned}
$$

and *lconst* and *lswap* are defined as in chapter 3.
The Evaluate method kept adding the case $B \neq A$ as a new case to be explored by reduction (having found a transition for $B = A$) and failed to recognise (after the first addition) that the condition was already in place.

7.7 Potential Problems with the Proof Strategy

While analysis of failures is important for the second aim of the experiment, some weakness in the proof strategy emerged which did not prove fatal on any of the development or test examples. These areas were nevertheless highlighted by the experimental process as requiring further study. In some cases overcoming these weaknesses could lead to greater efficiency in finding a plan.

7.7.1 Choosing Sinks

The experiment highlighted some weaknesses in rippling. Hitherto, no case had been reported where not only was there a choice of wave rules but also a choice of sinks for a wave front to ripple into. When a wave front ripples into a sink the rippling heuristic will not let it ripple out again since this would be a measure increasing step. Hence if a wave front ripples into an inappropriate sink there is a risk that this could prove fatal to the proof strategy.

In contrast to inductive proofs where there are generally very few sinks, in a coinductive proof all variables act as sinks. Moreover in cases where a generalised trial bisimulation was being used the choice of sinks meant there was always a choice of wave rules.

Example 7.1 *The following three wave rules are available to* CoCLAM

$$s^N(\boxed{s(\underline{Y})}^{\uparrow}) \overset{\text{red}}{\leadsto} \boxed{s(\underline{s^N(Y)})}^{\uparrow} \qquad (7.37)$$

$$\boxed{s(\underline{s^N(Y)})}^{\uparrow} \overset{\text{red}}{\leadsto} s^N(\boxed{s(\underline{Y})}^{\downarrow}) \qquad (7.38)$$

$$\boxed{s(\underline{s^N(Y)})}^{\uparrow} \overset{\text{red}}{\leadsto} s^{\boxed{s(\underline{N})}^{\downarrow}}(Y) \qquad (7.39)$$

Consider the expression

$$X + s^N(\boxed{s(\underline{Y})}^{\uparrow})$$

appearing on one side of a pair scheme in a trail bisimulation. (7.37) ripples this to

$$X + \boxed{s(\underline{s^N(Y)})}^{\uparrow}$$

Then there is a choice of (7.38) or (7.39) to ripple to either

$$X + s^{\boxed{s(\underline{N})}^{\downarrow}}(Y) \tag{7.40}$$

or

$$X + s^N(\boxed{s(\underline{Y})}^{\downarrow}) \tag{7.41}$$

In either of these cases rippling has terminated because an inward wave front can not be turned outwards again.

This choice frequently occurs in situations involving $(\cdots)^N$ since one wave rule, rippling differences into N, arises out of the definition of $(\cdots)^N$. The other, rippling differences into X, arises out of the lemma $F^N(F(X)) \rightsquigarrow F(F^N(X))$ this lemma, or a version of it, is often required in order to find the value of the expression. This is discussed in §7.7.2.

This choice causes two problems. Firstly fertilization may apply to one of the expressions but not the other. Backtracking is against the ethos of proof planning with critics, but a critic could be written to check for alternative wave rule applications in the proof. Such a critic would have a similar effect to backtracking across the possible wave rules in search of one that lead to fertilization.

The second problem is that if neither case leads to fertilization then the trial bisimulation needs to be modified and one modification may lead directly to a proof while the other may prompt further revisions. If a "Wave Revision" Critic were to be included alongside the Revise Bisimulation Critic, then care would have to be taken that the Revise Bisimulation Critic was called on a "good" terminal rewrite. This strongly suggests the need for some notion of a normal form for generalised expressions appearing in pair schema to guide this choice of terminal rewrite.

In general, wave rules like (7.40) are preferable to ones like (7.41) since they move the difference away from a variable that has already been generalised once. Y in (7.40) has almost certainly been generalised already because of the $s^N(\cdots)$ term around it – we would rather explore all other possibilities before returning to a second generalisation of Y.

At present *CoCLAM* relies on the wave rules being ordered in such a way as to avoid this problem. A heuristic was chosen during the developmental phase for this ordering. This heuristic always preferred wave rules that rippled into N in the expression $F^N(X)$ over those that rippled into X. This heuristic made an appropriate choice of sink in all the development and test examples.

7.7.2 The Need for Additional Lemmata

Despite the fact that *CLAM* aims to be fully automated, the provision of additional lemmata beyond function definitions is a widespread practice. The Lemma Speculation critic proposed by [Ireland & Bundy 96] was motivated by this observation and was an attempt to provide a way of automatically deriving such lemmata should they appear to be necessary.

It was hoped that the Revise Bisimulation Critic would remove the need for additional lemmata in *CoCLAM*, though at the cost of longer proofs (e.g. the proof of the commutativity of plus discussed in §7.6.1). However this proved not to be the case. Lemmata were found to be necessary in order to deal with $(\cdots)^N$ during the Evaluate method. There was some discussion of this point in chapter 5.

Recall that the proof of example 3.7 in chapter 3 required the lemmata:

$$(map(F))^N(nil) \overset{\text{red}}{\rightsquigarrow} nil \tag{7.42}$$

$$(map(F))^N(H :: T) \overset{\text{red}}{\rightsquigarrow} F^N(H) :: (map(F))^N(T) \tag{7.43}$$

These were needed in order to determine the transitions (i.e. by the Evaluate method).

The development and testing periods revealed that lemmata such as these were required for most proofs where a generalisation occurred. In §5.5.3 in chapter 5 it was observed that these lemmata corresponded to corecursive definitions for F^N for some specific value of F. In general $(\cdots)^N$ doesn't have a corecursive definition and this would appear to be the cause of the problem. It is unsatisfactory to have this need for additional lemmata inherent in the proof strategy. There is a discussion about possible ways round this in chapter 9

7.7.3 Choosing an Appropriate Hypothesis for Difference Matching

In a theorem that requires a bisimulation containing a lot of pair schema there can arise a choice of which pair schema to difference match the current coinduction conclusion against at the ripple phase.

Although this didn't prevent any of the development or test theorems from being planned, it is anticipated that this might be a potential cause of failure in future.

7.8 Non–theorems

It is important to establish that, even though *CoCLAM* comes with no guarantee of soundness (this would be supplied by the object–level theorem prover) it is not known to be unsound. In particular, it does not find plans for all input conjectures.

To test this a couple of non theorems were included at the developmental stages: $\forall x, y : nat.\ x \sim y$ and $\forall l_1, l_2 : list(nat).\ l_1 \sim l_2$ which *CoCLAM* failed to prove. *CoCLAM* also failed to prove two non–theorems that had been devised by myself, under the mistaken impression they were theorems, and included in the test set. These were (7.44) and (7.45).

$$lswap(0, 1) \sim inf\_list(0, flip01, id) \tag{7.44}$$

$$inf\_list(0, flip01, id) \sim inf\_list(1, id, flip01) \tag{7.45}$$

where $lswap$ and $flip01$ are defined as before, id is the identity function and inf_list is defined as:

$$inf\_list(N, H, T) \overset{\text{red}}{\rightsquigarrow} H(N) :: inf\_list(T(N), H, T) \tag{7.46}$$

This failure allowed the hypotheses to be identified as non–theorems when the trace was inspected. (7.44) was altered to $lswap(0, 1) \sim inf\_list(0, id, flip01)$ and so became a theorem. (7.45) didn't appear to be modifiable in this way.

7.8.1 A Disproof Method

Both these theorems and the two theorems used initially to check that *CoCLAM* didn't assign theorem–hood to every statement failed during the checking of transitions. That is, different transitions were found to apply to the two sides of the relation. This contradicts the conditions for bisimilarity. It also suggests that it might be possible to write a Disproof method that could determine certain conjectures to be non–theorems because of a mismatch between transitions. *CoCLAM* currently does not detect non–theorems explicitly: it simply fails to find a proof plan. A Disproof method would mean *CoCLAM* reached a stronger conclusion that the statement was a non–theorem.

Care would have to be taken in those situations in which a generalisation had occurred (because of the Revise Bisimulation critic) before a transition mismatch was discovered in case this mismatch were the result of over–generalisation.

7.9 Quality of the Examples

The major concern with the testing of the proof strategy was the nature of the examples used. While they are representative of the examples available in the literature (for proofs of the equivalence of functional programs), they remain essentially simple problems. The ability of *CoCLAM* to scale up to more realistic problems is an important consideration. The previous comments on its shortcomings relate to this.

The memory problems encountered by the Evaluate method and the problem of bisimulation explosion cast doubts on the ability of *CoCLAM* to scale up to larger problems although there may be simple fixes such as (in the case of the Evaluate method) changing the search strategy to depth first with iterative deepening and (in the case of bisimulation explosion) improving *CoCLAM*'s Wave method.

Perhaps more worryingly, anecdotal evidence from attempts to use coinduction for hardware verification [Collins & Hogg 97] indicates that those cases where new functions have to be synthesised in order to find a bisimulation may be more prevalent than originally thought. This would indicate a real need for more sophisticated generalisation techniques to be developed possibly drawing inspiration from techniques used in Inductive Logic Programming (discussed in chapter 8).

The strategy could be tested more thoroughly by adapting the labelled transition system to that of some of the recently developed semantics for object oriented programming [Jacobs 97] or the cryptographic Spi calculus [Abadi & Gordon 97]. However, it seems unlikely that a significant set of "real" examples will become available until the use of coinduction as a proof technique is more widespread.

7.10 Type Checking

An experiment in extending the system to allow type checking proofs was undertaken. This used the labelled transition system described in §3.7 of chapter 3.

It was checked in a very limited way with the following results

	Number of Theorems	Number of Theorems Proved
Development Set	3	3
Test Set	8	5

This labelled transition system is very similar in character to the one implemented for operational semantics with inductively defined types, in particular the transition rules tend not to have premises and depend upon the expression being reducible to a value in some reduction system. Not much can be drawn from these results, except that it was at least possible to input a new transition system.

7.10.1 Causes of Failure

Of the three theorems that failed one ran into the sort of memory error described in §7.6.1, the other two failed to prove

$$(X :_c nat \wedge F :_c (nat \rightarrow nat)) \Rightarrow \langle F(X),\ \langle F(X),\ nat\rangle\rangle \in \mathcal{R} \cup \sim$$

This is because the hypothesis states that $X \sim \langle X,\ nat\rangle$ and $F \sim \langle F, nat \rightarrow nat\rangle$ from which it can be inferred that $F(X) \sim \langle F(X), nat\rangle$. *CoCLAM* has no mechanism for obtaining this inference. Once again this arises out of shortcomings in the way *CoCLAM* deals with hypotheses. It seems likely that these shortcomings could well affect more theorems in this domain, where information about the types of the terms making up an expression is very important.

7.11 Conclusion

The number of theorems proved was encouraging, especially considering the number that failed for purely implementational reasons. However the lack of "real" examples for testing means that these results, while encouraging, can not be taken as incontrovertible evidence of the efficacy of the proof strategy. This also raises the question of whether a coinductive theorem prover is of any use. Further development and evaluation is needed but the general approach nevertheless appears very promising.

The experiment revealed a number of shortcomings with the Wave and Evaluate methods. The search strategy adopted by the Evaluate method needs to be changed to a more efficient one and thought has to be given to the control of rippling when there are several sinks. It is also necessary to improve the *CoCLAM*s efficiency when dealing with large expressions.

The handling of additional lemmata also needs to be considered. It may well be necessary to try and provide some pre–proof planning method which looks at the functions involved in the problem and tries to generate lemmata for $(\cdots)^N$ in advance.

The proposed strategy from chapter 6 was one of exploration of the transition space and revision using failure information. This is only challenged by the problem of bisimulation explosion. It is impossible to effectively evaluate the seriousness of this problem until a version of *CLAM* which combines a more efficient wave rule mechanism with support for critics is produced.

The experiment also revealed a number of shortcomings with the implementation of the proposed proof strategy, especially with the treatment of variables and substitutions. However, in principle, these were not problems for the strategy itself and in all but one case they were easily corrected.

Chapter 8

Related Work

8.1 Introduction

This chapter discusses work that is related to the development of a proof strategy for coinduction.

This falls into two areas, work on automating coinduction in other theorem proving environments and work on techniques for detecting divergence and speculating generalisations.

8.2 HOL

The HOL system [Gordon 88] [Gordon & Melham 93] is a mechanised proof–assistant for conducting proofs in a version of classical Higher Order Logic [Gordon 85]. This logic is based on Church's formulation of the simple theory of types [Church 40], adapted to allow type variables. Higher order functions are allowed (functions that take functions as arguments or return a functions as their result). It is also possible to quantify over functions.

HOL is a direct descendant of LCF [Gordon *et al* 79]. It supports secure interactive theorem proving by representing its logic in the strongly–typed functional programming language ML [Milner *et al* 90].

8.2.1 Coinduction

Collins [Collins 96] has created a system to support reasoning about lazy functional languages within HOL. This system defines the semantics of the language in an operational style using a labelled transition system and uses applicative bisimulation [Abramsky 90] to define equivalence of programs.

Embedding a Functional Programming Language in HOL

The syntax and semantics of functional languages are embedded in HOL using a framework that is very similar to that discussed in chapter 3, in fact the theories are both based on the work of Gordon [Gordon 95a].

Static semantics are formalised by an inductively defined relation **Type** as was shown in chapter 3, where **Type** $C\ e\ \tau$ means that expression e has type τ in the context C. For instance the rule for function application is

$$\frac{\mathbf{Type}\ C\ e_1\ (\tau_1 \rightarrow \tau_2) \quad \mathbf{Type}\ C\ e_2\ \tau_1}{\mathbf{Type}\ C\ (e_1\ e_2)\ \tau_2} \tag{8.1}$$

A second relation, **Prog**, is also introduced to represent programs that are well–typed according to **Type**. Thus, **Prog** $e\ t$ holds only if e has type t in the empty context.

Small and Large step reduction ($\overset{\text{red}}{\leadsto}$ and $\Downarrow$ from chapter 3) are similarly defined as the relations $\longrightarrow$ and **Eval**. **Eval** is defined in terms of $\longrightarrow$ as $\Downarrow$ was defined in terms of $\overset{\text{red}}{\leadsto}$.

The operational semantics are defined using the relation **LTS** which implements labelled transition systems. **LTS** is a ternary relation and the expression **LTS** $e_1\ e_2\ a$ means that the expression e_1 can make a transition to e_2 with label a.

Bisimilarity is represented by the equivalence relation, ==, which is introduced as the greatest fixedpoint of the function F:

$$\begin{array}{ll} \forall S\ a\ b. & (F\ S)\ a\ b = \\ & (\exists t.\mathbf{Prog}\ t\ a\ \wedge \mathbf{Prog}\ t\ b)\wedge \\ & (\forall a'\ act.\mathbf{LTS}\ a\ a'\ act \supset (\exists b'.(\mathbf{LTS}\ b\ b'\ act) \wedge S\ a'\ b')) \quad \wedge \\ & (\forall b'\ act.\mathbf{LTS}\ b\ b'\ act \supset (\exists a'.(\mathbf{LTS}\ a\ a'\ act) \wedge S\ a'\ b')) \end{array} \tag{8.2}$$

Collins has produced mechanised proofs in HOL that == is an equivalence and congruence. These mirror those in [Gordon 95a].

Basic Tools

Type Checking and Evaluation are prevalent, though low level, tasks in proofs about lazy functional programs and Collins identifies the provision of tools to automate these tasks as a strong requirement. As has been seen in chapters 3 and 5, evaluation to Weak Head Normal Form plays an important role in coinductive proofs. **Type** and **Eval** embody specifications of how to type or evaluate an expression on an abstract machine. Since HOL is built on ML, it is possible to write ML programs that implement these specifications and essentially act as an automatic type checker or evaluator.

Tactics for Coinduction

Collins provides limited support for forming bisimulations and the goals required to prove bisimilarity. The basic coinduction tactic when supplied with a relation $\mathcal{R}$ by the user, proves the first premise of the coinduction rule and forms goals equivalent to those formed by the Gfp Membership method.

This tactic has an associated more specialised tactic `GUESS_CO_INDUCT_TAC` that supplies a simple guess at a bisimulation to the program. This tactic performs the guess in the same way that the Coinduction Method makes its first guess. There are no tactics supplied to perform more sophisticated guesses. If the simple guess is incorrect then the user has to supply a bisimulation by hand.

When forming the Evaluate method as discussed in chapter 5. It was found that two processes had to be combined, reduction and case–splitting. The same experience was encountered when providing tactics for coinduction in HOL. Two tactics are provided `LTS_EVAL_CONV` and `LTS_CASE_CONV`. The first of these performs reduction and the second case splits variables. The user has to guide these tactics specifying when to case split and when to reduce. The tactics perform all the necessary reasoning about types and evaluation.

8.2.2 Non–terminating Programs

Collins' work, like the work reported here, is not equipped to deal with expressions that do not terminate. Assumptions are made that programs will evaluate and hence variables appearing in strict positions are also forced to evaluate by assumption.

8.2.3 Reported Results

Collins reports that the level of interaction required by these tools is similar to a proof on paper.

The tools have been used to investigate hardware verification in Ruby [Sheeran & Jones 90], a relational hardware description language. Ruby doesn't make any assumption about the direction in which data flows through circuits. This is problematic for many formalisations which assume some sort of directionality in order to obtain well–foundedness. Ruby was translated into Haskell so that Ruby specifications could be executed. The lazy properties of Haskell and the associated formal tools supplied by Collins allowed this translation to be performed without forcing any determination of the direction of the data flow. This work provided an encouraging test case for the implementation [Collins & Hogg 97].

8.3 Isabelle

One of the first theorem provers to offer any support for coinduction was Isabelle [Paulson 94a]. Isabelle is an interactive theorem prover which has been used to produce several coinductive proofs. It is a *Generic Theorem Prover.* The intention is that proofs in Isabelle are not tied to any one logic or formal system, but that the user may define his own logics as they may be appropriate, via the use of *theory files.* Isabelle comes with several different logics and packages containing definitions, object–level inference rules and lemmata. The different *object–logics* are defined in the Pure Isabelle *meta–logic* and object–level proofs are built up using meta–level rules.

8.3.1 Coinduction

As mentioned in chapter 2 coinductive definition packages have been implemented in two of Isabelle's object logics, ZF (Zermelo–Fraenkal Set Theory) [Paulson 94b] and HOL [Paulson 93], [Nipkow & Paulson 94]. In both cases the package is combined with a package for inductive definitions based around fixedpoint theory. The HOL implementation is examined here.

The Fixedpoint Types in HOL

Before developing a package, Paulson first had to implement a theory of lazy recursive structures in Isabelle/HOL. Paulson uses a single type, to formalise all recursive data structure definitions (both well–founded and non–well–founded). Recursive datatypes are viewed as sets of (infinite) trees. These trees can be represented as lists of the elements in the tree paired with their position (a numeric list description of the path to that node). The pair of an element and a position is formally defined as a type, α `node`, which is a complex type constructed as follows.

First it is necessary to define the list type that will represent the tree. Let the list $[k_0, \cdots, k_{n-1}]$ denote some function $f : nat \rightarrow nat$ such that

$$f(i) = \begin{cases} s(k_i) & \text{if } 0 \leq i < n; \\ 0 & \text{if } i = n. \end{cases} \tag{8.3}$$

'Consing' an element, a, onto the front of the list f is done by the function $push(a, f)$ which is defined in terms of nat_case (which is a similar to num_case defined in chapter 3).

$$push(a, f) \equiv \lambda i.nat\_case(i, s(a), f)$$

The elements of these lists are the labelled nodes of the tree. A labelled node in a tree is represented by a pair $\langle f,\ x\rangle$ where f stands for a list (as described above) indicating the position of the node in the tree and $x : \alpha + nat$ is a label, where $+$ is cartesian sum. The set of all nodes is defined as:

$$\mathcal{N}ode \equiv \{p \mid \exists f, x, n.\, p = \langle f,\, x\rangle \wedge f(n) = 0\}$$

$\mathcal{N}ode$ is of type $((nat \to nat) \times (\alpha + nat))set$. The type α `node` is the type of nodes taking labels from α. Possibly infinite binary trees represent all the data structures in the theory. The nodes of these trees are of type α `node`. The primitive constructors for these binary trees are *atom* and $(\cdot)$.

$$atom(a) \equiv \{\langle [],\, a\rangle\}$$

$$M \cdot N \equiv \{\langle push(0, f),\, x\rangle\}_{\langle f,\, x\rangle \in M} \cup \{\langle push(1, f),\, x\rangle\}_{\langle f,\, x\rangle \in N}$$

Having defined a type to represent general recursive data structures. It is then necessary to refine it to specific recursive types. The principal such type is lazy lists. In order to describe the type of lists it becomes necessary to define product and sum types on binary trees. Because of the complexity of the type the normal cartesian product and sum can't be used. $\otimes$ and $\oplus$ (similar to cartesian product and disjoint sum), are defined instead:

$$A \otimes B \equiv \bigcup_{x \in A} \cdot \bigcup_{y \in B} .\{x \cdot y\}$$

The sum type is derived from the normal cartesian sum type (with constructors *inl* and *inr*). *atom* has type $(\alpha + nat) \to (\alpha)$ `node` *set* this allows two constructors $leaf : \alpha \to \alpha$ `node` *set* and $numb : nat \to \alpha$ `node` *set*

$$leaf(a) \equiv atom(inl(a))$$

$$numb(k) \equiv atom(inr(k))$$

The "new" disjoint sum can then be derived as follows:

$$\begin{aligned} in0(M) &\equiv numb(0) \cdot M & (8.4)\\ in1(N) &\equiv numb(s(n)) \cdot N & (8.5)\\ A \oplus B &\equiv \{y \mid \exists x \in A.y = in0(x)\} \cup \{y \mid \exists x \in B.y = in1(x)\} & (8.6) \end{aligned}$$

Type assignment is equivalent to set membership (so much so that the two are sometimes used interchangeably).

Lazy Lists

Paulson concentrates exclusively upon list types and defines them in terms of the function $list\_fun$:

$$list\_fun(A) \equiv \lambda Z.1 \ \oplus (A \otimes Z)^1$$

$$list(A) \equiv lfp(list\_fun(A))$$

$$llist(A) \equiv gfp(list\_fun(A))$$

nil and :: are defined from this in terms of membership of 1 or $A \otimes Z$.

$$nil \equiv in0(1)$$

$$M :: N \equiv in1(\langle M,\ N\rangle)$$

These definitions supply all the properties commonly associated with the constructors.

Bisimulations over Lazy Lists

$\oplus$ and $\otimes$ are extended, in the obvious way, to $\oplus_D$ and $\otimes_D$ to act upon relations.

$$diag(A) \equiv \bigcup_{x \in A} \{\langle x, x\rangle\}$$

$$llistD\_fun(r) \equiv \lambda Z.diag(1) \oplus_D (r \otimes_D Z)$$

$$llistD(r) \equiv gfp(llistD\_fun(r))$$

The theorem that expresses list equality (or bisimilarity) is:

$$llistD(diag(A)) = diag(llist(A))$$

Like $llist(A)$, $llistD(diag(A))$ is also treated as a type in Isabelle proofs.

[1] 1 is used here to denote a singleton set of one distinguished element.

Corecursion

Just as definitions of inductive functions are given in terms of structural recursion specialised to list–recursion (or whatever) as required so, too, coinductive definitions are given in terms of corecursion using the function *llist_corec*. This was discussed in chapter 2.

Support for Coinduction in Isabelle

With the preceding theory in place it is possible to perform sound coinductive proofs in Isabelle. The Isabelle–94 release comes with two files supporting coinduction, `LList.thy` and `LList.ML`. `LList.thy` sets up the theory of lazy lists along with corecursive definitions of *map*, *iterates*, *append* and *lconst*. From this file Isabelle's coinductive definitions package derives the coinduction rule for *llist* and $llistD(diag)$.

$$\frac{a \in X \quad X \subseteq list\_fun(A, X)}{a \in llist(A)}$$

$$\frac{\langle a,\ b\rangle \in X \quad X \subseteq llistD\_fun(diag(A), X)}{a \equiv b}$$

`LList.ML` loads a series of standard theorems into Isabelle along with their proofs so that they may be used as derived rules in the proof process. This includes the standard introduction rules for *map* etc. and a theorem, `llist_fun_equalityI`:

$$\frac{f(nil) = g(nil) \qquad \begin{array}{c}\forall x, l.\ \langle f(x :: l),\ g(x :: l)\rangle \subseteq \\ llistD\_fun(rng(\lambda u.\langle f(u),\ g(u)\rangle) \cup rng(\lambda x.\langle x,\ x\rangle))\end{array}}{f(l) = g(l)} \tag{8.7}$$

which behaves similarly to Collins' `GUESS_CO_INDUCT_TAC` and the first heuristic used by the Coinduction method when it is combined with a resolution tactic, in that it speculates an instantiation for $\mathcal{R}$.

8.4 *CoCLAM* Compared to HOL and Isabelle

Both HOL and Isabelle are tactic based theorem provers. They are designed mainly as proof checkers with a user guiding the proof search. Tactics are algorithms for building up strings of inference rules. As a result HOL and Isabelle guarantee a sound proof in the particular logic that is being used. Tactics can be very sophisticated and tactic-based theorem provers can begin to blur into fully automated theorem provers. However it should be stressed that the work on coinduction in HOL and Isabelle was intended to provide tools for a user to perform coinductive proof interactively, whereas the work in *CoCLAM* was always with a view to full automation.

Unlike HOL and Isabelle, *CoCLAM* is not intended to provide soundness and its methods are not strings of inference rules, but specifications of tactics expressed as preconditions and results of their application. *CoCLAM* provides a greater degree of automation than the corresponding versions of HOL and Isabelle. HOL and Isabelle can only provide the simplest of bisimulations automatically and they also require a degree of guidance through the rewriting process. On the other hand *CoCLAM* only provides plans of proofs, it does not provide a record of the inference rules required.

CoCLAM should be regarded as an addition rather than a rival to the work done in tactic based theorem provers such as HOL and Isabelle. Proof planners are intended to link up with tactic based theorem provers, to do the guidance work otherwise done by a user. *CoCLAM*, while not linked up to any specific tactic based theorem prover has been developed with this in mind and is, as far as I'm aware, the only system of this kind.

8.5 Coinduction in Process Algebras

CCS is an example of a Process algebra. Process algebras describe systems comprised of states and the interactions between them. Theorems about systems expressed in process algebras fall into two groups, those which involve expressing the equivalence between two processes (and so are analogous to the problems considered in this thesis) and those which involve proving properties about some process (the property being expressed in some modal/temporal logic). Solving such problems for finite state processes is largely regarded as a solved problem (although there is still work on improving the efficiency of such algorithms). Algorithms for this are generally efficient techniques for exploring the entire state space of the process. Algorithms for proving properties of infinite state systems are still a major research search area. Large classes of such problems have been shown to be decidable. The general case, however, is undecidable.

8.5.1 The Modal Mu–Calculus

Most recent attempts at automating proofs about processes have centred around the use of the Modal Mu-Calculus. The syntax of the calculus is:

$$\Phi ::= Z \mid \neg\Phi \mid \Phi_1 \wedge \Phi_2 \mid [K]\Phi \mid \nu Z.\Phi$$

where Z ranges over propositional variables, and K over subsets of a label set $\mathcal{L}$. The intended meaning of most of the expressions should be obvious. $E \vdash_\Delta [K]\Phi$ means that Φ holds for all states reachable from states in E by a transition in K. ν is the greatest fixpoint operator and $\nu Z.\Phi$ can be interpreted as meaning that "always" Φ holds, i.e. it is true of this node/state and for all its successors. In the formula $\nu Z.\Phi$, $\nu Z.$ binds free occurrences of Z in Φ. Sequents are of the form $E \vdash_\Delta \Phi$ which can be interepreted as meaning that the nodes in E satisfy the formula Φ given the definitions in the list Δ.

8.5.2 Tableaux Proofs

A common approach to proving statements in the modal mu–calculus is to use a tableaux based proof system. Tableaux proof systems are generalisations of model–checking or equivalence checking algorithms. The tableaux rules are roughly backwards natural deduction style rules. A proof tree is built up with terminal nodes indicating either proof or contradiction. Proof rules include, for instance:

$$\frac{E \vdash_\Delta [K]\Phi}{E' \vdash_\Delta \Phi} E' = \{e' | \exists e \in E. e \xrightarrow{K} e'\} \tag{8.8}$$

$$\frac{E \vdash_\Delta \sigma Z.\Phi}{E \vdash_{\Delta'} U} U \notin \Delta \wedge \Delta' = \Delta \cdot (U = \sigma Z.\Phi) \tag{8.9}$$

where σ stands for the greatest fixpoint operator, ν, or its least fixpoint equivalent, μ. An algorithm for tableaux proof builds up a proof tree using the tableaux rules. Nodes are labelled terminal if the success or failure of that node can be determined without extending the tree further.

For instance, a node, $\mathbf{n} = E \vdash_\Delta \Phi$, is terminal if $\Phi = U$, $\Delta(U) = \sigma Z.\Psi$ and $\mathbf{n}$ has an ancestor node, $\mathbf{n}' = E' \vdash_{\Delta'} U$, such that $E \subseteq E'$. The node is successful if $\sigma = \nu$. This corresponds to the situation in coinduction where a loop is detected (We have assumed that U is true for all nodes in E' and for all their successors. Since $E \subseteq E'$, U is clearly true for all nodes in S and all their successors). It is more difficult in tableaux proofs to assess the truth of μ terminals (which are analogous to induction proofs) since they involve establishing well–foundedness conditions. Automated methods based on tableaux proof have been implemented in the CWB and in a prover discussed in a [Bradfield & Stirling 90].

8.5.3 Games

Another approach to proofs in Process Algebras is through game theory. Game theory represents the process of proof as a game played between two players, Abelard and Eloise (or Player and Opponent, Player 1 and Player 2, $\forall$ and $\exists$). Abelard attempts to find a contradiction to the assertion while Eloise tries to find a proof. In our case they are playing a bisimulation game. In the bisimulation game they take turns, Abelard starting. Abelard proposes a transition from one of the pairs in the bisimulation. This move is followed by Eloise matching that transition for the other pair. Abelard wins if Eloise can't move. Eloise wins if the play continues infinitely. The objective of provers using game theory is to show that either Abelard or Eloise has a winning strategy.

Example 8.1 *For example, consider games intended to establish whether the CCS processes in $B = in(x).o\bar{u}t(x).B$ and $C = in(x).\overline{out}(x).C$ are bisimilar. Obviously Abelard has a winning strategy. For example, from the initial position $\langle B, C\rangle$ he could pick the transition $B \stackrel{in(7)}{\rightarrow} \overline{out}(7).B$ meaning that Eloise must pick a $in(7)$ transition. This gives the position $\langle\overline{out}(7).B, \overline{out}(5).C\rangle$. Whichever transition Abelard chooses now, Eloise will be unable to match, so Abelard will win. Of course, had Abelard made the mistake of picking $B \stackrel{in(5)}{\rightarrow} \overline{out}(5).B$, Eloise would have been able to match his move.*

There are an infinite number of games here but Abelard has a winning strategy providing that the first number he selects isn't 5. However it isn't be possible to determine that Abelard has a winning strategy simply by searching through all possible games.

In [Stevens 98] Stevens describes how to define, given a "concrete" game, an associated *set game*, such that there is a correspondence between winning strategies for the two games. The paper presents an algorithm which, when instantiated with some functions describing the concrete game, can search for a winning strategy of the set game. The implementation of this algorithm in CWB is reported to perform better than the tableaux based algorithms implemented in it.

Essentially the set game abstracts away some of the details of the concrete game in a disciplined way which preserves winning strategies. If the algorithm terminates then the set game is a finite description of the infinite concrete game. This can be seen as similar to the way bisimulation generalisation abstracts infinite transition sequences.

Example 8.2 *The set game corresponding to the abstract game in example 8.1 allows the players to postpone decisions about exactly which of a set of plays is being followed. When a player chooses a move, s/he is permitted to restrict the set of plays which should be considered from here on, provided that this set remains non–empty. So Abelard could start with a move $B \stackrel{in(v)}{\rightarrow} \overline{out}(v).B$ simply restricting $v \neq 5$.*

In this example this one move by Abelard encompasses all his winning plays, representing them in a finite manner.

The algorithm requires the user to provide notions of *shape* for processes (e.g. $\{\overline{out}(v).B \mid v \neq 5\}$) and to assign winners to loops of shapes. The algorithm then searches for such loops in an attempts to decide bisimilarity. The idea behind this is similar in motivation to the ideas behind bisimulation generalisation. Shapes characterise an infinite set of processes in a finite manner much as the use of $(\cdots)^N$ characters an infinite set of expressions in a finite manner.

8.5.4 Bisimulation Bases

There are a number of other approaches to proof in Process Algebras for the decidable subsets of infinite state machines.

Burkhart et. al. [Burkhart *et al* 95] provide an algorithm for deciding bisimulations of context–free processes based upon calculating a *bisimulation base*, a rewriting relation which can then be used to decide bisimilarity if two processes rewrite to the same expression.

Their work is based on Basic Process Algebra systems defined as quadruples $(V, \mathcal{L}, \Delta, X_1)$ where V is a finite ordered set of variables, $\{X_1, \cdots, X_n\}$, $\mathcal{L}$ is a set of labels or actions, Δ a finite set of recursive process equations $\Delta = \{X_i \stackrel{\text{def}}{=} E_i \mid 1 \leq i \leq n\}$ where each E_i is an expression of form $E ::= a \mid X \mid E + E \mid E \cdot E$ such that a ranges over $\mathcal{L}$ and X over V, $+$ is the choice operator and $\cdot$ is sequential composition ($\cdot$ is generally omitted when writing expressions down). X_n is called the root.

Expressions in BPA systems can be expressed in K-GNF (Greibach normal form) as a sum of sequences. The *norm* of a process is the shortest sequence of actions that take it to the empty sequence ϵ. A variable is said to be normed if its norm is finite and V can be partitioned into V_N, normed variables, and V_U, unormed ones. A *seminorm* of a sequence αX (where α is a sequence of variables in $V_N^* \cup V_N^* V_U$ and X is a variable in V) is the norm of αX if X has a finite norm otherwise the seminorm of αX equals the norm of α. Seminorms can be used to define a well–founded ordering, $\sqsubseteq$, on $V_N^* \cup V_N^* V_U \times V_N^* \cup V_N^* V_U$.

- A base is a binary relation consisting of pairs $\langle X_i\alpha,\ X_j\beta\rangle$ with $i \leq j$, where X_i and X_j are variables in V and α and β are sequences of variables.

- A base B is called bisimulation–complete iff whether $X_i\alpha \sim X_j\beta$ with $i \leq j$ then one of the following conditions hold:

  1. $\langle X_i\alpha,\ X_j\beta\rangle$ is decomposable, i.e. we have in particular $X_i\gamma \sim X_j$ for some γ, and $\langle X_i\gamma',\ X_j\rangle \in B$ for some $\gamma' \sim \gamma$.
  2. $\langle X_i\alpha,\ X_j\beta\rangle$ is not decomposable and $\langle X_i\alpha',\ X_j\beta'\rangle \in B$ for some $\alpha' \sim \alpha$ and some $\beta` \sim \beta$ such that $(\alpha', \beta') \sqsubseteq (\alpha, \beta)$.

- The relation $\equiv_{\mathcal{B}}$ is defined recursively by:
  1. $\epsilon \equiv_{\mathcal{B}} \epsilon$ and
  2. $X_i\alpha \equiv_{\mathcal{B}} X_j\beta$ iff
     (a) $\langle X_i\gamma,\ X_j\rangle \in B$ and $\alpha \equiv_{\mathcal{B}} \gamma\beta$ or
     (b) $\langle X_i\alpha',\ X_j\beta'\rangle \in B$, $\alpha' \equiv_{\mathcal{B}} \alpha$ and $\beta' \equiv_{\mathcal{B}} \beta$

$\sim \subseteq \equiv_{\mathcal{B}}$ but $\sim$ can be reached from B by computing *subbases*. Given a base B the subbase $\mathcal{R}(B)$ by: $\langle \alpha,\ \beta\rangle \in \mathcal{R}(B)$ iff $\langle \alpha,\ \beta\rangle \in B$ and

1. $\alpha \xrightarrow{a} \alpha'$ implies $\exists \beta'.\beta \xrightarrow{a} \beta' \wedge \alpha' \equiv_{\mathcal{B}} \beta'$
2. $\beta \xrightarrow{a} \beta'$ implies $\exists \alpha'.\alpha \xrightarrow{a} \alpha' \wedge \alpha' \equiv_{\mathcal{B}} \beta'$

Burkhart et. al. present an algorithm for constructing an initial base from a the set of definitions Δ for an infinite state machine. A sequence of subbases can then be constructed from this until a fixedpoint is reached giving a bisimulation base. After this the bisimilarity of processes in the system can be determined by rewriting.

Once again the Bisimulation Base provides a finite description of an infinite relation by abstracting away details to a rewrite rule that can apply to more than one concrete process.

8.6 Inductive Inference

The general problem tackled by the Revise Bisimulation Critic is one of inferring a generalisation from examples. This is a problem from the field of inductive reasoning (inductive used here in the sense of reasoning from experience rather than as a mathematical proof technique).

A typical (though simple) example of an inductive inference problem is presented by Muggleton and De Raedt in their review of inductive logic programming techniques [Muggleton & De Raedt 94].

Example 8.3 *The example is one of inductively inferring (or synthesising) the definition of parent from the following information:*

$$B = \begin{cases} father(X,Z) \wedge parent(Z,Y) \Rightarrow grandfather(X,Y) \\ father(henry, jane) \\ mother(jane, john) \\ mother(jane, alice) \end{cases}$$

Positive and negative examples, E^+ and E^- respectively, concerning the relationship are also supplied:

$$E^+ = \begin{cases} grandfather(henry, john) \\ grandfather(henry, alice) \end{cases}$$

$$E^- = \begin{cases} \neg grandfather(john, henry) \\ \neg grandfather(alice, john) \end{cases}$$

Believing B, and faced with the new facts E^+ and E^- it would be reasonable to guess:

$$H = mother(X, Y) \Rightarrow parent(X, Y)$$

Muggleton and De Raedt observe three things about H

1. *H is not a consequence of B and E^-*
2. *E^+ is a consequence of B and H*
3. *B and H are consistent with E^-*

The object of inductive inference (inductive logic programming (ILP) in this case) is to develop techniques that will derive (even tentatively) a hypothesis, such as H, from the information given which conforms with the three properties noted.

Muggleton and De Raedt distinguish between two processes involved in inductive inference, hypothesis formation (*abduction*) and justification. It is hypothesis formation that is of particular interest to the problem of generalisation formation.

In inductive inference as whole the problem of generalisation formation can be framed as one of search [Mitchell 82] through a variety of rules. Michalski [Michalski 83] identifies a number of such rules, these include:

The Climbing Generalisation Tree Rule When the domain already contains some sort of generalisation tree and the examples are identical except for a conjunct $L = a_i$, where L is some expression and the a_i are all descendants of some node s in the generalisation tree.

This can be generalised by replacing the instances a_i with s. In this way if `triangle` and `square` are defined as instances of `polygon`, then the examples:

$$E^+ = \begin{cases} shape(P) = \texttt{triangle} \\ shape(P) = \texttt{square} \end{cases}$$

can be generalised by $shape(P) = \texttt{polygon}$.

The Turning Constants into Variables Rule If the examples differ only by the appearance of constants in some subterm, then a generalisation can be formed by replacing the constants with a variable.

Inductive Resolution This involves the use of the deductive resolution rule backwards to form expressions that will imply (deductively) the examples. It is discussed in more detail in what follows.

The sort of generalisation that is sought by the Revise Bisimulation critic is similar to use of the climbing generalisation tree rule with the background assumption that $F^N(X)$ is more general than the application of any specific number of Fs to X.

8.6.1 Inductive Logic Programming

The problem faced by the Revise Bisimulation Critic is also similar to that of inducing a recursive logic program.

Logic Programming as a field studies programming languages in which the program is a specification written in logic. There are a number of logic programming languages (which include Prolog) which embody this ideal to a greater or lesser extent. In this section no specific language is assumed. Programs are represented by logical expressions where $\Leftarrow$ is implication and , is conjunction (as opposed to the more common $\wedge$ or &). Universally quantified variables are represented by upper case letters unless stated otherwise (as in the absorption and inter–construction rules).

Definition 8.1 *A* **literal** *is a proposition or a negated proposition.*

A **clause** *is a formula in predicate logic of the form*

$$L_1 \vee \cdots \vee L_m \Leftarrow L_n, \cdots, L_s$$

where the L_i are literals.

Logic programs are sets of clauses. Most logic programs perform deductions based on the information contained in the clauses to reach a solution. This area is *deductive logic programming. Inductive Logic Programming* (ILP) studies languages in which programs can be synthesized from examples and definitions as in an inductive inference problem.

As stated above inductive inference is regarded as a search problem. There is a space of candidate solutions (i.e. the set of "well–formed" hypotheses) and an acceptance criterion. Based on this, the first approaches to ILP were generate and test algorithms, however, as might be expected this rapidly proved to be too computationally expensive to pursue. As an alternative ILP programs now have a number of generalisation and/or specialisation rules. Muggleton and De Raedt present what they term a *generic ILP algorithm* on a queue of candidate hypotheses:

QH := `Initialize`

repeat
    `Delete` H from QH
    `Choose` the inference rules $r_1, \cdots, r_k \in \mathtt{R}$ to be applied to H.
    Apply the rules $r_1, \cdots, r_k$ to H to yield $H_1, H_2, \cdots, H_n$
    Add $H_1, H_2, \cdots, H_n$ to QH
    `Prune` QH
until `stop-criterion`(QH) satisfied

This algorithm repeatedly deletes a hypothesis H from the queue, expands it using inference rules and then adds the expanded hypotheses back to the queue.

They identify "specific–to–general" and "general–to–specific" ILP systems. The first of these start from the background knowledge and the examples and have rules which generalise these, while the "general–to–specific" systems start with the most general hypothesis (the inconsistent clause) and repeatedly specialise. By these criterion *CoCLAM* can be viewed as a very specialised example of a specific–to–general inductive system. Most specific–to–general systems employ inductive or *inverted resolution.*

Inductive inference can be viewed as the inverse of deduction. Resolution (8.10) is a complete rule for deduction and as a result is a fundamental tool in deductive logic programming.

$$\frac{\begin{array}{c} P_1 \vee \cdots \vee P_n \Leftarrow Q_1, \cdots, Q_m, \\ R_1 \vee \cdots \vee R_k \Leftarrow S_1, \cdots, S_l \end{array}}{\begin{array}{c} (P_1 \vee \cdots \vee P_n \vee R_1 \cdots \vee R_{i-1} \vee R_{j+1} \vee \cdots R_K \Leftarrow \\ Q_1, \cdots, Q_{g-1}, Q_{h+1}, \cdots, Q_m, S_1, \cdots, S_l)\sigma \end{array}} \tag{8.10}$$

where σ is the most general unifier of $Q_g, \cdots, Q_h$ and $R_i, \cdots R_j$.

Deductive logic programming works in a similar way to backwards proof in theorem proving by setting up a goal which it subsequently attempts to break down to axioms. The general resolution rule given above can be specialised into a number of rules depending on the domain (e.g. Horn clauses – clauses with only one disjunct on the left).

ILP systems use inverted resolution (i.e. with the premises in place of the conclusion and vice versa) to create statements out of axioms and examples. Alternatively this can be viewed as forward chaining with the rule rather than backward chaining. An example of an inverted resolution rule is (8.11), Absorption:

$$\frac{q \Leftarrow A \quad p \Leftarrow A, B}{q \Leftarrow A \quad p \Leftarrow q, B} \tag{8.11}$$

where lower case letters represent single literals and upper case letters conjunctions of literals. The absorption rule could be used to form the sorts of generalisation produced by the Revise Bisimulation critic. Since B can be found by difference matching A and A, B.

The absorption rule is limited in similar ways to the Revise Bisimulation Critic. Absorption can't find a sufficiently general description for example 6.5 in chapter 6, for instance, which required the invention of a new function. However, there are inverted resolution rules which do allow the invention of new predicates. *Inter–construction*, (8.12) is one of these.

$$\frac{p \Leftarrow A, B \qquad\qquad q \Leftarrow A, C}{p \Leftarrow r, A, B \quad r \Leftarrow A \quad q \Leftarrow r, A, C} \tag{8.12}$$

where lower case letter denote single literals and upper case letters conjunctions of literals.

Notice that Inter–construction introduces a new predicate r. However while absorption simply required a match of the body of one predicate into the body of another. Inter–construction requires a match between some subset of the literals in the predicate bodies. This will increase the number of possible instantiations of the rule's premises and hence introduce a considerable element of search.

Since it is possible to view the Revise Bisimulation Critic as the absorption rule with difference matching used to determine the matches between clauses. It it possible that inter–construction could be adapted for use in a divergence check for coinduction (see chapter 9).

8.7 Divergence

The Revise Bisimulation critic used in this thesis is based on one developed for implicit induction. Implicit induction attempts to create confluent rewriting systems from the theorem and definitions. A major technique for creating such confluent systems is Knuth–Bendix Completion [Knuth & Bendix 70]. The Knuth–Bendix completion procedure generates a confluent set of rewrite rules by repeatedly superposing left hand sides of rewrite rules and adding any generated critical pairs as new rewrite rules. This process may fail to terminate. In effect it is producing an infinite (or divergent) set of rewrite rules.

The algorithm takes a set of rules, R, and within certain restrictions generates new rules of the form $l \rightsquigarrow r$, where $l = r$ in some equational theory. These rules are added to R and the process iterates. The object is to produce a confluent term rewriting system. There are three possible outcomes of the algorithm:

- The algorithm terminates with success, in which case a finite, confluent and terminating set of rules is output.
- The algorithm terminates with failure.
- The algorithm *diverges*, i.e. it fails to terminate, in which case the set of rules being derived is infinite.

Several generalisation techniques have also been developed to overcome the problem of infinite rule sets.

8.7.1 Overcoming Divergence

Thomas and Jantke [Thomas & Jantke 89] present two generalisation algorithms (or more specifically an algorithm and a semi–algorithm), which they describe as standard algorithms from inductive inference. The aim is to replace the infinite sequence of rewrite rules by a finite sequence of rules that are equivalent in some sense. This is an *enrichment* of the term rewriting system, R. This enrichment needs to be confluent and terminating and preserve the equational theory of R. It may be based on a larger signature (containing more operators or sorts).

The algorithms seeks out "smallest distinct terms". This is equivalent to identifying a witness for B in the absorption rule:

$$\frac{q \Leftarrow A \quad p \Leftarrow A, B}{q \Leftarrow A \quad p \Leftarrow q, B}$$

Smallest distinct terms are identified by their position in the expression viewed as a term tree (as discussed in chapter 4). Once found, the first algorithm replaces the subterms at these positions with a variable.

Example 8.4 *Assume the Knuth–Bendix procedure has generated the sequence of rewrites*

$$\begin{aligned} f(h(s(0), g(f(x)))) &\rightsquigarrow f(h(s(s(0)), x)) && (8.13)\\ f(h(s(s(0)), g(f(x)))) &\rightsquigarrow f(h(s(s(s(0))), x)) && (8.14)\\ f(h(s(s(s(0))), g(f(x)))) &\rightsquigarrow f(h(s(s(s(s(0)))), x)) && (8.15) \end{aligned}$$

The algorithm looks at the LHS of each rewrite to determine the first position where distinct terms appear.

The first distinct terms on the LHS of (8.13) and (8.14) occur at position 1.1.1 and are the subterms 0 *and* $s(0)$ *respectively. The algorithm replaces these by a variable* y*, occurrences of the subterms are then searched for on the RHS and where they occur in the same position in both terms they are again replaced by* y*. This creates a generalisation of the terms. This generalisation is Plotkin's least general generalisation [Plotkin 71].*

This gives a final rule

$$f(h(s(y), g(f(x)))) \rightsquigarrow f(h(s(s(y)), x)) \qquad (8.16)$$

The second (semi–)algorithm, called only if the first fails, searches the lexicographic order of terms in the language involving the variables used in the rewrites, for a term that rewrites to the distinct subterms.

Example 8.5 *Assume the procedure has generated the following set of rules*

$$f(h(s(0), g(h(y,x)))) \rightsquigarrow f(h(s(s(y)), x)) \tag{8.17}$$
$$f(h(s(s(0)), g(h(y,x)))) \rightsquigarrow f(h(s(s(s(y))), x)) \tag{8.18}$$
$$f(h(s(s(s(0))), g(h(y,x)))) \rightsquigarrow f(h(s(s(s(s(y)))), x)) \tag{8.19}$$

Following the same process on the LHS of the rewrites as the first algorithm, the second algorithm generates the term $f(h(s(z), g(h(y,x))))$. However the algorithm fails to find the subterms reappearing on the right since they do not involve 0*, the constant that has been generalised, and no analysis of distinct terms is performed for the expressions on the right. Hence the RHS of the rewrites remain the same.*

The second algorithm then searches through the lexicographic ordering of terms in the language involving the variables x, y and z for some term, t for which $t[0/z] \rightsquigarrow f(h(s(s(y)), x))$ and $t[s(0)/z] \rightsquigarrow f(h(s(s(s(y))), x))$. This search produces the solution $f(h(s(s(z+y)), x))$ and so the generated rewrite rule is

$$f(h(s(z), g(h(y,x)))) \rightsquigarrow f(h(s(s(z+y)), x)) \tag{8.20}$$

The first of these algorithms generalises only a subset of the sort of expressions the Revise Bisimulation Critic can handle (i.e. those where terms can be generalised by a variable). Both of Thomas and Jantke's algorithms are operating within a predefined sorted language and so don't allow new functions (i.e. $(\cdots)^N$) to be introduced, so for instance they would be unable to find the generalisation required for example 3.7. The first is restricted to cases where divergence can be avoided by replacing a constant with a variable.

The second has the ability to examine the set of pre–defined functions and this may be of use in cases where a new function needs to be introduced, although it doesn't have the ability to synthesize such a function. The second algorithm may also fail to terminate since there are infinite terms in the language and a pre–defined function which can provide a solution may not exist. As a result some sort of heuristic would be required to bound the search if it were to be used as part of a suite of generalisation techniques.

8.7.2 Repeated Function Application

Thomas and Watson [Thomas & Watson 93] improved upon the Thomas/Jantke method to introduce generalisations involving repeated function application. New sorts are introduced into the language which contain precisely terms of the form $g^n(f(x))$ (for instance). Expressions which contain instances of this sort e.g. $g(g(g(f(x))))$ can then be generalised by replacing the sort instance with a variable y of that new sort. The subterms that need to be generalised are identified once again by comparing rules and determining the positions of distinct subterms. These positions are called *varying positions*. There are a number of restrictions placed on the choice of subterms,

- The same subterm must appear at all the identified positions in some rule.
- All the divergent sets of rules are identical except at or below varying positions.
- No variable occurring at or below a varying position can occur elsewhere in the rule.
- Any variable occurring at or below a varying position may occur at most once below that position.

These restrictions limit the generalised rewrites that may be produced since at most one variable of at most one new sort may be introduced. Any rewrite requiring more than one new variable or sort would be excluded. It would also exclude rewrites where a variable needed to be generalised at one position but left ungeneralised at another.

8.7.3 Recurrence Terms

Chen et al. [Chen *et al* 90] develop an alternative method for detecting and representing the kind of generalisation employed by the Revise Bisimulation critic, which they call recurrence terms.

Note that in what follows t/p denotes the subterm at position p of t and $t[p \leftarrow s]$ denotes the term after replacing t/p by s.

Definition 8.2 *The set of* **recurrence terms** *is defined as the inductive closure of the following:*

1. *Every variable in $\mathcal{V}$ (the set of variables in the signature) is a recurrence–term.*
2. *If a function f is of arity n and $t_1, \ldots, t_n$ are recurrence terms, then $f(t_1, \cdots, t_n)$ is a recurrence–term.*
3. *If h is a term, p is a non–root position of h, N is a natural number (called the degree variable) and l is a recurrence–term, then $\Phi(h[p \leftarrow \diamond], N, l)$ is a recurrence–term, where $\diamond$ is a special symbol serving as a place holder.*

The unfolding of the recurrence terms containing place holders is as follows:

$$\Phi(h[p \leftarrow \diamond], 0, l) \stackrel{\text{def}}{=} l \tag{8.21}$$

$$\Phi(h[p \leftarrow \diamond], n+1, l) \stackrel{\text{def}}{=} h\sigma[p \leftarrow \Phi(h[p \leftarrow \diamond], n, l)] \tag{8.22}$$

where σ renames the variables in $h[p \leftarrow \diamond]$ into new variables. Hence recurrence terms are a way of coding $(\cdots)^N$ in a first order fashion ($f^n(c) \equiv \Phi(f(x)[1 \leftarrow \diamond], n, c)$)

Definition 8.3 *A binary relation on T_Σ is a* **homeomorphic embedding relation**, $\trianglelefteq^h$ *if:*

$$s = f(s_1, \cdots, s_n) \trianglelefteq^h g(t_1, \cdots, t_m) = t$$

if and only if

- $f = g$ *and* $s_1 \trianglelefteq^h t_{j_i}$, $\forall i, 1 \leq i \leq n$, *where* $1 \leq j_1 < j_2 < \cdots < j_n \leq m$, *or*
- $s \trianglelefteq^h t_j$ *for some* $j, 1 \leq j \leq m$.

Definition 8.4 *The extension of* $\trianglelefteq^h$ *to* $T_\Sigma(\mathcal{V})$, *called* **homeomorphic variable embedding relation**, $\trianglelefteq^{hv}$, *is defined as* $s \trianglelefteq^{hv} t$ *if and only if* $s \trianglelefteq^h t$ *when the variables in* s, $\mathcal{V}(s)$, *and* t, $\mathcal{V}(t)$ *are assumed to be new constants, this will define a new sort,* Σ', *and* $\trianglelefteq^h$ *is defined on* $\mathcal{T}_{\Sigma'}$.

Examples: $f(a, b) \trianglelefteq^h g(f(g(a), b))$, $f(a, b) \not\trianglelefteq^h f(g(b), a)$, $f(x) \trianglelefteq^{hv} f(f(x))$, $f(x) \not\trianglelefteq^{hv} f(f(y))$. These relations allow recurrence terms to be generated from two terms s and t in a nondeterministic way. Notice that if $s \trianglelefteq^h t$ then there is a difference match (chapter 4) that annotates s with respect to t [Smaill & Green 96].

The operation $cgen(s, t, N)$ is the equivalent of the Revise Bisimulation critic in this situation. It generates recurrence terms and combines the process of difference matching and generalisation formation into one.

Definition 8.5 *The operation* $cgen(s, t, N)$ *is equal to*

- t *if* $s = t$, *or*
- $f(cgen(s_1, t_1, N), \cdots, cgen(s_m, t_m, N))$ *if* $s = f(s_1, \cdots, s_m)$ *and* $t = f(t_1, \cdots, t_m)$, *or*
- $\Phi(t[p \leftarrow \diamond], N, cgen(s, t/p, N))$ *if* $s \trianglelefteq^{hv} t/p$

Sometimes no recurrence–term can be generated from two terms, in particular if every $f \in F$ has a unique arity then $cgen(s, t, N)$ if and only if $s \trianglelefteq^{hv} t$.

Chen et. al point out that their recurrence–terms are limited since their first argument can not be a proper recurrence–term and the degree variable is linear, i.e. they cannot accurately schematize the sequence:

$$\begin{gathered} f(x) \\ h(h(f(x))) \\ h(h(h(h(f(x))))) \\ \vdots \end{gathered}$$

they suggest a solution to this second problem of allowing N to be an expression (e.g. $2N$ making the above sequence $\Phi(h(x)[1 \leftarrow \diamond], 2N, f(x))$, but provide no algorithm for how this extended form could be generated. The Revise Bisimulation critic can cope with these sorts of sequences by using $(h \circ h)^N$ as the generalised term. It might be interesting to see, however, whether some method of transforming this to h^{2N} made any difference to the performance of *CoCLAM*.

They also supply a recurrence matching algorithm which matches a term with a recurrence–term. This is important for detecting when some expression is an instance of a recurrence term. Similarly for the coinduction proof strategy the Fertilize method had to be extended to recognise and match instances of $(\cdots)^N$ (as discussed in chapter 7). Since this extension of the Fertilize method was not entirely satisfactory Chen et. al's algorithm might be adapted to replace it.

8.8 Walsh's Divergence Critic

Walsh's [Walsh 96] divergence critic, on which the divergence check in the Revise Bisimulation method is based, was designed to work with an implicit induction theorem prover called SPIKE [Bouhoula & Rusinowitch 93].

Induction is performed in SPIKE by means of test sets (finite descriptions of the initial model). SPIKE attempts to instantiate induction variables in the conjecture to be proved with members of the test set and then to use rewriting to simplify the resulting expressions. The idea is to show that the expressions form a confluent set of rewrite rules. The process of generate and simplify is basically Knuth–Bendix completion and so often produces a divergent set of equations. It has been observed that this happens if an appropriate generalisation or lemma isn't present.

Example 8.6 *Walsh reports on SPIKE's attempts to prove the theorem*

$$length(append(a, b)) = length(append(b, a)) \tag{8.23}$$

where *length* and *append* are defined by the equations:

$$\begin{aligned} length(nil) &= nil && (8.24)\\ length(H :: T) &= s(length(T)) && (8.25) \end{aligned}$$

$$\begin{aligned} append(nil, L) &= L && (8.26)\\ append(H :: T, L) &= H :: append(T, L) && (8.27) \end{aligned}$$

SPIKE applies a `generate` rule that instantiates the induction variables, a, b, with members of the test set $\{nil, h :: t\}$. This produces 3 distinct equations which are rewritten by `simplify` rules using the definitions of *length* and *append*. These give a simple identity and the following two equations:

$$length(append(b, nil)) = s(length(b))$$

$$length(append(a, d :: b)) = length(append(b, c :: a))$$

SPIKE then repeats the process generating the equations:

$$length(append(b, c :: nil)) = s(s(length(b)))$$

$$length(append(a, f :: d :: b)) = length(append(b, e :: c :: a))$$

This process is repeated and the proof attempt diverges.

Walsh's critic partitions the equations using a parentage heuristic into two sequences and then applies difference matching to the sequences as described in chapter 6.

$$\begin{array}{c} length(append(b, nil)) = s(length(b)) \\ length(append(b, \boxed{c :: \underline{nil}})) = \boxed{s(\underline{s(length(b))})} \end{array}$$

$$\begin{array}{c} length(append(a, b)) = length(append(b, a)) \\ length(append(a, \boxed{d :: \underline{b}})) = length(append(b, \boxed{c :: \underline{a}})) \\ length(append(a, \boxed{f :: \underline{d :: b}})) = length(append(b, \boxed{e :: \underline{c :: a}})) \end{array}$$

Note that the first of these sequences is generated from a base case, while the second is generated from a step case.

Instead of speculating generalisations, however, Walsh's critic, at this point speculates a lemma, in this case:

$$length(append(a, d :: b)) = s(length(append(a, b)))$$

These lemmata are speculated using the pattern formed by the difference matching on the LHS of the equations and two heuristics, *cancellation* and *petering out* to instantiate the variable F in figure 8–1 (which gives the critic).

Cancellation uses difference matching to identify term structure accumulating on the RHS of the sequence which would allow cancellation to occur. Failing that, it looks for suitable term structure to cancel against in a new sequence (e.g. the sequence generated by the base case).

In the example, the critic is attempting to generate a lemma from the second sequence. To do this it refers to the RHS of the first sequence to provide the additional structure. The successor functions accumulating at the top of this sequence suggest that F be instantiated to $\lambda x.s(x)$. Thus the cancellation heuristic suggests the lemma,

$$length(append(a, f :: b)) = s(length(append(a, b)))$$

The petering out heuristic uses regular matching to identify cases where the the differences on one side of the sequence have disappeared. In these cases F is instantiated to $\lambda x.x$.

Preconditions:

1. There is a sequence of equations $s_i = t_i$ to which the `generate` rule is applied $(i = 0, 1, \cdots)$.
2. There exists (non trivial) G, H such that for each j, difference matching gives $s_j = G(U_j)$ and $s_{j+1} = G(\boxed{H(\underline{U_j})})$.

Postconditions:

1. The critic proposes a lemma of the form,

$$G(\boxed{H(\underline{U_0})}) = G(U_0, \boxed{F(\underline{G(U_0)})})$$

2. F is instantiated by the cancellation or petering out heuristics;
3. Lemmas are filtered through a conjecture disprover;
4. If several lemmas are suggested, the critic deletes any that are subsumed.

Figure 8–1: Walsh's Divergence Critic

Walsh notes that the proof of an insufficiently general lemma can also diverge and his critic attempts to generalise the lemma whilst filtering it through a conjecture disprover in an attempt to prevent over–generalisation.

Walsh has similar heuristics to speculate lemmata representing transverse wave rules and these are potentially more relevant to coinduction since they speculate lemmas that effectively move the difference around within the term.

8.8.1 Lemma Speculation vs. Generalisation

Walsh's critic concentrates on differences created on the RHS of an equality expression only using the difference matching on the left to instantiate some function, F, in a speculated lemma.

In some cases of coinductive proof the speculation of a lemma would lead to shorter and more elegant proofs (e.g. example 10 in chapter 3 used lemmata). However the heuristics supplied by Walsh for this speculation would not be sufficient, since, for instance, cancellation isn't always a valid simplification method in coinduction.

More importantly, lemma speculation will not always be sufficient since sometimes generalisations are genuinely required. If lemma speculation were to be included as a critic for coinduction then heuristics of some sort would be required to distinguish between situations where lemma speculation is appropriate and those where generalisation is appropriate.

8.9 Conclusion

This chapter covered two distinct topics. The implementation of coinduction in other theorem provers and generalisation procedures.

A number of theorem provers support coinduction. The use of coinduction in Isabelle and HOL was examined in some detail here. Neither of these support fully automated proofs. In HOL the focus has been on providing support for the operational semantics of functional languages, using applicative bisimulation as a notion of equality. Even though special tactics have been developed in HOL they are able to find only the simplest of bisimulations automatically. They also need to be guided through the simplification process for many proofs. This is also the case for Isabelle. *CoCLAM* has a different approach to both these systems, since it attempts to *plan* rather than *prove* a theorem. As such it doesn't offer the guarantees of soundness given by HOL and Isabelle, however this wasn't the object of the work that has been done. Proof planning is intended to provide guidance to tactic based theorem provers, such as HOL and Isabelle. *CoCLAM*, if connected to one of these systems should be able to provide the user with considerably more automation than is offered at present. In many ways the work on *CoCLAM* is complementary to the work on HOL and Isabelle.

Work on proving bisimilarity (and other properties of infinite systems) in process algebras was also surveyed. While process algebras provide a different setting from that presented in this thesis it could be seen that ideas similar to bisimulation extension and generalisation existed there. In particular they demonstrated that the provision and discovery of finite representations for infinite sets/sequences of states is a major requirement.

Generalisation procedures can be seen as special cases of inductive inference. The absorption rule (8.11) involves the exploitation of differences between terms, and the idea that one term is somehow embedded in another. The same ideas are used in Thomas and Jantke's "smallest distinct positions" and Thomas and Watson's "varying positions". The homeomorphic embeddings and recurrence term generator of Chen et. al. also seem to involve these ideas although they don't explicitly draw their inspiration from the field of inductive inference. Walsh linked the detection of distinct terms to difference matching and was able to use the information given by the detection of wave fronts and holes in forming generalisations. The work in this thesis is an adaptation of Walsh's work. The Revise Bisimulation critic uses the same preconditions as Walsh's Divergence Critic. However the post–conditions are new. It's generalisation technique, given the information received from difference matching is also new. Although it has many similarities with the techniques used by Thomas and Watson it can deal with a wider class of problems. As far as I'm aware this is the first time such techniques have been suggested for use in the automation of coinduction.

Chapter 9

Further Work

9.1 Introduction

Several extensions to the work presented in this thesis have already been discussed in previous chapters. Chapter 7 suggested:

- Improving the matching in the Fertilize method.
- Investigating a normal form for pair schema.
- Introducing lemma speculation for corecursive definitions.
- Improving the efficiency of dealing with large bisimulations.

Chapter 8 suggested:

- Using techniques, particularly inter–construction, from inductive logic programming to improve the Revise Bisimulation Critic.

This chapter will look at the speculation of corecursive definitions and the use of inter–construction in more detail and at extending *CoCLAM* to a wider variety of labelled transition systems and linking it to an object–level theorem prover.

9.2 Lemma Speculation and Divergence Analysis

Chapter 7 highlighted some deficiencies in the proof strategy. These were discussed at the time and improvements suggested. One of these improvements was to provide some automatic method of generating corecursive equivalents for function definitions (henceforth called *corecursive lemmata*). This is linked to the issue of functions whose evaluation diverges without the provision of additional lemmata.

9.2.1 Lemma Speculation

Given any recursively defined function, it is desirable to have lemmata available that will act like a corecursive definition of that function, i.e. lemmata that allow the function to be evaluated to a value for any given input. We want all expressions to be values in order to determine transitions. This is different from the situation in induction where theorems can be proved by "sinking" the differences between the induction hypothesis and conclusion and then using the induction hypothesis as a rewrite rule to complete the proof by rewriting the conclusion to an identity. In these cases cancellation (which would appear to be the equivalent process to taking transitions (see chapter 10)) may not have taken place.

Example 9.1 *Consider* $<>$*, append. The usual definition of append is:*

$$nil <> L \stackrel{\text{red}}{\leadsto} L \tag{9.1}$$
$$H :: T <> L \stackrel{\text{red}}{\leadsto} H :: (T <> L) \tag{9.2}$$

The corecursive definition is

$$nil <> nil \stackrel{\text{red}}{\leadsto} nil \tag{9.3}$$
$$nil <> H :: T \stackrel{\text{red}}{\leadsto} H :: (nil <> T) \tag{9.4}$$
$$H :: T <> L \stackrel{\text{red}}{\leadsto} H :: (T <> L) \tag{9.5}$$

In practice it is possible to evaluate $M <> N$ *to a value using either definition, so in this case the provision of corecursive lemmata isn't necessary. However they* are *more suitable, in a general sense, for coinduction since, for instance, they specify more accurately the number of casesplits that will be required to find a value.*

Such lemmata become necessary in situations where a function has no corecursive definition (or, more accurately, cannot be expressed with $unfold$). F^N was one such function. However given particular instantiations of F (e.g. $(map(F))^N$) it is possible to supply a corecursive definition provided that N is assumed to be finite[1]. Since F^N is fundamental to the Revise Bisimulation Critic, lemmata for finite N should clearly be calculated for F^N for all possible F in the function space otherwise evaluation will diverge (as was discussed in chapter 5).

The Lemma Speculation Critic of Ireland and Bundy [Ireland & Bundy 96], generates new lemmata when rippling becomes blocked. It is possible that something similar could be developed for *CoCLAM* to be used either *before* it has started attempting to plan a proof, or as a critic on the Evaluate method. That is, if the Evaluate method has failed or is diverging then the critic could be used to speculate a new lemma.

[1]Recall that integers are treated in a lazy (infinite) fashion elsewhere in this thesis.

Ireland and Bundy's Critic uses middle–out reasoning (described in chapter 4) for this task. The critic divides into two processes, *lemma calculation* and *lemma speculation*. Both these processes work by examining the skeleton of an expression in which rippling is blocked and using that skeleton and, in the case of lemma speculation, middle–out reasoning to find a new wave rule.

9.2.2 A New Critic

Given a function $F : \sigma \to \tau$ we want to provide corecursive lemmata (if they do not already exist) such that:

$$F(v_\tau) \rightsquigarrow v_\sigma$$

$$F^N(v_\tau) \rightsquigarrow v_\sigma$$

where v_τ and v_σ are values of type τ and σ respectively[2].

It may be possible to annotate such lemmata as outward Wave rules. Generating the LHS using meta–variables and potential wave fronts as done by [Ireland & Bundy 96]. So, for instance given $map(F)$ the critic might generate the following:

$$map(F)^N(\boxed{H :: \underline{T}}^{\uparrow}) \rightsquigarrow \boxed{F_1(F,H,T) :: \boxed{F_2(\underline{map(F)^N(T)})}^{\uparrow}}^{\uparrow}$$

However it isn't clear how instantiation of the meta–variables, F_1 and F_2, in this expression is to take place.

If this were done during the Evaluation process more information would be available. In this way attempts to find the lemma could be joined with attempts to prove the theorem. Specifically the results of the transitions on the other side of the relation could be inspected. If the results of transitions are assumed to match then they can be used to coerce the instantiation of the meta–variables.

Ideally though, since "Problematic" functions (those whose supplied definition is not corecursive) can be identified at the start of the process it is tempting to try and speculate lemmata in advance as this would save interrupting the proof process with the use of a critic. It might be possible to create such lemmata by exploitation of the function's recursive definition (which we are assuming is the definition supplied to *CoCLAM*) or examining the values of the function for specific inputs and attempting to generalise from that. This whole area would need to be investigated further to decide on appropriate techniques for speculating these lemmata and the appropriate points in the proofs at which to attempt it.

[2]Recall that in the systems we are considering values are more like patterns than is usual.

A further problem would be trying to identify functions (like $flatten$) which apparently do not have a corecursive definition or time will be wasted searching for one. It might be possible to include some variation of strictness analysis techniques into the critic in order to identify the possibility of divergence.

9.2.3 Divergence Analysis

This leads on to a second extension of *CoCLAM* which would be to implement procedures to recognise that evaluation is diverging and that there are no transitions from the expression. It seems likely that divergence recognition techniques will be relevant here. In the literature examples which involve divergence, such as example 3.5 in chapter 3, have a distinctive pattern of case splits. *CoCLAM* would be required to recognise such a pattern and that divergence occurred for the same inputs on both sides of the relations.

Divergence can be caused by a missing corecursive lemma. Recall example 3.7 from chapter 3. In that example evaluation of the expression

$$(map(F))^N(h(F, X))$$

diverges unless the lemma

$$map(F)^N(H :: T) \leadsto F^N(H) :: map(F)^N(T)$$

is present. This was discussed in chapter 5. This suggests that divergence analysis and lemma speculation may well go hand in hand. Either there are certain values for which a function diverges or a corecursive definition is possible. It might be possible to develop a process which attempts to find the set of values for which a function diverges and which in the process would develop a corecursive definition if such values didn't exist. This would involve a process like the Evaluate method which would replace variables with values e.g.

$$(map(F))^{s(N)}(L)$$

and then reduce them:

$$map(F, (map(F))^N(L)$$

If the reduced expression were not a value then attempts would be made to casesplit other variables in the hope that this would reduce to a value. If the original variable has to be split further then divergence patterns can be sought and the values that cause these patterns specified. In the example above the expression diverges if N is infinite[3]. If N is finite then evaluation will eventually terminate and a lemma could be speculated from observing this process in some way. Clearly this is only a vague outline of the process and further work would be needed to specify it more exactly and evaluate its effectiveness.

[3] As it could effectively be, if integers are being treated in a lazy fashion

9.3 Inter–construction

In chapter 8 a similarity was drawn between the absorption inverted resolution rule and the critic currently being used in *CoCLAM*. Another inverted resolution rule, inter–construction, was also looked at:

$$\frac{p \Leftarrow A, B \qquad\qquad q \Leftarrow A, C}{p \Leftarrow r, A, B \quad r \Leftarrow A \quad q \Leftarrow r, A, C} \tag{9.6}$$

where lower case letter denote single literals and upper case letters conjunctions of literals.

This is of interest because it introduces a new literal r, such that $r \Leftarrow A$. One of the desirable extensions for the revise bisimulation critic would be an ability to synthesize new functions (this was discussed in chapter 6). It is possible that the inter–construction rule could be adapted to an algorithm for performing this task within the critics setting.

An important consideration would be the identification of A. In the absorption rule:

$$\frac{q \Leftarrow A \quad p \Leftarrow A, B}{q \Leftarrow A \quad p \Leftarrow q, B} \tag{9.7}$$

A is fairly easy to identify $q \Leftarrow A$ and then B is identified by being the difference between A and A, B. Once transported into the functional setting this is performed by difference matching.

In inter–construction A is some set of literals shared by the RHSs of $p \Leftarrow A, B$ and $q \Leftarrow A, C$; however A, B is not necessarily embedded in A, C. Instead of difference matching we will need to *difference unify* the two expressions. Just as normal unification has a larger search space than matching so difference unification can be expected to have a large search space than difference matching. At present in *CoCLAM* the Revise Bisimulation critic doesn't really have a problem with the possibility of there being several difference matches since this occurs rarely – however if it were to be extended to difference unification as part of a process of allowing the synthesis of new function then it is possible that this would become a more serious problem.

9.4 Other Labelled Transition Systems

It is desirable that a user of *CoCLAM* should be able to "plug and play" with various labelled transition systems. At present only a very limited number of such systems can be used because of the assumptions made about the transition rules etc. There are a number of extensions that could be made that would allow a wider range of labelled transition systems.

9.4.1 Transition Rules with Non–empty Premises

Gordon presents a fairly simple system in [Gordon 93] which *CoCl^AM* would nevertheless be unable to use because the transition rules have extra premises. Gordon used the notions of bisimilarity and labelled transition systems to prove the semantic equivalence of three teletype input/output mechanisms: Synchronised–stream, Continuation–passing and Landin–stream.

I shall briefly examine Synchronised–stream I/O here in order to set out the sort of issues raised for *CoCl^AM*. These issues would also be present in adapting *CoCl^AM* to several other domains.

Stream Transformers

Stream transformers are functions from lists of input type to lists of output type. There are three important combinators on stream transformers, `getST`, `putST` and `nilST`

$$\begin{array}{l} \mathtt{getST}(k, h :: t) \overset{\text{red}}{\leadsto} k(h, t) \\ \mathtt{putST}(x, f, l) \overset{\text{red}}{\leadsto} x :: (f, l) \\ \mathtt{nilST}(l) \overset{\text{red}}{\leadsto} nil \end{array}$$

These can be used to form the stream transformers `giveST`, `nextST` and `skipST`. `nextST` requires the formation of a new datatype, $Maybe(\tau)$, described below.

$$Maybe(\tau) ::= Yes(\tau) \mid No$$

$$\begin{array}{rcl} \mathtt{giveST}(c, f, l) & \overset{\text{red}}{\leadsto} & f(c :: l) \\ f(\perp) \overset{\text{red}}{\leadsto} nil \Rightarrow \mathtt{nextST}(f) & \overset{\text{red}}{\leadsto} & No \\ f(\perp) \overset{\text{red}}{\leadsto} h :: t \Rightarrow \mathtt{nextST}(f) & \overset{\text{red}}{\leadsto} & Yes(h) \\ \mathtt{skipST}(f, l) & \overset{\text{red}}{\leadsto} & \mathtt{tail}(f(l)) \end{array}$$

where $\perp$ is a dummy argument used to test whether a stream transformer is ready to produce output. The use of $\perp$ may at first seem confusing, but it represents "unknown", which commonly implies divergence and is assumed to imply such in the rest of this thesis.

Synchronised–stream I/O

In *synchronised–stream I/O*, stream transformers produce streams of requests and consume streams of acknowledgements. The requests and acknowledgements are in one–to–one correspondence: the computing device specified by a stream transformer alternates between producing output requests and consuming input acknowledgements. The values of requests, `Req`, and acknowledgements, `Ack`, are:

$$\texttt{Req} ::= Get \mid Put(Char)$$

$$\texttt{Ack} ::= Got(Char) \mid Did$$

The transition rules embodying the semantics are shown in figure 9–1.

$$\frac{\texttt{nextST}(f) \overset{\text{red}}{\leadsto} Yes(r) \quad r \overset{\text{red}}{\leadsto} Get}{f \overset{n}{\to} \texttt{giveST}(Got(\underline{n}), \texttt{skipST}(f))} \tag{9.8}$$

$$\frac{\texttt{nextST}(f) \overset{\text{red}}{\leadsto} Yes(r) \quad r \overset{\text{red}}{\leadsto} Put(v) \quad v \overset{\text{red}}{\leadsto} \underline{n}}{f \overset{\bar{n}}{\to} \texttt{giveST}(Did, \texttt{skipST}(f))} \tag{9.9}$$

Figure 9–1: Transition Rules for Synchronised–stream I/O

9.4.2 Extending the Evaluate Method

In order to determine the transitions from a synchronised–stream program the Evaluate method would need to be extended. Take for instance the transition rule (9.8) in figure (9–1). This rule has several premises that various expressions can reduce to some value unlike the transition rules in $\mathcal{T}$ which, with one exception, have no premises. It was commented when the Evaluate method was first proposed that it was very specific. For the Evaluate method to be more general it would have to be able to backward chain and search through the premises of transition rules.

Reduction rules are a special sort of transition rule and labelled transition systems, especially those involving operational semantics will often include reduction rules. The Evaluate method incorporated a reduction strategy with the rule

$$\frac{a \overset{\text{red}}{\leadsto} b \quad b \overset{\alpha}{\to} c}{a \overset{\alpha}{\to} c} \tag{9.10}$$

This was the only transition rule with premises and was essentially hardwired into the method. A more general method would search the transition rules for one that applied to the goal. In many cases (9.10) will be the only such rule. One possibility at this point would be to carry on treating the premises in the same way (i.e. to reduce a once and then look for a transition from all resulting bs) an alternative would be to incorporate a reduction strategy in the method which could guide the eureka step of choosing a b.

Reduction may not be the only well-known process that frequently occurs in transition systems so it may be possible to create a method that can combine a brute–force search through the transition rules with heuristics for guiding certain special sorts of rule.

There may be some heuristic for guiding the general search, but it seems unlikely and a more probable scenario is that the general situation will require brute–force search and heuristics will only be available for specific domains.

9.4.3 Internal Actions

Recall that in Milner's CCS (described in chapter 2) it is possible to have internal τ–actions that are effectively invisible to the observer. These are a useful technique not only in CCS but also in other domains where we may wish to "ignore" certain actions. For instance if we wish to prove that a program meets a specification, the program is likely to be more detailed and may contain low–level actions that are not of interest to the specification. Currently *CoCLAM* has no facility to allow this sort of invisible action.

The solution might seem to be to simply ignore τ–actions and repeatedly apply transitions until a non τ–action is encountered. However this isn't possible:

Example 9.2 *If $P \equiv a.\mathbf{0} + b.\mathbf{0}$ and $Q \equiv a.\mathbf{0} + \tau.b.\mathbf{0}$, then P and Q do not exhibit the same behaviour since $Q \xrightarrow{\tau} b.\mathbf{0}$. $b.\mathbf{0}$ is not equivalent to P since it can not make an* a *action. So without making any action observable to a user Q has now become a state where its possible behaviours are different from P's.*

Even though the only observable actions for P and Q were a *and* b. *Q can perform an unobserved τ action which makes its behaviour different to P.*

The interaction of τ actions with objects that have more than one transition would have to be handled carefully.

9.4.4 Nondeterminism

A last extension to the system would be to handle nondeterministic labelled transition systems. Again, perhaps the best known of these is CCS. This allows a transition to have arbitrarily many results, e.g. the expression $a.C + a.D \xrightarrow{a} C$ and $a.C + a.D \xrightarrow{a} D$ for this to be bisimilar to some other expression, that expression would also have to be able to make an a transition to either C or D.

This increases the search space since it would introduce disjunctive goals into it. Consider showing the equivalence of A and B where:

$$
\begin{aligned}
A &= a.nil + a.A' && (9.11)\\
A' &= a.A && (9.12)\\
B &= a.nil + a.A' && (9.13)
\end{aligned}
$$

The possible combinations of matching transitions from A and B are

$$
\begin{gathered}
HYP \Rightarrow \\
(((\langle nil,\ nil\rangle \in \mathcal{R} \cup \sim) \wedge (\langle A',\ A'\rangle \in \mathcal{R} \cup \sim)) \vee \\
(((\langle A',\ nil\rangle \in \mathcal{R} \cup \sim) \wedge (\langle nil,\ A'\rangle \in \mathcal{R} \cup \sim))
\end{gathered}
$$

Obviously this goal is fairly simple (the first disjunct is easily shown to be true) but more complex problems can lead to a large tree of disjunctions which have to be searched.

9.5 Verifying the Proof Plans

For *CoCLAM* to be a useful system in practice, it must not only generate proof plans but also execute them to produce formal proofs. This requires the development of proof tactics to match the proof methods in some object–level theorem prover. This remains to be done. A partial investigation was undertaken into linking *CoCLAM* to Isabelle, the results of which are reported in appendix E.

The whole implementation, if it is to be pursued, needs to be made more robust and a problem with the naming of variables in rewriting needs to be solved, either by providing an Isabelle tactic that rewrites a named subterm or by adapting *CoCLAM*'s rewriting methods to provide sufficient information about instantiations for Isabelle to be able to identify the appropriate variables.

It then needs to be tested properly on the corpus of theorems planned by *CoCLAM*. At present what testing has been undertaken has involved only a small part of that corpus and has required some user intervention to get around the variable naming problem.

The other option is to attempt to link *CoCLAM* to HOL and use the tactics created by Collins for coinductive proof in HOL. This linkage will face the same problems of variable naming as is faced by the *CoCLAM*–Isabelle link. An "email" link has been attempted by which the bisimulations suggested by *CoCLAM* are supplied (by hand) to HOL. Again the testing has been fairly minimal though encouraging.

9.6 Conclusion

Four major areas of further work have been considered in detail by this chapter.

- Improve the current implementation by providing automated ways of discovering missing lemmata and introducing divergence analysis to recognise expressions with no transitions. This would extend *CoCLAM*'s ability to prove theorems within the kind of operational semantics systems examined in this thesis.
- Investigate the use of inter–construction to expand the scope of the Revise Bisimulation Critic.
- Provide better support for "plug and play" by creating a more general Evaluate method and support for τ actions and non–deterministic transition systems.
- Link *CoCLAM* with an object–level prover.

These are all important because they would extend the range of the system away from the current "toy" problems and would hopefully allow larger examples, less easily performed by hand, to be not only automatically planned but also formally verified.

There are also a number of other improvements that have been mentioned elsewhere. Several improvements to the implementation were suggested in chapter 7 where the results of testing the system were reported and improvements to the Revise Bisimulation Critic were considered in chapter 8 where generalisation techniques were surveyed.

In the next chapter I shall consider improvements that can be made to *CoCLAM* through a comparison with inductive proof in *CLAM*.

Chapter 10

Comparing the Process of Proof in Induction and Coinduction

10.1 Introduction

This chapter covers a programme of further work extending the proof methods for both coinduction and induction with reference to each other. As such it also covers some related work from the area of proof planning for induction.

There has been a lot of work on comparing induction and coinduction as proof principles and definition principles but little or no work comparing the processes of proof.

One of the original aims of the research reported in this thesis was to try and adapt the methods developed for induction to coinduction. In the end new methods were developed for coinduction and most of the methods developed for induction were not used. However there remain clear similarities between the processes of proof for the two principles and an examination of this in the light of the proposed proof plans seemed to be a profitable undertaking.

The discussion that follows attempts to draw out the observed similarities and then see if these observations can be justified by reference to the theory behind the proof principles. There are two aims:

1. To show how the theory supports the observed similarities between the proofs.
2. By clearly setting down the points of crossover to signpost the places where techniques can be transferred from one domain to the other.

10.2 Comparing Individual Proofs

The first step is to compare "anecdotally" the two proof styles by inspecting proofs done using them. This comparison is aided by the fact that many theorems can be proved either by Induction or Coinduction or have similar equivalents (except that in one theorem some variable is of strict type which is of lazy type in the other) which can be proved either way.

There are two ways of converting theorems about lazy lists into ones that can be proved inductively, either the type of some variable(s) of lazy type can be altered to an equivalent strict type or induction can be performed on the number of transitions from the objects.

Both of these are considered here and two examples are examined one using each of these forms of comparison. The coinductive proofs use the transition system, $\mathcal{T}'$, shown in figure 10–1. This transition system differs from the one

$$\frac{a : \tau_1 \to \tau_2 \qquad b : \tau_1}{a \xrightarrow{\mathtt{ap}(b)} a(b)} \tag{10.1}$$

$$\frac{}{\underline{bv} \xrightarrow{bv} \bot} \qquad \frac{}{\underline{n} \xrightarrow{n} \bot} \tag{10.2}$$

$$\frac{}{nil \xrightarrow{\mathtt{nil}} \bot} \tag{10.3}$$

$$\frac{}{a :: b \xrightarrow{\mathtt{hd}} a} \qquad \frac{}{a :: b \xrightarrow{\mathtt{tl}} b} \tag{10.4}$$

$$\frac{a : \tau \qquad \tau \neq \tau_1 \to \tau_2 \qquad a \overset{\mathrm{red}}{\leadsto} b \qquad b \xrightarrow{\alpha} c}{a \xrightarrow{\alpha} c} \tag{10.5}$$

Figure 10–1: Transition Rules for $\mathcal{T}'$

presented in chapter 3, $\mathcal{T}$, since the transitions from natural numbers are the numerals not zero and predecessor and there are no tree datatypes.

Proof planning will be a significant tool and comparisons will be made at the proof method level.

10.2.1 Restricting Theorems to Strict Lists: The Associativity of Append

One way to compare the two proof processes is to consider the coinductive proof for a theorem (taking lazy lists as an argument) and the inductive proof (taking strict lists).

Take, for instance the associativity of append. I shall present first an inductive proof and then a coinductive proof in turn highlighting the proof methods. This comparison shouldn't hinge on specific heuristics, such as rippling too much, so some abstraction has been performed in the proofs – using "Rewriting" as a proof method as opposed to "Rippling" or "Evaluate" or any of the more specialised rewriting methods that have been developed. The lists of proof methods will then be placed side by side in a tabular format in the hope that this will draw out some similarities.

Recall the definition of $<>$:

$$nil <> L \stackrel{\text{red}}{\rightsquigarrow} L \tag{10.6}$$

$$H :: T <> L \stackrel{\text{red}}{\rightsquigarrow} H :: (T <> L) \tag{10.7}$$

<u>**Inductive Proof**</u>

Theorem 10.1

$$\forall l_1, l_2, l_3.\ (l_1 <> l_2) <> l_3 = l_1 <> (l_2 <> l_3)$$

Proof.

Induction on l_1 with $h :: t$ as the Induction Scheme This gives two new goals, a base case:

$$(nil <> L_2) <> L_3 = nil <> (L_2 <> L_3)$$

and a step case:

$$(t <> L_2) <> L_3 = t <> (L_2 <> L_3) \Rightarrow$$
$$(h :: t <> L_2) <> L_3 = h :: t <> (L_2 <> L_3)$$

We shall examine the step case first.

Rewrite using (10.7) on LHS This rewrites the induction conclusion to:

$$\cdots \Rightarrow h :: (t <> L_2) <> L_3 = h :: t <> (L_2 <> L_3)$$

Rewrite using (10.7) on LHS

$$\cdots \Rightarrow h :: ((t <> L_2) <> L_3) = h :: t <> (L_2 <> L_3)$$

Rewrite using (10.7) on RHS

$$\cdots \Rightarrow h :: ((t <> L_2) <> L_3) = h :: (t <> (L_2 <> L_3))$$

Cancel h

$$\cdots \Rightarrow (t <> L_2) <> L_3 = t <> (L_2 <> L_3)$$

Fertilize This concludes the step case proof. The Base case remains.

Rewrite using (10.6) on the LHS

$$L_2 <> L_3 = nil <> (L_2 <> L_3)$$

Rewrite using (10.6) on the RHS

$$L_2 <> L_3 = L_2 <> L_3$$

Reflexivity $\Box$

<u>**Coinductive Proof**</u>

Theorem 10.2

$$\forall l_1, l_2, l_3.\ ((l_1 <> l_2) <> l_3) \sim (l_1 <> (l_2 <> l_3))$$

Proof.

Coinduction with $\{\langle (L_1 <> L_2) <> L_3, L_1 <> (L_2 <> L_3)\rangle\}$ **as Bisimulation**
This gives two new goals: $\langle (L_1 <> L_2) <> L_3, L_1 <> (L_2 <> L_3)\rangle \in \mathcal{R}$ and $\mathcal{R} \subseteq \langle \mathcal{R} \cup \sim \rangle$. Consider these in turn.

Set Membership The first goal is trivially true from the definition of $\mathcal{R}$

Casesplit L_1**.** This gives two goals:

$$\{\langle (nil <> L_2) <> L_3,\ nil <> (L_2 <> L_3)\rangle\} \subseteq \langle \mathcal{R} \cup \sim \rangle \quad (10.8)$$

$$\{\langle (H :: T <> L_2) <> L_3,\ H :: T <> (L_2 <> L_3)\rangle\} \subseteq \langle \mathcal{R} \cup \sim \rangle \quad (10.9)$$

Consider the second of these first.

Rewrite using (10.7) on LHS

$$\{\langle H :: (T <> L_2) <> L_3,\ H :: T <> (L_2 <> L_3)\rangle\} \subseteq \langle \mathcal{R} \cup \sim \rangle$$

Rewrite using (10.7) on LHS

$$\{\langle H :: ((T <> L_2) <> L_3),\ H :: T <> (L_2 <> L_3)\rangle\} \subseteq \langle \mathcal{R} \cup \sim \rangle$$

Rewrite using (10.7) on RHS

$$\{\langle H :: ((T <> L_2) <> L_3), H :: (T <> (L_2 <> L_3))\rangle\} \subseteq \langle \mathcal{R} \cup \sim\rangle$$

Take Transitions: $\stackrel{\texttt{hd}}{\to}$ **and** $\stackrel{\texttt{tl}}{\to}$ This produces two goals:

$$\{\langle (T <> L_2) <> L_3,\ T <> (L_2 <> L_3)\rangle\} \subseteq \mathcal{R} \cup \sim \tag{10.10}$$

$$\{\langle H,\ H\rangle\} \subseteq \mathcal{R} \cup \sim \tag{10.11}$$

Fertilize This discharges (10.10).

Reflexivity of $\sim$ This discharges (10.11).

This leaves only (10.8)

Rewrite using (10.6) on the LHS

$$\{\langle L_2 <> L_3,\ nil <> (L_2 <> L_3)\rangle\} \subseteq \langle \mathcal{R} \cup \sim\rangle$$

Rewrite using (10.6) on the RHS

$$\{\langle L_2 <> L_3,\ L_2 <> L_3\rangle\} \subseteq \langle \mathcal{R} \cup \sim\rangle$$

Determinacy of Transition System and Reflexivity of $\sim$ The determinacy of the transition system means that the results of all transitions from $L_1 <> L_3$ are the same and hence are in $\sim$. □

Comparing the Proofs of the Associativity of Append

If the above proofs are regarded as proof plans then a comparison of the method calls can be placed side by side as in the following table[1].

<table>
<tr><td>Induction on l_1 using $h :: t$</td><td>Coinduction Method $\mathcal{R}$</td></tr>
<tr><td>Step Case</td><td>Set Membership</td></tr>
<tr><td></td><td>Casesplit l_1</td></tr>
<tr><td colspan="2">Rewrite using (10.7) on LHS</td></tr>
<tr><td colspan="2">Rewrite using (10.7) on LHS</td></tr>
<tr><td colspan="2">Rewrite using (10.7) on RHS</td></tr>
<tr><td>Cancel h</td><td>Take Transitions $\stackrel{\texttt{hd}}{\to}$ and $\stackrel{\texttt{tl}}{\to}$</td></tr>
<tr><td colspan="2">Fertilize</td></tr>
<tr><td>Base Case</td><td>Reflexivity of $\sim$</td></tr>
<tr><td colspan="2">Rewrite using (10.6) on LHS</td></tr>
<tr><td colspan="2">Rewrite using (10.6) on RHS</td></tr>
<tr><td>Reflexivity</td><td>Determinacy of Transition</td></tr>
<tr><td></td><td>system and Reflexivity of $\sim$</td></tr>
</table>

[1]Cancellation is not necessarily a part of inductive proof, but is included here because it is proofs involving cancellation that give rise to many of the apparent similarities (more of this later).

The important parts of this pattern (i.e. those features that were observed over a number of proofs) are that the choice of induction scheme and bisimulation plus the casesplit of some variable occur at the same point; the sequence of rewrites is identical and that cancellation and reflexivity correspond to the use of transitions.

This comparison process was carried out all the applicable theorems *CoCLAM* successfully proved. This provided a total of 46 theorems. A number of patterns of proof like the above emerged, all of which contained the features mentioned. These patterns are shown in appendix F together with the theorems associated with each pattern. Only 1 proof didn't have this sort of pattern. In that proof + was cancelled in the inductive proof (i.e. a function that was not a constructor) while the coinductive proof required a generalisation.

10.2.2 Using Induction on the Number of Transitions

Not all theorems provable by coinduction have an equivalent that can be proved by induction just by changing the type of some variable. In such cases, if the overall type of the expressions is lazy lists, the function, nth, can be used to perform induction on lazy lists by examining each element of the list. In fact nth can be used as an alternative to changing the type of the variable in all cases. This technique was suggested by McAllester[2]. In this case I am assuming that only lists whose elements are of strict type are being considered but it would seem plausible that this technique could be extended to other datatypes.

$$nth(0, H :: T) \leadsto H \tag{10.12}$$
$$nth(s(N), H :: T) \leadsto nth(N, T) \tag{10.13}$$

Theorem 10.3 *(This theorem utilizes the assumption that the lists in question are lists of naturals). If L_1 and L_2 are both lazy lists of elements of strict type, τ, in a labelled transition system in which the only infinite data type is lists (so $\forall n_1, n_2 : \tau.\ n_1 \sim n_2 \Leftrightarrow n_1 = n_2$) then*

$$\forall l_1, l_2.\ l_1 \sim l_2 \ \Leftrightarrow\ \forall n.(nth(n, l_1) = nth(n, l_2) \vee (l_1 = l_2 \wedge length(l_1) \leq n))$$

where

$$length(nil) = 0 \tag{10.14}$$
$$length(h :: t) = s(length(t)) \tag{10.15}$$

and = is equality on finite lists.

Proof.

[2] Private Communication

1. $\forall l_1, l_2. l_1 \sim l_2 \Rightarrow \forall n. nth(n, l_1) = nth(n, l_2) \vee (l_1 = l_2 \wedge length(l_1) \leq n))$

   Proof by Induction on n.

   (a) $n = 0$.

   If $l_1 \overset{\texttt{nil}}{\rightarrow} \perp$ then by bisimilarity of l_1 and l_2, $l_2 \overset{\texttt{nil}}{\rightarrow} \perp$. The transition rules imply that $l_1 = l_2 = nil$ which means that for all n, $length(l_1) \leq n$.

   Otherwise $\exists h_1, t_1.\ l_1 \overset{\texttt{hd}}{\rightarrow} h_1 \wedge l_1 \overset{\texttt{tl}}{\rightarrow} t_1$. By bisimilarity of l_1 and l_2, $\exists h_2, t_2.\ l_2 \overset{\texttt{hd}}{\rightarrow} h_2 \wedge l_2 \overset{\texttt{tl}}{\rightarrow} t_2$. Furthermore $h_1 \sim h_2$ and $t_1 \sim t_2$. Hence $l_1 = h_1 :: t_1$, $l_2 = h_2 :: t_2$, $nth(0, l_1) = h_1$ and $nth(0, l_2) = h_2$. We know that $h_1 = h_2$ since they are natural numbers and $h_1 \sim h_2$. So $nth(0, l_1) = nth(0, l_2)$.

   (b) $n = s(n_1)$

   Assume that $\forall l_1, l_2. l_1 \sim l_2 \Rightarrow nth(n_1, l_1) = nth(n_1, l_2) \vee (l_1 = l_2 \wedge length(l_1) \leq n_1)$. We want to show that $\forall l_1, l_2.\ l_1 \sim l_2 \Rightarrow nth(s(n_1), l_1) = nth(s(n_1), l_2)$.
   If $l_1 = l_2 = nil$ then for all n, $length(l_1) \leq n$.
   Otherwise $l_1 = h_1 :: t_1$ and $l_2 = h_2 :: t_2$ and

   $$\begin{aligned} \cdots \Rightarrow nth(s(n_1), h_1 :: t_1) &= nth(s(n_1), h_2 :: t_2) \\ \cdots \Rightarrow nth(n_1, t_1) &= nth(n_1, t_2) \end{aligned}$$

   $t_1 \sim t_2$ since $l_1 \sim l_2$ so $nth(n, t_1) = nth(n, t_2)$ is true by appeal to the hypothesis.

2. $\forall l_1, l_2.\ (\forall n. nth(n, l_1) = nth(n, l_2) \vee (l_1 = l_2 \wedge length(l_1) \leq n)) \Rightarrow l_1 \sim l_2$.

   Proof by coinduction.

   Let $\mathcal{R} = \{\langle L_1,\ L_2 \rangle \mid \forall n. nth(n, L_1) = nth(n, L_2) \vee (L_1 = L_2 \wedge length(L_1) \leq n)\}$

   If $\langle l_1,\ l_2 \rangle \in \mathcal{R}$. The possible transitions from l_1 and l_2 are `nil` or `hd` and `tl`.

   (a) If $l_1 \overset{\texttt{nil}}{\rightarrow} \perp$ then $l_1 = nil$ so $\forall n. length(l_1) \leq n$. Since $\langle l_1,\ l_2 \rangle \in \mathcal{R}$ and $nth(n, l_1)$ has no solution (by inspection of the definition of nth) $l_1 = l_2$. Hence $l_2 = nil$ and $l_2 \overset{\texttt{nil}}{\rightarrow} \perp$. $\perp \sim \perp$ by the reflexivity of $\sim$.

   (b) If $\exists h_1, t_1$ such that $l_1 \overset{\texttt{hd}}{\rightarrow} h_1$ and $l_1 \overset{\texttt{tl}}{\rightarrow} t_1$ then $l_1 = h_1 :: t_1$. So $nth(0, l_1) = h_1$, clearly $length(l_1) > 0$ so $nth(0, l_2) = h_2 = h_1$ so $\exists t_2.\ l_2 = h_1 :: t_2$ and $l_2 \overset{\texttt{hd}}{\rightarrow} h_1$. $h_1 \sim h_1$ by reflexivity of $\sim$.

   Also $l_2 \overset{\texttt{tl}}{\rightarrow} t_2$ and $\forall n. nth(n, t_2) = nth(s(n), l_2) = nth(s(n), l_1) = nth(n, t_1)$ or $l_1 = l_2$ and $lenght(l_1) \leq s(n)$ so $t_1 = t_2$ and $length(t_1) \leq n$ hence $\langle t_1,\ t_2 \rangle \in \mathcal{R}$ by definition of $\mathcal{R}$.

In which case $\mathcal{R}$ is a bisimulation and by coinduction for all l_1 and l_2 if $\langle l_1,\ l_2\rangle \in \mathcal{R}$ then $l_1 \sim l_2$. $\square$

10.2.3 Using *nth*: The Mapiterates Theorem

Recall the following definitions:

$$\begin{aligned} map(F, nil) &\rightsquigarrow nil &(10.16)\\ map(F, H :: T) &\rightsquigarrow F(H) :: map(F, T) &(10.17)\end{aligned}$$

$$iterates(F, X) \rightsquigarrow X :: iterates(F, F(X)) \quad (10.18)$$

<u>**Inductive Proof**</u>

Theorem 10.4

$$\forall f, m, n.\ nth(n, map(f, iterates(f, m))) = nth(n, iterates(f, f(m)))$$

Proof

Induction on n **using** $s(n)$ This gives two new goals

$$nth(0, map(F, iterates(F, M))) = nth(0, iterates(F, F(M)))$$

and

$$\begin{gathered} nth(n, map(F, iterates(F, M))) = nth(n, iterates(F, F(M))) \Rightarrow \\ nth(s(n), map(F, iterates(F, M))) = nth(s(n), iterates(F, F(M))) \end{gathered}$$

Consider the step case first.

Rewrite using (10.18) on the LHS The induction conclusion becomes

$$\begin{gathered} \cdots \Rightarrow \\ nth(s(n), map(F, M :: (iterates(F, F(M))))) = \\ nth(s(n), iterates(F, F(M))) \end{gathered}$$

Rewrite using (10.17) on the LHS

$$\begin{gathered} \cdots \Rightarrow \\ nth(s(n), F(M) :: map(F, iterates(F, F(M)))) = \\ nth(s(n), iterates(F, F(M))) \end{gathered}$$

Rewrite using (10.18) on the RHS

$$\begin{gathered} \cdots \Rightarrow \\ nth(s(n), F(M) :: map(F, iterates(F, F(M)))) = \\ nth(s(n), F(M) :: iterates(F, F(F(M)))) \end{gathered}$$

Rewrite using (10.13) on both sides

$$\cdots \Rightarrow nth(n, map(F, iterates(F, F(M)))) = \\ nth(n, iterates(F, F(F(M))))$$

Fertilize This leaves the base case.

Rewrite using (10.18) on the LHS

$$nth(0, map(F, M :: (iterates(F, F(M))))) = \\ nth(0, iterates(F, F(M)))$$

Rewrite using (10.17) on the LHS

$$nth(0, F(M) :: map(F, iterates(F, F(M)))) = \\ nth(0, iterates(F, F(M)))$$

Rewrite using (10.18) on the RHS

$$nth(0, F(M) :: map(F, iterates(F, F(M)))) = \\ nth(0, F(M) :: iterates(F, F(F(M))))$$

Rewrite using (10.12) on both sides

$$F(M) = F(M)$$

Reflexivity □

<u>**Coinductive Proof**</u>

Theorem 10.5

$$\forall f, m.\ map(f, iterates(f, m)) \sim iterates(f, f(m))$$

Proof

Coinduction using $\{\langle map(F, iterates(F, M)),\ iterates(F, F(M))\rangle\}$
This gives two goals

$$\langle map(F, iterates(F, M)),\ iterates(F, F(M))\rangle \in \mathcal{R}$$

$$\mathcal{R} \subseteq \langle \mathcal{R} \cup \sim \rangle$$

Rewrite using (10.18) on the LHS

$$\{\langle map(F, M :: (iterates(F, F(M)))), \\ iterates(F, F(M))\rangle\} \subseteq \langle \mathcal{R} \cup \sim \rangle$$

Rewrite using (10.17) on the LHS

$$\{\langle F(M) :: map(F, iterates(F, F(M))),\\ iterates(F, F(M))\rangle\} \subseteq \langle \mathcal{R} \cup \sim\rangle$$

Rewrite using (10.18) on the RHS

$$\{\langle F(M) :: map(F, iterates(F, F(M))),\\ F(M) :: iterates(F, F(F(M)))\rangle\}\\ \subseteq \langle \mathcal{R} \cup \sim\rangle$$

Take Head and Tail Transitions This gives two goals:

$$\langle F(M),\ F(M)\rangle \in \mathcal{R} \tag{10.19}$$

$$\langle map(F, iterates(F, F(M))),\ iterates(F, F(F(M)))\rangle \in \mathcal{R} \tag{10.20}$$

Fertilize This discharges (10.20) leaving (10.19) to be proved.

Reflexivity of $\sim$ $\square$

Comparing the Proofs of Mapiterates

<table>
<tr><td colspan="2">Induction on n using $s(n)$</td><td rowspan="2">Coinduction $\mathcal{R}$</td></tr>
<tr><td>Base Case</td><td>Step Case</td></tr>
<tr><td colspan="3">Rewrite using (10.18) on the LHS</td></tr>
<tr><td colspan="3">Rewrite using (10.17) on the LHS</td></tr>
<tr><td colspan="3">Rewrite using (10.18) on the RHS</td></tr>
<tr><td>Rewrite using (10.12) on both sides</td><td>Rewrite using (10.13) on both sides</td><td>Take Transitions</td></tr>
<tr><td></td><td colspan="2">Fertilize</td></tr>
<tr><td>Reflexivity</td><td></td><td>Reflexivity of $\sim$</td></tr>
</table>

In these proofs a slightly different pattern emerges since the rewriting work needed for the base case and step case in the inductive proof is the same and is done only once in the coinductive proof since it doesn't separate out the two cases until transitions. However broadly speaking the same observations apply except that choice of induction scheme and bisimulation are at the same point and if any casesplitting is required then it is needed in both proofs and is identical in both proofs.

Once again this process was attempted on all the applicable theorems that *CoCLAM* proved, it applied to 43 theorems all of which exhibited the characteristics described above. Once again the results are reported in appendix F.

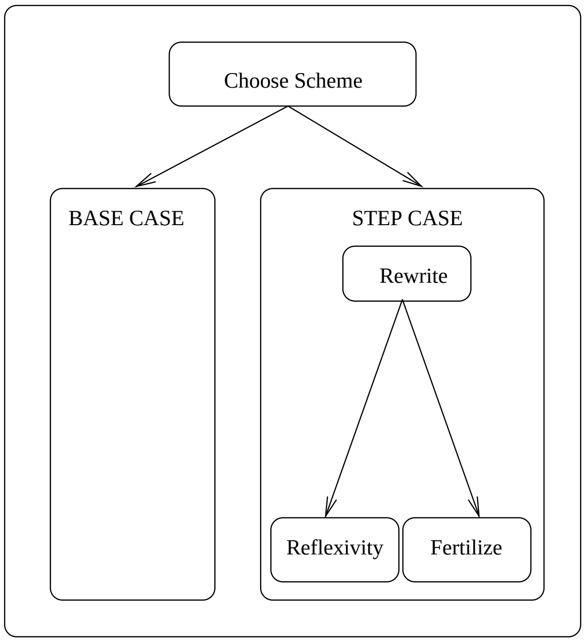

Figure 10–2: The Proof Plan for Induction

10.3 Comparison of Proof Methods

The tabular representation of the proof methods used in each proof suggest equivalences between certain steps in the proofs, e.g. The Choice of Induction Scheme and Bisimulation and the various rewriting steps. These similarities can be represented diagrammatically, superposing the proof strategies as shown in figures 10–2, 10–3 and 10–4. This places proof methods that appear to be performing similar tasks in the same general areas of the diagram.

Both these proof plan diagrams are for the proofs of theorems such as $map(f, map(g, l)) = map(f \circ g, l)$ where there is one base case and one step case in the inductive proof. In the base case both the expressions in the equation evaluate to nil and in the step case there is a casesplit after rewriting (induced by the transitions in the coinductive plan) in order to consider the heads and tails of the lists separately. The heads are shown to be equal and the reflexivity of $=$ or $\sim$ is used in the "head" case while the tail case requires fertilization.

Obviously these diagrams have been drawn so that they overlap in a certain way, however it is interesting to note that the general "shape" of the proof strategies has not had to be altered to do this. Methods have not had to be moved around the proof strategies in relation to other methods (although in the induction plan the method boxes have been enlarged). This provides supporting evidence of the comparable natures of those stages of the proof. It also suggests that proof strategy diagrams could well be a useful tool for the high level representation and discussion of proofs.

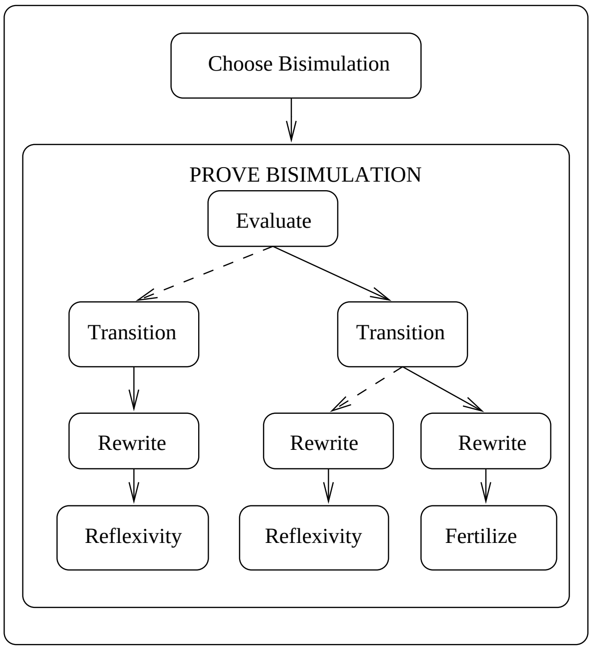

Figure 10–3: The Proof Plan for Coinduction

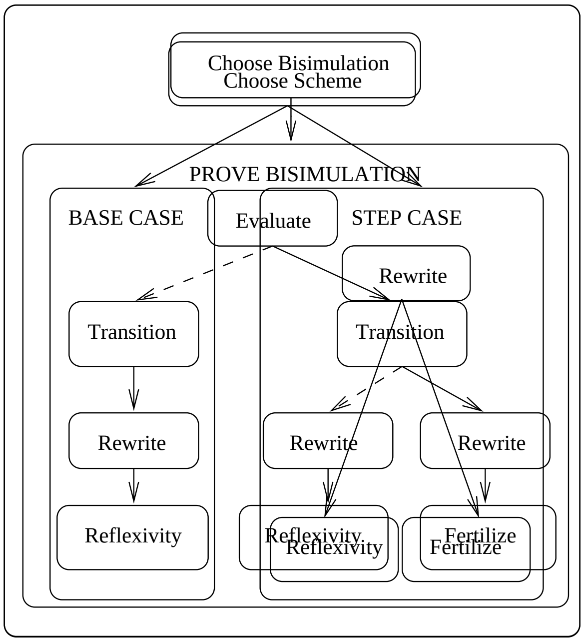

Figure 10–4: The Proof Plans Superposed

These diagrams further reinforce the suggestion that links should be sought between the choice of Bisimulation and induction scheme, and between Rippling and the Evaluate, Transition, Ripple sequence.

10.4 Standardizing the Representations

There are clearly similarities and dualities involved in the two proof processes. However, accepted methods for presenting the proofs tend to obscure the underlying causes of these similarities. I'm not necessarily advocating any change in the representations except for the purposes of comparison. The relative merits of the various proposed presentations are discussed further in §10.9.

For the purposes of comparison the process of inductive proof will be represented in a non–standard way that is closer to the representation for coinduction. There were two reasons for singling out the induction method for change rather than the coinduction method. Firstly coinduction is expressed in terms that are closer to the language of fixedpoints which is the theory that underlies induction and coinduction. Secondly inductive proof is better understood and therefore it is easier to alter the inductive proof and it still remain relatively comprehensible.

Recall the induction and coinduction rules as presented in chapter 2:

$$\frac{\mathcal{F}(\mathcal{S}) \subseteq \mathcal{S}}{lfp(\mathcal{F}) \subseteq \mathcal{S}} \qquad \frac{\mathcal{S} \subseteq \mathcal{F}(\mathcal{S})}{\mathcal{S} \subseteq gfp(\mathcal{F})}$$

This presentation makes the dualities between the two processes clear – however in the process of any specific proof this clear duality tends to get obscured, since the rules are not used in this form but specialised, for example as:

$$\frac{P(nil) \quad P(t) \Rightarrow P(h :: t)}{\forall l.\, P(l)} \qquad \frac{\langle x,\, y\rangle \in \mathcal{R} \quad \langle \mathcal{R} \cup \sim\rangle}{x \sim y}$$

Two rules that do not appear to have a lot in common.

Consider the presentation of the coinduction rule. $\langle - \rangle$ is used for $\mathcal{F}$ in many coinductive proofs since it expresses some generality through the use of labelled transition systems. It is also common practice to insert an intermediate set into the premise to make the rule

$$\frac{\mathcal{S} \subseteq \mathcal{R} \subseteq \langle \mathcal{R} \rangle}{\mathcal{S} \subseteq gfp(\langle - \rangle)}$$

Lastly the goal is written as $x \sim y$ rather than $\{\langle x,\, y\rangle\} \subseteq gfp(\langle - \rangle)$. As a result the coinduction rule is presented as

$$\frac{\{\langle x,\, y\rangle\} \subseteq \mathcal{R} \subseteq \langle \mathcal{R} \rangle^3}{x \sim y} \tag{10.21}$$

I here present a new function $\lceil - \rceil$ which looks at transitions from one object in a similar way that $\langle - \rangle$ looks at transitions from two in order to formulate the induction rule in a similar fashion to (10.21)

$$\lceil \mathcal{S} \rceil = \{a | \forall \alpha. (\forall a'.\, a \xrightarrow{\alpha} a' \Rightarrow (a' \in \mathcal{S} \vee a' \equiv \bot))\}$$

I shall use $\lceil - \rceil$ to provide a least fixedpoint for the induction rule in the same way that $\langle - \rangle$ provides a greatest fixedpoint for the coinduction rule[4]. Notice, for instance, that if the transition system contains `p` and `0` then the least fixedpoint of $\lceil - \rceil$ will contain the natural numbers.

If $\lceil - \rceil$ is substituted for $\mathcal{F}$ in the induction rule it becomes:

$$\frac{\lceil \mathcal{S} \rceil \subseteq \mathcal{S}}{lfp(\lceil - \rceil) \subseteq \mathcal{S}}$$

Introducing the sort of intermediate chaining of sets seen in the coinduction rule makes this:

$$\frac{\lceil \mathcal{S}' \rceil \subseteq \mathcal{S}' \subseteq \mathcal{S}}{lfp(\lceil - \rceil) \subseteq \mathcal{S}}$$

Since the induction rule is used to prove that some set of objects have some property, P, (i.e. it is a subset of $\{x | P(x)\}$). The final version of the induction rule is

$$\frac{\lceil \mathcal{S}' \rceil \subseteq \mathcal{S}' \subseteq \{x \mid P(x)\}}{\forall x \in lfp(\lceil - \rceil).P(x)} \tag{10.22}$$

This still appears unfamiliar as a formulation of the induction rule, but I hope to use this to draw out the similarities.

As an example of a proof utilising this induction rule, consider again the associativity of append using the LTS $\mathcal{T}'$.

Example 10.1

$$\forall l_1, l_2, l_3.\, (l_1 <> l_2) <> l_3 = l_1 <> (l_2 <> l_3)$$

---

[3]This is slightly different from the presentation used elsewhere in the thesis. In particular the chain has been left explicit not split into $\langle x,\ y \rangle \in \mathcal{R} \quad \mathcal{R} \subseteq \langle \mathcal{R} \rangle$ and the extension of $\langle \mathcal{R} \rangle$ to $\langle \mathcal{R} \cup \sim \rangle$ has been omitted since this made no difference to the theorems that could be proved, it simply made that proof simpler in some cases.

[4]NB. $\lceil - \rceil$ is monotonic. If $\mathcal{R} \subseteq \mathcal{S}$ then if $x \in \lceil \mathcal{R} \rceil$ it follows that $\forall \alpha. (\forall a'.\, x \xrightarrow{\alpha} a' \Rightarrow (a' \in \mathcal{R} \vee a' \equiv \bot))$ (by definition of $\lceil - \rceil$) and hence that $\forall \alpha. (\forall a'. x \xrightarrow{\alpha} a' \Rightarrow (a' \in \mathcal{S} \vee a' \equiv \bot))$ (since $\mathcal{R} \subseteq \mathcal{S}$) therefore $x \in \lceil \mathcal{S} \rceil$ and $\lceil \mathcal{R} \rceil \subseteq \lceil \mathcal{S} \rceil$.

Partial Proof

To use the induction rule a distinguished variable (the induction variable) must be chosen. Let this variable be l_1.

Let $\mathcal{S} = \{l \mid l : list(\tau) \Rightarrow (l <> L_2) <> L_3 = l <> (L_2 <> L_3)\}$.

(10.22)'s premise provides two new goals:

$$\lceil \mathcal{S} \rceil \subseteq \mathcal{S} \tag{10.23}$$

$$\mathcal{S} \subseteq \{x \mid P(x)\} \tag{10.24}$$

Using an inference rule similar to (3.22) (see Appendix D) we get the subgoal

$$\begin{gathered} l \overset{\alpha}{\rightarrow} \phi \Rightarrow \\ ((\phi <> L_1) <> L_3 = \phi <> (L_2 <> L_3) \Rightarrow \\ (l <> L_2') <> L_3' = l <> (L_2' <> L_3')) \end{gathered}$$

Objects which aren't of type $list(\tau)$ are trivially members of $\mathcal{S}$. This means the only transitions of interest are those which result in objects of this type. In fact strictly speaking we are interested in functions onto lists.

The possible transitions, α, from l which result in objects of list type are $\overset{\texttt{nil}}{\rightarrow}$ and $\overset{\texttt{tl}}{\rightarrow}$ ($\overset{\texttt{nil}}{\rightarrow}$ is included since $\perp$ is of arbitrary function type).

1. $\alpha = \texttt{nil}$

$$\begin{gathered} \perp <> (L_2 <> L_3) = (\perp <> L_2) <> L_3 \Rightarrow \\ nil <> (L_2 <> L_3) = (nil <> L_2) <> L_3 \end{gathered}$$

   Strictness analysis evaluates the induction hypothesis to $\perp = \perp$ which is trivially true, hence we get the goal

$$nil <> (L_2 <> L_3) = (nil <> L_2) <> L_3$$

   This should be familiar as a base case goal and the proof proceeds as in §10.2.1

2. $\alpha = \texttt{tl}$
   $l = h :: t$ the goal becomes

$$\begin{gathered} t <> (L_2 <> L_3) = (t <> L_2) <> L_3 \Rightarrow \\ h :: t <> (L_2 <> L_3) = (h :: t <> L_2) <> L_3 \end{gathered}$$

   which again should be familiar as a standard step case goal and the proof proceeds as normal.

Clearly if $l \in \mathcal{S}$ then $(l <> L_2) <> L_3 = l <> (L_2 <> L_3)$. This discharges (10.24). $\Box$

The "new" formulation of the induction rule has effectively placed an intermediate step into the proof process which uses transitions to decide upon the casesplitting of the induction variable.

10.5 Choice of Induction Scheme and Choice of Bisimulation

Inspection of the tables of proof methods for induction and coinduction and the superposed proof plans suggest a relationship between the choice of induction scheme and the choice of bisimulation. The new representation replaces the choice of induction scheme with the choice of a set (as the coinduction rule requires the choice of a relation). I contend that changing the choice of induction scheme is equivalent to altering the labelled transition system in this new representation and that both alteration of transition scheme and alteration of the set are valid ways to proceed.

In standard induction theorem proving we are free to use any suitable induction scheme, for example the schemes

$$\frac{P(nil) \quad P(t) \to P(h :: t)}{\forall l.\ P(l)} \qquad \frac{P(nil) \quad P(h :: nil) \quad P(t) \to P(h_1 :: h_2 :: t)}{\forall l.\ P(l)}$$

can both be used to prove $\forall l.\ P(l)$. However, if we view induction in terms of labelled transition systems, as suggested above, then these schemes arise out of *different* transition systems. For the first we need the standard transitions `nil`, `hd` and `tl`, while for the second we need transitions that are "chained" in some way (e.g. to give `nil`, `hd`, `tl.nil`, `tl.hd` and `tl.tl`).

What follows is some theoretical work to establish that transition systems that are "well–chained" can be used interchangeably like the induction schemes they give rise to.

First it is necessary to formalise the notion of "well–chained":

Notation

- $a \stackrel{\alpha.\beta}{\to} b$ is understood to mean that $\exists c.\ a \stackrel{\alpha}{\to} c \stackrel{\beta}{\to} b$.
- α^2 represents the transition $\alpha.\alpha$ similarly α^3 represents $\alpha.\alpha.\alpha$ and α^i represents a transition consisting of i α transitions.

Definition 10.1 *A list of transition systems* $\{\mathcal{T}_0, \ldots, \mathcal{T}_n\}$ *is* α–**well–chained** *if* $\mathcal{A}_{\mathcal{T}_0}$*, the set of transitions for* $\mathcal{T}_0$*, contains* α *and for all* $0 < k \leq n$ $\mathcal{A}_{\mathcal{T}_k}$ *is composed of the following transitions:*

1. *All transitions in* $A_{\mathcal{T}_0} - \{\alpha\}$

2. *All transitions of the form* $\alpha^i.\beta$ *where* $0 < i \leq k$ *and* $\beta \in A_{\mathcal{T}_0}^{\tau_\alpha} - \{\alpha\}$.
   $A_{\mathcal{T}}^{\tau_\alpha}$ *is the set of transitions in* $A_{\mathcal{T}}$ *whose domain is of the same type as the codomain of* α
3. α^{k+1}

10.5.1 Interchangeability in Induction

The interchangeability of induction schemes hinges on well–foundedness results so it should be no surprise that the interchangeability of the transition systems similarly depends upon well–foundedness. As a result, it is necessary to establish that the least fixedpoints of $\lceil - \rceil$ are well–founded according to some order. This order will be the one imposed by sequences of transitions.

Lemma 10.1 $\perp \in lfp(\lceil - \rceil)$ *for all transition systems*

Proof. By the definition of a fixedpoint $lfp(\lceil - \rceil) = \lceil lfp(\lceil - \rceil) \rceil$

$$\lceil lfp(\lceil - \rceil) \rceil = \{a \mid \forall \alpha. \forall a'.\ a \xrightarrow{\alpha} a' \Rightarrow (a' \in lfp(\lceil - \rceil) \vee a' \equiv \perp)\}$$

Since there are no transitions from $\perp$, $\perp$ trivially satisfies the formula $\forall \alpha. \forall a'.\ \perp \xrightarrow{\alpha} a' \Rightarrow (a' \in lfp(\lceil - \rceil) \vee a' \equiv \perp)$. Hence $\perp \in \lceil lfp(\lceil - \rceil) \rceil$ and so $\perp \in lfp(\lceil - \rceil)$. $\square$

Definition 10.2 *Let* $a \succ b$ *iff* $\exists \alpha . a \xrightarrow{\alpha} b$. $\succ$ *is the* **transition order** *on a domain.*

Theorem 10.6 *For all transition systems the transition order on* $lfp(\lceil - \rceil)$ *is well–founded.*

Proof. Let $\mathcal{S}$ contain all the members of $lfp(\lceil - \rceil)$ which are *only* involved in finite chains in $lfp(\lceil - \rceil)$. $\mathcal{S}$ is non–empty since it will contain $\perp$. We shall show that $\mathcal{S}$ is a fixedpoint for $\lceil - \rceil$ and hence that $\mathcal{S} = lfp(\lceil - \rceil)$.

$$\lceil \mathcal{S} \rceil = \{a \mid \forall \alpha. \forall a'.\ a \xrightarrow{\alpha} a' \Rightarrow (a' \in \mathcal{S} \vee a' \equiv \perp)\}$$

If $a \in \lceil \mathcal{S} \rceil$ then all the transitions result in members of finite chains in $lfp(\lceil - \rceil)$ and since $lfp(\lceil - \rceil)$ is a fixedpoint $a \in lfp(\lceil - \rceil)$ so $a \in \mathcal{S}$.

If $a \in \mathcal{S}$ then either all the transitions from a result in members of $\mathcal{S}$ or a has no transitions (i.e. $a \equiv \perp$) in both cases $a \in \lceil \mathcal{S} \rceil$.

Hence $\lceil \mathcal{S} \rceil = \mathcal{S}$ so $\mathcal{S}$ is a fixedpoint for $\lceil - \rceil$ since $\mathcal{S} \subseteq lfp(\lceil - \rceil)$, $\mathcal{S} = lfp(\lceil - \rceil)$. So $lfp(\lceil - \rceil)$ contains no infinite chains of transitions and the transition order is well–founded in $lfp(\lceil - \rceil)$. $\square$

If we want to show that two transition systems in an α–well–chained list are interchangeable then we need to show that for any α–well–chained list of transitions, $\{\mathcal{T}_0, \cdots, \mathcal{T}_n\}$, $lfp(\lceil - \rceil_{\mathcal{T}_0}) = lfp(\lceil - \rceil_{\mathcal{T}_k})$ for all $k \leq n$. We shall break this down into proving the two inclusions $lfp(\lceil - \rceil_{\mathcal{T}_0}) \subseteq lfp(\lceil - \rceil_{\mathcal{T}_k})$ and $lfp(\lceil - \rceil_{\mathcal{T}_k}) \subseteq lfp(\lceil - \rceil_{\mathcal{T}_0})$. To prove each of these we shall use the general form of the induction rule

$$\frac{\lceil \mathcal{S} \rceil \subseteq \mathcal{S}}{lfp(\lceil - \rceil) \subseteq \mathcal{S}}$$

to give us

$$\frac{\lceil lfp(\lceil - \rceil_{\mathcal{T}_k}) \rceil_{\mathcal{T}_0} \subseteq lfp(\lceil - \rceil_{\mathcal{T}_k})}{lfp(\lceil - \rceil_{\mathcal{T}_0}) \subseteq lfp(\lceil - \rceil_{\mathcal{T}_k})}$$

and

$$\frac{\lceil lfp(\lceil - \rceil_{\mathcal{T}_0}) \rceil_{\mathcal{T}_k} \subseteq lfp(\lceil - \rceil_{\mathcal{T}_0})}{lfp(\lceil - \rceil_{\mathcal{T}_k}) \subseteq lfp(\lceil - \rceil_{\mathcal{T}_0})}$$

This gives us two lemmata to prove namely that $\lceil lfp(\lceil - \rceil_{\mathcal{T}_k}) \rceil_{\mathcal{T}_0} \subseteq lfp(\lceil - \rceil_{\mathcal{T}_k})$ and $\lceil lfp(\lceil - \rceil_{\mathcal{T}_0}) \rceil_{\mathcal{T}_k} \subseteq lfp(\lceil - \rceil_{\mathcal{T}_0})$.

In the following proofs transitions are annotated with a subscript indicating which transition system they are occuring in. This is to aid the reader since the much of the proof depends upon using transitions made in one system to show that transitions can be made in another.

Lemma 10.2 $\lceil lfp(\lceil - \rceil_{\mathcal{T}_k}) \rceil_{\mathcal{T}_0} \subseteq lfp(\lceil - \rceil_{\mathcal{T}_k})$

Proof. We are going to show that for some expression, $a \in \lceil lfp(\lceil - \rceil_{\mathcal{T}_k}) \rceil_{\mathcal{T}_0}$, $a \in lfp(\lceil - \rceil_{\mathcal{T}_k})$. We do this by showing that all transitions from a in $\mathcal{A}_{\mathcal{T}_k}$ result in members of $lfp(\lceil - \rceil_{\mathcal{T}_k})$. In the course of this we will frequently exploit the fact that we know that the result of any transition in $\mathcal{A}_{\mathcal{T}_0}$ from a is a member of $lfp(\lceil - \rceil_{\mathcal{T}_k})$.

Since $lfp(\lceil - \rceil_{\mathcal{T}_k})$ is well–founded this means that its members are only part of finite chains of transitions in $lfp(\lceil - \rceil_{\mathcal{T}_k})$. We shall conduct the proof by induction on the maximum length of the transition chains in $lfp(\lceil - \rceil_{\mathcal{T}_k})$ from b where $a \xrightarrow{\beta}_{\mathcal{T}_0} b$.

Base Case. Assume that if $a \xrightarrow{\beta}_{\mathcal{T}_0} a_1$ then the maximum length of a chain in $lfp(\lceil - \rceil_{\mathcal{T}_k})$ from a_1 is 0. This means that $a_1 \equiv \bot_{\mathcal{T}_k}$

We shall examine each possible transition in $\mathcal{A}_{\mathcal{T}_k}$ from a, following the breakdown of the transitions in definition 10.1. We check that the results of these transitions are all in $lfp(\lceil - \rceil_{\mathcal{T}_k})$ so that $a \in lfp(\lceil - \rceil_{\mathcal{T}_k})$. The possible transitions are: $\beta \in A_{\mathcal{T}_0} - \{\alpha\}$, $\alpha^i.\beta$ where $\beta \in A_{\mathcal{T}_0} - \{\alpha\}$ and $0 < i \leq k$ and α^{k+1}.

No Transitions If a can't make any transition in $A_{\mathcal{T}_k}$ then $a \equiv \bot_{\mathcal{T}_k} \in lfp(\lceil - \rceil_{\mathcal{T}_k})$ and we are done.

<u>$a \xrightarrow{\beta}_{\mathcal{T}_k} b$ **where** $\beta \in A_{\mathcal{T}_0} - \{\alpha\}$</u> Since $\beta \in A_{\mathcal{T}_0}$ then $a \xrightarrow{\beta}_{\mathcal{T}_0} b$. So $b \in lfp(\lceil - \rceil_{\mathcal{T}_k})$ since $a \in \lceil lfp(\lceil - \rceil_{\mathcal{T}_k}) \rceil_{\mathcal{T}_0}$.

<u>$a \xrightarrow{\alpha^i.\beta}_{\mathcal{T}_k}$ **where** $\beta \in A_{\mathcal{T}_0} - \{\alpha\}$ **and** $0 < i \leq k$</u> If $a \xrightarrow{\alpha^i.\beta}_{\mathcal{T}_k} b$ then there is a chain of transitions from a:

$$a \xrightarrow{\alpha}_{\mathcal{T}_0} a_1 \xrightarrow{\alpha}_{\mathcal{T}_0} \cdots \xrightarrow{\alpha}_{\mathcal{T}_0} a_i \xrightarrow{\beta}_{\mathcal{T}_0} b$$

Since $a \in \lceil lfp(\lceil - \rceil_{\mathcal{T}_k}) \rceil_{\mathcal{T}_0}$, $a_1 \in lfp(\lceil - \rceil_{\mathcal{T}_k})$.
Now for $0 < i \leq k$, $\alpha^{i-1}.\beta$ is also in $A_{\mathcal{T}_k}$ by definition 10.1 so since $a_1 \in lfp(\lceil - \rceil_{\mathcal{T}_k})$ and $a_1 \xrightarrow{\alpha^{i-1}.\beta}_{\mathcal{T}_k} b$, $b \in lfp(\lceil - \rceil_{\mathcal{T}_k})$.

<u>$a \xrightarrow{\alpha^{k+1}}_{\mathcal{T}_k} b$</u> If $a \xrightarrow{\alpha^{k+1}}_{\mathcal{T}_k} b$ then there is a chain of transitions from a:

$$a \xrightarrow{\alpha} a_1 \xrightarrow{\alpha} \cdots \xrightarrow{\alpha} a_k \xrightarrow{\alpha} b$$

Since $a \in \lceil lfp(\lceil - \rceil_{\mathcal{T}_k}) \rceil_{\mathcal{T}_0}$, $a_1 \in lfp(\lceil - \rceil_{\mathcal{T}_k})$.
Recall that, by the base case assumption, the maximum possible length of a chain in $lfp(\lceil - \rceil_{\mathcal{T}_k})$ from a_1 is 0 (i.e. $a_1 \equiv \perp_{\mathcal{T}_k}$). Notice also that $a_1 \xrightarrow{\alpha^k}_{\mathcal{T}_{k-1}} b$. We use these observations to show that $b \equiv \perp_{\mathcal{T}_k}$
If b can make a transition in $A_{\mathcal{T}_k}$ to some expression, c, then that transition can be $\beta \in A_{\mathcal{T}_0}^{\mathcal{T}_\alpha} - \{\alpha\}$, $\alpha^i.\beta$ ($0 < i \leq k$, $\beta \in A_{\mathcal{T}_0}^{\mathcal{T}_\alpha} - \{\alpha\}$) or α^{k+1}.

<u>$b \xrightarrow{\beta}_{\mathcal{T}_k} c$ **where** $\beta \in A_{\mathcal{T}_0}^{\mathcal{T}_\alpha} - \{\alpha\}$</u> In this case $a_1 \xrightarrow{\alpha^k.\beta}_{\mathcal{T}_k} c$. This is a contradication because $a_1 \equiv \perp_{\mathcal{T}_k}$.

<u>$b \xrightarrow{\alpha^i.\beta}_{\mathcal{T}_k} c$ **(**$0 < i \leq k$**,** $\beta \in A_{\mathcal{T}_0}^{\mathcal{T}_\alpha} - \{\alpha\}$**) or** $b \xrightarrow{\alpha^{k+1}}_{\mathcal{T}_k} c$</u> This implies there is some expression c' such that $b \xrightarrow{\alpha}_{\mathcal{T}_0} c'$. In this case $a_1 \xrightarrow{a^{k+1}}_{\mathcal{T}_k} c'$ and once again we have a contradiction.

This excludes all possible transitions from b in $\mathcal{A}_{\mathcal{T}_k}$ so $b \equiv \perp_{\mathcal{T}_k} \in lfp(\lceil - \rceil_{\mathcal{T}_k})$.

So $\forall \beta \in A_{\mathcal{T}_k}$ if $a \xrightarrow{\beta}_{\mathcal{T}_k} b$ then $b \in lfp(\lceil - \rceil_{\mathcal{T}_k})$. Hence $a \in lfp(\lceil - \rceil_{\mathcal{T}_k})$.

Step Case. Assume that if $a \in \lceil lfp(\lceil - \rceil_{\mathcal{T}_k}) \rceil_{\mathcal{T}_0}$ and if $a \xrightarrow{\beta}_{\mathcal{T}_0} b$ implies the longest chain from b is of length n or less then $a \in lfp(\lceil - \rceil_{\mathcal{T}_k})$

Let $a \in \lceil lfp(\lceil - \rceil_{\mathcal{T}_k}) \rceil_{\mathcal{T}_0}$ be an expression such that for all a_1 where $a \xrightarrow{\beta}_{\mathcal{T}_0} a_1$ the longest chain from a_1 is of length $n + 1$ or less.

Again we examine each possible $\beta \in A_{\mathcal{T}_k}$ in turn to check that if $a \xrightarrow{\beta}_{\mathcal{T}_k} b$ then $b \in lfp(\lceil - \rceil_{\mathcal{T}_k})$

No transitions If a can't make any transition in $A_{\mathcal{T}_k}$ then $a \equiv \perp_{\mathcal{T}_k} \in lfp(\lceil - \rceil_{\mathcal{T}_k})$ and we are done.

<u>$a \xrightarrow{\beta}_{\mathcal{T}_k} b$ **where** $\beta \in A_{\mathcal{T}_0} - \{\alpha\}$</u> Since $\beta \in A_{\mathcal{T}_0}$, $a \xrightarrow{\beta}_{\mathcal{T}_0} b$ so $b \in lfp(\lceil - \rceil_{\mathcal{T}_k})$ since $a \in \lceil lfp(\lceil - \rceil_{\mathcal{T}_k}) \rceil_{\mathcal{T}_0}$.

<u>$a \xrightarrow{\alpha^i.\beta}_{\mathcal{T}_k}$ **where** $\beta \in A_{\mathcal{T}_0} - \{\alpha\}$ **and** $0 < i \leq k$</u> If $a \xrightarrow{\alpha^i.\beta}_{\mathcal{T}_k} b$ then there is a chain of transitions from a:

$$a \xrightarrow{\alpha}_{\mathcal{T}_0} a_1 \xrightarrow{\alpha}_{\mathcal{T}_0} \cdots \xrightarrow{\alpha}_{\mathcal{T}_0} a_i \xrightarrow{\beta}_{\mathcal{T}_0} b$$

Since $a \in \lceil lfp(\lceil - \rceil_{\mathcal{T}_k}) \rceil_{\mathcal{T}_0}$, $a_1 \in lfp(\lceil - \rceil_{\mathcal{T}_k})$.
Now for $0 < i \leq k$, $\alpha^{i-1}.\beta$ is also in $A_{\mathcal{T}_k}$ by definition 10.1 so since $a_1 \in lfp(\lceil - \rceil_{\mathcal{T}_k})$ and $a_1 \xrightarrow{\alpha^{i-1}.\beta}_{\mathcal{T}_k} b$, $b \in lfp(\lceil - \rceil_{\mathcal{T}_k})$.

<u>$a \xrightarrow{\alpha^{k+1}}_{\mathcal{T}_k} b$</u> If $a \xrightarrow{\alpha^{k+1}}_{\mathcal{T}_k} b$ then there is a chain of transitions from a such that

$$a \xrightarrow{\alpha}_{\mathcal{T}_0} a_1 \xrightarrow{\alpha}_{\mathcal{T}_0} \cdots \xrightarrow{\alpha}_{\mathcal{T}_0} a_k \xrightarrow{\alpha} b$$

Since $a \in \lceil lfp(\lceil - \rceil_{\mathcal{T}_k}) \rceil_{\mathcal{T}_0}$, $a_1 \in lfp(\lceil - \rceil_{\mathcal{T}_k})$.

We are interested in the transitions, $b \xrightarrow{\beta}_{\mathcal{T}_k} c$, to check that c and hence b is in $lfp(\lceil - \rceil_{\mathcal{T}_k})$. Once again we will break these down and use the step case assumption to show that $b \in lfp(\lceil - \rceil_{\mathcal{T}_k})$.

No transitions If b can't make any transition in $A_{\mathcal{T}_k}$ then $b \equiv \perp_{\mathcal{T}_k} \in lfp(\lceil - \rceil_{\mathcal{T}_k})$ and we are done.

<u>$b \xrightarrow{\beta}_{\mathcal{T}_k} c$ **where** $\beta \in A_{\mathcal{T}_0} - \{\alpha\}$</u> We have the following chain of transitions:

$$a \xrightarrow{\alpha}_{\mathcal{T}_0} a_1 \xrightarrow{\alpha}_{\mathcal{T}_0} \cdots \xrightarrow{\alpha}_{\mathcal{T}_0} a_k \xrightarrow{\alpha}_{\mathcal{T}_0} b \xrightarrow{\beta}_{\mathcal{T}_0} c$$

Recall that $a_1 \in lfp(\lceil - \rceil_{\mathcal{T}_k})$. $a_1 \xrightarrow{\alpha^k.\beta}_{\mathcal{T}_k} c$ so $c \in lfp(\lceil - \rceil_{\mathcal{T}_k})$

<u>$b \xrightarrow{\alpha^i.\beta} c$ **where** $\beta \in A_{\mathcal{T}_0} - \{\alpha\}$ **and** $0 < i \leq k$</u> We have the following chain of transitions

$$a \xrightarrow{\alpha}_{\mathcal{T}_0} a_1 \xrightarrow{\alpha}_{\mathcal{T}_0} \cdots \xrightarrow{\alpha}_{\mathcal{T}_0} a_k \xrightarrow{\alpha}_{\mathcal{T}_0} b \xrightarrow{\alpha}_{\mathcal{T}_0} b_1 \xrightarrow{\alpha}_{\mathcal{T}_0} \cdots \xrightarrow{\alpha}_{\mathcal{T}_0} b_i \xrightarrow{\beta}_{\mathcal{T}_0} c$$

This means that $a_1 \xrightarrow{\alpha^{k+1}}_{\mathcal{T}_k} b_1$ so $b_1 \in lfp(\lceil - \rceil_{\mathcal{T}_k})$ since $a_1 \in lfp(\lceil - \rceil_{\mathcal{T}_k})$.
By the step case assumption the maximum length of the transition chains in $lfp(\lceil - \rceil_{\mathcal{T}_k})$ from a_1 can be $n+1$ so maximum length for such chains from b_1 can be n. We know that $b \xrightarrow{\alpha}_{\mathcal{T}_0} b_1$.
To exploit the induction hypothesis we want to show that $b \in \lceil lfp(\lceil - \rceil_{\mathcal{T}_k}) \rceil_{\mathcal{T}_0}$ and that for all β if $b \xrightarrow{\beta} b'$ then the maximum length of any transition from b' is n.
We already know that if $\beta = \alpha$ then $b' = b_1 \in lfp(\lceil - \rceil_{\mathcal{T}_k})$ and the maximum length of any chain

from b_1 is n. We need to check what happens if $\beta \in A_{\mathcal{T}_0} - \{\alpha\}$. We know from the previous section that if b can make a transition from $A_{\mathcal{T}_0} - \{\alpha\}$, $b \xrightarrow{\beta}_{\mathcal{T}_0} b'$, the result is in $lfp(\lceil - \rceil_{\mathcal{T}_k})$ so $b \in \lceil lfp(\lceil - \rceil_{\mathcal{T}_k}) \rceil_{\mathcal{T}_0}$. Furthermore if $b \xrightarrow{\beta}_{\mathcal{T}_0} b'$ then $a_1 \xrightarrow{\alpha^k;\beta}_{\mathcal{T}_0} b'$ which means that the maximum length for chains from b' can be n.

By the induction hypothesis $b \in lfp(\lceil - \rceil_{\mathcal{T}_k})$.

$\underline{\alpha^{k+1}}$ We have the following chain of transitions

$$a \xrightarrow{\alpha}_{\mathcal{T}_0} a_1 \xrightarrow{\alpha}_{\mathcal{T}_0} \cdots \xrightarrow{\alpha}_{\mathcal{T}_0} a_k \xrightarrow{\alpha}_{\mathcal{T}_0} b \xrightarrow{\alpha}_{\mathcal{T}_0} b_1 \xrightarrow{\alpha}_{\mathcal{T}_0} \cdots \xrightarrow{\alpha}_{\mathcal{T}_0} b_k \xrightarrow{\alpha}_{\mathcal{T}_0} c$$

This means that $a_1 \xrightarrow{\alpha^{k+1}}_{\mathcal{T}_k} b_1$ so $b_1 \in lfp(\lceil - \rceil_{\mathcal{T}_k})$ since $a_1 \in lfp(\lceil - \rceil_{\mathcal{T}_k})$.

By the step case assumption the maximum length of the transition chains in $lfp(\lceil - \rceil_{\mathcal{T}_k})$ from a_1 can be $n+1$ so maximum length for such chains from b_1 can be n hence by similar arguments, involving the induction hypothesis, as used in the last section $b \in lfp(\lceil - \rceil_{\mathcal{T}_k})$.

Hence all the possible transitions from a in $A_{\mathcal{T}_k}$ are to members of $lfp(\lceil - \rceil_{\mathcal{T}_k})$ so by definition $a \in lfp(\lceil - \rceil_{\mathcal{T}_k})$. Since $a \in \lceil lfp(\lceil - \rceil_{\mathcal{T}_k}) \rceil_{\mathcal{T}_0}$ this means that $\lceil lfp(\lceil - \rceil_{\mathcal{T}_k}) \rceil_{\mathcal{T}_0} \subseteq lfp(\lceil - \rceil_{\mathcal{T}_k})$. □

Lemma 10.3 $\lceil lfp(\lceil - \rceil_{\mathcal{T}_0}) \rceil_{\mathcal{T}_k} \subseteq lfp(\lceil - \rceil_{\mathcal{T}_0})$

Proof. We conduct the proof by induction on k.

Base Case. If $k = 0$ then $\mathcal{T}_k = \mathcal{T}_0$ so clearly $\lceil lfp(\lceil - \rceil_{\mathcal{T}_0}) \rceil_{\mathcal{T}_k} \subseteq lfp(\lceil - \rceil_{\mathcal{T}_0})$

Step Case. Assume that for all $j < k$ $\lceil lfp(\lceil - \rceil_{\mathcal{T}_0}) \rceil_{\mathcal{T}_j} \subseteq lfp(\lceil - \rceil_{\mathcal{T}_0})$.

We are going to show that for some expression, $a \in \lceil lfp(\lceil - \rceil_{\mathcal{T}_0}) \rceil_{\mathcal{T}_k}$ (where $k \neq 0$), $a \in lfp(\lceil - \rceil_{\mathcal{T}_0})$. We do this by showing that all transitions from a in $\mathcal{A}_{\mathcal{T}_0}$ result in members of $lfp(\lceil - \rceil_{\mathcal{T}_0})$. In the course of this we will exploit the fact that we know that the result of any transition in $\mathcal{A}_{\mathcal{T}_k}$ from a is a member of $lfp(\lceil - \rceil_{\mathcal{T}_0})$.

As in the last proof we examine each possible $\beta \in A_{\mathcal{T}_0}$ in turn to check that if $a \xrightarrow{\beta}_{\mathcal{T}_0} b$ then $b \in lfp(\lceil - \rceil_{\mathcal{T}_0})$. The transitions in this case are $\beta \in A_{\mathcal{T}_0} - \{\alpha\}$ and α.

No Transitions If no transitions apply to a then $a \equiv \bot_{\mathcal{T}_0} \in lfp(\lceil - \rceil_{\mathcal{T}_0})$.

$\underline{a \xrightarrow{\beta}_{\mathcal{T}_0} b \text{ **where** } \beta \in A_{\mathcal{T}_0} - \{\alpha\}}$ If $\beta \in A_{\mathcal{T}_0} - \{\alpha\}$ then $\beta \in A_{\mathcal{T}_k}$, $a \xrightarrow{\beta}_{\mathcal{T}_k} b$ so b is in $lfp(\lceil - \rceil_{\mathcal{T}_0})$ since $a \in \lceil lfp(\lceil - \rceil_{\mathcal{T}_0}) \rceil_{\mathcal{T}_k}$

$a \xrightarrow{\alpha}_{\mathcal{T}_0} b$ We will show that $b \in \lceil lfp(\lceil - \rceil_{\mathcal{T}_0}) \rceil_{\mathcal{T}_{k-1}}$ and hence, by the induction hypothesis, that $b \in lfp(\lceil - \rceil_{\mathcal{T}_0})$

We need to show that for every transition, β, in $\mathcal{A}_{\mathcal{T}_{k-1}}$ if $b \xrightarrow{\beta}_{\mathcal{T}_{k-1}} c$ then $c \in lfp(\lceil - \rceil_{\mathcal{T}_0})$. We shall break the transitions down slightly differently from in the previous proof. Instead of $\beta \in A_{\mathcal{T}_0} - \{\alpha\}$, $\alpha^i.\beta$ $(0 < i \leq k-1,\ \beta \in A_{\mathcal{T}_0} - \{\alpha\})$ and α^k we shall examine $\alpha^i.\beta$ $(0 \leq i < k-1,\ \beta \in A_{\mathcal{T}_0} - \{\alpha\})$ – so this transition now includes $\beta \in A_{\mathcal{T}_0} - \{\alpha\}$ alone as well as a chain of αs followed by β – and $\alpha^{k-1}.\beta$ (where it is possible that $\beta = \alpha$).

No Transitions If b can make no transitions in $\mathcal{A}_{\mathcal{T}_{k-1}}$ then it is trivially in $\lceil lfp(\lceil - \rceil_{\mathcal{T}_0}) \rceil_{\mathcal{T}_{k-1}}$

$b \xrightarrow{\alpha^i.\beta}_{\mathcal{T}_{k-1}} c$ **where** $0 \leq i < k-1$ **and** $\beta \in A_{\mathcal{T}_0} - \{\alpha\}$ In this case we have the following chain of transitions:

$$a \xrightarrow{\alpha}_{\mathcal{T}_0} b \xrightarrow{\alpha}_{\mathcal{T}_0} b_1 \xrightarrow{\alpha}_{\mathcal{T}_0} \cdots \xrightarrow{\alpha}_{\mathcal{T}_0} b_i \xrightarrow{\beta}_{\mathcal{T}_0} c$$

Let $j = i + 1$. Since $0 \leq i < k-1$ this means that $0 < j < k$ so $\alpha^j.\beta \in \mathcal{A}_{\mathcal{T}_k}$ and $a \xrightarrow{\alpha^j.\beta}_{\mathcal{T}_k} c$. Since $a \in \lceil lfp(\lceil - \rceil_{\mathcal{T}_0}) \rceil_{\mathcal{T}_k}$ this means $c \in lfp(\lceil - \rceil_{\mathcal{T}_0})$.

$b \xrightarrow{\alpha^{k-1}.\beta}_{\mathcal{T}_{k-1}} c$ **(where it is possible that** $\beta = \alpha$**)** In this case we have the following chain of transitions:

$$a \xrightarrow{\alpha}_{\mathcal{T}_0} b \xrightarrow{\alpha}_{\mathcal{T}_0} b_1 \xrightarrow{\alpha}_{\mathcal{T}_0} \cdots \xrightarrow{\alpha}_{\mathcal{T}_0} b_{k-1} \xrightarrow{\beta}_{\mathcal{T}_0} c$$

Clearly $a \xrightarrow{\alpha^k.\beta}_{\mathcal{T}_k} c$ (if $\beta = \alpha$ this means $a \xrightarrow{\alpha^{k+1}}_{\mathcal{T}_k} c$). Since $a \in \lceil lfp(\lceil - \rceil_{\mathcal{T}_0}) \rceil_{\mathcal{T}_k}$ this means $c \in lfp(\lceil - \rceil_{\mathcal{T}_0})$.

So $b \in \lceil lfp(\lceil - \rceil_{\mathcal{T}_0}) \rceil_{\mathcal{T}_{k-1}}$ because all possible transitions $\mathcal{A}_{\mathcal{T}_{k-1}}$ from b result in $c \in lfp(\lceil - \rceil_{\mathcal{T}_0})$. Hence $b \in lfp(\lceil - \rceil_{\mathcal{T}_0})$ by the induction hypothesis.

Since all the possible transitions in $\mathcal{A}_{\mathcal{T}_0}$ from a result in members of $lfp(\lceil - \rceil_{\mathcal{T}_0})$ $a \in lfp(\lceil - \rceil_{\mathcal{T}_0})$. So $\lceil lfp(\lceil - \rceil_{\mathcal{T}_0}) \rceil_{\mathcal{T}_k} \subseteq lfp(\lceil - \rceil_{\mathcal{T}_0})$. □

Theorem 10.7 *If* $\{\mathcal{T}_0, \cdots, \mathcal{T}_n\}$ *is* α*–well–chained then for all* $k \leq n$

$$lfp(\lceil - \rceil_{\mathcal{T}_0}) = lfp(\lceil - \rceil_{\mathcal{T}_k})$$

Proof. By 10.2 $\lceil lfp(\lceil - \rceil_{\mathcal{T}_k}) \rceil_{\mathcal{T}_0} \subseteq lfp(\lceil - \rceil_{\mathcal{T}_k}$ so by coinduction $lfp(\lceil - \rceil_{\mathcal{T}_0}) \subseteq lfp(\lceil - \rceil_{\mathcal{T}_k})$.

This result allows us to modify labelled transition systems by chaining to allow a proof to go through without affecting the theorem. Consider the two proofs given for the following example. This should be familiar as the example used to discuss the Induction Revision Critic in chapter 4.

Example 10.2 $\forall l_1 : list(\tau), l_2 : list(\tau).even(length(l_1 <> l_2)) \Leftrightarrow even(length(l_1) + length(l_2))$

I shall present two partial proofs of this theorem one with an extended transition system to $\mathcal{T}$ and one using $\mathcal{T}$ but a larger set than is used in the first proof.

Partial Proof 1

Let $\mathcal{T}^*$ be a labelled transition system containing the transitions of $\mathcal{T}$ with `tl.nil`, `tl.hd` and `tl.tl` in place of `tl`. $\{\mathcal{T}, \mathcal{T}^*\}$ is `tl`–well–chained . Therefore $lfp(\lceil - \rceil_{\mathcal{T}}) = lfp(\lceil - \rceil_{\mathcal{T}^*})$.

Let $\mathcal{S} = \{l_1 \mid l_1 : list(\tau) \Rightarrow even(length(l_1 <> l_2)) \Leftrightarrow even(length(l_1) + length(l_2))\}$

Using rule (10.22) we get two goals:

$$l \xrightarrow{\alpha} \phi \Rightarrow (\phi \in \mathcal{S} \Rightarrow l \in \mathcal{S})$$

$$\mathcal{S} \subseteq \{l \mid l_1 : list(\tau) \Rightarrow even(length(l <> l_2)) \Leftrightarrow even(length(l) + length(l_2))\}$$

Consider the first of these. We want to consider all possible transitions in the system to objects of list type. These are $\xrightarrow{\texttt{nil}}$, $\xrightarrow{\texttt{tl.nil}}$ and $\xrightarrow{\texttt{tl.tl}}$ therefore there are three subgoals, one for each of these transitions.

1. $\alpha = \texttt{nil} \Rightarrow l = nil$

$$even(length(nil <> l_2)) \Leftrightarrow even(length(nil) + length(l_2))$$

2. $\alpha = \texttt{tl.nil} \Rightarrow l = h :: nil$

$$even(length(h :: nil <> l_2)) \Leftrightarrow even(length(h :: nil) + length(l_2))$$

3. $\alpha = \texttt{tl.tl} \Rightarrow l = h_1 :: h_2 :: t$

$$\begin{gathered} even(length(t <> l_2)) \Leftrightarrow even(length(t) + length(l_2)) \Rightarrow \\ even(length(h_1 :: h_2 :: t <> l_2)) \Leftrightarrow \\ even(length(h_1 :: h_2 :: t) + length(l_2)) \end{gathered}$$

These should be the familiar goals expected for the use of the induction scheme (10.25).

The second goal follows trivially from the definition of $\mathcal{S}$. $\Box$

This corresponds to the choice of the induction scheme

$$\frac{P(nil) \quad P(h :: nil) \quad P(t) \vdash P(h_1 :: h_2 :: t)}{\forall x.P(x)} \tag{10.25}$$

because of the structure that the transitions imply for l.

Partial Proof 2

An alternative suggested by the common practice in coinductive proofs would be to chose the set

$$\mathcal{S} = \{l_1 \mid l_1 : list(\tau) \Rightarrow even(length(l_1 <> l_2)) \Leftrightarrow even(length(l_1) + length(l_2))\} \cap \{l_1 \mid l_1 : list(\tau) \Rightarrow even(s(length(l_1 <> l_2))) \Leftrightarrow even(s(length(l_1) + length(l_2))))\}$$

Notice that the $\cup$ which appears in similar situations in coinductive proofs has here changed to the dual $\cap$.

Using this set we get two goals:

$$l \xrightarrow{\alpha} \phi \Rightarrow (\phi \in \mathcal{S}) \Rightarrow l \in \mathcal{S} \tag{10.26}$$

$$\mathcal{S} \subseteq \{l \mid l : list(\tau) \Rightarrow even(length(l <> l_2)) \Leftrightarrow even(length(l) + length(l_2))\} \tag{10.27}$$

If l_1 is a list then there are two possible list to list transitions, $\alpha = \texttt{nil}$ or $\texttt{tl}$.

1. If $\alpha = \texttt{nil}$ then $even(length(\bot <> l_2))$ and $even(length(\bot) + length(l_2))$ both diverge. The hypothesis is trivially true leaving the goal

$$even(length(nil <> l_2)) \Leftrightarrow even(length(nil) + length(l_2)) \wedge$$
$$even(s(length(nil <> l_2))) \Leftrightarrow even(s(length(nil) + length(l_2)))$$

   both of which conjuncts evaluate to true like the two base cases used in the first proof.

2. If $\alpha = \texttt{tl}$, we can create two step cases (for each of the schema in $\mathcal{S}$)

$$\begin{gathered} even(length(t <> l_2)) \Leftrightarrow even(length(t) + length(l_2)) \wedge \\ even(s(length(t) <> l_2)) \Leftrightarrow even(s(length(t) + length(l_2))) \Rightarrow \\ even(length(h :: t <> l_2)) \Leftrightarrow even(length(h :: t) + length(l_2)) \end{gathered} \tag{10.28}$$

$$\begin{gathered} even(length(t <> l_2)) \Leftrightarrow even(length(t) + length(l_2)) \wedge \\ even(s(length(t) <> l_2)) \Leftrightarrow even(s(length(t) + length(l_2))) \Rightarrow \\ even(s(length(h :: t) <> l_2)) \Leftrightarrow even(s(length(h :: t) + length(l_2))) \end{gathered} \tag{10.29}$$

   both of which are provable because of the conjunction in the hypothesis.

This leaves (10.27) which is trivial since $\mathcal{S} \subseteq \{l \mid l : list(\tau) \Rightarrow even(length(l <> l_2)) \Leftrightarrow even(length(l) + length(l_2))\}$. $\square$

By 10.3 $\lceil lfp(\lceil - \rceil_{\mathcal{T}_0}) \rceil_{\mathcal{T}_k} \subseteq lfp(\lceil - \rceil_{\mathcal{T}_0}$ so by coinduction $lfp(\lceil - \rceil_{\mathcal{T}_k}) \subseteq lfp(\lceil - \rceil_{\mathcal{T}_0})$.

Hence $lfp(\lceil - \rceil_{\mathcal{T}_k}) = lfp(\lceil - \rceil_{\mathcal{T}_0})$. $\square$

10.5.2 Interchangeability in Coinduction

The result about the equivalence of least fixedpoints of well–chained transition systems can be extended in a modified form to greatest fixedpoints. It turns out that α–well–chainedness alone is not sufficient in greatest fixedpoints to guarantee equivalence. This is illustrated in example 10.3.

Example 10.3 *Consider the transitions systems $\mathcal{N}_0$ and $\mathcal{N}_1$ shown in figures 10–5 and 10–6. $\{\mathcal{N}_0, \mathcal{N}_1\}$ is* p*–well–chained. However $gfp(\langle - \rangle_{\mathcal{N}_0})$ is not equivalent*

$$\frac{}{0 \stackrel{\mathtt{0}}{\rightarrow} \perp}$$

$$\frac{}{s(n) \stackrel{\mathtt{p}}{\rightarrow} n}$$

$$\frac{a : \tau \quad \tau \not\equiv \tau_1 \rightarrow \tau_2 \quad a \stackrel{\text{red}}{\rightsquigarrow} a' \quad a' \stackrel{\alpha}{\rightarrow} b}{a \stackrel{\alpha}{\rightarrow} b}$$

Figure 10–5: $\mathcal{N}_0$

$$\frac{}{0 \stackrel{\mathtt{0}}{\rightarrow} \perp}$$

$$\frac{}{s(0) \stackrel{\mathtt{p.0}}{\rightarrow} \perp}$$

$$\frac{}{s(s(n)) \stackrel{\mathtt{p.p}}{\rightarrow} n}$$

$$\frac{a : \tau \quad \tau \not\equiv \tau_1 \rightarrow \tau_2 \quad a \stackrel{\text{red}}{\rightsquigarrow} a' \quad a' \stackrel{\alpha}{\rightarrow} b}{a \stackrel{\alpha}{\rightarrow} b}$$

Figure 10–6: $\mathcal{N}_1$

to $gfp(\langle - \rangle_{\mathcal{N}_1})$. Consider $\langle p(\perp), \perp \rangle$ this pair is not in $gfp(\langle - \rangle_{\mathcal{N}_0})$ since $p(\perp)$ can make a p *transition whilst $\perp$ can not make any transitions. However the pair is in $gfp(\langle - \rangle_{\mathcal{N}_1})$ since neither $p(\perp)$ nor $\perp$ can make a transition in $A_{\mathcal{N}_1}$.*

However, it is possible to add a new notion of *guardedness* which will give equivalence.

Definition 10.3 *A transition* α *is* **guarded** *in a transition system* $\mathcal{T}$ *if* $\exists \alpha_g.\forall a.\exists a_1.\ a \stackrel{\alpha}{\rightarrow} a_1 \Leftrightarrow \exists a_2.\ a \stackrel{\alpha_g}{\rightarrow} a_2$.

Example 10.4 *Consider the transitions systems* $\mathcal{L}_0$ *and* $\mathcal{L}_1$ *shown in figures 10–7 and 10–8.* $\{\mathcal{L}_0, \mathcal{L}_1\}$ *is* `tl`*–well–chained. What is more* `tl` *is guarded by* `hd` *in* $\mathcal{L}_0$*.*

$$\frac{}{nil \stackrel{\texttt{nil}}{\rightarrow} \bot}$$

$$\frac{}{h :: t \stackrel{\texttt{hd}}{\rightarrow} h}$$

$$\frac{}{h :: t \stackrel{\texttt{tl}}{\rightarrow} t}$$

$$\frac{a : \tau \quad \tau \not\equiv \tau_1 \rightarrow \tau_2 \quad a \stackrel{\text{red}}{\rightsquigarrow} a' \quad a' \stackrel{\alpha}{\rightarrow} b}{a \stackrel{\alpha}{\rightarrow} b}$$

Figure 10–7: $\mathcal{L}_0$

Say we try to construct a similar counterexample as that in example 10.3. So we consider the pair $\langle h :: \bot, \bot\rangle$*. As in example 10.3 this pair in not in* $gfp(\langle - \rangle_{\mathcal{L}_0})$ *since* $h :: \bot$ *can make a* `tl` *transition whilst* $\bot$ *can not make any transitions. However the pair isn't in* $gfp(\langle - \rangle_{\mathcal{L}_1})$ *either since although neither* $h :: \bot$ *nor* $\bot$ *can make a* `tl.nil`*,* `tl.hd` *or* `tl.tl` *transition* $h :: \bot$ *can make a* `hd` *transition which* $\bot$ *can not.*

We show that if α is guarded in $\mathcal{T}_0$ then the greatest fixedpoints of the transition systems in any α–well–chained list from $\mathcal{T}_0$ are equivalent.

The general shape of the proofs here follows that of the proofs for the equivalence of the least fixedpoints of $\lceil - \rceil$. We use the coinduction rule and try to show that if $\{\mathcal{T}_0, \cdots, \mathcal{T}_n\}$ is α–well–chained then for all $0 \leq k \leq n$, $gfp(\langle - \rangle_{\mathcal{T}_k}) \subseteq \langle gfp(\langle - \rangle_{\mathcal{T}_k})\rangle_{\mathcal{T}_0}$ and $gfp(\langle - \rangle_{\mathcal{T}_0}) \subseteq \langle gfp(\langle - \rangle_{\mathcal{T}_0})\rangle_{\mathcal{T}_k}$. This is done mainly by the analysis of the transitions according to definition 10.1. But in one place it involves an intermediate coinduction. We shall also exploit a lemma (10.4) about guardedness.

Lemma 10.4 *Suppose* $\{\mathcal{T}_0, \cdots, \mathcal{T}_n\}$ *is* α*–well–chained and* α *is guarded by* α_g *in* $\mathcal{T}_0$*. If, for some* k*,* $\langle a, b\rangle \in gfp(\langle - \rangle_{\mathcal{T}_k})$ *then* $a \stackrel{\alpha}{\rightarrow}_{\mathcal{T}_0} a'$ *implies* $\exists b'.b \stackrel{\alpha}{\rightarrow}_{\mathcal{T}_0} b'$ *(and vice versa).*

$$\frac{}{0 \xrightarrow{\texttt{nil}} \bot}$$

$$\frac{}{h :: nil \xrightarrow{\texttt{tl.nil}} \bot}$$

$$\frac{}{h :: t \xrightarrow{\texttt{hd}} h}$$

$$\frac{}{h_1 :: h_2 :: t \xrightarrow{\texttt{tl.tl}} t}$$

$$\frac{}{h_1 :: h_2 :: t \xrightarrow{\texttt{tl.hd}} h_2}$$

$$\frac{a : \tau \qquad \tau \not\equiv \tau_1 \to \tau_2 \qquad a \overset{\text{red}}{\rightsquigarrow} a' \qquad a' \xrightarrow{\alpha} b}{a \xrightarrow{\alpha} b}$$

Figure 10–8: $\mathcal{L}_1$

Proof. Guardedness implies that if $a \xrightarrow{\alpha}_{\mathcal{T}_0} a'$ there is some a'' such that $a \xrightarrow{\alpha_g}_{\mathcal{T}_0} a''$. Since $\alpha_g \in \mathcal{A}_{\mathcal{T}_0} - \{\alpha\}$ this means that $a \xrightarrow{\alpha_g}_{\mathcal{T}_k} a''$. Since $\langle a,\, b\rangle \in gfp(\langle - \rangle_{\mathcal{T}_k})$ then $\exists b''.b \xrightarrow{\alpha_g}_{\mathcal{T}_k} b''$ and hence $b \xrightarrow{\alpha_g}_{\mathcal{T}_0} b''$. Since α_g is guarded by α in $\mathcal{T}_0$ then $\exists b'.b \xrightarrow{\alpha}_{\mathcal{T}_0} b'$. Exactly the same argument can be used to show that if $b \xrightarrow{\alpha}_{\mathcal{T}_0} b'$ then $\exists a'.a \xrightarrow{\alpha}_{\mathcal{T}_0} a'$.
□

Lemma 10.5 *If α is guarded by α_g in $\mathcal{T}_0$ then*

$$gfp(\langle - \rangle_{\mathcal{T}_k}) \subseteq \langle gfp(\langle - \rangle_{\mathcal{T}_k})\rangle_{\mathcal{T}_0}$$

Proof. Let $\langle a,\, b\rangle \in gfp(\langle - \rangle_{\mathcal{T}_k})$.

We shall consider all the possible transitions in $A_{\mathcal{T}_0}$ that a can make to ensure that they are matched by a transition from b and the resulting pair are in $gfp(\langle - \rangle_{\mathcal{T}_k})$ and similarly that if b can make a transition it is matched by one from a.

$\underline{a \xrightarrow{\beta}_{\mathcal{T}_0} a' \textbf{ where } \beta \in A_{\mathcal{T}_0} - \{\alpha\}}$ We know that $\beta \in A_{\mathcal{T}_k}$ so $a \xrightarrow{\beta}_{\mathcal{T}_k} a'$.
Since $\langle a,\, b\rangle \in gfp(\langle - \rangle_{\mathcal{T}_k})$ there is some b' such that $b \xrightarrow{\beta}_{\mathcal{T}_k} b'$ and $\langle a',\, b'\rangle \in gfp(\langle - \rangle_{\mathcal{T}_k})$. Similarly we can show that if $b \xrightarrow{\beta}_{\mathcal{T}_k} b'$ then $\exists a'.a \xrightarrow{\beta}_{\mathcal{T}_k} a'$ and $\langle a',\, b'\rangle \in gfp(\langle - \rangle_{\mathcal{T}_k})$.

$\underline{a \xrightarrow{\alpha}_{\mathcal{T}_0} a'}$ By lemma 10.4 we know $\exists b'.b \xrightarrow{\alpha}_{\mathcal{T}_0} b'$.

It now remains to show that $\langle a',\, b'\rangle \in gfp(\langle - \rangle_{\mathcal{T}_k})$. This is done by coinduction. Let

$$\mathcal{R} = \{\langle a_1,\, b_1\rangle \mid \exists a, b.\langle a,\, b\rangle \in gfp(\langle - \rangle_{\mathcal{T}_k}) \wedge a \xrightarrow{\alpha}_{\mathcal{T}_0} a_1 \wedge b \xrightarrow{\alpha}_{\mathcal{T}_0} b_1\} \\ \cup\, gfp(\langle - \rangle_{\mathcal{T}_k})$$

Clearly $\langle a',\, b'\rangle \in \mathcal{R}$. We want to show that $\mathcal{R}$ is a $\langle - \rangle_{\mathcal{T}_k}$–bisimulation. Clearly $gfp(\langle - \rangle_{\mathcal{T}_k})$ is a $\langle - \rangle_{\mathcal{T}_k}$–bisimulation so we only need to consider the transitions from pairs in

$$\{\langle a_1,\, b_1\rangle \mid \exists a, b.\langle a,\, b\rangle \in gfp(\langle - \rangle_{\mathcal{T}_k}) \wedge a \xrightarrow{\alpha}_{\mathcal{T}_0} a_1 \wedge b \xrightarrow{\alpha}_{\mathcal{T}_0} b_1\}$$

This is done by analysis of the transitions from some arbitrary pair, $\langle a_1,\, b_1\rangle \in \mathcal{R}$. We split these transitions as $\alpha^i.\beta$ where $\beta \in A_{\mathcal{T}_0} - \{\alpha\}$ and $0 \leq i < k$ and $\alpha^k.\beta$ where $\beta \in A_{\mathcal{T}_0}$ (possibly equal to α).

<u>$a_1 \xrightarrow{\alpha^i.\beta}_{\mathcal{T}_k} a_1'$ **where** $\beta \in A_{\mathcal{T}_0} - \{\alpha\}$ **and** $0 \leq i < k$</u> This means that $\alpha^{i+1}.\beta \in A_{\mathcal{T}_k}$. We know there are $\langle a,\, b\rangle \in gfp(\langle - \rangle_{\mathcal{T}_k})$ such that $a \xrightarrow{\alpha}_{\mathcal{T}_0} a_1$ and $b \xrightarrow{\alpha}_{\mathcal{T}_0} b_1$. This implies that $a \xrightarrow{\alpha^{i+1}.\beta}_{\mathcal{T}_k} a_1'$ which in turn implies there is some b_1' such that $b \xrightarrow{\alpha^{i+1}.\beta}_{\mathcal{T}_k} b_1'$ and $\langle a_1',\, b_1'\rangle \in gfp(\langle - \rangle_{\mathcal{T}_k})$. So $\exists b_1'.b_1 \xrightarrow{\alpha^i.\beta}_{\mathcal{T}_k} b_1'$ and $\langle a_1', b_1'\rangle \in gfp(\langle - \rangle_{\mathcal{T}_k})$, hence $\langle a_1',\, b_1'\rangle \in \mathcal{R}$.

<u>$a_1 \xrightarrow{\alpha^k.\beta} a_1'$ **where** $\beta \in A_{\mathcal{T}_0}$ **(so** β **could equal** α**)**</u> We know there are $\langle a,\, b\rangle \in gfp(\langle - \rangle_{\mathcal{T}_k})$ such that $a \xrightarrow{\alpha}_{\mathcal{T}_0} a_1$ and $b \xrightarrow{\alpha}_{\mathcal{T}_0} b_1$. Consider the chains of transitions:

$$\begin{array}{l} a \xrightarrow{\alpha}_{\mathcal{T}_0} a_1 \xrightarrow{\alpha}_{\mathcal{T}_0} \cdots \xrightarrow{\alpha}_{\mathcal{T}_0} a_{k+1} \xrightarrow{\beta}_{\mathcal{T}_0} a_1' \\ b \xrightarrow{\alpha}_{\mathcal{T}_0} b_1 \xrightarrow{?}_{\mathcal{T}_0} \cdots ? \end{array}$$

Clearly $a \xrightarrow{\alpha^{k+1}}_{\mathcal{T}_k} a_{k+1}$ this means there is some b_{k+1} such that $b \xrightarrow{\alpha^{k+1}}_{\mathcal{T}_k} b_{k+1}$ and that $\langle a_{k+1}, b_{k+1}\rangle \in gfp(\langle - \rangle_{\mathcal{T}_k})$. If $\beta \in A_{\mathcal{T}_0} - \{\alpha\}$ then $\beta \in A_{\mathcal{T}_k}$ so there is some b_1' such that $b_{k+1} \xrightarrow{\beta}_{\mathcal{T}_k} b_1'$ and $\langle a_1',\, b_1'\rangle \in gfp(\langle - \rangle_{\mathcal{T}_k})$ hence $\langle a_1',\, b_1'\rangle \in \mathcal{R}$.
Else $\beta = \alpha$, by lemma 10.4 we know that there is some b_1' such that $b_{k+1} \xrightarrow{\alpha}_{\mathcal{T}_0} b_1'$. $\langle a_1',\, b_1'\rangle \in \mathcal{R}$ by the definition of $\mathcal{R}$.

Similarly if $\exists b_2.b_1 \xrightarrow{\beta}_{\mathcal{T}_k} b_2$ then $\exists a_2.a_1 \xrightarrow{\beta}_{\mathcal{T}_k} a_2$ and $\langle a_2,\, b_2\rangle \in \mathcal{R}$. Hence $\mathcal{R}$ is a $\langle - \rangle_{\mathcal{T}_k}$–bisimulation. This means that $\mathcal{R} \subseteq gfp(\langle - \rangle_{\mathcal{T}_k})$ so since $\langle a',\, b'\rangle \in \mathcal{R}$, $\langle a',\, b'\rangle \in gfp(\langle - \rangle_{\mathcal{T}_k})$.

Similarly if $\exists b'.b \xrightarrow{\beta}_{\mathcal{T}_k} b'$ then $\exists a'.a \xrightarrow{\beta}_{\mathcal{T}_k} a'$ and $\langle a',\, b'\rangle \in \mathcal{R}$ so $\langle a',\, b'\rangle \in gfp(\langle - \rangle_{\mathcal{T}_k})$.

Hence $gfp(\langle - \rangle_{\mathcal{T}_0}) \subseteq \langle gfp(\langle - \rangle_{\mathcal{T}_0})\rangle_{\mathcal{T}_k}$. $\square$

Lemma 10.6 *If α is guarded by α_g in $\mathcal{T}_0$ then*

$$gfp(\langle - \rangle_{\mathcal{T}_0}) \subseteq \langle gfp(\langle - \rangle_{\mathcal{T}_0}) \rangle_{\mathcal{T}_k}$$

Proof. Let $\langle a,\, b \rangle \in gfp(\langle - \rangle_{\mathcal{T}_0})$.

We consider all the possible transitions in $A_{\mathcal{T}_k}$ that a can make to ensure that they are matched by a transition from b and the resulting pair are in $gfp(\langle - \rangle_{\mathcal{T}_0})$

All the transitions in $A_{\mathcal{T}_k}$ are of the form $a \stackrel{\alpha^i.\beta}{\rightarrow} a'$ where $0 \leq i \leq k$ and $\beta \in A_{\mathcal{T}_0}$ (so β could equal α) This means there is a chain of transitions in $A_{\mathcal{T}_0}$ such that

$$a \stackrel{\alpha}{\rightarrow}_{\mathcal{T}_0} a_1 \stackrel{\alpha}{\rightarrow}_{\mathcal{T}_0} \cdots \stackrel{\alpha}{\rightarrow}_{\mathcal{T}_0} a_i \stackrel{\beta}{\rightarrow}_{\mathcal{T}_0} a'$$

It is a simple matter to show by induction that there must be a similar sequence of transitions

$$b \stackrel{\alpha}{\rightarrow}_{\mathcal{T}_0} b_1 \stackrel{\alpha}{\rightarrow}_{\mathcal{T}_0} \cdots \stackrel{\alpha}{\rightarrow}_{\mathcal{T}_0} b_i \stackrel{\beta}{\rightarrow}_{\mathcal{T}_0} b'$$

Base Case. If $i = 0$ then $a \stackrel{\beta}{\rightarrow}_{\mathcal{T}_0} a'$, since $\langle a,\, b \rangle \in gfp(\langle - \rangle_{\mathcal{T}_0})$ there must be a b' such that $b \stackrel{\beta}{\rightarrow}_{\mathcal{T}_0} b'$ and $\langle a',\, b' \rangle \in gfp(\langle - \rangle_{\mathcal{T}_0})$

Step Case. Assume that for all $i < j$ if $a \stackrel{\alpha^i.\beta}{\rightarrow}_{\mathcal{T}_k} a'$ then there is some b' such that $b \stackrel{\alpha^i.\beta}{\rightarrow}_{\mathcal{T}_k} b'$ and $\langle a',\, b' \rangle \in gfp(\langle - \rangle_{\mathcal{T}_0})$. Suppose $a \stackrel{\alpha^j.\beta}{\rightarrow}_{\mathcal{T}_k} a'$ this means there is a sequence of transitions

$$a \stackrel{\alpha}{\rightarrow}_{\mathcal{T}_0} a_1 \stackrel{\alpha}{\rightarrow}_{\mathcal{T}_0} \cdots \stackrel{\alpha}{\rightarrow}_{\mathcal{T}_0} a_j \stackrel{\beta}{\rightarrow}_{\mathcal{T}_0} a'$$

Now $a \stackrel{\alpha^{j-1}.\alpha}{\rightarrow}_{\mathcal{T}_k} a_j$ so by the induction hypothesis there must be some b_j such that $b \stackrel{\alpha^{j-1}.\alpha}{\rightarrow}_{\mathcal{T}_k} b_j$ and $\langle a_j,\, b_j \rangle \in gfp(\langle - \rangle_{\mathcal{T}_0})$. Since $a_j \stackrel{\alpha}{\rightarrow}_{\mathcal{T}_0} a'$ there must be some b' such that $b_j \stackrel{\alpha}{\rightarrow}_{\mathcal{T}_0} b'$ and $\langle a',\, b' \rangle \in gfp(\langle - \rangle_{\mathcal{T}_0})$.

This same analysis works to show that any transition in $A_{\mathcal{T}_k}$ which b can make can be matched by a transition from a and that the resulting pair are in $gfp(\langle - \rangle_{\mathcal{T}_0})$.

Hence $gfp(\langle - \rangle_{\mathcal{T}_0}) \subseteq \langle gfp(\langle - \rangle_{\mathcal{T}_0}) \rangle_{\mathcal{T}_k}$. $\square$

Theorem 10.8 *If $\{\mathcal{T}_0, \cdots, \mathcal{T}_n\}$ is α–well–chained and α is guarded in $\mathcal{T}_0$ then for all $k \leq n$*

$$gfp(\langle - \rangle_{\mathcal{T}_0}) = gfp(\langle - \rangle_{\mathcal{T}_k})$$

Proof. By 10.5 $gfp(\langle - \rangle_{\mathcal{T}_k}) \subseteq \langle gfp(\langle - \rangle_{\mathcal{T}_k}) \rangle_{\mathcal{T}_0}$ so by coinduction $gfp(\langle - \rangle_{\mathcal{T}_k}) \subseteq gfp(\langle - \rangle_{\mathcal{T}_0})$.

By 10.6 $gfp(\langle - \rangle_{\mathcal{T}_0}) \subseteq \langle gfp(\langle - \rangle_{\mathcal{T}_0}) \rangle_{\mathcal{T}_k}$ so by coinduction $gfp(\langle - \rangle_{\mathcal{T}_0}) \subseteq gfp(\langle - \rangle_{\mathcal{T}_k})$.

Hence $gfp(\langle - \rangle_{\mathcal{T}_k}) = gfp(\langle - \rangle_{\mathcal{T}_0})$. $\square$

It seems that both proof principles require a eureka step at the point where the proof rule is applied. There is a choice in both situations between choosing a set and choosing a function (labelled transition system). In Coinduction we want to choose a set that contains the set we are interested in (from the premise $\{\langle x,\, y\rangle\} \subseteq \mathcal{R}$ where $\mathcal{R}$ is the chosen set). In induction we want to choose a set that is contained in the set we're interested in (from the premise $\mathcal{S}' \subseteq \{x \mid P(x)\}$ where $\mathcal{S}'$ is the chosen set).

10.6 Choice of Induction Scheme and Evaluation

There also seems to be a comparison between the choice of induction scheme and evaluation. The induction scheme always involves case analysis between one or more base cases and a step case. Evaluation often chooses to case split a variable. Moreover the proof comparisions suggest that if the proof goes through given a casesplit on some variable x using one proof principle then it will go through, given a case split on x using the other proof principle and the process of the proofs after these casesplits are related in some way.

Since transitions only act on values it is easy to see (using $\lceil - \rceil$) that the induction rule will induce a casesplit on a cover set of values. Evaluate on the other hand seeks to find the values of the expression for all values of its arguments. This will also sometimes introduce a casesplit over a cover set of the values of those arguments. The reasons for the apparent equivalence of the subsequent proofs is argued in the next section.

10.7 Rewriting and Transitions

One startling fact to be observed in the tabular comparison of the proofs is that the same sequence of rewrite rules is used in both the inductive and coinductive proofs. This strongly suggests some link between these processes.

Consider the two examples which started this chapter. In the first, rewriting moved the constructor outwards until it was lost due to cancellation. In the second example, rewriting terminated with the loss of the successor constructor in the first argument of *nth*. The rewrite rules used before cancellation/*nth* were the same as those used before taking transitions in coinduction. In proofs where rewriting takes place after transitions are taken (none of which were included in the examples at the start of this chapter, however the patterns can be seen in appendix F) the rules are also the same as those after cancellation/*nth*.

Looking back to the unified presentation it is clear that in coinduction, rewriting is used to determine transitions whereas in induction the transitions are assumed to start with and all the rewriting process is concentrated on fertilization. The apparent link between the processes of cancellation, the *nth* rules and taking transitions comes about because, in the transition systems we are examining, the transitions correspond to the use of destructors[5]. Cancellation rules that strip away constructor functions are doing something similar and *nth* is explicitly examining successive destructor steps.

Hence inductive proofs that involve the cancellation of constructors or *nth* will contain the same set of rewriting steps to bring those constructors to the top of the expressions as coinductive proofs. Inductive proofs that rely on weak fertilization or the sinking of differences (without the use of cancellation) do not easily translate into coinductive proofs since they involve considering divergence (as no transitions can be found) or generalisations. Similarly inductive proofs in which non–constructors are cancelled proceed differently from the coinductive proofs.

This also explains why the choice of induction variable(s) and case split variable(s) in the two proof styles are the same since these introduce constructors to be moved outwards by rewriting. If Rippling were to be used instead of Evaluate then this outward movement would be more obvious.

10.8 Generalisation

To extend the process of comparison it is worth considering that the use of larger bisimulations is not always equivalent to the use of a more complex induction scheme. The only alternative form of bisimulation that has been considered in this thesis has been using generalised expressions in the bisimulation. This is equivalent to generalising the goal in induction.

Example 10.5 *Recall example 3.7 from chapter 3 that was used to motivate the Revise Bisimulation Critic in chapter 6.*

$$\forall f, x.\, h(f,x) \sim iterates(f,x)$$

where

$$h(F,X) \overset{\text{red}}{\leadsto} map(F, h(F,X))$$

$$iterates(F,X) \overset{\text{red}}{\leadsto} X :: iterates(F, F(X))$$

[5] It would also be possible to compare coinduction to destructor–style induction, however this would really require some form of destructor–style coinduction in which destructors/transitions were involved in the function definitions, e.g. $L \neq nil \Rightarrow map(F,L) \overset{\mathtt{tl}}{\rightarrow} map(F,T) \leadsto L \overset{\mathtt{tl}}{\rightarrow} T$ so that the rewrite rules employed in the proofs could be correctly compared. Since this thesis has concentrated on coinduction presented in a constructor style a comparison with destructor–style induction has not been attempted

To prove this coinductively it is sufficient to use the bisimulation

$$\{\langle map(F)^N(h(F,X)),\ iterates(F, F^N(X))\rangle$$

To prove this inductively using nth it is also sufficient to generalise the conjecture to

$$nth(M, map(F)^N(h(F,X))) = nth(M, iterates(F, F^N(X)))$$

If this isn't done then the proof gets blocked at the step case with the goal

$$\begin{gathered} nth(n, h(F,X)) = nth(n, iterates(F,X)) \Rightarrow \\ nth(n, map(F, h(F,X))) = nth(n, iterates(F, F(X))) \end{gathered}$$

This fits into the story of extension/restriction of sets.

For generalisations $gen(f)$ and $gen(g)$ of the functions f and g then $\{\langle f(x),\ g(x)\rangle\} \subseteq \{\langle gen(f)(x),\ gen(g)(x)\rangle\}$. So $\{\langle gen(f)(x),\ gen(g)(x)\rangle\}$ is an extension of $\{\langle f(x),\ g(x)\rangle\}$. However there are fewer values of x that will satisfy $gen(f)(x) = gen(g)(x)$ or in the more general case $P(gen(f)(x))$ so $\{x \mid P(gen(f)(x))\} \subseteq \{x \mid P(f(x))\}$ so $\{x \mid P(gen(f)(x))\}$ is a restriction of $\{x \mid P(f(x))\}$.

10.9 Summary of Proof Comparison

The presentation developed in the last section was more abstract than the more usual representation of induction rules. The change has highlighted two different approaches to inductive proof which I shall call *scheme induction* and *set induction*. The first of these relies on the correct choice of induction scheme (or function/labelled transition system) whilst the second relies on the correct choice of set. The results about the equivalence of the least fixedpoints of certain well–chained LTSs also have duals for greatest fixedpoints. This means that *scheme* coinduction also exists, as well as the more common *set* coinduction used in this thesis. It isn't clear at the time of writing if one approach has any benefits over the other.

The analysis allowed us to provide theoretical justifications for the comparisons that were drawn between the various proof methods. The set induction rule allows us to see the relationship between induction schemes and trial bisimulations whilst the link between transitions and cancellation of constructors allows the observed similarities in the rewriting processes to be explained. The duality between induction and coinduction introduces, unsurprisingly, dualities throughout the proof process.

10.10 Transferring *CLAM* Proof Methods

As stated at the beginning of this chapter, the identified equivalences will now be used to look more closely at the *CLAM* and *CoCLAM* methods and see whether these can be transfered between the two systems.

10.10.1 Induction Scheme/Set and Bisimulation Choice

The previous analysis showed a duality between the induction scheme and the trial bisimulation. If the original goal is used to provide a set and this fails then it has to be restricted or enlarged depending upon which type of proof is being attempted. Furthermore as the rewriting processes are expected to be very similar we can reasonably expect them to fail at the same point.

All this suggests comparing the critics and methods which perform these tasks in *CLAM* and *CoCLAM*. Lemma Speculation and Induction Scheme Revision are triggered by the same Wave precondition failing (there is no Wave Rule match) this is also the main precondition that will trigger Bisimulation Revision in coinduction. If Bisimulation Revision is divided into two parts, Bisimulation Extension (adding new pair schema into the relation) and Bisimulation Generalisation (generalising the expressions in the relation) then each principle can be seen to have two critics attached to this Wave rule precondition.

CLAM decides whether to use Lemma Speculation or Induction Scheme Revision depending upon whether any partial wave rule matches are present in the expression. *CoCLAM* decides whether to use Bisimulation Extension or Bisimulation Generalisation depending upon whether it can observe difference matches among subexpressions. A major thread running through the following discussion will be a comparison of these two heuristics.

Bisimulation Extension vs. Induction Scheme Revision

Consider proving theorem 3.4 from chapter 3 inductively using *nth*. Recall that this theorem requires bisimulation extension. For simplicity consider only the LHS of the equation, Rippling becomes blocked with the goal:

$$\text{Induction Hypothesis} \Rightarrow \underbrace{nth(n, \boxed{B :: \underline{lswap(A,B)}}^{\uparrow})}_{blocked} = \cdots$$

Annotating this with potential wave fronts (as for Induction Revision) gives:

$$nth(\boxed{F_1(\underline{n})}^{\uparrow}, \boxed{B :: F_2(\underline{lswap(A,B)})}^{\uparrow}) = \cdots$$

There is a partial wave rule match here with (10.13) so in the inductive proof the Induction Scheme Revision critic would fire at this point. Returning to the coinductive proof, this suggests that it might be appropriate to use the presence of a partial wave rule match as a condition for bisimulation extension. Although, in coinductive proofs there is no *nth* function to provide a wave rule match it should be possible to introduce the idea of *potential transitions* instead which would be an equivalent concept given that *nth* has been shown to be related to taking transitions.

Care needs to be taken if partial wave rule matching is to be incorporated into coinduction in cases where mutual recursion is employed. Consider the functions *tick* and *tock*:

$$tick \overset{\text{red}}{\rightsquigarrow} 0 :: tock$$

$$tock \overset{\text{red}}{\rightsquigarrow} 1 :: tick$$

$$\begin{array}{rcl} flip(0) & = & 1 \\ flip(1) & = & 0 \end{array}$$

and the theorem

Example 10.6

$$tick \sim map(flip, tock)$$

This requires the bisimulation $\{\langle tick,\ map(flip, tock)\rangle\} \cup \{\langle tock,\ map(flip, tick)\rangle\}$ *in the proof (i.e. the original pair scheme has been extended). If only the first of these pair schema is provided (as is done by the Coinduction method) the rippling will become blocked at:*

$$\cdots \Rightarrow \langle tock,\ map(flip, tick)\rangle$$

which has no potential transition or partial wave rule match.

Mutual recursion is perceived as a hard problem for induction in general and particularly for the theory of Rippling since it doesn't conform to ideas of skeleton preservation. It is standard practice when using CLAM to attempt to transform mutually recursive functions into functions that do not require mutuality. e.g. tock could become $tock \rightsquigarrow 1 :: 0 :: tock$ *with this sort of definition there would be a partial wave rule match.*

There is certainly a strong suggestion that the idea of a partial match of some sort could be useful for coinduction critics though, as the above example shows, more work would be required before it could be introduced with confidence. The fact that these critics are not directly transferable may well be because one applies to set coinduction while the other applies to scheme induction.

Generalisation vs. Lemma Speculation

It is interesting to note that Ireland and Bundy [Ireland & Bundy 96] observe similar divergence patterns to those that occur in coinduction with the use of the Induction Revision Critic. The Lemma Speculation Critic was developed to overcome this. This suggests that Lemma Speculation might be an alternative to Bisimulation Generalisation in coinduction.

Looking at the proof of example 10.5 the first naive attempt (using a first guess at trial bisimulation) becomes stuck at

$$\begin{gathered}\langle h(F,X),\ iterates(F,X)\rangle \in \mathcal{R} \Rightarrow \\ \langle \boxed{map(F, \underline{h(F,X)})}^{\uparrow},\ iterates(F, \boxed{F(\underline{X})}^{\uparrow})\rangle \in \mathcal{R} \cup \sim\end{gathered}$$

Should we be looking for some lemma to rewrite one side of the relation ? Taking the cue from the Ireland and Bundy's [Ireland & Bundy 96] Lemma Speculation Critic this would suggest the lemma

$$iterates(F, \boxed{F(\underline{X})}^{\uparrow}) \rightsquigarrow \boxed{map(F, \underline{iterates(F,X)})}^{\uparrow} \tag{10.30}$$

which is the theorem in example 5.1 so the lemma is true.

Using (10.30) the goal rewrites to

$$\begin{gathered}\langle h(F,X),\ iterates(F,X)\rangle \in \mathcal{R} \Rightarrow \\ \langle \boxed{map(F, \underline{h(F,X)})}^{\uparrow},\ \boxed{map(F, \underline{iterates(F,X)})}^{\uparrow}\rangle \in \mathcal{R} \cup \sim\end{gathered}$$

We would need to be able to make a $map(F)$ transition (or cancel the $map(F)$s) to prove the theorem and such rules are not available. So Lemma Speculation can't simply replace Bisimulation Generalisation.

Including Lemma Speculation in Coinductive Proofs

Just because lemma speculation cannot replace generalisation does not mean it should be excluded altogether. Recall the bisimulation for the commutativity of plus discussed in chapter 7:

$$\begin{gathered}\{\langle V_0 + s^N(V_1),\ V_1 + s^N(V_0)\rangle\} \cup \\ \{\langle V_0 + 0,\ V_0\rangle\} \cup \\ \{\langle V_0,\ V_0 + 0\rangle\} \cup \\ \{\langle s^{s(N)}(V_1),\ V_1 + s^{s(N)}(0)\rangle\} \cup \\ \{\langle V_0 + s^{s(N)}(0),\ s^{s(N)}(V_0)\rangle\}\end{gathered}$$

The additional pair schema do not, in this bisimulation, give rise to partial wave rule matches even though Bisimulation Extension was used to patch the proofs. However, it should also be recalled that the complexity of this bisimulation could have been avoided if the additional lemmata $X+0 \rightsquigarrow X$ and $X+s(Y) \rightsquigarrow s(X+Y)$ had been present in which case $\langle X+Y,\ Y+X\rangle$ would have been provably a bisimulation. As a result, we wouldn't necessarily expect partial wave rule matches here. Moreover it suggests that it would definitely be desirable to have a Lemma Speculation critic for coinduction which could be used in situations like this and, if nothing else, could help contain the problem of bisimulation explosion.

Including a new Generalisation Critic in Inductive Proofs

Similarly a Generalisation critic like *CoCLAM*'s could be of use in *CLAM*.

If we examine the inductive proof of example 10.5 using *nth* it becomes clear that Lemma Speculation isn't sufficient even in induction to prove this theorem. The step case becomes blocked at:

$$\begin{gathered} nth(n, h(F,X)) = nth(n, iterates(F,X)) \Rightarrow \\ \underbrace{nth(n, \boxed{map(F, \underline{h(F,X)})}^{\uparrow})}_{blocked} = \underbrace{nth(n, iterates(F, \boxed{F(\underline{X})}^{\uparrow}))}_{blocked} \end{gathered}$$

Rippling is blocked on both sides so it is possible to speculate two lemmata:

$$nth(n, \boxed{map(F, \underline{h(F,X)})}^{\uparrow}) \rightsquigarrow nth(\boxed{F_1(\underline{n})}^{\uparrow}, h(F, \boxed{F_2(\underline{X})}^{\downarrow})) \tag{10.31}$$

$$nth(n, iterates(F, \boxed{F(\underline{X})}^{\uparrow})) \rightsquigarrow nth(\boxed{F_1(\underline{n})}^{\uparrow}, \boxed{F_2(\underline{iterates(F,X)})}^{\uparrow}) \tag{10.32}$$

(10.32) would lead to the speculation of the inductive version of mapiterates (theorem 10.4) but would fail to prove the theorem since $\boxed{map(F, \underline{\cdots})}^{\uparrow}$ can't be rippled through *nth* as was discussed above. (10.31) would suggest the lemma:

$$nth(n, \boxed{map(F, \underline{h(F,X)})}^{\uparrow}) \rightsquigarrow nth(n, h(F, \boxed{F(\underline{X})}^{\downarrow}))$$

which would allow fertilization by sinking the differences in X.

However attempts at proving this lemma fail becoming blocked at the step case goal:

$$\begin{gathered} nth(n, map(F, h(F,X))) = nth(n, h(F, F(X))) \Rightarrow \\ \underbrace{nth(n, \boxed{map(F, \underline{map(F, h(F,X))})}^{\uparrow})}_{blocked} = \underbrace{nth(n, h(F, \boxed{F(\underline{F(X)})}^{\uparrow}))}_{blocked} \end{gathered}$$

This would in turn speculate the lemma

$$nth(N, \boxed{map(F, \underline{map(F, h(F,X))})}^{\uparrow}) \rightsquigarrow nth(n, h(F, \boxed{F(\underline{F(X)})}^{\downarrow}))$$

and a divergent process of lemma speculation would have been embarked upon.

This last observation suggests a possible strategy of using generalisation if the lemma sequence appears to be divergent with the generalisation formed by difference matching between different elements in that sequence.

A simple alternative would be to apply lemma speculation in situations where no partial wave rule matches existed but some divergence check had not identified any divergence patterns. This strategy would apply to coinduction and could possibly apply to induction if an appropriate "divergence check" could be devised.

10.10.2 Ripple and Evaluate

Ripple Analysis is the process used by *CLAM* to determine the induction scheme and induction variable. These choices are related to the choice of variables to casesplit in the Evaluate method in *CoCLAM*. If this casesplitting could be performed accurately then it would no longer be necessary to have a separate Evaluate method but instead it might be possible to use ripple analysis and then rippling towards cancellation/transitions, followed by more rippling to fertilize.

Ripple analysis relies on a process called "lookahead". This examines each variable in turn, casesplitting it to see if the structure this would introduce would allow a wave rule to apply involving that variable. Variables are scored according to their suitability as induction variable. If wave rules apply to all instances of a variable then it scores highly and is likely to be chosen as an induction variable. In coinduction the situation is often more complex than it is in induction due to the presence of functions that "generate" lists without casesplitting any variables (e.g. *iterates*).

Consider proofs involving *nth*. Induction on n will not automatically introduce any annotation into the 2nd argument. However an outward wave front is needed there before *nth* can be used. Such a wave front may not come from a revised induction scheme (further casesplitting) but from a function such as *iterates* (or a combination of both !).

Where these "list generators" do not appear there would seem to be a strong case for using *CLAM*'s lookahead methods and a version of the Induction Revision critic in *CoCLAM*. It would be useful if lookahead could be extended to cover list generators as well.

The problem would seem to be identifying and using "list generators". At present I'm unaware of an adequate formal definition of what constitutes a "list generator" although a first attempt is

Definition 10.4 *A* **list generator** *is a function, f, which is completely defined by the one rule*

$$\forall \bar{x}.\ f(\bar{x}) \leadsto \alpha(\bar{x}) :: \beta(\bar{x}, f(\bar{x}))$$

for some functions α and β.

This definition is unsatisfactory because it doesn't account for list generators which have more than one rule but which can nevertheless be "infinitely applied" to some expressions they apply to once. However, it may well be sufficient for many situations. The lookahead process would have to be extended to identify list generators and score them for unfolding as well scoring variables for casesplitting. It would also be necessary to extend the concept of generators to other datatypes. As with generalisation the fact that these functions can appear in inductive proofs (via the use of *nth*) suggests that strategies for list generators also need to be developed in *CLAM*.

The Evaluate method works (whether extended with a lookahead or not) for the problems considered by this thesis but it is heavily dependent on the specifics of $\mathcal{T}$. Middle-Out Reasoning might prove more appropriate for this proof step in the general case. Work would have to be done to extend the MOR approach to transition systems where reduction/rewriting of an expression was not the only requirement for determining the transitions from it. In these case MOR would have to be adapted to guide search through general inference rules.

10.11 Conclusion

This chapter has examined the links between inductive and coinductive proof. It is clear that the dualities that exist theoretically have implications on the practical process of proof providing dualities throughout the proof process. This observation suggests strongly that anyone intending to implement support for coinduction in a theorem prover should look at the support they have for induction and draw from that. It also suggests a major course of further work in the development of *CoCLAM* to integrate these observations and suggestions. However, the current implementation should not be regarded as a wasted effort since it was the formation of the methods for coinduction and their testing in *CoCLAM* that enabled much of this discussion which centred upon the comparison of those methods.

These observations also apply the other way around. Although, at present, it seems unlikely that there are any coinductive theorem provers that do not also support induction, the observations have suggested improvements to the implementation of proof planning for induction in *CLAM* such as suggestions for extending the critics with a second generalisation critic. So far as I'm aware no comparison of this nature has been undertaken before, though those working in the field are no doubt aware of the practical links between the two methods.

CoCLAM is basically a set coinduction theorem prover whilst *CLAM* is a scheme induction theorem prover[6]. An interesting investigation would be to implement set induction and scheme coinduction theorem provers and see how well these perform

[6] *CLAM* has generalisation capabilities but its main paradigm is clearly that of scheme induction.

compared to the current systems. For instance, at present *CLAM* has to store a set of well–founded orders from which it chooses induction schemes. The use of set restriction to patch proofs may avoid some of the need for this and provide more flexibility.

The above analysis was intended to show in detail where these similarities and dualities reside in the proof process and hence to give guidance for method transfer. It is worth noting that the discussion above was framed by proof planning which allowed the high–level proof steps to be parcelled together intuitively and allowed a discussion of corresponding parts in the proofs in terms of proof methods. It should be clear from this that proof planning is not only a useful tool for guiding proofs, but it also has uses in the discussion, understanding and comparison of proofs.

Chapter 11

Conclusion

11.1 Introduction

In chapter 1 I listed the contributions of this thesis as:

- Demonstrating that the choice of bisimulation relation in coinductive proof can, in many cases, be performed automatically.
- Providing some theoretical results about the smallest bisimulation required to prove a given theorem.
- Developing a number of heuristics for forming such bisimulations.
- Providing a proof strategy for coinduction which allows many coinductive proofs to be performed completely automatically.
- Showing that Rippling which was developed for induction is also of use for coinduction and that the idea of *difference matching* that underlies Rippling can be used to form bisimulations.
- Showing that these ideas can be implemented naturally in a proof planning system.
- Drawing out the links between induction and coinduction in the light of the proposed proof strategy.
- Providing theoretical results about the links between the choice of induction scheme and the choice of an appropriate labelled transition system when performing induction on programs in lazy functional languages and in particular how transition systems can be created from each other by "chaining" transitions and how some of these systems can be used interchangeably in induction.

In this last chapter I shall discuss these contributions in more depth.

11.2 Generation of Bisimulations

In chapter 6 I observed that there was a least relation which could discharge the first premise of the coinduction rule. I also showed that the minimum bisimulation required to prove a theorem was the set of chopped transition sequences from this relation. These sequences are the sequence of pairs formed by repeated use of transitions. From these observations I advocated a strategy, based on the idea of proof critics, of gradual extension of the least relation with pairs obtained by taking transitions. These pairs naturally arise when attempting a coinductive proof with a relation that is too small, hence they are the failure information that informs the critic. I have successfully demonstrated two methods of extending this bisimulation, bisimulation extension and generalisation (used when the transition sequence is infinite), using this paradigm.

The fact that such a minimal starting point exists strongly suggests that this strategy together with heuristics for expressing infinite chopped transition sequences provides a good framework for the generation of bisimulations. The associated heuristics discussed in this thesis have drawn much of their inspiration from proof planning techniques. However, there is no reason to suppose that other techniques could not be used for alternative versions of this strategy and one such alternative, inverse resolution, as used in inductive logic programming, was considered.

11.3 Theoretical Results about the Smallest Bisimulation

Although the general strategy I've propounded is known in the field it is not clear who originated it. It has been described as folklore by [Barwise & Moss 96]. However, I'm not aware of the existence of any theoretical results about minimum bisimulations justifying the strategy such as have been used to justify it here. Even the strategy as it appears in the "folklore" so far as I'm aware only concerns itself with the process I've termed bisimulation extension, and does not consider the possible need for generalisations.

11.4 Heuristics for Bisimulation Formation

The heuristics adopted for bisimulation formation involve the gradual extension of the "least" relation with additional pair schema whilst at the same time looking for patterns in this sequence that may be generalised. There are a large number of generalisation heuristics available, many of which would probably be suitable for this task. The particular heuristic described here relies on difference matching to identify divergent patterns and is based on one used to perform a similar task in an implicit induction prover [Walsh 96]. This heuristic has allowed bisimulations to be found for many of the theorems that appear in the literature.

The language of difference matching provides a more abstract view of the generalisation process and enables us to consider extensions and alternatives to the strategy depending on the observed patterns of differences. It may well be that in future the critic can be extended to cope with more complicated difference patterns (e.g. those situations where a new function needs to be synthesized). The framework difference matching provides for examining differences appears to be both general and useful and so a good basis for further work in this area.

11.5 The Proof Strategy for Coinduction

A proof strategy for coinduction as a whole has been developed which incorporates the strategies for bisimulation discovery already discussed. This strategy has been used to generate proof plans in a fully automated fashion for a large number of theorems. There are still a number of cases where the proof strategy fails and improvements and modifications were discussed in chapters 7, 9 and 10. The problems do not, at present, seem to indicate fundamental flaws in the strategy.

The strategy has been tailored for a particular flavour of operational semantics. However the proof strategy does express a more abstract notion of coinductive proof. Most of the tailoring occurs in one method, the Evaluate method, which makes specific assumptions about the rules in the labelled transition system. More general forms of this method were discussed in chapter 9. The assertion that the proof strategy expresses some generality is also supported by the following four facts.

- An earlier version of the proof strategy was used to perform coinductive proofs in a framework that didn't use transitions [Dennis *et al* 96].
- In chapter 10 where the proof strategy was used to make comparisons with induction, these comparisons were justified.
- Isabelle tactic sequences were chosen with relative ease to correspond to the *CoCLAM* methods (even if the methods didn't always present the precise information required by Isabelle – see appendix E).

- In chapter 9 I briefly examined some common features of labelled transition systems which $CoCL^AM$ cannot handle at present. None of these features required a radical change to the proof plan.

I believe the proof strategy to be a good representation of coinductive proof and that much of the detail that makes it specific to the operational semantics of lazy functional programming languages lies in the specifics of the methods (particularly the evaluation method) and not in the overall strategy.

11.6 Rippling and Coinduction

Rippling plays an apparently minor role in the coinductive proof strategy presented here, that of guiding rewriting after the determination of transitions. However it should be recalled that it is this step that attempts to rewrite towards fertilization and so it is the failure of this step that causes the Revise Bisimulation critic to be called and this is central to the process of forming bisimulations. Since this stage of the proof when it occurs, often involves using rewrites in the opposite direction to that imposed by the reduction order, it is important to have a more flexible rewriting method to hand.

Chapter 10 looked at bringing the two proof strategies (for induction and coinduction) closer together and its suggestions included replacing evaluation with rippling. Among other things, attempts to speculate corecursive lemmata (as discussed in chapter 9) could well benefit from the addition of difference annotations at this point in the proof process.

Chapter 10 also suggested including a general Lemma Speculation critic in coinduction which would exploit blocked wave fronts produced at the rippling stage and exploit the ideas of partial wave rule matches when choosing critics.

Rippling embodies important concepts about manipulating differences between expressions and it should come as no surprise that these ideas appear in coinduction since a major part of the proof is an attempt to prove two sets equal, one of which has been formed from the other.

11.7 Contribution of Proof Planning to Coinductive Proof

There are two levels of contribution of proof planning to coinduction.

At one level proof methods for inductive proof have been used to inspire proof methods for coinductive proof. Furthermore, suggestions have been made for further proof methods and critics to be used in this way. At this level of contribution the fact that proof planning has been used for automating inductive proofs is an aid in the automation of coinductive proof.

At the other level, the formation of a proof strategy for coinduction and the ideas of interacting proof methods and critics are of themselves useful tools in automating coinductive proof. Heuristically guided search is clearly needed for coinductive proof and proof planning provides an efficient way to undertake such search. This served as an illustration of how proof strategies can allow specific theorem proving tasks to be identified and examined and how the process of linking them together to produce proofs can be guided.

11.8 Contribution of Inductive Proof Techniques to Coinductive Proof

Inductive proof techniques have clear relevance to coinduction grounded in the duality of the two proof methods. The natural correspondence between cancellation and transitions further reinforces this.

It was disappointing that more of this correspondence wasn't used in the final proof strategy proposed for coinduction. However the formation of the proof strategy made the links between the proof processes clearer and allowed them to be compared again suggesting several points at which techniques could be transferred from one proof strategy to the other.

It should be noted, however, that although many new methods were created, rippling, a heuristic originally developed for induction, was used in the proof process and that one suggestion for further work is that it should be adapted for use in the Evaluation method. Eval_def, a reduction method implemented in CL^AM3 was also used. The idea of using a critic to revise the bisimulation also originated in the use of a critic to revise the induction scheme and furthermore that critic was based on one developed for implicit induction.

On a more general level the comparison indicated points at which similar theorem proving tasks are required in the two methods and so provided guidance for their transfer in other theorem proving environments.

11.9 Theoretical Results about the Link between Induction Scheme and Bisimulation choice

In chapter 10 the induction and coinduction rules were examined in some detail to try and ascertain how particular specialisations of the rules related to one another. This resulted in a formulation of the induction rule in which the choice of some set was the eureka step in the proof, not the choice of induction scheme.

I showed that using this rule the choice could be placed back on some sort of scheme if the labelled transition system was modified by chaining some of the transitions together. I then defined a property of α–well–chained and showed that the least fixedpoints of my induction function, $\lceil - \rceil$, were the same given two systems in an α–well–chained list.

As far as I'm aware, no one has previously suggested performing induction in this way and as a result the theorems about the equivalence of least fixedpoints are new (although unsurprising given that the equivalent process can be undertaken in "normal" induction).

I showed that these results can be extended to coinduction, so that instead of choosing a bisimulation it is possible to choose a transition system.

11.10 General Conclusions

Coinduction is a method of increasing interest in computer science. It can be partially automated using proof planning and for many proofs it can be fully automated. The key elements of the proof strategy are the use of critics and generalisation techniques to find a bisimulation for use in the proof.

This proof strategy was very successful on the examples that are currently available. This suggests that the proof strategy and implementation based upon it would be of practical use to people attempting to prove theorems coinductively.

The development of a clear proof strategy for coinduction allowed the proof principle to be compared with induction on a practical level to examine points where the techniques used for one can be adapted to the other.

Proof planning clearly has a contribution to make to both the automation of coinductive proof and in a more general way to the understanding, comparison and reuseability of theorem proving techniques.

Bibliography

[Abadi & Gordon 97] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. In *Proceedings of the Fourth ACM Conference on Computer and Communications Security*, pages 36 – 47. ACM Press, April 1997.

[Abramsky 90] S. Abramsky. The lazy lambda calculus. In D. Turner, editor, *Research Topics in Functional Programming*, pages 65–117. Addison Wesley, 1990.

[Aczel & Mendler 89] P. Aczel and N. Mendler. A final coalgebra theorem. In D. H. Pitt, D. E. Ryeheard, P. Dybjer, A. M. Pitts, and A. Poigne, editors, *Proceedings Category Theory and Computer Science*, pages 357–365. Springer–Verlag, 1989. Lecture Notes in Computer Science No. 389.

[Aczel 88] P. Aczel. *Non–Well–Founded Sets.* CSLI Lecture Notes, Number 14. LSCI/Stanford, 1988.

[Aczel 97] P. Aczel. Lectures on semantics: The initial algebra and final coalgebra perspectives. In H. Schwichtenberg, editor, *Logic of Computation*, number 157 in Series F: Computer and Systems Sciences, pages 1–34. Springer-Verlag, 1997.

[Barendregt 84] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* North–Holland, 1984. revised edition.

[Barwise & Moss 96] J. Barwise and L. Moss. *Vicious Circles.* CSLI Lecture Notes Number 59. CSLI Publications, Stanford, 1996.

[Basin & Walsh 92] D. Basin and T. Walsh. Difference matching. In Deepak Kapur, editor, *11th Conference on Automated Deduction*, pages 295–309, Saratoga Springs, NY, USA, June 1992. Published as Springer Lecture Notes in Artificial Intelligence, No 607.

[Basin & Walsh 96] David Basin and Toby Walsh. Annotated rewriting in inductive theorem proving. *Journal of Automated Reasoning*, 16(1–2):147–180, 1996.

[Benzmüller *et al* 97] C. Benzmüller, L. Cheikhrouhou, D Fehrer, A. Fiedler, X. Huang, M. Kerber, K. Kohlhase, A Meirer, W. Melis, E. aand Schaarschmidt, J. Siekmann, and V. Sorge. Ωmega: Towards a mathematical assistant. In W. McCune, editor, *14th Conference on Automated Deduction*, pages 252–255. Springer-Verlag, 1997.

[Berry *et al* 86] G. Berry, P.-L. Curien, and J.-J. Lévy. Full abstraction for sequential languages: the state of the art. In M. Nivat and J. Reynodls, editors, *Algebraic Semantics*, pages 89–132. Cambridge University Press, 1986.

[Bird & Wadler 88] Richard S. Bird and Philip Wadler. *Introduction to Functional Programming.* Prentice-Hall, 1988.

[Bouhoula & Rusinowitch 93] A. Bouhoula and M. Rusinowitch. Automatic case analysis in proof by induction. In *Proceedings of the 13th IJCAI.* International Joint Conference on Artificial Intelligence, 1993.

[Boyer & Moore 90] R. S. Boyer and J S. Moore. A theorem prover for a computational logic. In *Proceedings of the Tenth International Conference on Automated Deduction*, 1990. Kaiserlauten, Germany.

[Bradfield & Stirling 90] J. Bradfield and C. Stirling. Verifying Temporal Properties of Processes. In *Lecture Notes in Conputer Science, v.458*, pages 115–125. Springer-Verlag, 1990.

[Bruns 91] G. Bruns. A language for value-passing CCS. LFCS Report Series ECS-LFCS-91-175, Department of Computer Science, University of Edinburgh, 1991.

[Bundy 83] Alan Bundy. *The Computer Modelling of Mathematical Reasoning.* Academic Press, 1983. Second Edition.

[Bundy 88] Alan Bundy. The use of explicit plans to guide inductive proofs. In R. Lusk and R. Overbeek, editors, *9th Conference on Automated Deduction*, pages 111–120. Springer-Verlag, 1988. Longer version available from Edinburgh as DAI Research Paper No. 349.

[Bundy *et al* 90a] A. Bundy, A. Smaill, and J. Hesketh. Turning eureka steps into calculations in automatic program synthesis. In S. L.H. Clarke, editor, *Proceedings of UK IT 90*, pages 221–6, 1990. Also available from Edinburgh as DAI Research Paper 448.

[Bundy *et al* 90b] A. Bundy, F. van Harmelen, C. Horn, and A. Smaill. The Oyster-Clam system. In M. E. Stickel, editor, *10th International Conference on Automated Deduction*, pages 647–648. Springer-Verlag, 1990. Lecture Notes in Artificial Intelligence No. 449. Also available from Edinburgh as DAI Research Paper 507.

[Bundy *et al* 91] Alan Bundy, Frank van Harmelen, Jane Hesketh, and Alan Smaill. Experiments with proof plans for induction. *Journal of Automated Reasoning*, 7:303–324, 1991. Earlier version available from Edinburgh as DAI Research Paper No 413.

[Bundy *et al* 93] A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62:185–253, 1993. Also available from Edinburgh as DAI Research Paper No. 567.

[Burkhart *et al* 95] O. Burkhart, D. Caucal, and B. Steffen. An elementary decision procedure for arbitrary context–free processes. In *MFCS'95*, LNCS. Springer–Verlag, 1995.

[Chen *et al* 90] H. Chen, J. Hsiang, and H.-C. Kong. On finite representations of infinite sequences of terms. In M. Okada, editor, *Proceedings of the 2nd International Workshop of Conditional and Typed Rewriting Systems*, Lecture Notes in Computer Science, v. **516**, pages 100–114, Berlin, 1990. Springer-Verlag.

[Church 40] A. Church. A formulation of the simple theory of types. *Symbolic Logic*, 5(1):56–68, 1940.

[Cleaveland *et al* 89] R. Cleaveland, Parrow J., and B. Steffen. The concurrency workbench: A semantics-based verification tool for finite-state systems. In *Proceedings of the Workshop on Automated Verification Methods for Finite-State Systems*. Springer-Verlag, 1989.

[Collins & Hogg 97] G. Collins and J. Hogg. The circuit that was too lazy to fail. Unpublished Paper, 1997.

[Collins 96] G. Collins. A proof tool for reasoning about functional programs. In J. von Wright, J. Grundy, and J Harrison, editors, *9th International Conference of Theorem Proving in Higher Order Logics*, volume 1125 of *Lecture Notes in Computer Science*, pages 109–124. Springer, 1996.

[Dennis *et al* 96] L. Dennis, A. Bundy, and I. Green. Using a generalisation critic to find bisimulations for coinductive proofs. In W. McCune, editor, *14th Conference on Automated Deduction*, Lecture Notes in Artificial Intelligence, Vol. 1249, pages 276–290, Townsville, Australia, 1996. Springer-Verlag.

[Fiore 93] M. Fiore. A coinduction principle for recursive data types based on bisimulation. In *Proceedings of the Eight IEEE Symposium on Logic in Computer Science*, pages 110–119, 1993.

[Frost 95] J. Frost. A case study of co-induction in isabelle. Forthcoming technical report, University of Cambridge, Computer Laboratory, 1995.

[Gordon & Melham 93] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A theorem proving environment for higher order logic.* Cambridge University Press, 1993.

[Gordon 85] M. Gordon. Hol: A machine oriented formulation of higher order logic. Technical Report 68, Computer Laboratory, University of Cambridge, July 1985. revised version.

[Gordon 88] M. Gordon. HOL: A proof generating system for higher-order logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis.* Kluwer, 1988.

[Gordon 93] A. D. Gordon. Functional programming and input/output. Technical Report 285, University of Cambridge, Computer Laboratory, 1993.

[Gordon 95a] A. D. Gordon. Bisimilarity as a theory of functional programming. In *Proceedings of 11th Conference on the Mathematical Foundations of Programming Semantics*, Electronic Notes in Computer Science, v.1, New Orleans, 1995. Elsevier.

[Gordon 95b] A. D. Gordon. Operational methods. Lecture notes for a course at the University of Cambridge, 1995.

[Gordon 95c] A. D. Gordon. A tutorial on co–induction and functional programming. In *Proceedings of the 1994 Glasgow Workshop on Functional Programming*, Springer Workshops in Computing, 1995.

[Gordon 96] A. D. Gordon. Bisimilarity for a first-order calculus of objects with subtyping. In *Proceedings, 23rd Symposium on Principles of Programming Languages*, pages 386–395, St. Petersburg Beach, Florida, USA, 21–24 January 1996. ACM SIGPLAN-SIGACT.

[Gordon *et al* 79] M. J. Gordon, A. J. Milner, and C. P. Wadsworth. *Edinburgh LCF - A mechanised logic of computation*, volume 78 of *Lecture Notes in Computer Science*. Springer Verlag, 1979.

[Groote & Vaandrager 92] J.F Groote and F.W. Vaandrager. Structured operartional semantics and bisimulation as congruence. *Information and Computation*, 100(2):202–260, 1992.

[Hensel & Jacobs 97] U. Hensel and B. Jacobs. Coalgebraic theories of sequences in PVS. Technical Report CSI-R9708, Computing Science Institute, University of Nijmegen, 1997.

[Hesketh 91] J. T. Hesketh. *Using Middle-Out Reasoning to Guide Inductive Theorem Proving.* Unpublished PhD thesis, University of Edinburgh, 1991.

[Horn & Smaill 90] C. Horn and A. Smaill. Theorem proving and program synthesis with Oyster. In *Proceedings of the IMA Unified Computation Laboratory*, Stirling, 1990.

[Horn 88] C. Horn. The Nurprl proof development system. Working paper 214, Dept. of Artificial Intelligence, University of Edinburgh, 1988. The Edinburgh version of Nurprl has been renamed Oyster.

[Howe 89] Douglas J. Howe. Equality in lazy computation systems. In *Proceedings, Fourth Annual Symposium on Logic in Computer Science*, pages 198–203, Asilomar Conference Center, Pacific Grove, California, 5–8 June 1989. IEEE Computer Society Press.

[Huet & Oppen 80] G. Huet and D. C. Oppen. Equations and rewrite rules: a survey. In R. Book, editor, *Formal languages: perspectives and open problems*. Academic Press, 1980. Presented at the conference on formal language theory, Santa Barbara, 1979. Available from SRI International as technical report CSL-111.

[Huet 75] G. Huet. A unification algorithm for lambda calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[Hutton 98] G. Hutton. Fold and unfold for program semantics. Technical report, Department of Computer Science, University of Nottingham, 1998. Submitted to ICFP'98.

[Ireland & Bundy 96] A. Ireland and A. Bundy. Productive use of failure in inductive proof. *Journal of Automated Reasoning*, 16(1–2):79–111, 1996. Also available as DAI Research Paper No 716, Dept. of Artificial Intelligence, Edinburgh.

[Ireland 92] 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, St. Petersburg*, Lecture Notes in Artificial Intelligence No. 624, pages 178–189. Springer-Verlag, 1992. Also available from Edinburgh as DAI Research Paper 592.

[Jacobs & Rutten 97] B. Jacobs and J. Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 1997. to appear.

[Jacobs 97] B. Jacobs. Invariants, bisimulations and the correctness of coalgebraic refinements. Technical Report CSI–R9704, Computing Science Institute, University of Nijmegen, 1997.

[Knuth & Bendix 70] D. E. Knuth and P. B. Bendix. Simple word problems in universal algebra. In J. Leech, editor, *Computational problems in abstract algebra*, pages 263–297. Pergamon Press, 1970.

[Luger & Stubblefield 93] G. F. Luger and W. A Stubblefield. *Artificial Intelligence: Structures and Strategies for Complex Problem Solving*. Benjamin/Cummings Publishing Company, 1993.

[Mac Lane 71] S. Mac Lane. *Categories for the Working Mathematician*. Springer-Verlag, New York, 1971.

[Manning *et al* 93] A. Manning, A. Ireland, and A. Bundy. Increasing the versatility of heuristic based theorem provers. In A. Voronkov, editor, *International Conference on Logic Programming and Automated Reasoning – LPAR 93, St. Petersburg*, number 698 in Lecture Notes in Artificial Intelligence, pages pp 194–204. Springer-Verlag, 1993.

[Michalski 83] R. S. Michalski. A theory and methodology of inductive learning. *Artificial Intelligence*, 20:111–161, 1983.

[Milner & Tofte 91] Robin Milner and Mads Tofte. Co-induction in relational semantics. *Theoretical Computer Science*, 87:209–220, 1991.

[Milner 77] R. Milner. Fully abstract models of typed lambda–calculi. *Theoretical Computer Science*, 4:1–23, 1977.

[Milner 89] R. Milner. *Communication and Concurrency.* Prentice Hall, London, 1989.

[Milner *et al* 90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.

[Mitchell 82] T. M. Mitchell. Generalization as search. *Artificial Intelligence*, 18:203–226, 1982.

[Monroy *et al* 95] R. Monroy, A. Bundy, A. Ireland, and J. Hesketh. Proof Planning the Verification of CCS Programs. Research Paper 781, Dept. of Artificial Intelligence, University of Edinburgh, 1995.

[Morris 68] J. H. Morris. *Lambda–Calculus Models of Programmins Languages.* Unpublished PhD thesis, MIT, December 1968.

[Muggleton & De Raedt 94] S. Muggleton and L. De Raedt. Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19, 20:629–679, 1994.

[Nipkow & Paulson 94] T. Nipkow and L. Paulson. Datatypes and (co)inductive definitions in isabelle/hol, 1994. Sections of [Paulson 96] that could not be included in [Paulson 94a].

[Owre *et al* 92] S. Owre, J. M. Rushby, and N. Shankar. PVS: An integrated approach to specification and verification. Forthcoming, SRI International, 1992.

[Owre *et al* 96] S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. K. Srivas. PVS: Combining specification, proof checking, and model checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the 1996 Conference on Computer-Aided Verification*, number 1102 in LNCS, pages 411–414, New Brunswick, New Jersey, U. S. A., 1996. Springer-Verlag.

[Park 70] D. Park. Fixpoint Induction and Proofs of Program Properties. *Machine Intelligence*, 5:59–78, 1970.

[Park 80] D. Park. On the Semantics of Fair Parallelism. In D. Bjϕrner, editor, *Proceedings 1979 Copenhagen Winter School*, pages 504–526, 1980. LNCS 86.

[Park 81] D. Park. Concurrency and Automata on Infinite Sequences. In P. Deussen, editor, *Proceedings of the 5th GI-Conference on Theoretical Computer Science*, pages 167–183, 1981. LNCS 104.

[Paulin-Mohring 95] Christine Paulin-Mohring. Circuits as streams in Coq verification of a sequential multiplier. Technical Report RR95–16, Laboratoire de l'Informatique du Parallelisme, 1995.

[Paulson 93] L. C. Paulson. Co-induction and Co-recursion in Higher-order Logic. Technical Report 304, University of Cambridge, Computer Laboratory, 1993.

[Paulson 94a] L. C. Paulson. *Isabelle: A generic theorem prover*. Springer-Verlag, 1994.

[Paulson 94b] Lawrence C. Paulson. A fixedpoint approach to implementing (co)inductive definitions. In Alan Bundy, editor, *12th Conference on Automated Deduction*, Lecture Notes in Artificial Intelligence, Vol. 814, pages 148–161, Nancy, France, 1994. Springer-Verlag.

[Paulson 96] L. C. Paulson. Isabelle's object logics. Documentation for Isabelle available online `ftp://ftp.cl.cam.ac.uk/ml/index.html`, 1996. Included in [Paulson 94a].

[Peyton Jones 87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

[Pitts 92] A. M. Pitts. A Co-Induction Principle for Recursively Defined Domains. Technical Report 252, University of Cambridge, Computer Laboratory, April 1992.

[Plotkin 71] G. D. Plotkin. *Automatic Methods of Inductive Inference*. Unpublished PhD thesis, University of Edinburgh, 1971.

[Richardson 95] J. D. C. Richardson. *Proof planning data type changes in pure functional programs*. Unpublished PhD thesis, Department of Artificial Intelligence, University of Edinburgh, September 1995.

[Rutten 96] J.J.M.M. Rutten. Universal coalgebra: A theory of systems. Technical Report CS–R9652, CWI, Amsterdam, 1996.

[Sheeran & Jones 90] M. Sheeran and G. Jones. *Circuit Design in Ruby*. North Holland, 1990.

[Smaill & Green 96] A. Smaill and I. Green. Higher-order annotated terms for proof search. volume 1125 of *Lecture Notes in Computer Science*, pages 399–414. Springer, 1996. Also available as DAI Research Paper 799.

[Smyth & Plotkin 82] M.B. Smyth and G.D. Plotkin. The category–theoretic solution of recursive domain equations. *SIAM Journal of Computing*, 11(4):761–783, 1982.

[Stevens 98] P. Stevens. Abstract games for inifinite state processes. In *CONCUR'98*. Springer-Verlag, 1998.

[Tarski 55] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[Thomas & Jantke 89] M. Thomas and K. P. Jantke. Inductive inference for solving divergance in Knuth-Bendix. In K. P. Jantke, editor, *Analogical and Inductive Inference, Procs. of AII'89*, pages 288–303. Springer-Verlag, 1989.

[Thomas & Watson 93] M. Thomas and P. Watson. Solving divergence in Knuth-Bendix completion by enriching signatures. *Theoretical Computer Science*, 112:145–185, 1993.

[vanGlabbeek 96] R. van Glabbeek. The meaning of negative premises in transition system specifications ii. Technical Report STAN–CS–TN–95–16, Department of Computer Science, Stanford University, 1996. Extended Abstract in: ICALP-96.

[Walsh 96] Toby Walsh. A divergence critic for inductive proof. *Journal of Artificial Intelligence Research*, 4:209–235, 1996.

[Walsh 97] T. Walsh. Depth–bounded discrepancy search. In M.E. Pollack, editor, *Proceedings of the 15th International Joint Conference on Artificial Intelligence*, volume 1, pages 528–533. International Joint Conference on Artificial Intelligence, Morgan Kaufmann Publishers, 1997.

Appendix A

Glossary of Terms

Action A term originating in CCS indicating an action taking place on a state, e.g. the input or output of a data item. If a state, A, becomes the state A' as a result of the action a this is written $A \xrightarrow{a} A'$. Transitions are sometimes referred to as actions.

Algebra [Aczel 97] If F is an endofunctor then an F-**algebra** is a pair, $(A,\ \alpha)$, where A is an object of the category and $\alpha : F(A) \to A$.

Applicative Bisimulation [Abramsky 90] A **quasi–applicative transition system** (quasi–ats) is a structure (A, ev) where: $ev : A \rightharpoonup (A \to A)$ (ev is a partial function from A to functions from A to A).

Notation:

$$\begin{array}{rcl} a \Downarrow_{qats} b & \equiv & a \in dom(ev) \wedge ev(a) = b \\ a \Downarrow_{qats} & \equiv & a \in dom(ev) \\ a \Uparrow_{qats} & \equiv & a \notin dom(ev) \\ Rel(A) & \equiv & \text{the set of relations on } A \end{array}$$

Let (A, ev) be a quasi-ats. Define:

$F(\mathcal{R}) \stackrel{\text{def}}{=} \{\langle a, b\rangle : \exists f.a \Downarrow_{qats} f \Rightarrow (\exists g.b \Downarrow_{qats} g \wedge \forall c \in A.f(c)\mathcal{R}g(c))\}$

$\mathcal{R} \in Rel(A)$ is an **applicative bisimulation** iff $\mathcal{R} \subseteq F(\mathcal{R})$.

Big Step Evaluation [Gordon 95b] **Big step evaluation** is specified inductively by the two rules:

$$\frac{}{\lambda x.e \Downarrow_u \lambda x.e} \tag{A.1}$$

$$\frac{a \Downarrow_u \lambda x.e \quad e[b/x] \Downarrow_u v}{a\ \ b \Downarrow_u v} \tag{A.2}$$

Bisimilar Two expressions are said to be bisimilar if they are related by some equivalence $\sim$ which is the greatest fixedpoint of a function on relations. In particular $\sim$ has the property that all pairs of expression obtained from some pair in $\sim$ by the use of transitions or destructors are also in $\sim$.

Bisimulation If $\mathcal{F}$ is a monotone function on relations then a bisimulation is a relation $\mathcal{R}$ such that $\mathcal{R} \subseteq \mathcal{F}(\mathcal{R})$. A bisimulation is a subset of the greatest fixedpoint of $\mathcal{F}$.

Bisimulation Explosion An effect encountered when running *CoCLAM*. The size of the representation of the bisimulation becomes so large that the program can no longer process it in reasonable time.

Bisimulation Extension The process by which a trial bisimulation is extended by a new pair scheme taken from a failed goal.

Bisimulation Generalisation The process by which a trial bisimulation is amended by replacing one or more pair schema by a generalisation.

Bisimulation Proof Method A term from CCS. The Bisimulation Proof method is used to show the equivalence of two CCS states. It is essentially the same as coinduction.

Call–By–Need Evaluation An evaluation strategy for functional programming languages by which functions are only evaluated if their results are needed by another function and are only evaluated as far as they are required by that function.

Category [Aczel 97] A **category** consists of **objects** and **arrows**. An arrow, f, between two objects A and B is written $f : A \to B$. A category also comes with two operations

1. If $f : A \to B$ and $g : B \to C$ then $(g,\ f)$ is called a **composable pair** and the **composite** arrow is $g \circ f : A \to C$. Furthermore for composable pairs $(g,\ f)$ and $(h,\ g)$, $(h \circ g) \circ f = h \circ (g \circ f)$.
2. An assignment of an arrow id_A to each object A, called the **identity** on A, such that for $f : A \to B$, $f \circ id_A = id_B \circ f = f$.

Category Theory Category theory is a formalism that describes the passage from one type of mathematical object to another.

CCS An abbreviation for the *Calculus of Communication Systems* originated by Milner [Milner 89]. CCS represents states and communications between states as labelled transition systems. Moreover, coinduction, as the Bisimulation Proof method, and greatest fixedpoints are a central part of the Calculus.

CLAM *CLAM* is a proof planning system intended primarily for planning inductive proofs which has been developed by the Edinburgh Mathematical Reasoning Group.

Coalgebra [Aczel 97] If F is an endofunctor then an F**-coalgebra** is a pair, $(A,\ \alpha)$, where A is an object of the category and $\alpha : A \to F(A)$.

CoCLAM *CoCLAM* is a proof planning system built from *CLAM* for performing coinductive proofs.

Coinduction Coinduction is a proof principle. It is the dual of induction and is used to prove the equivalence of lazy or infinite objects.

Coinduction Conclusion That part of the statement of a subgoal in a coinductive proof that requires us to prove that some pair of expressions is in the trial bisimulation.

Coinduction Hypothesis That part of the statement of a subgoal in a coinductive proof that states what pairs of expressions are in the trial bisimulation.

Coinductive Datatype It is usual for recursive datatypes to be defined inductively, essentially as the least fixedpoint of a function. However, this will exclude some objects that may be desired, such as $[M, M, \cdots]$, the infinite list of Ms. Hence a datatype may be defined coinductively as the greatest fixedpoint of the function.

Coinductive Definition This term can be applied both to functions and datatypes. When applied to datatypes it means that the members of that type are defined as being the members of some greatest fixedpoint. When applied to a function it is sometimes referred to as corecursion. It defines a functions in terms of the outcome of transitions applied to the values of that function, or the effect of destructors upon that function or the weak head normal form of the function. It is related to $unfold$.

Confluence [Bundy 83] A set of rules is confluent if whenever an expression, exp, can be rewritten in two different ways, say to $int1$ and to $int2$, then $int1$ and $int2$ can both be rewritten to some common rewriting, $comm$.

Congruence A relation, $\mathcal{R}$, is a congruence relation if it is an equivalence relation and if when $a\,\mathcal{R}\,b$ then for any experiment $\mathcal{E}(-)$ $\mathcal{E}(a)\,\mathcal{R}\,\mathcal{E}(b)$

Constructor A constructor is an operator on objects of some type. A constructor builds a new object of that type. Constructors are used to define types in terms of how the elements of the type can be built from each other.

Context [Gordon 95b] Let a **context** $\mathcal{C}$, be an expression such that there are no free variables in $\mathcal{C}$, except for $\{-\}$. This variable is a **hole** that will be filled in.

Contextual Equivalence [Gordon 95b] Let $\mathcal{C}$ be a context, define the relation $\simeq$, **contextual equivalence**, as follows:

$$\mathcal{C}[a] \stackrel{\text{def}}{=} \mathcal{C}[a/-] \tag{A.3}$$

$$a \Downarrow \stackrel{\text{def}}{=} \exists v.(a \Downarrow v) \tag{A.4}$$

$$a \precsim b \stackrel{\text{def}}{=} \forall \mathcal{C}(\mathcal{C}[a] \Downarrow \Rightarrow \mathcal{C}[b] \Downarrow) \tag{A.5}$$

$$a \simeq b \stackrel{\text{def}}{=} a \precsim b \& b \precsim a \tag{A.6}$$

Concurrency Workbench (CWB) The Concurrency Workbench is an automated environment for reasoning about finite automata in CCS. It includes coinduction, as the Bisimulation Proof Method, as a proof principle.

Corecursion See Coinductive Definition.

Denotational Semantics Denotational semantics describes the objects and processes in a programming language in terms of objects and processes in some other language (often the language of mathematics). Denotational semantics are *compositional* that is, given some expression in the language it doesn't matter whether the expression is evaluated or executed in the language and then the result translated to the other language or whether the constituent parts of the expression are translated first and the result determined in the new language.

Destructor Destructors are generally linked to constructors. If a constructor specifies that $c(t)$ is a member of a type constructed from t then the appropriate destructor applied to $c(t)$ brings you back to t. Destructors have been used to describe types as constructors do.

Difference Matching If one expression is embedded in another then difference matching is a process which annotates the expression it is embedded in to identify the additional structure in this new expression.

Divergence Check The Divergence check is the process used by the Revise Bisimulation critic in *CoCLAM* to check a sequence of expressions to see if they are diverging.

Dynamic Semantics The dynamic semantics of a program are determined at run–time. In functional languages the dynamic semantics describe its evaluation behaviour.

Endofunctor [Aczel 97] A functor from a category C to itself is called an **endofunctor** on C.

Environment An environment is a finite map from variables to types.

Erasure The erasure of a term annotated by difference matching is the term without any of the annotations present.

Eureka Step A Eureka step in a proof is one that requires more than just the blind following of some rules. It usually involves the introduction of some new information, such as an existential witness.

Experiment An experiment, $\mathcal{E}$, is a context of the form $-a$, where a is a program.

F–**Dense** [Gordon 95c] A set is F–**dense** iff $X \subseteq F(X)$

Fertilization Fertilization is the term given to the process by which a hypothesis (usually an induction hypothesis) is utilised to prove some goal. Either the hypothesis is used directly (strong fertilization) or it may be used as a rewrite rule (weak fertilization).

F–Homomorphism [Rutten 96] If $(S,\ \alpha_S)$ and $(T,\ \alpha_T)$ are F–coalgebras for some arbitrary endofunctor, F then $f : S \to T$ is an F–**homomorphism** if $F(f) \circ \alpha_S = \alpha_T \circ f$.

Final Coalgebra A coalgebra, A, is a final coalgebra if for every other coalgebra, B, in the category there is a unique arrow $f : B \to A$.

Fixedpoint A **fixedpoint** of a function $\mathcal{F}$ is a domain D such that

$$\mathcal{F}(D) = D$$

Fold A program transformation. A single fold step rewrites an expression by matching it with the left–hand side of an equation or function definition and replacing it with the RHS. $fold$, the function, can be used to generalise recursive/inductive definitions. It is also an arrow in category theory. Given an initial F–algebra $\pi : F(P) \to P$ and an arrow $\alpha : F(P) \to F(S)$ then $fold : F(S) \to S$.

Functional Programming Programming in a functional language consists of building definitions and using the computer to evaluate expressions. The primary role of the programmer is to construct a function to solve a given problem. This function, which may involve a number of subsidiary functions, is expressed in notation that obeys normal mathematical principles. The primary role of the computer is to act as an evaluator or calculator: its job is to evaluate expressions and print the results.

A characteristic feature of functional programming is that if an expression possesses a well–defined value, then the order in which a computer may carry out the evaluation does not affect the outcome. In other words, the meaning of an expression is its value and the task of the computer is simply to obtain it. It follows that expressions in a functional language can be constructed, manipulated and reasoned about, like any other kind of mathematical expression, using more or less familiar algebraic laws. [Bird & Wadler 88]

Functor [Aczel 97] If C_1 and C_2 are categories, a **functor**, F, from C_1 to C_2, written $F : C_1 \to C_2$, consists of:

1. An assignment of an object $F(A)$ of C_2 to each object A of C_1.
2. An assignment of an arrow $F(f) : F(A) \to F(B)$ of C_2 to each arrow $f : A \to B$ of C_1.

These assignments have the following properties:

- For each object, A, in C_1, $F(id_A) = id_{F(A)}$.
- For each composable pair $(g,\ f)$ in C_1, $(F(g),\ F(f))$ is composable in C_2 and $F(g \circ f) = F(g) \circ F(f)$

Greatest Fixedpoint The greatest fixedpoint of a function $\mathcal{F}$ is defined as [Paulson 93]

$$gfp(\mathcal{F}) = \bigcup\{\mathcal{A} \mid \mathcal{A} \subseteq \mathcal{F}(\mathcal{A})\}$$

Head Normal Form [Barendregt 84] A λ-term M is a **head normal form**(HNF) if M is of the form $\lambda x_1 \cdots x_n.xM_1 \cdots M_m, n, m \geq 0$. Where x is a variable (possibly one of the x_i, $i \leq n$ and the M_j are λ–terms (not necessarily in HNF).

Haskell In the mid-1980s, there was no "standard" non-strict, purely-functional programming language. A language-design committee was set up in 1987, and the Haskell language is the result. At the time of writing, version 1.4 is the latest version of the language (from the `comp.lang.functional` FAQ).

HOL The HOL system is a powerful and widely used computer program for constructing formal specifications and proofs in higher order logic. The system is used in both industry and academia to support formal reasoning in many different areas, including hardware design and verification, reasoning about security, proofs about real-time systems, semantics of hardware description languages, compiler verification, program correctness, modelling concurrency, and program refinement. HOL is also used as an open platform for general theorem-proving research. (`http://www.dcs.gla.ac.uk/~tfm/fmt/hol.html`)

Implicit Induction In implicit induction the conjecture to be proved is added to the axioms. A Knuth–Bendix completion procedure is then applied to the whole system. If no inconsistency is derived by the procedure, the the conjecture is an inductive theorem. [Bouhoula & Rusinowitch 93]

Induction A logical inference rule. Application of induction usually produces one or more base cases together with one or more step cases (implications in which assuming the theorem to be true for one (or more cases) allows its truth to be inferred for other cases). In the proof plan for induction, the former subgoals contain wave fronts which mark the differences between the induction hypothesis and the induction conclusion, and they are proved by rippling these differences away until fertilisation can be performed. [Richardson 95]

Induction Hypothesis That part of the step case goal in induction in which the theorem is assumed to be true for some value, or all values less than one particular value (according to the well–founded order).

Induction Conclusion That part of the step case goal in induction which is a subgoal requiring the proof that the theorem is true for some particular expression. In structural constructor style induction, as used in this thesis, this value is generally constructed from the expression in the induction hypothesis using constructors.

Induction Scheme An instance of the induction principle or rule used in a proof gives rise to an induction scheme which determines the expression used in the induction conclusion.

Inductive Datatype An inductive datatype is a datatype that contains members of a least fixedpoint. It is generally described in terms of how members of the datatype may be constructed from other members.

Inductive Inference Inductive inference is an area of artificial intelligence which examines how rules may be generalised from examples.

Inductive Logic Programming Inductive logic programming examines techniques by which logic programs can be synthesized from examples of their intended truth values for given inputs.

Initial Algebra An algebra, A, is an initial algebra if for every other algebra, B, in the category there is a unique arrow $f : A \to B$.

Isabelle Isabelle is a generic theorem prover. New logics are introduced by specifying their syntax and rules of inference. Proof procedures can be expressed using tactics and tacticals. (`http://www.cl.cam.ac.uk/Research/HVG/Isabelle/`).

Knuth–Bendix Completion The Knuth–Bendix completion procedure generates a confluent set of rewrite rules by repeatedly superposing left hand sides of rewrite rules and adding any generated critical pairs as new rewrite rules. This process may fail to terminate, producing a divergent set of rewrites.

Labelled Transition System In its most basic form a labelled transition system is a binary relation on terms or processes indexed by a set of labels. Within the semantics of lazy languages the labels tend to be type destructors and the RHS of the relation is the effect of applying that type destructor to the LHS. Often the labels (or *transitions*) are considered to be things you can "observe" about the program.

λ–**Calculus** The λ–calculus is a model of computation developed by Alonzo Church. It is one of the most widely used models and most functional languages are elaborated forms of the λ–calculus. It's most distinctive feature is the use of λ–abstractions to form functions from object. $\lambda x.t$ indicates that x is some variable in t, a function expression, to which an argument may be supplied.

Lazy Evaluation Lazy Evaluation is an evaluation technique which only evaluates function calls when they are needed and as far as they are needed. It is also commonly used to mean that any given function call is only evaluated once and it's results saved to be re–used later should that call be made again.

Lazy List A lazy list is a list that may be either finite or infinite in length.

LCF Edinburgh LCF is a programmable proof checker. Users can write proof procedures in ML, *proof tactics*, rather than typing repetitive commands. LCF has an abstract type of theorems which ensures that each inference rule is a function from theorems to theorems by type–checking. LCF was the first theorem–prover to do this. The widely used functional programming language, ML, was designed for LCF.

Least Fixedpoint The least fixed point of a function $\mathcal{F}$ is defined as [Paulson 93]

$$lfp(\mathcal{F}) = \bigcap\{\mathcal{A} \mid \mathcal{F}(\mathcal{A}) \subseteq \mathcal{A}\} \tag{A.7}$$

Lemma Calculation Lemma calculation is part of the lemma speculation critic in CL^AM. It is used to hypothesize new lemmata in the case of a failed proof attempt. It does not require the use of any middle–out reasoning or higher order unification. It is a straightforward calculation procedure.

Lemma Speculation Lemma speculation is part of a critic in CL^AM. It is used to hypothesize new lemmata in the case of a failed proof attempt. Lemma speculation uses middle–out reasoning and higher order unification in the process of hypothesizing lemmata.

LTS An abbreviation for labelled transition system.

Meta–Level Meta–level terms are an abstraction of object–level terms in which hypotheses and type information may be missing. Certain information may be added in the form of annotations, either annotating object–level terms to give wave terms or marking certain hypotheses as induction hypotheses, amongst other things. [Richardson 95]

The axioms and expressions of some logic are often described as the object–level terms and the rules of inference of that logic allow object–level reasoning about the axioms and expressions. At the meta–level reasoning is about the inference rules.

Middle–Out Reasoning Middle–out reasoning was first described by Bundy et al [Bundy *et al* 90a]. Middle–out reasoning postpones making eureka steps for a long as possible in the proof process, adapting methods to cope with partial information where necessary. In this way more information about the nature of the eureka step can be determined.

Non–Strict Evaluation Non–Strict Evaluation is an evaluation procedure that does not require all expressions in a function call to be evaluated only those that are needed to determine the result and then these need only be evaluated as far as is required. It is often referred to as *call–by–need* or *lazy evaluation*, although the latter of these often implies that any function call is only evaluated once.

Non–Well–Founded Sets Non–well–founded sets are sets containing infinite sequences of nested subsets. They were studied first by Aczel [Aczel 88].

Object–Level Well–formed terms, proofs and inference steps of the logic in use. [Richardson 95]

Operational Semantics Operational semantics describe the idealised behaviour of a programming language on some abstract machine. It is often expressed with the use of labelled transition systems.

Oyster A Prolog implementation of a variant of Martin–Löf's Type Theory. Originally Oyster was based on NuPRL. Both Oyster and NuPRL are tactic–based, goal–directed theorem provers. [Richardson 95]

Pair Scheme A set consisting of one related pair of objects containing free or universally quantified variables is a **pair scheme**.

Bisimulations in functional languages are invariably the unions of one or more pair schema.

Partial Wave Rule Match A partial wave rule match occurs when an expression can be embedded into the LHS of a wave rule, but would need to have some extra structure present before the two expressions could be unified.

Potential Wave Front Potential wave fronts are used in *middle–out reasoning* and indicate a possible position for a wave front.

Proof Critic A proof critic describes a process for patching a failed proof attempt. Generally a proof critic analyses failed method pre–conditions.

Proof Method A proof method is a partial specification of a tactic. It is composed of several slots. When a method applies to a goal, it generates a list of subgoals to be proved. Associated with each method is the tactic which constructs the part of the object–level proof which corresponds to the part of the meta–level proof constructed by the method. [Richardson 95]

Proof Planning Proof planning is a technique for guiding the search for a proof in automated theorem proving. A proof plan is an outline or plan of a proof. To prove a conjecture, proof planning first constructs the proof plan for a proof and then uses it to guide the construction of the proof itself. (Proof planning FAQ)

Proof Strategy A sequence of methods and submethods in the method database which generates proof plans of a particular form [Richardson 95]. Extended in this thesis to include the presence of critics.

Proof Tactic A proof tactic is a sequence of inference rules in some proof checking system. It is a program which when executed performs part of a proof.

PVS PVS is a verification system: that is, a specification language integrated with support tools and a theorem prover. It is intended to capture the state-of-the-art in mechanized formal methods and to be sufficiently rugged that it can be used for significant applications. (`http://www.csl.sri.com/pvs.html`).

Reduction Rule A reduction rule is a measure decreasing rewrite rule. They are used in the evaluation of functional programs and are generally derived from function definitions.

Rewrite Rule [Thomas & Jantke 89] A **rewrite rule** is an ordered pair of terms of the form, $l \leadsto r$.

Ripple Analysis Ripple analysis is the process by which an induction variable is chosen by CL^AM. It uses a process of lookahead through possible applications of wave rules.

Rippling The successive application of wave rules [Richardson 95]

Set Induction Set induction is the process of using induction with the general induction rule: $\frac{[\mathcal{S}]\subseteq\mathcal{S}\subseteq\mathcal{S}'}{lfp([-])\subseteq\mathcal{S}'}$. In set induction the eureka step is the choice of the set $\mathcal{S}$.

Scheme Induction Scheme induction is the process of using induction with a specialised induction scheme. In this form of induction the correct choice of induction scheme is a eureka step.

Skeleton In the case of a single wave hole, the skeleton of a wave term is the part of the wave term which is either completely outside the wave fronts, or is inside a wave front and underlined. The wave hole is the set of the underlined expressions. When a wave rule is applied to a wave term, the skeleton of the result is a subset of the skeleton before wave rule application. [Richardson 95]

Small Step Reduction [Gordon 95b] **Small step reduction**, $a \overset{\text{red}}{\leadsto} b$, is specified inductively by the rules:

$$\frac{}{(\lambda x.e)b \overset{\text{red}}{\leadsto} e[b/x]} \tag{A.8}$$

$$\frac{a \overset{\text{red}}{\leadsto} a'}{\mathcal{E}[a] \overset{\text{red}}{\leadsto} \mathcal{E}[a']} \tag{A.9}$$

Static Semantics A program's static semantics is determined at compile time. The static semantics of a typed functional programming language describe the typing rules for the language.

Strict List A strict list is of finite length.

τ **Action** In CCS a τ–action is an internal action on two states that is unobserved by the user.

Transition A transition is a label on the transition relation in a labelled transition system.

Trial Bisimulation A trial bisimulation is a relation used in the course of a coinductive proof. The object of the part of the proof where the relation appears will be to prove that it is a bisimulation.

Unfold A program transformation. A single unfold step rewrites an expression by matching it with the right–hand side of an equation or function definition. $unfold$, the function, can be used to generalise corecursive/coinductive definitions. It is also an arrow in category theory. Given a final F–coalgebra $\pi : P \to F(P)$ and an arrow $\alpha : S \to P$ then $unfold : S \to F(S)$.

Value In functional languages values are a distinguished set of expressions which indicate a satisfactory end to an evaluation process.

Wave Front A wave front is represented by a box surrounding a term, some of whose subterms are underlined. The underlined subterms are the *wave holes*. A wave front has a direction which is indicated by an arrow attached to the box. [Richardson 95]

Wave Hole See *wave front*

Wave Measure A measure on annotated terms, i.e. a mapping from annotated terms to some well–founded order. [Richardson 95]

Wave Rule A rewrite rule which has been annotated such that the skeleton of the left hand side is a superset of the skeleton of the right hand side, and the wave measure of the right and side is strictly less than the wave measure of the left hand side. The skeleton preserving and measure–decreasing properties allow a proof of the termination of rippling. [Richardson 95]

Weak Head Normal Form (Adapted from [Peyton Jones 87]) A λ–term M is a **weak head normal form**(WHNF) if M is of the form
$\lambda x.M'$ or $xM_1 \cdots M_m, m \geq 0$.

Well–Founded Order An order, $\succ$, is well–founded if there are no infinite descending chains $a \succ a' \succ \cdots$.

Appendix B

Results

B.1 Introduction

This appendix lists the function definitions, lemmata and theorems used in the experimental evaluation of *CoCLAM* discussed in chapter 7.

B.2 Function Definitions

addl	$addl(L, nil) = L$ $addl(nil, L) = L$ $addl(H_1 :: T_1, H_2 :: T_2) = (H_1 + H_2) :: addl(T_1, T_2)$
ap	$ap(ap(<>, X), Y) = X <> Y$
$<>$ (append)	$nil <> L = L$ $H :: T <> L = H :: (T <> L)$
p$<>_l$ (append–lazy)	$nil <>_l nil = nil$ $nil <>_l H :: T = H :: (nil <>_l T)$ $H :: T <>_l L = H :: (T <>_l L)$
atend	$atend(X, nil) = X :: nil$ $atend(X, Y :: Z) = Y :: atend(X, Z)$
brsearch	$brsearch(node(A, nil), nil) = A :: nil$ $brsearch(node(A, nil), H :: T) = A :: brsearch(H, T)$ $brsearch(node(A, H :: T), nil) = A :: brsearch(H, T)$ $brsearch(node(A, L), H :: T) = A :: brsearch(H, T <> L)$
brswap	$brswap(leaf(X)) = leaf(X)$ $brswap(node(A, L, R)) = node(A, brswap(R), brswap(L))$
del	$del(N, nil) = nil$ $X = Y \Rightarrow del(X, Y :: Z) = del(X, Z)$ $X \neq Y \Rightarrow del(A, Y :: Z) = Y :: del(X, Z)$
double	$double(0) = 0$ $double(s(N)) = s(s(double(N)))$

dpsearch	$dpsearch(node(A, nil), nil) = A :: nil$ $dpsearch(node(A, nil), H :: T) = A :: dpsearch(H, T)$ $dpsearch(node(A, H :: T), L) = A :: dpsearch(H, T <> L)$
drop	$drop(0, L) = L$ $drop(N, nil) = nil$ $drop(s(N), H :: T) = drop(N, T)$
dup	$dup(X, 0) = nil$ $dup(N, s(M)) = N :: dup(N, M)$
evenl	$evenl(nil) = nil$ $evenl(H :: nil) = nil$ $evenl(H_1 :: H_2 :: T) = H_2 :: evenl(T)$
exp	$exp(N, 0) = s(0)$ $exp(0, s(I)) = 0$ $exp(s(0), s(I)) = s(0)$ $exp(s(s(N)), s(I)) = s(N + (s(s(N)) * exp(s(s(N)), I)))$
explode	$explode(nil) = nil$ $explode(H :: T) = (H :: nil) :: explode(T)$
flip	$flip(0) = s(0)$ $flip(s(0)) = 0$
flip$_{bv}$	$flip_{bv}(\bot) = T$ $flip_{bv}(T) = \bot$
flattenA	$flattenA(nil) = nil$ $flattenA(H :: T) = H <> flattenA(T)$
foldr	$foldr(F, nil, E) = E$ $foldr(F, H :: T, E) = F(H, foldr(F, T, E))$
from	$from(N) = N :: from(s(N))$
h	$h(F, X) = X :: map(F, h(F, X))$
half	$half(0) = 0$ $half(s(0)) = 0$ $half(s(s(N))) = s(half(N))$
idlist	$idlist(nil) = nil$ $idlist(H :: T) = H :: idlist(T)$
idnat	$idnat(0) = 0$ $idnat(s(N)) = s(idnat(N))$
∞ (infinity)	$\infty = s(\infty)$
inflist	$inflist(X, F, G) = F(X) :: inflist(G(X), F, G)$
iterates	$iterates(F, M) = M :: iterates(F, F(M))$
jump	$jump(N, M) = N :: jump(N + M, M)$
lconst	$lconst(M) = M :: lconst(M)$
length	$length(nil) = 0$ $length(H :: T) = s(length(T))$
loop	$loop(F, A, B) = map(F, merge2(A :: loop(F, A, B), B))$
loop2	$loop2(F, A, nil) = nil$ $loop2(F, A, H :: T) = F(A, H) :: loop2(F, F(A, H), T)$
lswap	$lswap(A, B) = A :: lswap(B, A)$
map	$map(F, nil) = nil$ $map(F, H :: T) = F(H) :: map(F, T)$

map2	$map2(F, nil) = nil$ $map2(F, H :: T) = map(F, H) :: map2(F, T)$
map3	$map3(F, nil) = nil$ $map3(F, H :: T) = ap(F, H) :: map(F, T)$
merge	$merge(nil, A) = A$ $merge(A, nil) = A$ $merge(H_1 :: T_1, H_2 :: T_2) = H_1 :: H_2 :: merge(T1, T2)$
merge2	$merge2(nil, A) = nil$ $merge2(A, nil) = nil$ $merge2(H_1 :: A, H_2 :: B) = (H_1, H_2) :: merge2(A, B)$
$-_l$ (minus–lazy)	$0 -_l 0 = 0$ $0 -_l N = s(0 -_l s(N))$ $s(N) -_l Y = s(N -_l Y)$
numparity	$numparity(X, nil) = nil$ $numparity(X, \perp :: L) = even_l(X) :: numparity(s(X), L)$ $numparity(X, T :: L) = odd_l(X) :: numparity(s(X), L)$
oddl	$oddl(nil) = nil$ $oddl(H :: nil) = H :: nil$ $oddl(H_1 :: H_2 :: T) = H_1 :: oddl(T)$
ones	$ones = 1 :: ones$
parity	$parity(B, nil) = nil$ $parity(B, \perp :: L) = B :: parity(B, L)$ $parity(B, T :: L) = flip_{bv}(B) :: parity(flip_{bv}(B), L)$
$+$ (plus)	$0 + X = X$ $s(N) + X = s(N + X)$
$+_l$ (plus–lazy)	$0 +_l 0 = 0$ $0 +_l s(N) = s(0 +_l N)$ $s(N) +_l X = s(N +_l X)$
pred-l	$pred - l(0) = 0$ $pred - l(s(0)) = 0$ $pred - l(s(s(N))) = s(N)$
$(\cdots)^N$ (repeat apply)	$F^0(X) = X$ $F^{s(N)}(X) = F(F^N(X))$
replace	$replace(A, B, nil) = nil$ $B \neq H \Rightarrow replace(A, B, H :: T) = H :: replace(A, B, T)$ $B = H \Rightarrow replace(A, B, H :: T) = A :: replace(A, B, T)$
tconst	$tconst(M) = node(M, tconst(M), tconst(M))$
tconstinf	$tconstinf(M) = node(M, tconstinf(M) :: nil)$
tick	$tick = s(0) :: tock$
$*$ (times)	$0 * Y = 0$ $Y * 0 = 0$ $s(N) * Y = Y + X * Y$
$*_l$ (times–lazy)	$0 *_l Y = 0$ $Y *_l 0 = 0$ $s(N) *_l s(Y) = s(X +_l s(X) *_l Y)$
tock	$tock = 0 :: tick$
tswap	$tswap(A, B) = node(A, tswap(A, B), tswap(B, A))$

tswap2	$tswap2(A, B) = node(B, tswap2(B, A), tswap2(A, B))$
zigzag	$zigzag(nil, nil, nil) = nil$ $zigzag(nil, nil, H :: T) = zigzag(nil, H :: T, nil)$ $zigzag(nil :: T, nil, L) = zigzag(T, L, nil)$ $zigzag((H_H :: T_H) :: T, nil, L) = H_H :: zigzag(T, L, T_H :: nil)$ $zigzag(L_1, nil :: T, L_2) = zigzag(L_1, T, L_2)$ $zigzag(L_1, (H_H :: T_H) :: T, L_2) = H_H :: zigzag(L_1, T, T_H :: L_2)$

B.3 Theorems

In order to proof plan any theorem it is necessary to have at least the relevant function definitions available. A standard set of lemmata shown in §B.5.1 were also made available to all proof plan attempts. If any additional lemmata were required by the attempt to prove a theorem then they are listed with the theorem below.

The result column indicates the result of the proof plan attempt. The theorem is either proved or is subject to one of the errors discussed in chapter 7, i.e. bisimulation explosion, false hypothesis not recognised, memory error during evaluation, matching of $(\cdots)^n$, incorrect generalisation, initial definition of bisimulation, substitution error or transition error.

B.3.1 Development Set

Name	Theorem	Lemmata	Result
app1right	$X <> nil = X$		Proved
appntconst	$\lambda l.node(A, L, L)^{\infty}(nil) = tconst(A)$		Proved
assapp	$L <> (M <> N) = (L <> M) <> N$		Proved
asslapp	$(X <>_l Y) <>_l Z = X <>_l (Y <>_l Z)$		Proved
asslplus	$(M +_l N) +_l L = M +_l (N +_l L)$		Proved
assp	$X + (Y + Z) = (X + Y) + Z$		Proved
comapp	$length(X <> Y) = length(Y <> X)$		Proved
commthree	$(Z *_l X) *_l Y = (Z *_l Y) *_l X$	disttwo times2right	Bisimulation Explosion
comp	$A + B = B + A$	ssid	Proved
doublehalf	$double(half(N)) = N$		Proved
dpbrtconst	$dpsearch(tconstinf(M), nil) =$ $brsearch(tconstinf(M), nil)$		Proved
everylconst	$lconst(M) = evenl(lconst(M))$		Proved
everylswap	$lconst(M) = evenl(lswap(M, N))$		Proved
expplus	$exp(N, I + J) = exp(N, I) * exp(N, J)$	disttwo dist times1right times2right	Bisimulation Explosion
exptimes	$exp(X, N * M) = exp(exp(X, N), M)$	expplus dist	Bisimulation Explosion

Name	Theorem	Lemmata	Result
flattenexplode	$flattenA(explode(L)) = L$		Proved
grahamsthm	$loop2(F, A, B) = loop(F, A, B)$		Gen. Error
halfplus1	$half(X + X) = X$		Proved
hiterates	$h(F, X) = iterates(F, X)$		Proved
infl1iter	$iterates(\lambda y.A + y, B) = inflist(0, \lambda x.(x * A) + B, s)$		Bisimulation Explosion
infl1lconst	$lconst(N) = inflist(M, idnat, idnat)$		Proved
infl1nat	$inflist(A, idnat, s) = jump(A, 1)$		Proved
iterateslc	$lconst(M) = iterates(idnat, M)$		Proved
jumpfrom	$jump(N, 1) = from(N)$		Proved
lappnilr	$L <>_l nil = L$		Proved
lendouble	$length(X <> X) = double(length(X))$		Proved
lenlconst	$length(M :: lconst(M)) = \infty$		Proved
lenplus	$length(X <> Y) = length(X) + length(Y)$		Proved
lplus0l	$0 +_l N = N$		Proved
lminus–plus	$(Z + X) -_l (Z + Y) = X -_l Y$		Matching of $(\cdots)^N$
lminus–succ	$X -_l Y = s(X) -_l s(Y)$		Proved
lswaplconst	$A \neq B \Rightarrow del(B, lconst(A)) = del(B, lswap(A, B))$		Proved
lswaplmerge	$merge(lconst(A), lconst(B)) = lswap(A, B)$		Proved
map3iter	$map(*(s(s(s(0)))), jump(0, s(0))) = iterates(+(s(s(s(0)))), 0)$		Matching of $(\cdots)^N$
mapapp	$map3(W, X <> Y :: Z) = map3(W, X) <> map3(W, Y :: nil <> Z)$		Proved
mapidnat	$lconst(M) = map(idnat, lconst(M))$		Proved
mapdouble	$map(double, L) = map(\lambda X + X, L)$		Proved
mapfinfl1	$map(H, iterates(T, A)) = inflist(A, H, T)$		Proved
mapflip	$map(flip_{bv}, map(flip_{bv}, L)) = L$		Proved
mapfold	$map(F, L) = foldr(\lambda x.\lambda t.F(x) :: t, L, nil)$		Proved
mapid	$map(L, idnat) = L$		Proved
mapiter	$iterates(F, F(M)) = map(F, iterates(F, M))$		Proved
mapiter2	$iterates(F, X) = X :: map(F, iterates(F, X))$		Proved
mapjump	$map(lconst, jump(0, 1)) = iterates(map(s), lconst(0))$		Proved
mapthm	$map(F, map(G, L)) = map(F \circ G, L)$	compfunc	Proved
mergezigzag	$merge(merge(oddl(A), oddl(B)), merge(evenl(A), evenl(B))) = zigzag(nil, A :: nil, B :: nil)$		Memory Error
nat1	$iterates(s, s(s(0))) = jump(s(s(0)), s(0))$		Proved
nat2	$jump(0, s(0)) = merge(jump(0, s(s(0))), jump(s(0), s(s(0))))$		Proved Proved
oneslconst	$ones = lconst(s(0))$		Proved

Name	Theorem	Lemmata	Result
parityT0	$parity(T, L) = numparity(0, T)$	evenlem oddlem evenlem2 oddlem2	Proved
plus2right	$X + s(Y) = s(X + Y)$		Proved
pluslem2	$X + s(0) = s(X)$		Proved
plusxx	$X + s(X) = s(X + X)$	ssid	Proved
tbrswap	$tswap(A, B) = brswap(tswap2(B, A))$		Proved
zeroplus	$X = 0 \wedge Y = 0 \Rightarrow X + Y = 0$		Proved

B.3.2 Test Set

Name	Theorem	Lemmata	Result
appatend	$X <> Y :: Z = atend(Y, X) <> Z$		Proved
appiterates	$iterates(F, X) <> N = iterates(F, X)$		Proved
applapp	$L_1 <> L_2 = L_1 <>_l L_2$		Proved
appnlconst	$:: (N)^{\infty}(nil) = lconst(N)$		Proved
assconsapp	$P <> V_0 :: L = (P <> V_0 :: nil) <> L$		Proved
assm	$A * (B * C) = (A * B) * C$		Bisimulation Explosion
brswap	$T = brswap(brswap(T))$		Proved
comaddl	$addl(M, N) = addl(N, M)$		Generalisation Error
comlplus	$M +_l N = N +_l M$		Proved
comm	$X * Y = Y * X$		Bisimulation Explosion
dist	$A * (B + C) = A * B + A * C$		Bisimulation Explosion
disttwo	$(B + C) * A = (B * A) + (C * A)$		Bisimulation Explosion
doubleplus	$double(X) = X + X$		Proved
doubletimes1	$double(N) = s(s(0)) * N$		Proved
doubletimes2	$double(N) = N * s(s(0))$		Proved
dpsearchlc	$dpsearch(node(M, lconst(node(M, nil))), nil)$ $= lconst(M)$		Proved
dptconst	$dpsearch(tconstinf(M), nil) = lconst(M)$		Proved
dupinf	$dup(M, \infty) = lconst(M)$		Proved
flattenfold	$flattenA(map(\lambda a.map(\lambda b.a :: b :: nil),$ $iterates(F, N), lconst(M))) =$ $foldr(\lambda x.\lambda p.foldr(\lambda y.\lambda l.(x :: y :: nil) :: l,$ $iterates(F, N), p), lconst(M), nil)$		Bisimulation Explosion
gordon1	$map(\lambda x.idnat(x), Y) = idlist(Y)$		Proved
halfdouble	$half(double(N)) = N$		Proved
halflenapp1	$half(length(A <> B)) =$ $half(length(B <> A))$	ssid	Generalisation Error
halfplus2	$half(X + Y) = half(Y + X)$	ssid	Proved

Name	Theorem	Lemmata	Result
identrm	$X * s(0) = X$		Proved
infl1lswap	$lswap(0,1) = inflist(0, idnat, flip)$		Proved
lappnill	$nil <>_l L = L$		Proved
lconsta	$map(F, lconst(M)) = lconst(F(M))$		Proved
lconstaddl	$addl(lconst(M), lconst(N)) = lconst(M+N)$		Proved
lconstapp	$lconst(M) = lconst(M) <> L$		Proved
lconsteven	$lconst(T) = map(even, jump(0, s(s(0))))$	evenlem	Proved
lconstiter	$lconst(iterates(F, F(M))) = lconst(map(F, iterates(F, M)))$		Proved
lconstzigzag	$lconst(N) = zigzag(lconst(N) :: nil, nil, nil)$		Proved
lenlcit	$length(lconst(M)) = length(iterates(F, N))$		Proved
lenlconst	$\infty = length(lconst(M))$		Proved
lenmapcar	$length(map(F, L)) = length(L)$		Proved
lplus0r	$N +_l 0 = N$		Proved
ltimes2right	$X *_l s(Y) = X +_l X *_l Y$		Proved
mergedrop	$merge(merge(evenl(A), evenl(B)), drop(s(s(0)), merge(oddl(A), oddl(B)))) = drop(s(s(0)), zigzag(A :: B :: nil, nil, nil))$		Memory Error
mergeolel	$merge(oddl(L), evenl(L)) = L$		Proved
minus–pred	$pred(X -_l Y) = pred(X) -_l Y$		Proved
map2thm	$map2(F, map2(G, L)) = map2(F \circ G, L)$		Proved
mapaddl	$addl(L, L) = map(double, L)$		Proved
mapcarapp	$map(F, L_1 <> L_2) = map(F, L_1) <> map(F, L_2)$		Proved
maplconst	$map(\lambda m.L <> m, lconst(M)) = lconst(L <> M)$		Proved
natmap	$jump(0, s(0)) = 0 :: map(s, jump(0, s(0)))$		Proved
parityTeven	$even(N) \Rightarrow parity(T, L) = numparity(N, L)$		Def. of Bisimulation
pluslplus	$A +_l B = B + A$		Proved
plus1right	$X + 0 = X$		Proved
replacelswap	$replace(A, B, lswap(A, B)) = lconst(A)$		Finding Transitions
tconsttswap	$tconst(M) = tswap(M, M)$		Proved
ticktock	$tick = map(flip, tock)$		Substitution Error
times2right	$X * s(Y) = X + X * Y$		Proved
timesltimes	$X * Y = X *_l Y$		Bisimulation Explosion
zerotimes2	$X = 0 \vee Y = 0 \Rightarrow X * Y = 0$		Hypothesis Handling

B.4 Type Checking Theorems

B.4.1 Development Set

Name	Theorem	Lemmata	Result
brsearch	$T :_c int \Rightarrow brsearch(T, nil) :_c list(int)$		Proved
lswap	$M :_c int \wedge N :_c int \Rightarrow lswap(M, N) :_c list(int)$		Proved
tconst	$N :_c inttconst(N) :_c binarytree(int)$		Proved

B.4.2 Test Set

Name	Theorem	Lemmata	Result
app–type	$M :_c int \wedge N :_c int \Rightarrow M <> N :_c list(int)$		Proved
dpsearch	$T :_c tree(int) \Rightarrow dpsearch(T, nil) :_c list(int)$		Memory Error
infl1typ	$A :_c int \wedge H :_c (int \rightarrow int) \wedge T :_c (int \rightarrow int) \Rightarrow$ $inflist(A, H, T) :_c list(int)$		Failed
iterates	$M :_c int \wedge F :_c (int \rightarrow int) \Rightarrow$ $iterates(F, M) :_c list(int)$		Proved
lconst	$M :_c int \Rightarrow lconst(M) :_c list(int)$		Proved
map–type	$F :_c (int \rightarrow int) \wedge L :_c list(int) \Rightarrow$		Proved
	$map(F, L) :_c list(int)$		Failed
ones–type	$ones :_c list(int)$		
tswap	$M :_c int \wedge N :_c int \Rightarrow$ $tswap(M, N) :_c binarytree(int)$		Proved

B.5 Lemmata

B.5.1 Standard Lemmata

These are all special cases of the rules:

$$f^0(X) = X$$

$$f^{s(N)}(X) = f(f^N(X))$$

$$f^N(c(X)) = g(f^N(X))$$

where c is a constructor and f and g are functions.

conapn	$(::(H))^0(X) = X$ $(::(H))^{s(N)}(X) = H :: ((::(H))^N(X))$
conslem1	$(::(X))^N(H :: T) = X :: ((::(X))^N(T))$
hitlem1	$(map(F))^N(map(F, X)) = map(F, (map(F))^N(X))$
idnatapn	$idnat^N(0) = 0$ $idnat^N(s(X)) = s(idnat^N(X))$
mapapn	$(map(F))^N(nil) = nil$ $(map(F))^N(H :: T) = F^N(H) :: (map(F))^N(T)$
plusapn	$(+A)^N(0) = N * A$ $(+A)^N(s(X)) = s((+A)^N(X))$
plusapn2thm	$(+X)^N(X+Y) = X + ((+X)^N(Y))$
$(\cdots)^N$ (repeat apply)	$F^0(X) = X$ $F^{s(N)}(X) = F(F^N(X))$
sapn	$s^0(X) = X$ $s^{s(N)}(X) = s(s^N(X))$
sapn2thm	$s^N(s(X)) = s(s^N(X))$
ssapn	$(s \circ s)^0(X) = X$ $(s \circ s)^{s(N)}(X) = s(s((s \circ s)^N(X))$
ssapn2thm	$(s \circ s)^N(s(s(X))) = s(s((s \circ s)^N(X)))$
timesapn	$(*0)^N(X) = 0$ $(*(s(X)))^N(Y) = ((+Y)^N(X * Y)) + ((*X)^N(Y))$
timesapn2thm	$(*X)^{s(N)}(Y) = X * ((*X)^N(Y))$

It should be noted that some of these lemmata are not actually theorems (e.g. conslem1). These were only used when the list was contained by a *length* function so the actual value of the head was not important. This simplifying assumption was used to avoid having to provide a large number of such lemmata for, say, $length(L)$, $length(map(F, L))$ etc. However this does highlight the need for a lemma speculation critic where such lemmata could be speculated as needed rather than having to be supplied by hand.

B.5.2 Other Lemmata

compfunc	$\langle F(G(X)),\ F \circ G(X)\rangle \in\sim$
evenlem	$\langle T,\ even((s \circ s)^N(0))\rangle \in\sim$
evenlem2	$\langle \perp,\ even(s((s \circ s)^N(0)))\rangle \in\sim$
oddlem	$\langle \perp,\ odd((s \circ s)^N(0))\rangle \in\sim$
oddlem2	$\langle T,\ odd(s((s \circ s)^N(0)))\rangle \in\sim$
ssid	$s(s(X)) = s(s(X))$

Appendix C

Program Traces

C.1 Introduction

CoCLAM is built on top of the SICStus Prolog version of *CLAM*3 an implementation of *CLAM* that incorporated the critics mechanism developed by Ireland and Bundy [Ireland & Bundy 96]. This particular version of *CLAM* has never been made generally available, although other versions (which do not support critics) are available from `file://dream.dai.ed.ac.uk/pub/oyster-clam/`. Anyone wishing to obtain the core code for *CLAM*3 should contact `dream@dai.ed.ac.uk`.

The additional code for *CoCLAM* can be obtained from `http://dream.dai.ed.ac.uk/systems/coclam`.

This appendix shows traces of *CoCLAM* in operation, planning the proofs of $map(F, iterates(F, X)) = iterates(F, F(X))$ and $h(F, X) = iterates(F, X)$ – both these examples were discussed in chapter 3.

C.2 Traces and Plans

C.2.1 mapiter

```
| ?- plan(iteratesa).
loading thm(iteratesa)...done
iteratesa([])
==>m:int=>f:(int=>int)=>iterates(f,f of m)=mapcar(iterates(f,m),f)in int list
SELECTED METHOD at depth 0: coinduction_lts(union([range(int list-lam(m-int,lam\
(f-(int=>int),related(iterates(f,f of m),mapcar(iterates(f,m),f)))))]),[])
|iteratesa([1])
|==>subset(union([range(int list-lam(m-int,lam(f-(int=>int),related(iterates(f,\
f of m),mapcar(iterates(f,m),f)))))]),obs_fun of union([range(int list-lam(m-in\
t,lam(f-(int=>int),related(iterates(f,f of m),mapcar(iterates(f,m),f)))))]))in \
set(int list)
|SELECTED METHOD at depth 1: gfp_membership(obs_fun)
||iteratesa([1,1])
||v0:ih(m:int=>f:(int=>int)=>related(iterates(f,f of m),mapcar(iterates(f,m),f)\
```

```
)in int list)
||==>m:int=>f:(int=>int)=>(act(iterates(f,f of m))and act(mapcar(iterates(f,m),\
f)))in int list
||SELECTED METHOD at depth 2: intro1([iterates1-[1,1]],[])
|||iteratesa([1,1,1])
|||v0:ih(m:int=>f:(int=>int)=>related(iterates(f,f(m)),mapcar(iterates(f,m),f))\
in int list)
|||==>m:int=>f:(int=>int)=>(act(f(m)::iterates(f,f(f(m))))and act(mapcar(iterat\
es(f,m),f)))in int list
|||SELECTED METHOD at depth 3: intro1([iterates1-[1,1,2],mapcar2-[1,2]],[])
||||iteratesa([1,1,1,1])
||||v0:ih(m:int=>f:(int=>int)=>related(iterates(f,f(m)),mapcar(iterates(f,m),f)\
)in int list)
||||==>m:int=>f:(int=>int)=>(act(f(m)::iterates(f,f(f(m))))and act(f(m)::mapcar\
(iterates(f,f(m)),f)))in int list
||||SELECTED METHOD at depth 4: transition([hd,tl])
|||||iteratesa([2,1,1,1,1])
|||||v0:ih(m:int=>f:(int=>int)=>related(iterates(f,f(m)),mapcar(iterates(f,m),f\
))in int list)
|||||==>m:int=>f:(int=>int)=>related(iterates(\f/,f(``f({\m/})''<out>)),mapcar(\
iterates(\f/,``f({\m/})''<out>),\f/))in int list
|||||TERMINATING METHOD at depth 5: fertilize(strong(v0))
|||||iteratesa([1,1,1,1,1])
|||||v0:ih(m:int=>f:(int=>int)=>related(iterates(f,f(m)),mapcar(iterates(f,m),f\
))in int list)
|||||==>m:int=>f:(int=>int)=>related(f(\m/),f(\m/))in int
|||||TERMINATING METHOD at depth 5: strong(f(\m/))
Planning complete for iteratesa

-----------------------------------------------------------
iteratesa:

[m:int,f:int=>int]

|- iterates(f,f of m)=mapcar(iterates(f,m),f)in int list

coinduction_lts(union([range(int list-lam(m-int,lam(f-(int=>int),related(iterat\
es(f,f of m),mapcar(iterates(f,m),f)))))]),[]) then
  gfp_membership(obs_fun) then
    intro1([iterates1-[1,1]],[]) then
      intro1([iterates1-[1,1,2],mapcar2-[1,2]],[]) then
        transition([hd,tl]) then
          [strong(f(\m/)),
           fertilize(strong(v0))
          ]


yes
| ?-
```

C.2.2 hiterates

```
| ?- plan(hiterates).
loading thm(hiterates)...done
hiterates([])
==>f:(int=>int)=>x:int=>h(f,x)=iterates(f,x)in int list
SELECTED METHOD at depth 0: coinduction_lts(union([range(int list-lam(f-(int=>i\
nt),lam(x-int,related(h(f,x),iterates(f,x)))))]),[])
|hiterates([1])
|==>subset(union([range(int list-lam(f-(int=>int),lam(x-int,related(h(f,x),iter\
ates(f,x)))))]),obs_fun of union([range(int list-lam(f-(int=>int),lam(x-int,rel\
ated(h(f,x),iterates(f,x)))))]))in set(int list)
|SELECTED METHOD at depth 1: gfp_membership(obs_fun)
||hiterates([1,1])
||v0:ih(f:(int=>int)=>x:int=>related(h(f,x),iterates(f,x))in int list)
```

```
||==>f:(int=>int)=>x:int=>(act(h(f,x))and act(iterates(f,x)))in int list
||SELECTED METHOD at depth 2: intro1([h1-[1,1]],[])
|||hiterates([1,1,1])
|||v0:ih(f:(int=>int)=>x:int=>related(h(f,x),iterates(f,x))in int list)
|||==>f:(int=>int)=>x:int=>(act(x::mapcar(h(f,x),f))and act(iterates(f,x)))in i\
nt list
|||SELECTED METHOD at depth 3: intro1([iterates1-[1,2]],[])
||||hiterates([1,1,1,1])
||||v0:ih(f:(int=>int)=>x:int=>related(h(f,x),iterates(f,x))in int list)
||||==>f:(int=>int)=>x:int=>(act(x::mapcar(h(f,x),f))and act(x::iterates(f,f(x)\
)))in int list
||||SELECTED METHOD at depth 4: transition([hd,tl])
|||||hiterates([2,1,1,1,1])
|||||v0:ih(f:(int=>int)=>x:int=>related(h(f,x),iterates(f,x))in int list)
|||||==>f:(int=>int)=>x:int=>related(``mapcar({h(\f/,\x/)},\f/)''<out>,iterates\
(\f/,``f({\x/})''<out>))in int list
|||||SELECTED METHOD at depth 5: wave(sinkexp,[f:(int=>int)=>x:int=>related(``m\
apcar({h(\f/,\x/)},\f/)''<out>,iterates(\f/,\``f({x})''<out>/))in int list,b])
||||||hiterates([1,2,1,1,1,1])
||||||v0:ih(f:(int=>int)=>x:int=>related(h(f,x),iterates(f,x))in int list)
||||||==>f:(int=>int)=>x:int=>related(``mapcar({h(\f/,\x/)},\f/)''<out>,iterate\
s(\f/,\``f({x})''<out>/))in int list

>>>>> INVOKING revise-bisimulation CRITIC <<<<<

SELECTED METHOD at depth 0: coinduction_lts(union([range(int list-lam(v0-pnat,l\
am(f-(int=>int),lam(x-int,related(appn(v0,mapcar of f,h(f,x)),iterates(f,appn(v\
0,f,x)))))))]),[])
|hiterates([1])
|==>subset(union([range(int list-lam(v0-pnat,lam(f-(int=>int),lam(x-int,related\
(appn(v0,mapcar of f,h(f,x)),iterates(f,appn(v0,f,x)))))))]),obs_fun of union([\
range(int list-lam(v0-pnat,lam(f-(int=>int),lam(x-int,related(appn(v0,mapcar of\
 f,h(f,x)),iterates(f,appn(v0,f,x)))))))]))in set(int list)
|SELECTED METHOD at depth 1: gfp_membership(obs_fun)
||hiterates([1,1])
||v1:ih(v0:pnat=>f:(int=>int)=>x:int=>related(appn(v0,mapcar of f,h(f,x)),itera\
tes(f,appn(v0,f,x)))in int list)
||==>v0:pnat=>f:(int=>int)=>x:int=>(act(appn(v0,mapcar of f,h(f,x)))and act(ite\
rates(f,appn(v0,f,x))))in int list
||SELECTED METHOD at depth 2: intro1([h1-[3,1,1],mapapn2-[1,1]],[])
|||hiterates([1,1,1])
|||v1:ih(v0:pnat=>f:(int=>int)=>x:int=>related(appn(v0,mapcar(f),h(f,x)),iterat\
es(f,appn(v0,f,x)))in int list)
|||==>v0:pnat=>f:(int=>int)=>x:int=>(act(appn(v0,f,x)::appn(v0,mapcar(f),mapcar\
(h(f,x),f)))and act(iterates(f,appn(v0,f,x))))in int list
|||SELECTED METHOD at depth 3: intro1([iterates1-[1,2]],[])
||||hiterates([1,1,1,1])
||||v1:ih(v0:pnat=>f:(int=>int)=>x:int=>related(appn(v0,mapcar(f),h(f,x)),itera\
tes(f,appn(v0,f,x)))in int list)
||||==>v0:pnat=>f:(int=>int)=>x:int=>(act(appn(v0,f,x)::appn(v0,mapcar(f),mapca\
r(h(f,x),f)))and act(appn(v0,f,x)::iterates(f,f(appn(v0,f,x)))))in int list
||||SELECTED METHOD at depth 4: transition([hd,tl])
|||||hiterates([2,1,1,1,1])
|||||v1:ih(v0:pnat=>f:(int=>int)=>x:int=>related(appn(v0,mapcar(f),h(f,x)),iter\
ates(f,appn(v0,f,x)))in int list)
|||||==>v0:pnat=>f:(int=>int)=>x:int=>related(appn(\v0/,mapcar(\f/),``mapcar({h\
(\f/,\x/)},\f/)''<out>),iterates(\f/,``f({appn(\v0/,\f/,\x/)})''<out>))in int l\
ist
|||||SELECTED METHOD at depth 5: wave([1,1,2,2,2,2,2,2],[hitlem1,equ(left)])
||||||hiterates([1,2,1,1,1,1])

||||||v1:ih(v0:pnat=>f:(int=>int)=>x:int=>related(appn(v0,mapcar(f),h(f,x)),ite\
rates(f,appn(v0,f,x)))in int list)
||||||==>v0:pnat=>f:(int=>int)=>x:int=>related(``mapcar(\f/,{appn(\v0/,mapcar(\\
f/),h(\f/,\x/))})''<out>,iterates(\f/,``f({appn(\v0/,\f/,\x/)})''<out>))in int \
list
||||||TERMINATING METHOD at depth 6: fertilize(strong(v1))
```

```
|||||hiterates([1,1,1,1,1])
|||||v1:ih(v0:pnat=>f:(int=>int)=>x:int=>related(appn(v0,mapcar(f),h(f,x)),iter\
ates(f,appn(v0,f,x)))in int list)
|||||==>v0:pnat=>f:(int=>int)=>x:int=>related(appn(\v0/,\f/,\x/),appn(\v0/,\f/,\
\x/))in int
|||||TERMINATING METHOD at depth 5: strong(appn(\v0/,\f/,\x/))
Planning complete for hiterates

--------------------------------------------------------------
hiterates:

[f:int=>int,x:int]

|- h(f,x)=iterates(f,x)in int list

coinduction_lts(union([range(int list-lam(v0-pnat,lam(f-(int=>int),lam(x-int,re\
lated(appn(v0,mapcar of f,h(f,x)),iterates(f,appn(v0,f,x))))))]),[]) then
  gfp_membership(obs_fun) then
    intro1([h1-[3,1,1],mapapn2-[1,1]],[]) then
      intro1([iterates1-[1,2]],[]) then
        transition([hd,tl]) then
          [strong(appn(\v0/,\f/,\x/)),
           wave([1,1,2,2,2,2,2,2],[hitlem1,equ(left)]) then
             fertilize(strong(v1))
          ]


yes
| ?-
```

Appendix D

Various Theorems with Proofs

D.1 Derived Inference Rule for $\sim$

Theorem D.1

$$\frac{\begin{array}{l}\forall 1 \leq i \leq n.\ \bigwedge_{i=1}^{n}\langle a_i,\ b_i\rangle \in \mathcal{R} \Rightarrow \\ \qquad \forall \alpha.((a_i \xrightarrow{\alpha} a_i' \vee b_i \xrightarrow{\alpha} b_i') \Rightarrow \\ \qquad ((a_i \xrightarrow{\alpha} a_i' \wedge b_i \xrightarrow{\alpha} b_i') \wedge \\ \qquad \langle a_i',\ b_i'\rangle \in \mathcal{R} \cup \sim))\end{array}}{\bigcup_{i=1}^{n}\langle a_i,\ b_i\rangle \subseteq \langle \bigcup_{i=1}^{n}\langle a_i,\ b_i\rangle \cup \sim\rangle} \tag{D.1}$$

Proof. The preconditions imply that for all $\langle a,\ b\rangle \in \bigcup_{i=1}^{n}\langle a_i,\ b_i\rangle$ and for all α if $a \xrightarrow{\alpha} a'$ there is a b' with $b \xrightarrow{\alpha} b'$ and $\langle a',\ b'\rangle \in \bigcup_{i=1}^{n}\langle a_i,\ b_i\rangle \cup \sim$. This implies that $\bigcup_{i=1}^{n}\langle a_i,\ b_i\rangle \subseteq [\bigcup_{i=1}^{n}\langle a_i,\ b_i\rangle \cup \sim]$ (by the definition of $[\cdots]$ from chapter 3).

Similarly for all $\langle a,\ b\rangle \in \bigcup_{i=1}^{n}\langle a_i,\ b_i\rangle$ and for all α if $b \xrightarrow{\alpha} b'$ there is a a' with $a \xrightarrow{\alpha} a'$ and $\langle a',\ b'\rangle \in \bigcup_{i=1}^{n}\langle a_i,\ b_i\rangle \cup \sim$,so $\langle b',\ a'\rangle \in (\bigcup_{i=1}^{n}\langle a_i,\ b_i\rangle \cup \sim)^{op}$. This implies that $\langle b,\ a\rangle \in [(\bigcup_{i=1}^{n}\langle a_i,\ b_i\rangle \cup \sim)^{op}]$ so $\bigcup_{i=1}^{n}\langle a_i,\ b_i\rangle \subseteq [(\bigcup_{i=1}^{n}\langle a_i,\ b_i\rangle \cup \sim)^{op}]^{op}$.

Hence $\mathcal{R} \subseteq [\mathcal{R} \cup \sim] \cup [(\mathcal{R} \cup \sim)^{op}]^{op}$, i.e. $\mathcal{R} \subseteq \langle \mathcal{R} \cup \sim\rangle$ (by definition of $\langle\cdots\rangle$).$\square$

D.2 Derived Inference Rule for $LlistD\_fun$

Theorem D.2

$$\frac{\begin{array}{c}\forall 1 \leq i \leq n. \\ \bigwedge_{i=1}^{n}\langle a_i,\ b_i\rangle \in \mathcal{R} \Rightarrow hd(a_i) = hd(b_i)\end{array} \qquad \begin{array}{c}\forall 1 \leq i \leq n. \\ \bigwedge_{i=1}^{n}\langle a_i,\ b_i\rangle \in \mathcal{R} \Rightarrow \langle tl(a_i),\ tl(b_i)\rangle \in \mathcal{R}\end{array}}{\bigcup_{i=1}^{n}\langle a_i,\ b_i\rangle \subseteq LlistD\_fun(\bigcup_{i=1}^{n}\langle a_i,\ b_i\rangle)} \tag{D.2}$$

Proof. The preconditions imply that for all $\langle a,\ b\rangle \in \bigcup_{i=1}^{n}\langle a_i,\ b_i\rangle$ $hd(a) = hd(b)$ so $\langle a,\ b\rangle = \langle hd(a) :: tl(a),\ hd(a) :: tl(b)\rangle$ and $\langle tl(a),\ tl(b)\rangle \in \bigcup_{i=1}^{n}\langle a_i,\ b_i\rangle$ so $\langle a,\ b\rangle \in LlistD\_fun(\bigcup_{i=1}^{n}\langle a_i,\ b_i\rangle)$ (by definition of $LlistD\_fun$) so $\bigcup_{i=1}^{n}\langle a_i,\ b_i\rangle \subseteq LlistD\_fun(\bigcup_{i=1}^{n}\langle a_i,\ b_i\rangle)$.

D.3 Derived Inference Rule for $list\_fun$

Theorem D.3

$$\frac{\begin{array}{cc} \forall 1 \leq i \leq n. & \forall 1 \leq i \leq n. \\ \bigwedge_{i=1}^{n} a_i \in \mathcal{S} \Rightarrow hd(a_i) \in U & \bigwedge_{i=1}^{n} a_i \in \mathcal{S} \Rightarrow tl(a_i) \in \mathcal{S} \end{array}}{\bigcup_{i=1}^{n}\{a_i\} \subseteq list\_fun(\bigcup_{i=1}^{n}\{a_i\}, U)} \tag{D.3}$$

Proof. The preconditions imply that for all $a \in \bigcup_{i=1}^{n}\{a_i\}$ $hd(a) \in U$ and $tl(a) \in \mathcal{S}$ so $hd(a) :: tl(a) \in list\_fun(\bigcup_{i=1}^{n}\{a_i\}, U)$ (by definition of $list\_fun$) so $\mathcal{S} \subseteq list\_fun(\mathcal{S}, U)$.□

D.4 Derived Rule for Induction

Theorem D.4

$$l \stackrel{\alpha}{\rightarrow} \phi \Rightarrow \frac{\bigwedge_{i=1}^{n}((P_i(\phi) \rightarrow P_i(l)))}{\lceil \bigcap_{i=1}^{n}\{x \mid P_i(x)\} \rceil \subseteq \bigcap_{i=1}^{n}\{x \mid P_i(x)\}} \tag{D.4}$$

Proof. Let $a \in \lceil \bigcap_{i=1}^{n}\{x \mid P_i(x)\} \rceil$ then there is some α and a' such that $a \stackrel{\alpha}{\rightarrow} a'$ and $a' \in \bigcap_{i=1}^{n}\{x \mid P_i(x)\}$ or $a \equiv \bot$. So if $a \stackrel{\alpha}{\rightarrow} a'$, then $P_i(a')$ is true (for all i). By the above premises this implies that $P_i(a)$ is true which implies that $a \in \bigcap_{i=1}^{n}\{x \mid P_i(x)\}$ so $\lceil \bigcap_{i=1}^{n}\{x \mid P_i(x)\} \rceil \in \bigcap_{i=1}^{n}\{x \mid P_i(x)\}$.□

Appendix E

Verifying the Proof Plans in a Tactic–based Theorem Prover

E.1 Introduction

The object of this appendix is to overview the work done towards linking *CoCLAM* with the object–level theorem prover Isabelle, in order to make use of the support for coinduction present in Isabelle. This work is very preliminary.

E.2 Isabelle

Isabelle is a *Generic Theorem Prover* [Paulson 94a]. The intention is that proofs in Isabelle are not tied to any one logic or formal system, but that the user may define their own logics as they may be appropriate, via the use of *theory files*. Isabelle comes with several different logics and packages containing definitions, object–level inference rules and lemmas.

The different *object–logics* are defined in the Isabelle *meta–logic* and object–level proofs are built up using meta–level rules. Proof construction in Isabelle is based on resolution and uses proof tactics.

Each use of a meta–level axiom corresponds to the application of an object–level rule. The meta–logic is defined by a collection of inference rules, including equational rules for the λ–calculus and logical rules. Proofs performed using the primitive meta–rules would be lengthy so Isabelle often uses derived rules e.g. resolution.

E.2.1 Tactics and Tacticals

The process of proof in Isabelle can be forwards or backwards as in HOL. Coinductive proofs as planned by *CoCLAM* are backwards. Each resolution step is a case of resolving one of the object level inference rules with the current proof state, to produce a new proof state analogous to application of that rule at the object level.

Tactics are built up out of sets of rules. In Isabelle they act directly on the proof state, or the set of subgoals rather than upon theorems. An Isabelle tactic is a function that takes a proof state and returns a sequence of possible successor states. Isabelle represents proof states as theorems. Some of these tactics are extremely powerful and allow theorems to be proved with minimal interaction.

Most proofs make heavy use of resolution and coinductive proofs are no exception. As a result one of the central tactics is `resolve_tac`:

> "`resolve_tac` *thms* i is the basic resolution tactic, used for most proof steps. The *thms* represent object–rules, which are resolved against subgoal i of the proof state. For each rule, resolution forms next states by unifying the conclusion with the subgoal and inserting instantiated premises in its place. A rule can admit many higher–order unifiers. The tactic fails if none of the rules generates next states." [Paulson 94a]

An example of a particularly powerful tactic is the simplification tactic which performs numerous processes such as reduction etc. The tactic `simp_tac` ss i simplifies subgoal i using the rules in ss. ss is a *simpset.* A simpset consists of sets of rewrite rules, congruence rules, a subgoaler, solver and looper. These come as defaults with Isabelle's supplied logics, although further equivalences can be added using the command `addsimps`. `simp_tac` is very powerful and can solve a large number of complex goals.

Isabelle doesn't supply the user with any trace of the proof process, beyond the tactics supplied by the user and this can sometimes be a disadvantage. There are a number of tactics that break down `simp_tac` into smaller steps so the process can be viewed in more detail.

An overview of Isabelle's coinduction package was presented in chapter 8.

E.3 Linking *CoCLAM* to Isabelle

A prototype translator was developed which took sequences of *CoCLAM* proof method calls and translated them into sequences of Isabelle tactic calls. Considerable further work needs to be done for this link to work without user intervention on a wide range of theorems.

Ideally a full link up between the two systems would allow a user to specify a problem in Isabelle which would then call *CoCLAM* to make a proof plan which could then be re–applied to the Isabelle problem. A more modest approach was adopted of taking a *CoCLAM* proof plan for a goal specified in *CoCLAM* and translating both the goal and the plan into Isabelle syntax (i.e. the translation only goes one way from *CoCLAM* to Isabelle). This translation mechanism required three basic components.

1. A translation of object level *CoCLAM* terms to object level Isabelle terms.
2. A translation of *CoCLAM* rule/theorem names to Isabelle rule/theorem names
3. Provision of Isabelle tactics corresponding to the *CoCLAM* methods (in the basic implementation described here these tactics were composed of sequences of tactics already present in Isabelle).

E.3.1 Translating Object Level Terms and Rules

Most of the basic translation was fairly simple. A database of equivalent function names were supplied (e.g. *CoCLAM*'s *map* function has the same introduction rules as Isabelle's `lmap`), variable names were preserved and the term structure changed from an uncurried to a curried format. An extension of this, of course, would be for *CoCLAM*'s functions to be redefined in Isabelle automatically rather than relying upon a pre–specified set of equivalences.

The only translation that was at all complicated was between the representations of bisimulations. *CoCLAM* uses a fairly complicated representation partly because of the difficulties of dealing with higher order information.

Free Variables

Variable naming conventions are different in *CoCLAM* and Isabelle. More significantly the two systems choose to rename and standardise apart variables at different points in the proof process. This remains the biggest hurdle in providing a fully automated link. It is hard to provide appropriate substitution instances for higher–order unification to those tactics that require them.

E.3.2 Implementation: Providing Tactics for the Methods

The Isabelle tactics developed for *CoCLAM* methods are summarised by the following table.

Method	Tactic

Coinduction($\mathcal{R}$)	`by (res_inst_tac [("r", "` *tran*($\mathcal{R}$) `")]`
	`llist_equalityI i);`
	`by (REPEAT (resolve_tac [UN1_I, rangeI, UnI1] i));`

Gfp Membership	`by (safe_tac set_cs);`
Transition([*nil*])	No tactic
Transition([*hd*, *tl*])	`by (rtac llistD_Fun_LCons_Case_I i);`
Fertilize	`by (REPEAT` `(resolve_tac [UN1_I RS UnI1, UN1_I, rangeI] i));`
Reflexivity	`by ((rtac llistD_Fun_range_I i)` `ORELSE (Simp_tac i);`
Eval_def(*Rule*)	`by (rtac` *Rule* `i);`

Method	Tactic
Wave(*Rule*)	`by (res_inst_tac` *Insts Rule* `i );` with User Intervention providing the set of variable instantiations, *Insts*

Eval(*Rules*, *Cases*)	`by (res_inst_tac` *Insts Rule* `i);` with User Intervention suppling the set of variable instantiations, *Insts*,
	and *Rule* $\in$ *Rules*
	If case splits are involved add
	`by (res_inst_tac` *Insts* `llistE i);` with User Intervention suppling the set of variable instantiations, *Insts*
	`by (etac ssubst i);`
	`by (etac ssubst (i+1));`

Note that most of the rules contain information on which subgoal they are to be applied, (`i`, `(i+1)`, etc.). $tran(\mathcal{R})$ represents the translation of $\mathcal{R}$ in *CL^{A}M* syntax into Isabelle syntax. The need for a user to intervene in the Wave and Eval tactics and supply *Insts* is because *CL^{A}M* identifies redexes by position in the term while the built–in Isabelle tactics used identify them by pattern matching. This is not a requirement of the translation just an implementation detail of the one undertaken.

The rest of this section discusses some of the methods with their tactics and further work needed for their implementation.

The Coinduction Method

Recall that the Coinduction Method applies the rule

$$\frac{\langle l_1,\ l_2\rangle \in \mathcal{R} \quad \mathcal{R} \subseteq \langle \mathcal{R} \cup \sim\rangle}{l_1 \sim l_2}$$

supplying an instantiation for $\mathcal{R}$ and assuming that the first premise is automatically discharged.

The coinduction rule for bisimilarity in Isabelle/HOL is called `llist_equalityI` and is the rule:

$$\frac{\langle l_1,\ l_2\rangle : \mathcal{R} \quad \mathcal{R} \subseteq llistD\_fun(\mathcal{R} \cup rng(\lambda x.\langle x,\ x\rangle))}{l_1 = l_2} \tag{E.1}$$

These two rules can be shown to be equivalent for lazy lists.

An instantiation for $\mathcal{R}$ is provided by the proof method (once translated into an Isabelle Object Term). So the tactic `res_inst_tac` is used which applies resolution with substitution information supplied by the user.

As stated in chapter 5, the coinduction method also assumes that the first premise of the coinduction rule will be discharged. This involves showing that some pair, $\langle l_1, l_2\rangle$, is an element of some set $\{\langle\phi_1(\bar{x}), \psi_1(\bar{x})\rangle\}\cup\cdots\cup\{\langle\phi_n(\bar{x}), \psi_n(\bar{x})\rangle\}$. This can be done by resolution with standard set theoretic axioms already implemented in Isabelle/HOL.

The Gfp Membership Method

The translation from $\mathcal{R} \subseteq LlistD\_fun(\mathcal{R})$ can be handled automatically by the `safe_tac` tactic, using the set of definitions and derived rules in Isabelle's `set_cs`. `safe_tac` is a deterministic tactic, with at most one outcome. It applies "safe steps" to the goal. Safe steps are a loosely defined class of rules that may be performed blindly. They include proof by assumption and Modus Ponens. No safe step instantiates unknowns. `set_cs` is a collection of rules for the propositional and higher order logics extended with rules for bounded quantifiers, subsets, comprehensions, unions and intersections, complements, finite sets, images and ranges.

Rewriting

There are several rewriting methods implemented in *CoCIAM*. Coinduction uses three: Evaluate, Wave and Eval_def. Although Isabelle contains a rewriting tactic, `rewrite_tac`, this attempts to unfold *all* occurrences of a function and so cannot cope with definitions that can be infinitely unfolded (e.g. *lconst* and *iterates*). To apply just one rewrite (in the way the Wave and Eval_def methods do) it is necessary to use resolution. Sometimes there is a choice, however, of where a rewrite rule may apply. For instance, in the goal

$$\begin{gathered}\langle iterates(F, F(M)),\ map(F, iterates(F, M))\rangle : \\ llistD\_fun(\{\langle iterates(F', F'(M')),\ map(F', iterates(F', M'))\rangle\} \cup rng(\lambda x.\langle x,\ x\rangle))\end{gathered} \tag{E.2}$$

the definition of *iterates*:

$$iterates(F, M) \rightsquigarrow M :: iterates(F, F(M)) \tag{E.3}$$

can apply to either $iterates(F', F'(M'))$ or $iterates(F', M')$. Moreover once applied to one of these expressions it can be applied to a sub–expression of the rewritten expression. Hence resolution, applied blindly, can be non–terminating. This means the `res_inst_tac` tactic needs to be used to guide the application of (E.3) by nominating variable instantiations, which, in turn, means that the translator has to supply substitution information.

This raised several problems. The various rewriting methods implemented in *CoCIAM* specify the position of a sub–term to be rewritten rather than providing any substitution information. Moreover, even if they were modified to provide such information there would remain problems in translating *CoCIAM* object–level term names to Isabelle object–level term names because of the inconsistent renaming of variables.

The desirable solution would be to implement a new tactic in Isabelle that performed rewriting based on position information rather than substitution information. This wasn't achieved and would be the next task to be undertaken in providing a link.

Case Splits

The Eval method may also require case-splitting to be performed. This was done using resolution with the rule, `llistE`:

$$\frac{l = nil \Rightarrow P \quad l = x :: l' \Rightarrow P}{P} \tag{E.4}$$

As with the rewriting tactic additional substitution information frequently has to be supplied by the user.

`by (etac ssubst 1)` was used to replace $l = nil \Rightarrow \phi$ and $l = x :: l' \Rightarrow \phi$ with $\phi[l/nil]$ and $\phi[l/x :: l']$. `ssubst` is a substitution rule. `etac` performs *Elim–resolution*. Elim–resolution seeks to eliminate a premise (by assumption). This can be used to guide the choice of instantiations.

Transitions

Since only lazy lists were being considered only three transitions were possible $\stackrel{\texttt{nil}}{\rightarrow}$, $\stackrel{\texttt{hd}}{\rightarrow}$ and $\stackrel{\texttt{tl}}{\rightarrow}$. Of these $\stackrel{\texttt{hd}}{\rightarrow}$ and $\stackrel{\texttt{tl}}{\rightarrow}$ always occur together. Since the actual formalisation being used by Isabelle isn't that of labelled transitions systems, no explicit analysis transitions was involved. However Isabelle had to be provided with a new rule `llistD_Fun_LCons_Case_I` which asserts that

$$\frac{h_1 = h_2 \quad \langle t_1,\, t_2\rangle : \mathcal{R}}{\langle h_1 :: t_1,\, h_2 :: t_2\rangle : llistD\_Fun(\mathcal{R})} \tag{E.5}$$

This proved simple to derive in Isabelle.

E.4 Conclusion

This is the extent of the current work on linking *CoCLAM* to an object–level theorem prover. Obviously a great deal more work needs to be done before this is a sufficiently robust system to test the proof plans *CoCLAM* has generated.

Appendix F

Proof Comparisons

F.1 Introduction

This appendix lists the comparison patterns that were discussed in chapter 10. It does this by the use of tables (like abstracted versions of those shown in that chapter) which attempt to list the sequence proof methods without giving any of the details. The intention is simply to present the raw data from which the conclusions in chapter 10 were drawn.

The theorems were drawn from those that *CoCL^AM* successfully planned, however the plans shown here are not necessarily those produced by *CL^AM* and *CoCL^AM*. This is for a number of reasons. *CL^AM* cannot handle several of the theorems proved using *nth* since it can't cope with "list generators". In several cases additional lemmata (such as $0 + X \leadsto X$) have been used in the proofs to avoid the need to discuss critics etc. when the object of the exercise was to look at the shape of the proofs in a general way, not in a *CL^AM* specific way. This has made some plans simpler than those produced by *CoCL^AM*. In all cases, if a lemma was available to one proof (e.g. the inductive proof) then it was also available for the coinductive proof.

F.2 Comparisons Using Type Changes

F.2.1 Pattern 1

In the first pattern the inductive base case is associated with the coinductive "base case" produced by the casesplit. The pattern has been abstracted over all three lazy types, this caused difficulties when trying to represent parts of the pattern that involved cases and transitions. The cases caused by heads of lists are placed in brackets and do not appear in proof plans involving natural numbers.

<table>
<tr><td colspan="2">Induction</td><td colspan="2">Coinduction</td></tr>
<tr><td colspan="2"></td><td colspan="2">Casesplit</td></tr>
<tr><td colspan="2">Base Case</td><td colspan="2">Nil/0 Case</td></tr>
<tr><td colspan="4">Rewrite</td></tr>
<tr><td colspan="2"></td><td>(Head Transition)</td><td>Destructor Transition</td></tr>
<tr><td colspan="4">Rewrite</td></tr>
<tr><td colspan="4">Reflexivity</td></tr>
<tr><td colspan="2">Step Case</td><td colspan="2">Constructor Case</td></tr>
<tr><td colspan="4">Rewrite</td></tr>
<tr><td colspan="2">Cancel Constructor</td><td>(Head Transition)</td><td>Destructor Transition</td></tr>
<tr><td colspan="4">Rewrite</td></tr>
<tr><td>(Reflexivity)</td><td>Fertilize</td><td>(Reflexivity)</td><td>Fertilize</td></tr>
</table>

The theorems that displayed this pattern are listed below. They are referred to by the names used in appendix B.

app1right	appatend
assconsapp	flattenexplode
halfdouble	lappnill
lappnilr	lendouble
lenmapcar	lplus0l
lplus0r	mapflip
mapfold	mapthm
parityT0	plus1right
pluslem2	plusxx
times2right	

F.2.2 Pattern 2

The second pattern is very similar to the first except that here in the "base case" transitions are not taken but arguments about the determinacy of the transition system. This is used when the base case has reduced to a case which isn't a value. e.g. $map(f, nil <> l) = map(f, nil) <> map(f, l)$ reduces to $map(f, l) = map(f, l)$ this is proved in induction by the reflexivity of equals, but in coinduction the goal is $map(f, l) \xrightarrow{\alpha} \phi \wedge map(f, l) \xrightarrow{\alpha} \psi \wedge \langle \phi, \psi \rangle \in \mathcal{R}$. $CoCL^{A}M$ would perform further casesplits on l and use reduction to determine transitions, however it is also possible to argue, from the determinacy of the transition system that all the results of transitions from $map(f, l)$ will be related by $\sim$. It is this course that has been taken here.

Induction		Coinduction	
		Casesplit	
Base Case		Nil/0 Case	
Rewrite			
Reflexivity		Determinacy Arguments	
Step Case		Constructor Case	
Rewrite			
Cancel Constructor		(Head Transition)	Destructor Transition
Rewrite			
(Reflexivity)	Fertilize	(Reflexivity)	Fertilize

The theorems exhibiting this proof pattern were:

assapp	assp
comapp	comp
comp2	halfplus2
lenplus	mapcarapp
mapapp	plus2right

F.2.3 Pattern 3

In pattern 3, coinduction is using two pair schema (indicated by the two columns under coinduction in the pattern), however the second pair schema is identical to the first except that there is an extra outer constructor (e.g. $\{\langle double(N),\ N+N\rangle\} \cup \{\langle s(double(N)),\ s(N+N)\rangle\}$). No rewriting is required on this second pair scheme to determine the transitions and once determined it fertilizes immediately. In the inductive case the constructor is cancelled twice.

Induction		Coinduction			
		Casesplit			
Base Case		Nil/0 Case			
Rewrite					
		Nil/0 transition			
Reflexivity					
Step Case		Constructor Case			
Rewrite					
Cancel Constr. Twice		(Hd Trans.)	Dest. Trans.	(Hd Trans.)	Dest. Trans.
Rewrite					
(Reflexivity)	Fertilize	(Reflexivity)	Fertilize	(Reflexivity)	Fertilize

The theorems exhibiting this proof pattern were:

doublehalf
doubleplus
doubletimes1
doubletimes2
mergeolel

F.2.4 Pattern 4

Pattern 4 links two other patterns together by extra pair schema (in Coinduction) and a secondary induction in the step case (in induction). The two patterns are referred to as Pattern A and Pattern B below, however the last few steps (involving the step/constructor cases) of pattern A are shown to illustrate where a new induction occurs in place of a fertilize step.

<table>
<tr><td colspan="2">Induction</td><td colspan="3">Coinduction</td></tr>
<tr><td colspan="2"></td><td colspan="2">Casesplit</td><td></td></tr>
<tr><td colspan="4">Pattern A</td><td></td></tr>
<tr><td colspan="2">Step Case</td><td colspan="2">Constructor Case</td><td></td></tr>
<tr><td colspan="4">Rewrite</td><td></td></tr>
<tr><td colspan="2">Cancel Constructor</td><td>(Head Transition)</td><td>Destructor Transition</td><td></td></tr>
<tr><td colspan="4">Rewrite</td><td></td></tr>
<tr><td>(Reflexivity)</td><td>Induction</td><td>(Reflexivity)</td><td>Fertilize</td><td></td></tr>
<tr><td></td><td>Pattern B</td><td></td><td></td><td>Pattern B</td></tr>
</table>

The theorems exhibiting this proof pattern were:

gordon1
map2thm
mapaddl
mapdouble
mapid

F.2.5 Pattern 5

Pattern 5 links two other patterns together by a second casesplit (in Coinduction) and a secondary induction in the base case (in induction). In this case only the step/constructor case part of one of the patterns is used.

<table>
<tr><td>Induction</td><td>Coinduction</td></tr>
<tr><td></td><td>Casesplit</td></tr>
<tr><td>Base Case</td><td>Nil/0 Case</td></tr>
<tr><td>Induction 2</td><td>Casesplit 2</td></tr>
<tr><td colspan="2">Pattern A</td></tr>
<tr><td>Step Case</td><td>Constructor Case</td></tr>
<tr><td colspan="2">Step/Constructor Case Pattern B</td></tr>
</table>

The theorems exhibiting this proof pattern were:

applapp
asslapp
asslplus
comlplus
pluslplus

F.2.6 Cancellation of +

times2right was the only theorem that didn't fit into any of these patterns. Here induction used cancellation with + while coinduction was forced to perform generalisations to enable the proof to go through.

F.3 Comparisons Using nth

F.3.1 Pattern 1

This case is very similar to Pattern 1 for type changes.

<table>
<tr><td colspan="2">Induction</td><td colspan="2">Coinduction</td></tr>
<tr><td>Base Case</td><td>Step Case</td><td colspan="2"></td></tr>
<tr><td colspan="4">Casesplit</td></tr>
<tr><td colspan="4">Nil Case</td></tr>
<tr><td colspan="4">Rewrite</td></tr>
<tr><td>$nth(0)$</td><td>$nth(s)$</td><td>Head Transition</td><td>Tail Transition</td></tr>
<tr><td colspan="4">Reflexivity</td></tr>
<tr><td colspan="4">Constructor Case</td></tr>
<tr><td colspan="4">Rewrite</td></tr>
<tr><td>$nth(0)$</td><td>$nth(s)$</td><td>Head Transition</td><td>Tail Transition</td></tr>
<tr><td colspan="4">Rewrite</td></tr>
<tr><td>Reflexivity</td><td>Fertilize</td><td>Reflexivity</td><td>Fertilize</td></tr>
</table>

The theorems exhibiting this proof pattern were:

app1right	appatend
assconsapp	asslapp
flattenexplode	lappnill
lappnilr	mapfold
mapflip	mapthm

F.3.2 Pattern 2

Pattern 2 is the case for those theorems involving list generators, so no casesplitting is involved in the proof.

<table>
<tr><td colspan="2">Induction</td><td colspan="2">Coinduction</td></tr>
<tr><td>Base Case</td><td>Step Case</td><td colspan="2"></td></tr>
<tr><td colspan="4">Rewrite</td></tr>
<tr><td>$nth(0)$</td><td>$nth(s)$</td><td>Head Transition</td><td>Tail Transition</td></tr>
<tr><td colspan="4">Rewrite</td></tr>
<tr><td>Reflexivity</td><td>Fertilize</td><td>Reflexivity</td><td>Fertilize</td></tr>
</table>

The theorems exhibiting this proof pattern were:

appiterates	dpbrtconst
dptconst	everylswap
lconsta	lconstaddl
lconstapp	mapfinfl1
mapiter	oneslconst

F.3.3 Pattern 3

Pattern 3 is similar to Pattern 2 for type changes.

<table>
<tr><td colspan="2">Induction</td><td colspan="2">Coinduction</td></tr>
<tr><td>Base Case</td><td>Step Case</td><td colspan="2"></td></tr>
<tr><td colspan="4">Casesplit</td></tr>
<tr><td colspan="4">Nil Case</td></tr>
<tr><td colspan="4">Rewrite</td></tr>
<tr><td>Reflexivity</td><td>Reflexivity</td><td colspan="2">Determinacy Arguments</td></tr>
<tr><td colspan="4">Constructor Case</td></tr>
<tr><td colspan="4">Rewrite</td></tr>
<tr><td>$nth(0)$</td><td>$nth(s)$</td><td>Head Transition</td><td>Tail Transition</td></tr>
<tr><td colspan="4">Rewrite</td></tr>
<tr><td>Reflexivity</td><td>Fertilize</td><td>Reflexivity</td><td>Fertilize</td></tr>
</table>

The theorems exhibiting this proof pattern were:

assapp
comaddl
gordon1
mapaddl
mapcarapp
maplconst

F.3.4 Pattern 4

Pattern 4 joins two patterns together much as Pattern 3 in type checking does.

<table>
<tr><td colspan="2">Induction</td><td colspan="3">Coinduction</td></tr>
<tr><td colspan="4">Pattern A</td><td></td></tr>
<tr><td>$nth(0)$</td><td>$nth(s)$</td><td>Head Transition</td><td>Tail Transition</td><td></td></tr>
<tr><td colspan="4">Rewrite</td><td></td></tr>
<tr><td>Reflexivity</td><td>Induction 2</td><td>Reflexivity</td><td>Fertilize</td><td></td></tr>
<tr><td></td><td>Pattern B</td><td colspan="2"></td><td>Pattern B</td></tr>
</table>

The theorems exhibiting this proof pattern were:

dpsearchlc	infl1lswap
infl1nat	lconstzigzag
lswaplconst	lswaplmerge
mapdouble	mapidnat
mapiter2	mergeolel

F.3.5 Pattern 5

Pattern 5 generalises the conjecture first for induction and uses a generalisation in the pair schema it then proceeds according to one of the other patterns

Generalise	
Induction	Coinduction Generalised
Pattern	

The theorems exhibiting this proof pattern were:

applapp
lconsteven
hiterates
infl1lconst
nat1
natmap