Making a Productive Use of Failure to Generate Witnesses for Coinduction from Divergent Proof Attempts *

L. A. Dennis ** A. Bundy I. Green

Division of Informatics, University of Edinburgh, 80 South Bridge, Edinburgh
E-mail: louised@dai.ed.ac.uk

Coinduction is a proof rule. It is the dual of induction. It allows reasoning about non–well–founded structures such as lazy lists or streams and is of particular use for reasoning about equivalences. A central difficulty in the automation of coinductive proof is the choice of a relation (called a bisimulation).

We present an automation of coinductive theorem proving. This automation is based on the idea of proof planning [7]. Proof planning constructs the higher level steps in a proof, using knowledge of the general structure of a family of proofs and exploiting this knowledge to control the proof search. Part of proof planning involves the use of failure information to modify the plan by the use of a proof critic [23] which exploits the information gained from the failed proof attempt.

Our approach to the problem was to develop a strategy that makes an initial simple guess at a bisimulation and then uses generalisation techniques, motivated by a critic, to refine this guess, so that a larger class of coinductive problems can be automatically verified.

The implementation of this strategy has focused on the use of coinduction to prove the equivalence of programs in a small lazy functional language which is similar to Haskell [22].

We have developed a proof plan for coinduction and a critic associated with this proof plan. These have been implemented in *CoCL^AM*, an extended version of *CL^AM* [9], with encouraging results. The planner has been successfully tested on a number of theorems.

1. Introduction

Proof planning is a technique for automating proof. It exploits similarities across a family of proofs to provide guidance heuristics. Among these heuristics are various techniques for formulating existential witnesses. The provision of a

* Research funded by EPSRC grant GR\L\11724 and EPSRC postgraduate studentship 94314689. Our thanks to the three anonymous referees who made many helpful comments about this paper.
** Corresponding Author.

witness relation is a fundamental step in the process of proof by coinduction. These techniques are based on middle–out reasoning (delaying the choice of witness for as long as possible by using meta–variables and higher order unification) and proof critics (exploiting information from failed proof attempts to modify witnesses).

Coinduction is the dual of induction and is used to deal naturally with infinite processes. It was first investigated seriously in the field of concurrency [25] where looping communication networks are commonplace. It is also used in so–called "lazy" functional languages where the evaluation procedure only evaluates functions when they are required and may not fully evaluate[1] them. In this way a potentially infinite process may be present in a program without forcing the entire program to be non–terminating. The semantics of lazy languages are generally expressed in an operational style. This work concentrates on the use of coinduction with the operational semantics of a lazy functional language. Coinduction has also been proposed for use with object–oriented languages [20], cryptographic protocols [1] and the calculus of mobile ambients [21].

Tools have been provided for coinduction in several theorem proving environments. One of these, the Edinburgh Concurrency Workbench [12], is fully automated. This deals with problems described in Process Algebras. In other areas, such as functional languages, automation has not been attempted. The choice of the bisimulation needed by a proof is equivalent to the choice of induction scheme in inductive proofs [15]. Like the choice of induction scheme, the choice of bisimulation is a hard step in coinductive proof.

This work presents an automation of coinductive proof based on proof planning as a system, *CoCLAM*. The general structure of coinductive proof is informally analysed. This allows techniques for the provision of bisimulations (existential witnesses) to be developed. *CoCLAM* has been tested on a number of theorems with encouraging results.

2. Theoretical Background on Coinduction

The approach to modelling infinite processes is based on the theory of fixedpoints. The study of fixedpoints grew up out of Tarksi's work [33].

Definition 1. (Fixedpoint) A fixedpoint of a function $\mathcal{F}$ is an element, D, of its domain such that

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

[1] This assertion is not completely 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.

Definition 2. (Monotone) A function, f, on sets is monotone iff

$$\forall A, B.\ A \subseteq B \Rightarrow f(A) \subseteq f(B)^2$$

Tarski showed that a monotone function over a complete lattice has a lattice of fixedpoints which has a greatest and a least element. The least fixedpoint is associated with the induction rule, the greatest fixedpoint with the coinduction rule.

2.1. Least Fixed Points and Induction

The least fixedpoint is defined as [33,26]

$$\mathit{lfp}(\mathcal{F}) = \bigcap\{\mathcal{A} \mid \mathcal{F}(\mathcal{A}) \subseteq \mathcal{A}\} \tag{1}$$

Induction acts as an inference rule for deducing the presence of a least fixedpoint. Inductive domains (e.g. the natural numbers) can be regarded as the least fixedpoints of monotone functions. A general form of the induction rule is

$$\frac{\mathcal{F}(\mathcal{A}) \subseteq \mathcal{A}}{\mathit{lfp}(\mathcal{F}) \subseteq \mathcal{A}} \tag{2}$$

2.2. Greatest Fixed Points and Coinduction

The greatest fixedpoint is defined as [33,26]

$$\mathit{gfp}(\mathcal{F}) = \bigcup\{\mathcal{A} \mid \mathcal{A} \subseteq \mathcal{F}(\mathcal{A})\} \tag{3}$$

Coinduction acts as an inference rule for greatest fixedpoints. Coinductive domains are the greatest fixedpoints of monotone functions. The general coinduction rule is

$$\frac{\mathcal{A} \subseteq \mathcal{F}(\mathcal{A})}{\mathcal{A} \subseteq \mathit{gfp}(\mathcal{F})} \tag{4}$$

A more usual form is [26]

$$\frac{a \in \mathcal{A} \quad \mathcal{A} \subseteq \mathcal{F}(\mathcal{A})}{a \in \mathit{gfp}(\mathcal{F})} \tag{5}$$

2.3. Coinduction Specific to Functional Languages

Operational semantics describe how programs in some language execute on some abstract machine. The operational semantics of lazy functional languages are based on Abramsky's work [2] and use his notion of applicative bisimulation to build up an equational theory.

[2] We use $\Rightarrow$ to indicate implication.

In Abramsky's lazy λ–calculus the "meaning" of a term is its weak head normal form (WHNF).

Definition 3. (Weak Head Normal Form)(Adapted from [27]) A λ-term M is a **weak head normal form**(WHNF) if M is of the form $\lambda x.M'$ or $xM_1 \cdots M_m, n, m \geq 0$.

WHNFs are often referred to as *values* and this convention is adopted here. The calculus also contains two relations, big step evaluation and small step reduction. Big step evaluation, $\Downarrow$, relates expressions to values. Small step reduction[3], $\overset{\text{red}}{\to}$, represents the individual steps taken in reducing an expression to a value. $\overset{\text{red}}{\to}$ is stated in the semantics of the language and, in the case of functional languages will include β–reduction, $((\lambda x.e)b \overset{\text{red}}{\to} e[b/x])$, and rules governing the reduction of subterms. Evaluation and reduction are related as follows (where $\overset{\text{red}^*}{\to}$ is the reflexive transitive closure of $\overset{\text{red}}{\to}$):

$$a \overset{\text{red}}{\to} \overset{\text{def}}{=} \exists b.(a \overset{\text{red}}{\to} b) \tag{6}$$

$$a \Downarrow b \overset{\text{def}}{=} a \overset{\text{red}^*}{\to} b \wedge \neg(b \overset{\text{red}}{\to}) \tag{7}$$

$$a \Downarrow \overset{\text{def}}{=} \exists v.(a \Downarrow v) \tag{8}$$

$$a \Uparrow \overset{\text{def}}{=} \neg a \Downarrow \tag{9}$$

The last component of operational semantics is a 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, labels are type destructors, and the right hand side of the relation is the result of applying that type destructor to the left hand side. Often the labels (or *transitions*) are considered to be things that can be "observed" about the execution of the program.

Observational equivalence is built up as an order using $[-]$, a function on relations. $[-]$ 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 $\mathcal{S}^{\text{op}}$, where $a\mathcal{S}^{\text{op}}b$ iff $b\mathcal{S}a$, to form a second function $\langle - \rangle$. Definitions follow:

$$\overset{\text{def}}{=} \{\langle a,\, b\rangle | \forall \alpha.a \overset{\alpha}{\to} a' \Rightarrow \exists b'.b \overset{\alpha}{\to} b' \wedge a'\mathcal{S}b'\} \tag{S}$$

$$\langle \mathcal{S} \rangle \overset{\text{def}}{=} [\mathcal{S}] \cap [\mathcal{S}^{\text{op}}]^{\text{op}} \tag{11}$$

[3] $\overset{\text{red}}{\to}$ indicates a reduction order. We use $\to$ to indicate general rewriting.

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 $a \sim b$. This means that any transition on one expression can be matched by a transition on the other and the resulting expressions are also bisimilar.

Theorem 4. [19] $\sim$ is an equivalence relation.

Definition 5. (Bisimulation) A bisimulation is a relation, $\mathcal{R}$, such that $\mathcal{R} \subseteq \langle \mathcal{R} \rangle$

This means that $\mathcal{R}$ is a bisimulation if

$$\begin{aligned}\mathcal{R} \subseteq \{\langle a,\, b\rangle \mid\, & \forall \alpha.(\exists a'.a \stackrel{\alpha}{\rightarrow} a') \Rightarrow (\exists b'.b \stackrel{\alpha}{\rightarrow} b' \wedge (a'\mathcal{R}b'))\wedge \\ & \forall \alpha.(\exists b'.b \stackrel{\alpha}{\rightarrow} b') \Rightarrow (\exists a'.a \stackrel{\alpha}{\rightarrow} a' \wedge (a'\mathcal{R}b'))\}\end{aligned}$$

Theorem 6. [19] $\mathcal{R} \subseteq \sim$ if $\mathcal{R} \subseteq \langle \mathcal{R} \cup \sim \rangle$.

This means we can also consider sets, $\mathcal{R}$, satisfying the following formula

$$\begin{aligned}\mathcal{R} \subseteq \{\langle a,\, b\rangle \mid\, & \forall \alpha.(\exists a'.a \stackrel{\alpha}{\rightarrow} a') \Rightarrow (\exists b'.b \stackrel{\alpha}{\rightarrow} b' \wedge (a'\mathcal{R}b' \vee a' \sim b'))\wedge \\ & \forall \alpha.(\exists b'.b \stackrel{\alpha}{\rightarrow} b') \Rightarrow (\exists a'.a \stackrel{\alpha}{\rightarrow} a' \wedge (a'\mathcal{R}b' \vee a' \sim b'))\}\end{aligned}$$

Theorem 6 is useful since it allows known properties of $\sim$, most notably its reflexivity, to be exploited during the course of a coinductive proof. This is a *bisimulation up-to* $\sim$, throughout the rest of this paper these will be called bisimulations for ease of reference but the reader should bear in mind this is an abuse of notation. The coinduction rule for $\sim$ (based on (5) and taking theorem 6 into account) is:

$$\frac{\langle a,\, b\rangle \in \mathcal{R} \quad \mathcal{R} \subseteq \langle \mathcal{R} \cup \sim \rangle}{a \sim b} \tag{12}$$

This is the standard coinduction rule for a lazy functional language[4].

3. An Example of Coinduction

We are not going to provide the full semantics for a lazy functional language here (full details can be found in [15]). The important facts are that the following proof takes place in a language with a lazy list type in which *nil* and $H :: T$ are values. We assume that the language permits terms to be reduced if a subterm reduces. We also assume that the following rules express the only transitions that act on expressions of list type.

[4] This rule holds for any monotone function on a complete lattice, not just $\langle - \rangle$.

$$\frac{}{nil \stackrel{\texttt{nil}}{\rightarrow} \bot}$$

$$\frac{}{a :: b \stackrel{\texttt{hd}}{\rightarrow} a} \qquad \frac{}{a :: b \stackrel{\texttt{tl}}{\rightarrow} b}$$

$$\frac{a \Downarrow b \quad b \stackrel{\alpha}{\rightarrow} c}{a \stackrel{\alpha}{\rightarrow} c}$$

The following is an example of a coinductive proof in such a language:

Example 7. Consider the theorem:

$$\forall f, x.\, h(f, x) \sim iter(f, x) \tag{13}$$

together with the following reduction rules derived from the function definitions[5] ($\stackrel{\mathrm{red}}{\rightarrow}^{+}$ indicates at least one use of reduction).

$$h(F, X) \stackrel{\mathrm{red}}{\rightarrow}^{+} X :: map(F, h(F, X)) \tag{14}$$

$$map(F, nil) \stackrel{\mathrm{red}}{\rightarrow}^{+} nil \tag{15}$$

$$map(F, H :: T) \stackrel{\mathrm{red}}{\rightarrow}^{+} F(H) :: map(F, T) \tag{16}$$

$$iter(F, X) \stackrel{\mathrm{red}}{\rightarrow}^{+} X :: iter(F, F(X)) \tag{17}$$

Proof. The process of proof will be divided into stages to help facilitate later analysis.

3.1. Application of the Coinduction Rule

Recall the coinduction rule for labelled transition systems:

$$\frac{\langle a,\ b\rangle \in \mathcal{R} \quad \mathcal{R} \subseteq \langle \mathcal{R} \cup \sim\rangle}{a \sim b} \tag{18}$$

To show that $h(F, X) \sim iter(F, X)$ we need to introduce some $\mathcal{R}$ such that $\langle h(F, X),\ iter(F, X)\rangle \in \mathcal{R}$ and $\mathcal{R} \subseteq \langle \mathcal{R} \cup \sim\rangle$. As commented in §1, this step is one of the barriers to full automation of coinductive proof.

[5] We use capitals throughout this paper to indicate universally quantified variables. The scope of such quantifiers is always the entire formula.

For this proof we define the relation $\mathcal{R}$ as

$$\mathcal{R} \stackrel{\text{def}}{=} \{\langle (map(F))^N(h(F,X)),\ iter(F, F^N(X))\rangle\} \tag{19}$$

Where $(\cdots)^N$ obeys the following reduction rules:

$$F^0(X) \stackrel{\text{red}^+}{\to} X \tag{20}$$

$$F^{s(N)}(X) \stackrel{\text{red}^+}{\to} F(F^N(X)) \tag{21}$$

$$F^N(F(X)) \stackrel{\text{red}}{\to} F(F^N(X)) \tag{22}$$

$$(map(F))^N(H :: T) \stackrel{\text{red}}{\to} F^N(H) :: (map(F))^N(T) \tag{23}$$

The first premise of (18) applied to (13) gives the subgoal

$$\langle h(F,X),\ iter(F,X)\rangle \in \mathcal{R} \tag{24}$$

We know that $\langle (map(F))^0(h(F,X)),\ iter(F, F^0(X))\rangle \in \mathcal{R}$ by the definition of $\mathcal{R}$. What is more

$$(map(F))^0(h(F,X)) \stackrel{\text{red}^+}{\to} h(F,X) \tag{25}$$

$$iter(F, F^0(X)) \stackrel{\text{red}^+}{\to} iter(F,X) \tag{26}$$

Hence $\langle h(F,X),\ iter(F,X)\rangle \in \mathcal{R}$. This discharges the first premise of (18).

The second premise of (18) is $\mathcal{R} \subseteq \langle \mathcal{R} \cup \sim\rangle$.

3.2. Dealing with $\mathcal{R} \subseteq \langle \mathcal{R} \cup \sim\rangle$

In order to show that $\mathcal{R} \subseteq \langle \mathcal{R} \cup \sim\rangle$ we will use a derived inference rule for *deterministic* labelled transition systems (A labelled transition system is deterministic if for all a and for all transitions α if $a \stackrel{\alpha}{\to} a'$ and $a \stackrel{\alpha}{\to} a''$ then a' is syntactically identical to a''), (27), the proof of which can be found in appendix A.

In order to discharge the subgoal $\mathcal{R} \subseteq \langle \mathcal{R} \cup \sim\rangle$, it is necessary to express how the function $\langle - \rangle$ relates $\mathcal{R}$ to transitions on its members. The central concept of a coinductive proof is to show that the results of transitions on members of a relation (or some function(s) applied to members of a relation if using coinduction outside of labelled transition systems) are also members of that relation. The inference rule shown is one way of getting formally from the subgoal $\mathcal{R} \subseteq \langle \mathcal{R} \cup \sim\rangle$ to something that explicitly expresses this concept. We make no claims that this is the only or the best way to reach such an expression, or even that the need for explicitly stating this requirement is always necessary.

In essence (27) converts the subset expression into an implication based on the definition of $\langle - \rangle$ stating that the result of matching transitions from a related pair of expressions is also a related pair of expressions.

$$\frac{\begin{array}{l}\forall \mathcal{R}. \forall 1 \leq i \leq n. \bigwedge_{i=1}^{n} \langle a_i,\, b_i \rangle \in \mathcal{R} \Rightarrow \\ \quad (\forall \alpha. \exists a_i', b_i'. ((a_i \xrightarrow{\alpha} a_i' \vee b_i \xrightarrow{\alpha} b_i') \Rightarrow \\ \qquad ((a_i \xrightarrow{\alpha} a_i' \wedge b_i \xrightarrow{\alpha} b_i') \wedge \\ \qquad \langle a_i',\, b_i' \rangle \in \mathcal{R} \cup \sim)))\end{array}}{\bigcup_{i=1}^{n} \langle a_i,\, b_i \rangle \subseteq \langle \bigcup_{i=1}^{n} \langle a_i,\, b_i \rangle \cup \sim \rangle} \tag{27}$$

In this rule $\mathcal{R}$ is universally quantified since it is possible to construct a relation $\{\langle a_1, b_1 \rangle, \ldots, \langle a_n, b_n \rangle\}$ which is not a bisimulation but which has some superset $\mathcal{R}$ which nevertheless satisfies the property. We can existentially quantify a_i' and b_i' since the system is deterministic. In a non-deterministic system a more complex rule would be needed. The rule gives a conjunction of statements listing pairs in $\mathcal{R}$ as hypotheses. Obviously in reality we are only actually interested in $\{\langle a_1, b_1 \rangle, \ldots, \langle a_n, b_n \rangle\}$ and there are more elegant rules that could be used. However this rule proved itself the most practical for the purposes of our implementation.

Returning to the proof and using (27) we get the following subgoal:

$$\begin{array}{c}\forall \mathcal{R}. \langle (map(F))^{N}(h(F,X)),\ iter(F, F^{N}(X)) \rangle \in \mathcal{R} \Rightarrow \\ \forall \alpha. \exists \phi, \psi. ((map(F'))^{N'}(h(F',X')) \xrightarrow{\alpha} \phi \vee iter(F', F'^{N'}(X')) \xrightarrow{\alpha} \psi) \Rightarrow \\ ((map(F'))^{N'}(h(F',X')) \xrightarrow{\alpha} \phi \wedge iter(F', F'^{N'}(X')) \xrightarrow{\alpha} \psi \wedge \\ \langle \phi,\, \psi \rangle \in \mathcal{R} \cup \sim)\end{array} \tag{28}$$

3.3. Reducing the Expressions to Weak Head Normal Form

We want to check all the possible transitions, α, from $(map(F'))^{N'}(h(F',X'))$ and $iter(F', F'^{N'}(X'))$. We cannot do this immediately because the expressions are not values and transitions only explicitly apply to values. So our first task is to evaluate these terms.

$$(map(F'))^{N'}(h(F',X')) \Downarrow F'^{N'}(X') :: (map(F'))^{N'}(map(F', h(F',X'))) \tag{29}$$

$$iter(F', F'^{N'}(X')) \Downarrow F'^{N'}(X') :: iter(F', F'(F'^{N'}(X'))) \tag{30}$$

In this case reduction using rules (14), (23) and (17) leads directly to values, however it should be noted that sometimes, due to the presence of free variables in the expressions, reduction leads to irreducible terms that are not values. In this case more work (i.e. casesplitting of free variables) has to be done to discover the possible values of the expression. Induction is not appropriate here since the process undertaken is one of search to determine the WHNF, rather than a proof that something *is* the WHNF. It might be possible to use induction together with middle-out reasoning [10] but this seems an overly complex approach when a process of reduction and casesplitting will suffice.

3.4. Taking Transitions

Applying the evaluations in (29) and (30) to (28) allows us to determine the results of the transitions by inspection of the transition rules for the language. The possible transitions are `hd` and `tl` which can be applied to the expressions obtained in (29) and (30) as follows:

$$F'^{N'}(X') :: (map(F'))^{N'}(map(F', h(F', X'))) \overset{\texttt{hd}}{\rightarrow} F'^{N'}(X')$$
$$F'^{N'}(X') :: iter(F', F'(F'^{N'}(X'))) \overset{\texttt{hd}}{\rightarrow} F'^{N'}(X')$$

$$F'^{N'}(X') :: map(F')^{N'}(map(F', h(F', X'))) \overset{\texttt{tl}}{\rightarrow} map(F')^{N'}(map(F', h(F', X')))$$
$$F'^{N'}(X') :: iter(F', F'(F'^{N'}(X'))) \overset{\texttt{tl}}{\rightarrow} iter(F', F'(F'^{N'}(X')))$$

Since there are two possible transitions there are two subgoals resulting from this step. The first is obtained by substituting the results of the head transitions into (28) followed by tidying up any conjuncts and disjuncts that have become trivially true. This leaves the goal:

$$\begin{array}{c} \forall \mathcal{R}.\, \langle (map(F))^N(h(F, X)),\ iter(F, F^N(X)) \rangle \in \mathcal{R} \Rightarrow \\ \langle F'^{N'}(X'),\ F'^{N'}(X') \rangle \in \mathcal{R} \cup \sim \end{array} \tag{31}$$

A similar process using the tail transitions, leaves the goal:

$$\begin{array}{c} \forall \mathcal{R}.\, \langle map(F)^N(h(F, X)),\ iter(F, F^N(X)) \rangle \in \mathcal{R} \Rightarrow \\ \langle map(F')^{N'}(map(F', h(F', X'))),\ iter(F', F'(F'^{N'}(X'))) \rangle \in \mathcal{R} \cup \sim \end{array} \tag{32}$$

3.5. Rewriting, Reflexivity and Fertilization

(31) is true by the reflexivity of $\sim$.

Some general rewriting is needed to prove (32), in some cases such rewriting is also needed for the goal resulting from head transitions. The rules $F(F^N(X)) \rightarrow F^{s(N)}(X)$ and $F^N(F(X)) \rightarrow F(F^N(X))$ rewrite the goal to

$$\begin{array}{c} \forall \mathcal{R}.\, \langle (map(F))^N(h(F, X)),\ iter(F, F^N(X)) \rangle \in \mathcal{R} \Rightarrow \\ \langle (map(F'))^{s(N')}(h(F', X')),\ iter(F', F'^{s(N')}(X'))) \rangle \in \mathcal{R} \end{array}$$

We can then appeal to the hypothesis to finish the proof since N is universally quantified. This process is called *fertilization*. $\square$

4. Proof Planning

Proof plans were first proposed by Bundy [7] and have been successfully applied to inductive theorem proving and other domains. Proof plans have two basic components, proof *methods* and proof *tactics*. Tactics are algorithms for constructing combinations of low-level inference rule applications. Methods characterise the tactics by specifying the preconditions 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 in a tactic based theorem prover which provides a formal verification of the theorem. The objective is to separate proof discovery from proof checking.

Proof planning has been implemented in *CLAM* [9] and Ωmega [5]. The discussion here 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. This makes proof planning an attractive option in any attempt to automate or provide proof tools to support coinduction.

4.1. Proof Methods

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

Inputs	A pattern which must match with the current goal.
Preconditions	Conditions which must hold for the method to apply stated in some meta–logic.
Outputs	Subgoals Generated by the method, a list of meta–level sequents.
Effects	Post–conditions of the method's application stated in some meta–logic.
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. The work reported here hasn't been linked to an object level prover so the tactic slot is omitted in the rest of the discussion. In practice the Effects slot was not used by the methods described here and so has also been omitted.

A sequence of expected method applications forms a proof strategy for a given type of proof, for instance the proof strategy for induction involves splitting the goal using the Induction method into a base case and a step case. These goals are in turn solved by simplification methods or further induction for the base case and rippling (a rewriting method) followed by fertilization for the step case. This sequence of methods forms a proof strategy for induction.

4.2. Proof Critics

A proof strategy provides a guide 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* [23] can be employed to analyse the reasons for failure and propose alternative choices.

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 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. Critics patch the proof in different ways depending upon which of a method's preconditions fail.

5. A Proof Strategy for Coinduction

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.

A number of specific assumptions are made about the relationship of transitions and values

- It is assumed that transition rules are defined on some set of values and require no additional hypotheses for application.
- At least one transition is defined for every value.
- There is a small step reduction order, $\stackrel{\text{red}}{\rightarrow}$.
- If a descending chain of ground expressions in this order has a least element, that element is a value and the following rule applies:

$$\frac{a \Downarrow a' \quad a' \stackrel{\alpha}{\rightarrow} b}{a \stackrel{\alpha}{\rightarrow} b} \tag{33}$$

These assumptions are all met by standard transition systems representing the operational semantics of functional languages.

Another choice was not to pursue problems which contained terms with no value (i.e. terms for which evaluation fails to terminate - e.g. problems with the $flatten$ function (see [26])). Such problems involve additional analysis to show that evaluating each side of the relation diverges under the same conditions. No criteria are presented for detecting such theorems.

The following strategy embodies an analysis of the process of coinductive proof within the limits specified.

1. A coinductive proof starts with the application of the coinduction rule which produces two subgoals ($\langle x,\, y\rangle \in \mathcal{R}$ and $\mathcal{R} \subseteq \langle \mathcal{R} \cup \sim\rangle$). To produce these subgoals some relation has to be introduced.
2. (a) The first subgoal is discharged relatively easily.
   (b) The second subgoal is more complicated. First it is converted into some goal of the form *coinduction hypothesis* $\Rightarrow$ *coinduction conclusion* which explicitly expresses how the function $\langle - \rangle$ relates a relation to transitions on its members. This transformation is based on (27).
3. The subgoals produced by step 2b can only be discharged by determining transitions from one or more terms. This requires the evaluation of those terms to WHNF. This process of reduction can follow the strategy of non–strict evaluation, where terms are only reduced if needed.

   Definition 8. (Redex) A term M is a redex (reducible expression) if it matches the LHS of a reduction rule.

   Definition 9. (Non–strict Evaluation) 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 is appropriate because, when applied to closed terms, it always terminates in a WHNF if one exists.
4. Transitions can then be determined by reference to the transition rules of the system.
5. Further rewriting leads to goals that can be solved by appeal either to the hypotheses or the reflexivity of $\sim$.

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[6].

[6] Proof plans provide a lot of detail, equivalent to the detail of a proof on paper and so inspection of the proof plans can provide a reasonable degree of confidence in the output of *CoCLAM*.

6. Proof Methods for Coinduction

This section discusses the proof methods required by the proof strategy for coinduction.

6.1. Coinduction Method

This method starts out a coinductive proof by applying the coinduction rule. By inspection of rule (18) we can see that the coinduction method applies if the goal is of the form $a \sim b$. The method's outputs, or the new goals, will be $\langle a,\, b\rangle \in \mathcal{R}$ 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. The obvious heuristic for choosing a relation is to pick the smallest possible relation that discharges the first precondition of (18). 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 there is a possibility that $\mathcal{R}$ may need to be revised. The choice of $\mathcal{R}$ is defeasible, that is its definition can be changed in the event of the proof failing to go through: we will use proof critics to identify such situations (see §7.1).

The coinduction method is described in figure 1.

NB. For a verification it would be necessary to check that $\langle f(\bar{X}), g(\bar{X})\rangle \in \mathcal{R}$. *CoCLAM* doesn't do this: it assumes that the $\mathcal{R}$ chosen (either by the coinduction method itself or by the critic (see §7.1)) satisfies this property. This is not an inherent shortcoming of proof planning. It would be possible to set up a method to check this. The nature of the coinductive proof strategy means that $\langle f(\bar{X}),\, g(\bar{X})\rangle \in \mathcal{R}$ will be true for the chosen $\mathcal{R}$ therefore it was unnecessary to provide the method in order to generate a plan.

6.2. Gfp Membership Method

The second stage of a coinductive proof involves showing that $\mathcal{R}$ is a included in 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 inference rule (27). It is described in figure 2.

Input

$f(\bar{X}) \sim g(\bar{X})$.

Preconditions

None.

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 (by a critic) 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 1. The Coinduction($\mathcal{R}$) Method

Input

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

Preconditions

None.

Outputs

For each $i.i \in \{1, \cdots, n\}$ form the goal:

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

Figure 2. The Gfp Membership($\langle\cdots\rangle$) Method

6.3. Evaluate Method

The evaluate method embodies a non–strict evaluation strategy with narrowing [28]. The object is to reduce the expressions l_i in goals of the form

$$\begin{gathered}\textit{coinduction hypothesis} \Rightarrow \\ \forall\alpha.((l_a \xrightarrow{\alpha} \phi \vee l_b \xrightarrow{\alpha} \psi) \Rightarrow \\ ((l_a \xrightarrow{\alpha} \phi \wedge l_b \xrightarrow{\alpha} \psi)\wedge \\ \langle\phi,\ \psi\rangle \in \mathcal{R}\cup \sim))\end{gathered} \tag{34}$$

to weak head normal form. To do this, non–strict evaluation has to be extended with some case splitting ability (i.e. narrowing). l_a and l_b are of the form $\forall \bar{x}.exp(\bar{x})$ where the $\bar{x}$ are variables occurring free in exp, if any exist, which represent arbitrary input terms for the functional program. Reduction is applied to $exp(\bar{x})$ to obtain the possible values of $exp(\bar{x})$ given arbitrary values of $\bar{x}$. Non–strict reduction is only guaranteed to terminate in a value if the original expression is ground (a program would be ground for any specific input). Hence $exp(\bar{x})$ may be irreducible even though it is not a value (since it is non–ground because we are reasoning about arbitrary input).

As an example consider the term $\forall x : (nat)list.x$. In this example, x is not in WHNF. To be in WHNF it would have to be either nil or $h :: t$ (for some free variables h and t). However x is irreducible.

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). For instance in the example if x is case split as nil and $H :: T$ (the values of list type) it is immediately in WHNF. 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.

6.3.1. A Reduction Strategy (Non-strict Evaluation with Narrowing)

The reduction strategy on a term is

1. Perform non–strict evaluation (without case–splitting) on the term until it terminates 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. 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 3. This method is not guaranteed to terminate.

6.4. Transition Method

The Transition method unfolds the goals according to the rules that specify the transition system.

Input

$$
\begin{gathered}
\textit{coinduction hypothesis} \Rightarrow \\
\forall \alpha.((l_a \xrightarrow{\alpha} \phi \vee l_b \xrightarrow{\alpha} \psi) \Rightarrow \\
((l_a \xrightarrow{\alpha} \phi \wedge l_b \xrightarrow{\alpha} \psi) \wedge \langle \phi,\, \psi \rangle \in \mathcal{R} \cup \sim))
\end{gathered}
$$

Preconditions

For some i ($i = a, b$), l_i reduces to v_i, a value, using the reduction strategy with the reduction rules, *rules*, and possibly case–splitting some variables which form a set, $\mathcal{V}$. v_i depends upon a particular set V of variable/value pairs, one pair for each variable in $\mathcal{V}$.

Outputs

- Form one goal by replacing l_i with v_i everywhere it appears in the input.
- Assume $\mathcal{V}$ is size k (where k may equal 0).
  Any variable, $var_j (1 \leq j \leq k)$, in $\mathcal{V}$ can have two or more values depending upon its type. Say var_j has n possible values. This gives rise to a set, A^j, of variable/value pairs $(\{(var_j, value_{j1}), ..., (var_j, value_{jn})\})$. Each variable $\mathcal{V}$ is associated with such a set.
  Form sets, B^m, $(var_1, value_{1m^1}), \cdots, (var_k, value_{km^k})$ by picking one pair from each A^j so that one such set exists for every combination of these pairs.
  Form goals for each B^m by replacing all the var_j with $value_{jm^j}$ $(1 \leq j \leq k)$ everywhere they appear in the goal.

Figure 3. The Evaluate(*rules*, $\mathcal{V}$) Method

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 and a is a value. This means α and ϕ can now be instantiated by inspecting the transition rules.

This situation is slightly complicated by the requirement that if a transition applies to one side of a relation it should apply to both. However the method remains a fairly straightforward process of applying the transition rules for the language and tidying up the expressions $a \xrightarrow{\alpha} \phi$ which have become trivially true.

Lastly difference matching is performed to motivate the rippling used in the next part of the proof strategy (§6.5). Difference matching is a process by which two expressions are compared. If one expression can be embedded in the other then this is indicated by placing boxes (or *wave fronts*) round the additional parts of the larger expression. The embedded expression can be identified as those parts of the expression that are outside the boxes or those parts within boxes which are underlined (called *wave holes*). As an example the result of difference matching $s(x)$ and $s(x+y)$ would be $s(\boxed{\underline{x}+y})$. Here the first expression, $s(x)$, appears outside the wave fronts or in the wave holes of the annotated expression $s(\boxed{\underline{x}+y})$. If the annotations of $s(\boxed{\underline{x}+y})$ are stripped away then the second expression, $s(x+y)$, is left. A full description of difference matching and algorithm for it is given in [3]. The important point is that difference matching picks out the differences between two expressions.

The transition method is shown in figure 4.

Input

Any goal.

Preconditions

1. The expression $a_1 \xrightarrow{\alpha} \phi \wedge a_2 \xrightarrow{\alpha} \psi$ appears in the goal.
2. For one $i \in \{1,2\}$, the set of all transitions that apply to a_i is *transitions* and this set is non–empty.
3. All the transitions in *transitions* also apply to $a_j, j \in \{1,2\}, j \neq i$. There is no transition that applies to a_j that isn't in *transitions*.

Outputs

Form a new goal for each $\alpha_k \in$ *transitions*, by the following process:

1. Replace $a_1 \xrightarrow{\alpha} \phi$ and $a_2 \xrightarrow{\alpha} \psi$ with *true* wherever they appear in the goal.
2. Replace all remaining occurrences of ϕ with a_1' where $a_1 \xrightarrow{\alpha_k} a_1'$ and all remaining occurrences of ψ with a_2' where $a_2 \xrightarrow{\alpha_k} a_2'$.
3. Remove all the occurrences of *true* using the rules
$$(true \Rightarrow P) \rightarrow P \qquad (true \wedge P) \rightarrow P \qquad (true \vee P) \rightarrow true$$
4. If the resulting expression differences matches against one of the hypotheses then annotate the expression accordingly.

Figure 4. The Transition(*transitions*) Method

6.5. Wave Method

Recall that in example 7 the tail transitions leave the goal:

$$\forall \mathcal{R}.\ \langle (map(F))^{N}(h(F,X)),\ iter(F, F^{N}(X))\rangle \in \mathcal{R} \Rightarrow$$
$$\langle (map(F'))^{N'}(map(F', h(F',X'))),\ iter(F', F'(F'^{N'}(X')))\rangle \in \mathcal{R}$$

This can't be solved immediately by appeal to the hypothesis 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 $\langle map(F')^{s(N')}(h(F',X')),\ iter(F', F'^{s(N')}(X'))\rangle\rangle$ using the rewrite rules $F(F^{N}(X)) \to F^{s(N)}(X)$ and $F^{N}(F(X)) \to F(F^{N}(X))$. We can then appeal to the hypothesis since N is a universally quantified variable. The first of these rewrite rules is the reverse of a reduction rule, (21), we have already used to evaluate the expression. This means we can't use reduction for this step but need a more general rewriting method.

This is similar to the purpose for which rippling (a heuristic for controlling rewriting in induction) [8] is designed, in fact it corresponds to "rippling into a sink" since the aim is to sink the differences between the hypothesis and conclusion in a universal variable.

Rippling controls rewriting by the use of *wave rules*, annotated rewrite rules. These annotations are the same as those produced by difference matching. Annotations consist of contexts (expressions with holes) indicated by a wave front (box) with a directional arrow. The holes in the context are wave holes (i.e. they are filled with an expression which is underlined). In induction, the induction hypothesis is embedded in the induction conclusion. Annotations are used to indicate this with the *skeleton* of the induction conclusion (the expression with the contents of wave fronts that are not included in wave holes removed) matching the induction hypothesis. So the goal

$$a + (B + C) = (a + B) + C \Rightarrow$$
$$s(a) + (B' + C') = (s(a) + B') + C'$$

is annotated as

$$a + (B + C) = (a + B) + C \Rightarrow$$
$$\boxed{s(\underline{a})}^{\uparrow} + (B' + C') = (\boxed{s(\underline{a})}^{\uparrow} + B') + C' \tag{35}$$

The annotation is done by difference matching as described in §6.4. Wave rules are skeleton preserving so all possible rewrites of the above induction conclusion using wave rules will result in terms with the induction hypothesis as the skeleton.

The ripple method implements a restricted form of rewriting: The induction conclusion is rewritten using only wave rules. Wave rules require both the object–

level term structure and the meta–level annotations to match. So, for instance, the LHS of the conclusion of (35) matches the LHS of the wave rule:

$$\boxed{s(\underline{A})}^{\uparrow} + B \to \boxed{s(\underline{A + B})}^{\uparrow}$$

while it does not match the wave rule

$$\boxed{s(\underline{A}) + B}^{\uparrow} \to \boxed{s(\underline{A} + B)}^{\uparrow}$$

Rippling, unlike general rewriting, is terminating. Rippling either moves wave fronts outwards (indicated by the upward arrow) in the term structure so that they can be cancelled away or inwards (indicated by a downward arrow) so that the differences surround a universally quantified variable (or *sink*). Rippling is embodied by the Wave method which is described in [8]. Formal definitions for annotated terms and wave rules and a termination proof can be found in [4].

In order to use rippling in coinduction, it is necessary to provide annotations. Taking a cue from the use of rippling in induction, the obvious approach is to difference match against the coinduction hypothesis. This difference matching is performed by the Transition method when forming its output in the anticipation, based on the proof strategy, that rippling is the next method to be applied.

Returning to the example, the goal is annotated by difference matching as:

$$\begin{gathered}\forall \mathcal{R}.\ \langle (map(F))^N(h(F, X)),\ iter(F, F^N(X))\rangle \in \mathcal{R} \Rightarrow \\ \langle (map(F'))^{N'}(\boxed{map(F', \underline{h(F', X')})}^{\uparrow}),\ iter(F', \boxed{F'(\underline{F'^{N'}(X')})}^{\uparrow})\rangle \in \mathcal{R}\end{gathered}$$

CoCL^AM automatically annotates all function definitions and lemmata as wave rules. In this case the rules of interest are

$$\boxed{F(\underline{F^N(X)})}^{\uparrow} \to F^{\boxed{s(\underline{N})}^{\downarrow}}(X) \tag{36}$$

$$F^N(\boxed{F(\underline{X})}^{\uparrow}) \to \boxed{F(\underline{F^N(X)})}^{\uparrow} \tag{37}$$

These ripple the wave fronts in the coinduction conclusion as follows:

$$\begin{aligned}&\cdots \Rightarrow \langle (map(F'))^{N'}(\boxed{map(F', \underline{h(F', X')})}^{\uparrow}),\ iter(F', \boxed{F'(\underline{F'^{N'}(X')})}^{\uparrow})\rangle \in \mathcal{R} \\ &\cdots \Rightarrow \langle \boxed{map(F', \underline{map(F')^{N'}(h(F', X'))})}^{\uparrow},\ iter(F', \boxed{F'(\underline{F'^{N'}(X')})}^{\uparrow})\rangle \in \mathcal{R} \\ &\cdots \Rightarrow \langle (map(F'))^{\boxed{s(\underline{N'})}^{\downarrow}}(h(F', X')),\ iter(F', F'^{\boxed{s(\underline{N'})}^{\downarrow}}(X'))\rangle \in \mathcal{R}\end{aligned}$$

The wave fronts have been rippled into the sink, N', and fertilization (as described in §3.5) may now occur.

6.6. Fertilization and Other Methods

The fertilization method (appeal to the coinduction hypothesis) is a standard method used in induction but had to be extended to recognise matches involving $(\cdots)^N$ (see §8.3). The only other method required is appeal to the reflexivity of $\sim$.

7. The Coinduction Method Heuristic

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 that the chosen relation might not be a bisimulation.

Example 10. Consider example 7.

$$\forall x, f.\, h(f,x) \sim iter(f,x)$$

The Coinduction method, as described, will choose the trial relation

$$\{\langle h(F,X),\ iter(F,X)\rangle\} \tag{38}$$

The Gfp Membership and Evaluate methods will provide the subgoal

$$\begin{gathered}\forall \mathcal{R}.\, \langle h(F,X),\ iter(F,X)\rangle \in \mathcal{R} \Rightarrow \\ \forall \alpha.((X' :: map(F', h(F',X')) \xrightarrow{\alpha} \phi \vee X' :: iter(F'), F'(X') \xrightarrow{\alpha} \psi) \Rightarrow \\ (X' :: map(F', h(F',X')) \xrightarrow{\alpha} \phi \wedge X' :: iter(F'), F'(X') \xrightarrow{\alpha} \psi) \wedge \\ \langle \phi,\ \psi\rangle \in \mathcal{R} \cup \sim)\end{gathered} \tag{39}$$

There are two possible values for α, `hd` and `tl`. We will only 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

$$\begin{gathered}\forall \mathcal{R}.\, \langle h(F,X),\ iter(F,X)\rangle \in \mathcal{R} \Rightarrow \\ \langle \boxed{map(F', \underline{h(F',X')})}^{\uparrow},\ iter(F', \boxed{F'(\underline{X'})}^{\uparrow})\rangle \in \mathcal{R}\end{gathered} \tag{40}$$

The attempt now fails because $\langle map(F', h(F',X')),\ iter(F', F'(X'))\rangle \notin \mathcal{R}$.

There are two possible solutions to this problem, either the heuristic is improved or proof critics are employed to modify the choice of relation.

The trial relation 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 relation. 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.

7.1. The Revise Bisimulation Critic

A critic is added to the proof strategy for coinduction which extends the current trial relation by adding in any new successor states. At the same time it seeks for patterns that will allow sequences which, if formed in this way, would be infinite, but finitely described. The critic is shown in figure 5.

Preconditions

1. The coinduction method has been used in this branch of the proof.
2. The current goal is $\forall \mathcal{R}.\, \langle f_1,\, g_1 \rangle \in \mathcal{R} \wedge \ldots \wedge \langle f_n,\, g_n \rangle \in \mathcal{R} \Rightarrow \langle \phi,\, \psi \rangle \in \mathcal{R}$. Where ϕ and ψ are both instantiated and $\langle \phi,\, \psi \rangle$ is not unifiable with any $\langle f_i,\, g_i \rangle, i \leq n$.

Patch

If

A pattern has been spotted (see §7.1.1 for details) in 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 relation 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 to the method.

Figure 5. The Revise Bisimulation Critic

Although this is a very general description it should be clear that the coinduction proof strategy, together with this critic, provides a method for extending the trial relation in a controlled way.

Example 11 shows the critic in action.

Example 11.

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

where:

$$lswap(A, B) \stackrel{\text{red}^+}{\rightarrow} A :: lswap(B, A) \tag{41}$$

$$lconst(A) \stackrel{\text{red}^+}{\rightarrow} A :: lconst(A) \tag{42}$$

$$merge(nil, L) \stackrel{\text{red}^+}{\rightarrow} L \tag{43}$$

$$merge(L, nil) \stackrel{\text{red}^+}{\rightarrow} L \tag{44}$$

$$merge(H_1 :: T_1, H_2 :: T_2) \stackrel{\text{red}^+}{\rightarrow} H_1 :: H_2 :: merge(T_1, T_2) \tag{45}$$

The Coinduction method chooses the trial relation

$$\{\langle lswap(A, B),\ merge(lconst(A), lconst(B))\rangle\}$$

The Gfp Membership and Evaluate methods give the goal

$$\begin{array}{c} \forall \mathcal{R}.\ \langle lswap(A, B),\ merge(lconst(A), lconst(B))\rangle \in \mathcal{R} \Rightarrow \\ ((A' :: lswap(B', A') \stackrel{\alpha}{\rightarrow} \phi \vee \\ A' :: B' :: merge(lconst(A'), lconst(B')) \stackrel{\alpha}{\rightarrow} \psi) \Rightarrow \\ A' :: lswap(B', A') \stackrel{\alpha}{\rightarrow} \phi \wedge \\ A' :: B' :: merge(lconst(A'), lconst(B')) \stackrel{\alpha}{\rightarrow} \psi \wedge \\ \langle \phi,\ \psi\rangle \in \mathcal{R} \cup \sim) \end{array} \tag{46}$$

There are two possible values for α, namely `hd` and `tl`. We will only 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

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

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 §7.1.1), 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. This time there are two goals, each with two coinduction hypotheses. After taking `tl` transitions these goals are:

$$\begin{array}{c}\forall \mathcal{R}.\ \langle lswap(A,B),\ merge(lconst(A), lconst(B))\rangle \in \mathcal{R} \wedge \\ \langle lswap(B',A'),\ B' :: merge(lconst(A'), lconst(B'))\rangle \in \mathcal{R} \Rightarrow \\ \langle lswap(B'',A''),\ B'' :: merge(lconst(A''), lconst(B''))\rangle \in \mathcal{R} \cup \sim\end{array} \tag{47}$$

$$\begin{array}{c}\forall \mathcal{R} \langle lswap(A,B),\ merge(lconst(A), lconst(B))\rangle \in \mathcal{R} \wedge \\ \langle lswap(B',A'),\ B' :: merge(lconst(A'), lconst(B'))\rangle \in \mathcal{R} \Rightarrow \\ \langle lswap(A'',B''),\ merge(lconst(A''), lconst(B''))\rangle \in \mathcal{R} \cup \sim\end{array} \tag{48}$$

These can both be solved by fertilization.

7.1.1. The Divergence Check

Up until now we have been vague about the process of detecting infinite sequences which is the final part of the Revise Bisimulation critic.

The "spotting of patterns" mentioned in figure 5 is performed using a *divergence check* based on work by Walsh [34]. The check attempts to find some term structure, introduced by the revisions, accumulating in the sequence of equations which describe the relation. It is this structure which is preventing fertilization solving the goals. The critic identifies the accumulating structure using difference matching (§6.4) which highlights the differences between two terms. It is described in figure 6.

Conditions

1. There is a subsequence of pairs, $\bigcup_{i=1}^{n}\{\langle l_i,\ r_i\rangle\}$, within the relation, $\mathcal{R}$, 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:

$$\{\langle l_j,\ r_j\rangle\} = \{\langle G_l(U_{l_j}),\ G_r(U_{r_j})\rangle\},$$
$$\{\langle l_{j+1},\ r_{j+1}\rangle\} = \{\langle G_l(\boxed{H_l(\underline{U_{l_j}})}),\ G_r(\boxed{H_r(\underline{U_{r_j}})})\rangle\}$$

Proposed Replacement of Sequence

$$\{\langle G_l((H_l)^N(U_{l_0})),\ G_r((H_r)^N(U_{r_0}))\rangle\}$$

Figure 6. The Divergence Check

Once potential divergence is spotted it is necessary to find an appropriate patch. Walsh's divergence critic patched proofs by speculating and proving additional lemmata. In coinductive proofs, a generalisation using $(\cdots)^N$ replaces the detected subsequence, since the divergence is assumed to be caused by the repeated addition of H_i (as defined by the divergence check) every time the tail of the latest addition to the relation is examined.

The need for the divergence check can be illustrated by returning to example 7.

Example 12. Recall, from examples 7 and 10, that the coinduction method will choose the trial relation

$$\{\langle h(F,X),\ iter(F,X)\rangle\} \tag{49}$$

and as a result that the proof attempt becomes stuck at the goal

$$\begin{array}{c}\forall \mathcal{R}.\ \langle h(F,X),\ iter(F,X)\rangle \in \mathcal{R} \Rightarrow \\ \langle \boxed{map(F',\underline{h(F',X')})}^{\uparrow},\ iter(F',\boxed{F'(\underline{X'})}^{\uparrow})\rangle \in \mathcal{R}\end{array} \tag{50}$$

According to the critic (figure 5) this new pair is added into the trial bisimulation.

If the process implemented by the critic (without the divergence check) were repeated two more subgoals would be produced. The first subgoal would be similar to (40) but with an additional hypothesis. This extra hypothesis can be used to fertilize at the point where the proof attempt became blocked (50). The second new subgoal, after taking `tl` transitions, would be:

$$\begin{array}{c}\langle h(F,X),\ iter(F,X)\rangle \in \mathcal{R}\wedge \\ \langle map(F',h(F',X')),\ iter(F',F'(X'))\rangle \in \mathcal{R} \Rightarrow \\ \langle map(F'',\boxed{map(F'',\underline{h(F'',X'')})}^{\uparrow}),\ iter(F'',\boxed{F''(\underline{F''(X'')})}^{\uparrow})\rangle \in \mathcal{R}\end{array} \tag{51}$$

Once again the Revise Bisimulation critic would intervene and suggest adding $\{\langle map(F'',map(F'',h(F'',X''))),\ iter(F'',F''(F''(X'')))\rangle\}$ to $\mathcal{R}$. A human can tell that this process can be infinitely repeated, i.e. the sequence of `tl` successors represents unwanted divergence.

To prevent this, *CoCLAM* checks whether one pair in the bisimulation can be embedded in another each time a new pair is produced, since this implies the presence of some accumulating structure. The first two elements the above sequence of pairs were:

$$\begin{array}{c}\langle h(F,X),\ iter(F,X)\rangle \in \mathcal{R} \\ \langle map(F',h(F',X')),\ iter(F',F'(X'))\rangle \in \mathcal{R}\end{array}$$

Difference matching these[7], produces the annotated sequence:

$$\langle h(F,X),\ iter(F,X)\rangle \in \mathcal{R}$$
$$\langle \boxed{map(F', \underline{h(F',X')})},\ iter(F', \boxed{F'(\underline{X'})})\rangle \in \mathcal{R}$$

This suggests that there is some potentially accumulating term structure. Moreover the difference matching singles out the differences between the two expressions which are preventing fertilization occurring.

The critic instantiates G_l, G_r, H_l, H_r and U_{l_0} and U_{r_0} (from figure 6) as id $(\lambda x.x)$, $iter(F)$, $map(F)$, F, $h(F,X)$ and X respectively and so proposes that $\mathcal{R} = \{\langle (map(F))^N(h(F,X)),\ iter(F, F^N(X))\rangle\}$. This allows the proof to go through as it did when example 7 was originally presented.

7.1.2. Limitations of the Divergence Check

Ideally a divergence check for the Revise Bisimulation critic should:

1. Fire if and only if the proof strategy was in the process of exploring an infinite transition sequence.
2. Only extend the trial bisimulation by elements in some minimal bisimulation.

Neither of these is necessarily the case with the divergence check proposed here (in fact, since bisimilarity to a given program is an extensional non–trivial property, it is undecidable [29]. No critic is going to be perfect). There are situations in which it can be "fooled" into generalising when in fact it is looking at a cycle of transitions which nevertheless have a common structure. Alternatively sometimes more complicated generalisations are required than those using $(\cdots)^N$, in particular, some new function may need to be speculated. These situations are discussed more fully in [15].

8. Experimental Results and Evaluation

The proof methods and critic proposed for coinduction 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. To provide an empirical evaluation of our ideas we have implemented them in a

[7] This difference matching should not be confused with the difference matching after the Transition method which resulted in the annotations on goals (50) and (51) which was intended to motivate rippling if possible. Here the difference matching is being used to identify a common pattern of differences.

planning system called *CoCLAM* [8]. *CoCLAM* was used to plan a representative selection of theorems using the methods and critics described in this paper and so to determine how effective was the proposed strategy.

This served a double purpose of evaluating the effectiveness of the proposed strategy and highlighting areas of the strategy that need improvement.

8.1. Source of Examples

The proof plans make no claim to deal with cases where there are terms with no value, so such theorems were excluded from consideration. These theorems were identified by inspection.

Examples of coinduction were drawn from the literature, in particular [16,24,26,31]. Examples are relatively few in number, 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 ourselves, in order to provide a reasonably sized database.

The *CLAM* corpus is a selection of theorems used for testing inductive proof plans. Many of these are over natural numbers and so a labelled transition system involving $\xrightarrow{\mathtt{p}}$ [9] as a transition was employed so that these theorems could be used in testing as well as those involving lists. 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 a small functional language containing lazy natural, list, labelled binary tree and labelled infinite tree types. Details can be found in [15].

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.

8.2. 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 and the number that the strategy failed

[8] *CoCLAM* is based on *CLAM*3 and is written in sicstus prolog 2.1. The code for *CLAM*3 and *CoCLAM* is available from `http://dream.dai.ed.ac.uk/software/systems/clam3/`.

[9] Predecessor over natural numbers, i.e. -1. This works analogously to $\xrightarrow{\mathtt{hd}}$ and $\xrightarrow{\mathtt{tl}}$.

to find a proof for. There is a third column, Imp. Failures. This is the number of theorems that failed because of perceived problems with the implementation, as opposed to the proof strategy. All but one of these implementation failures were subsequently eradicated without any change to the proof strategy. A second pass was made of the theorems after minor modifications were made (including the elimination of all but one implementation failure), the results of which are also shown.

	Number of Theorems	Theorems Proved	Theorems not Proved	Imp. Failures
Development Set	53	47	5	1
Test Set	54	41	8	5
Dev. Set (2nd pass)	53	47	5	1
Test Set (2nd pass)	54	45	9	0

It is reassuring to note that the similarity between the success rates 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.

8.3. Failure Analysis

The most interesting errors are those that pose problems for the proof strategy and these are examined here.

False Hypothesis not Recognised No additional methods were supplied to *CoCLAM* to enable it to exploit hypotheses other than the coinduction hypothesis. 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 limited and this caused one theorem to fail.

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).

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. For larger problems it is foreseeable that heuristics would be necessary to guide this search.

Matching of $(\cdots)^N$ 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 several proofs this failure led to unnecessary and infinitely repeating revisions of the trial relation. This lack of generality in the method clearly needs to be addressed. There are algorithms for performing such tasks (e.g [11]) which need to be incorporated into the Fertilize method.

8.4. 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 [23] 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. 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.

The proof of example 7 required the rewrite rule (23). This was needed in order to evaluate the expression to WHNF.

The development and testing periods revealed that such lemmata were required for most proofs where a generalisation occurred. It was observed that these lemmata corresponded to corecursive definitions [26] for F^N for some specific value of F. It is unsatisfactory to have this need for additional lemmata inherent in the proof strategy. It might be possible to extend the existing Lemma Speculation critic in some way to perform this task.

8.5. 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 ourselves, under the mistaken impression they were theorems, and included in the test set. These were (52) and (53).

$$lswap(0,1) \sim \mathit{inf\_list}(0, \mathit{flip}, id) \tag{52}$$

$$\mathit{inf\_list}(0, \mathit{flip}, id) \sim \mathit{inf\_list}(1, id, \mathit{flip}) \tag{53}$$

where $lswap$ is defined as before, id is the identity function and flip and $\mathit{inf\_list}$ are defined as in appendix B.

The failure allowed the hypotheses to be identified as non–theorems when the trace was inspected.

8.5.1. A Disproof Method

Both these hypotheses, and the two non–theorems used initially to check that *CoCL^AM* 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 develop 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.

8.6. 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 in many ways simple problems. The ability of *CoCLAM* to scale up to large problems is an important consideration. The previous comments on its shortcomings relate to this.

The memory problems encountered by the Evaluate method cast doubts on the ability of *CoCLAM* to tackle larger problems although there may be simple fixes such as changing the search strategy.

Perhaps more worryingly, anecdotal evidence[10] from attempts to use coinduction for hardware verification indicate 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 suggest a need for more sophisticated generalisation techniques to be developed.

[10] T. Melham, personal communication

The strategy could be tested more thoroughly by adapting it to other labelled transition systems. 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.

9. Related Work

The first and most widely used proof tool for coinduction is the Edinburgh Concurrency Workbench (CWB) [12] which was developed for use with Process Algebras and provides a fully automated prover for finite bisimulations over ground terms. There has also been an effort to incorporate tools for coinduction in several large theorem proving environments including HOL [13] and Isabelle [26] (among others). The work done in HOL is most closely related to this work.

9.1. HOL

Collins [13] 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 to define equivalence of programs.

Collins provides support for forming bisimulations and solving 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 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 conversions are provided, `LTS_EVAL_CONV` and `LTS_CASE_CONV` which are used with tactics. The first of these performs reduction and the second case splits variables. The user has to guide these conversions specifying when to case split and when to reduce.

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.

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 [32], a relational hardware description language. This work provided an encouraging test case for the implementation.

9.1.1. Co*CLAM Compared to HOL*

HOL is a tactic based theorem prover. It is designed mainly as a proof checker with a user guiding the proof search. Tactics are algorithms for constructing strings of inference rules. As a result HOL guarantees a sound proof. 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 in the other tactic based theorem proving environments) was intended to provide tools for a user to perform coinductive proof interactively, whereas the work in *CoCLAM* was always with a view to automation.

Unlike HOL, *CoCLAM* is not intended to provide soundness and its methods do not construct inference rules sequences, but specifications of tactics expressed as preconditions and results of their application. *CoCLAM* provides a greater degree of automation than the corresponding version of HOL.

CoCLAM should be regarded as complementary rather than a rival to the work done in tactic based theorem provers such as HOL. 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 we're aware, the only system of this kind. *CLAM* has been linked to HOL [6] and it is possible that *CoCLAM* will be adapted to use Collins' HOL tactics.

9.2. Circular Coinductive Rewriting

More recently a powerful technique called *circular coinductive rewriting* [18] has been developed. Circular coinductive rewriting combines behavioural rewriting with circular coinduction [30]. Broadly speaking this treats what we have termed the coinduction hypothesis as a rewrite rule, instead of extending the relation, the process extends the set of available rewrites. The authors report that circular coinductive rewriting can automatically prove all the examples to which *CoCLAM* has been applied.

9.3. Walsh's Divergence Critic

Walsh's [34] 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.

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 often produces a divergent set of equations. It has been observed that this happens if an appropriate generalisation or lemma isn't present.

We use the same pre–conditions for the critic as Walsh, but our patch is different.

10. Further Work

There are a number of extensions to be made to *CoCLAM* among these are:

- 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 paper.
- Investigate the use of other generalisation techniques to extend the Revise Bisimulation Critic.
- Provide better support for other labelled transition systems to test the generality of the proof strategy by creating a more general Evaluate method and support for non–deterministic transition systems. It should be noted that the revise bisimulation critic has already been tested in a non–labelled transition system environment [14].
- Link *CoCLAM* with HOL so that the proof plans can be verified using Collins' tactics.
- Extend the testing of the system.

These are all important because they would extend the range of the system away from the current "toy" problems and would allow more realistic theorems to be not only automatically planned but also formally verified.

11. Conclusion

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.

References

[1] M. Abadi and A. D. Gordon, A calculus for cryptographic protocols: The Spi Calculus, in : *Fourth ACM Conference on Computer and Communications Security*, ACM Press, 1997, pp. 36–47. Full version available as Technical Report 414, University of Cambridge Computer Laboratory, January 1997.

[2] S. Abramsky, The lazy lambda calculus, in: *Research Topics in Functional Programming.* ed. D. Turner, Addison Wesley, 1990, pp. 65–117".

[3] D. Basin and T. Walsh, Difference matching, in *11th Conference on Automated Deduction*, ed. D. Kapur, Lecture Notes in Artificial Intelligence 607, Springer–Verlag, 1992, pp. 295–309.

[4] D. Basin and T. Walsh, A calculus for and termination of rippling, Journal of Automated Reasoning. 16(1–2), 1996, pp. 147–180.

[5] C. Benzmüller, L. Cheikhrouhou, D. Fehrer, A. Fiedler, X. Huang, M. Kerber, M. Kohlhase, A. Meier, E. Melis, W. Schaarschmidt, J. Siekmann and V. Sorge, Ωmega, Towards a mathematical assistant, in: *14th Conference on Automated Deduction.*, ed. W. McCune, Lecture Notes in Artificial Intelligence 1249, Springer–Verlag, 1997, pp. 252–255.

[6] R. Boulton, K. Slind, A. Bundy and M. Gordon, An interface between CLAM and HOL, in: *Proceedings of the 11th International Conference on Theorem Proving in Higher Order Logics*, eds. J. Grundy and M. Newey, Lecture Notes in Computer Science 1479, Springer, pp. 87–104.

[7] A. Bundy, The use of explicit plans to guide inductive proofs, in: *9th Conference on Automated Deduction*, eds. R. Lusk and R. Overbeek, 1988, pp. 111–120. Longer version available from Edinburgh as DAI Research Paper No. 349.

[8] A. Bundy, A. Stevens, F. van Harmelen, A. Ireland and A. Smaill, Rippling: A heuristic for guiding inductive proofs, Artificial Intelligence 62, 1993, pp. 185–253. Also available from Edinburgh as DAI Research Paper No. 567.

[9] A. Bundy, F. van Harmelen, C. Horn and A. Smaill, The Oyster-Clam system, in: *10th International Conference on Automated Deduction*, ed. M. E. Stickel, Lecture Notes in Artificial Intelligence 449, Springer–Verlag, 1990, pp. 647–648. Also available from Edinburgh as DAI Research Paper 507.

[10] A. Bundy, A. Smaill and J. Hesketh, Turning eureka steps into calculations in automatic program synthesis, in: *Proceedings of UK IT 90*, ed. S. L. H. Clarke, 1990, pp. 221–226. Also available from Edinburgh as DAI Research Paper 448.

[11] H. Chen, J. Hsiang and H.–C. Kong, On finite representations of infinite sequences of terms, in: *Proceedings of the 2nd International Workshop of Conditional and Typed Rewriting Systems*, ed. M. Okada, Lecture Notes in Computer Science 516, Springer-Verlag, 1990, pp. 100–114.

[12] R. Cleaveland, J. Parrow 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*, Lecture Notes in Computer Science 407, Springer–Verlag, 1989. Also available from Edinburgh, as ECS-LFCS-89-83.

[13] G. Collins, A proof tool for reasoning about functional programs. in: *9th International Conference of Theorem Proving in Higher Order Logics*, eds. J. von Wright, J. Grundy and J. Harrison, Lecture Notes in Computer Science 1125, Springer–Verlag, 1996, pp.109–124.

[14] L. Dennis, A. Bundy and I. Green, Using a generalisation critic to find bisimulations for coinductive proofs, in: *14th Conference on Automated Deduction*, ed. W. McCune, Lecture Notes in Artificial Intelligence 1249, Springer–Verlag, 1996, pp. 276–290.

[15] L. Dennis, *Proof Planning Coinduction*, 1998. Unpublished Ph.D. thesis, Edinburgh University.

[16] M. Fiore, A coinduction principle for recursive data types based on bisimulation, in: *Proceedings of the Eight IEEE Symposium on Logic in Computer Science*, 1993, pp. 110–119.

[17] J. Goguen, How to prove algebraic inductive hypotheses without induction, with applications to the correctness of data type implementation, in: *5th Conference on Automated Deduction*, eds. W. Bibel and R. Kowalski, Lecture Notes in Computer Science 87, pp. 356–373, Springer Verlag, 1980.

[18] J. Goguen, K. Lin and G. Rosu, Circular Coinductive Rewriting, in: *Proceedings, Automated Software Engineering (ASE)'00*, 2000. To Appear.

[19] 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 1, Elsevier, 1995.

[20] A. D. Gordon, Bisimilarity for a first-order calculus of objects with subtyping, in: *Proceedings, 23rd Symposium on Principles of Programming Languages*, ACM SIGPLAN-SIGACT, 1996, pp. 386–395.

[21] A. D. Gordon and L. Cardelli, Mobile ambients, in: *Proceedings FoSSaCS'98*, Lecture Notes in Computer Science 1578, Springer–Verlag, 1998. Full version to appear in Theoretical Computer Science.

[22] P. Hudak, S. Peyton–Jones, P. Wadler and et al., Report on the functional programming language Haskell: A non–strict, purely functional language version 1.2., in: *ACM SIGPLAN Notices* **27** (5), 1992.

[23] A. Ireland and A. Bundy, Productive use of failure in inductive proof, Journal of Automated Reasoning 16(1–2), 1996, pp. 79–111. Also available as DAI Research Paper No. 716, Dept. of Artificial Intelligence, Edinburgh.

[24] B. Jacobs and J. Rutten, A tutorial on (co)algebras and (co)induction, EATCS Bulletin 2, 1997, pp.222-259.

[25] D. Park, Fixpoint induction and proofs of program properties, in: Machine Intelligence 5, eds. D. Michie and B. Meltzer, 1970, pp. 59–78.

[26] L. C. Paulson, Co-induction and co-recursion in higher-order logic, Technical Report 304, University of Cambridge, Computer Laboratory, 1993.

[27] S. L. Peyton Jones. *The Implementation of Functional Programming Languages*, Prentice-Hall, 1987.

[28] U. S. Reddy. Narrowing as the Operational Semantics of Functional Languages, in: *Proc. of Second IEEE Int'l Symp. on Logic Programming* pp. 138–151. IEEE, New York, 1985.

[29] H. G. Rice, Classes of recursively enumerable sets and their decision problems, in: Transactions of the American Mathematical Society 89, 1953, pp .25–59.

[30] G. Roşu and J. Goguen, Circular Coinduction, UCSD Technical Report, CSE2000-064, 1999.

[31] J. Rutten, Universal coalgebra: A theory of systems, Technical Report CS–R9652, CWI, Amsterdam, 1996.

[32] M. Sheeran and G. Jones, *Circuit Design in Ruby*, North Holland, 1990.

[33] A. Tarski, A lattice-theoretical fixpoint theorem and its applications, Pacific Journal of Mathematics 5, 1955, pp. 285–309.

[34] T. Walsh, A divergence critic for inductive proof, Journal of Artificial Intelligence Research 4, 1996, pp. 209–235

Appendix

A. Proof of the Gfp Membership Rule of Inference

Theorem 13.

$$\frac{\begin{array}{l}\forall \mathcal{R}.\forall 1 \le i \le n.\bigwedge_{i=1}^{n}\langle a_i,\, b_i\rangle \in \mathcal{R} \Rightarrow \\ \qquad\qquad \forall \alpha.\exists a_i', b_i'.((a_i \xrightarrow{\alpha} a_i' \vee b_i \xrightarrow{\alpha} b_i') \Rightarrow \\ \qquad\qquad ((a_i \xrightarrow{\alpha} a_i' \wedge b_i \xrightarrow{\alpha} b_i')\wedge \\ \qquad\qquad \langle a_i',\, b_i'\rangle \in \mathcal{R}\cup \sim))\end{array}}{\bigcup_{i=1}^{n}\langle a_i,\, b_i\rangle \subseteq \langle \bigcup_{i=1}^{n}\langle a_i,\, b_i\rangle \cup \sim\rangle} \tag{54}$$

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 $[\cdots]$).

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 the definition of $\langle\cdots\rangle$). □

B. Results

This appendix lists the function definitions, lemmata and theorems used in the experimental evaluation of *CoCLAM*. p

B.1. Function Definitions

Name	Reduction Rules
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)	$nil <> L = L$
	$H :: T <> L = H :: (T <> L)$
$<>_l$	$nil <>_l nil = nil$
(append–lazy)	$nil <>_l H :: T = H :: (nil <>_l T)$
	$H :: T <>_l L = H :: (T <>_l L)$
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	$brswap(leaf(X)) = leaf(X)$
	$brswap(node(A, L, R)) = node(A, brswap(R), brswap(L))$

Name	Reduction Rules
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	$evenl(nil) = nil$ $evenl(H :: nil) = nil$ $evenl(H_1 :: H_2 :: T) = H_2 :: evenl(T)$
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	$inflist(X, F, G) = F(X) :: inflist(G(X), F, G)$
iter	$iter(F, M) = M :: iter(F, F(M))$
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	$map(F, nil) = nil$ $map(F, H :: T) = F(H) :: map(F, T)$

Name	Reduction Rules
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 tconst	$replace(A, B, nil) = nil$ $B \neq H \Rightarrow replace(A, B, H :: T) = H :: replace(A, B, T)$ $B = H \Rightarrow replace(A, B, H :: T) = A :: replace(A, B, T)$ $tconst(M) = node(M, tconst(M), tconst(M))$
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	$tswap2(A, B) = node(B, tswap2(B, A), tswap2(A, B))$

Name	Reduction Rules
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.2. 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.4.1 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 on the second pass through the sets. The theorem is either proved (in which case CPU Time is given) or is subject to one of the errors discussed in §8, i.e. false hypothesis not recognised (E1), memory error during evaluation (E2), implementation error (E3) or it timed out (usually due to the matching of $(\cdots)^N$ but sometimes because of inefficiencies in the implementation of the rippling method).

B.3. Development Set

Name	Theorem	Lemmata	Result
app1right	$X <> nil = X$		3841
appntconst	$\lambda l.node(A, L, L)^{\infty}(nil) = tconst(A)$		6360
assapp	$L <> (M <> N) = (L <> M) <> N$		15057
asslapp	$(X <>_l Y) <>_l Z = X <>_l (Y <>_l Z)$		36611
asslplus	$(M +_l N) +_l L = M +_l (N +_l L)$		28936
assp	$X + (Y + Z) = (X + Y) + Z$		11499
comapp	$length(X <> Y) = length(Y <> X)$		926734
commthree	$(Z *_l X) *_l Y = (Z *_l Y) *_l X$	disttwo tms2right	Time out
comp	$A + B = B + A$	ssid	340248
doublehalf	$double(half(N)) = N$		20897
dpbrtconst	$dpsearch(tconstinf(M), nil) =$ $brsearch(tconstinf(M), nil)$		21851
everylconst	$lconst(M) = evenl(lconst(M))$		4329
everylswap	$lconst(M) = evenl(lswap(M, N))$		4808
flattenex	$flattenA(explode(L)) = L$		62636
grmsthm	$loop2(F, A, B) = loop(F, A, B)$		E3
halfplus1	$half(X + X) = X$	463379	
hiterates	$h(F, X) = iter(F, X)$	21586	
infl1iter	$iter(\lambda y.A + y, B) =$ $inflist(0, \lambda x.(x * A) + B, s)$		Time out
infl1lc	$lconst(N) = inflist(M, idnat, idnat)$		24869
infl1nat	$inflist(A, idnat, s) = jump(A, 1)$		15614

Name	Theorem	Lemmata	Result
iterateslc	$lconst(M) = iter(idnat, M)$		23371
jumpfrom	$jump(N, 1) = from(N)$		5535
lappnilr	$L <>_l nil = L$		4770
lendouble	$length(X <> X) = double(length(X))$		102933
lenlconst	$length(M :: lconst(M)) = \infty$		3852
lenplus	$length(X <> Y) = length(X) + length(Y)$		839082
lplus0l	$0 +_l N = N$		4130
lminuspl	$(Z + X) -_l (Z + Y) = X -_l Y$		Time out
lminussc	$X -_l Y = s(X) -_l s(Y)$		7356
lswaplc	$A \neq B$ $\Rightarrow del(B, lconst(A)) = del(B, lswap(A, B))$		29250
lswaplm	$merge(lconst(A), lconst(B)) = lswap(A, B)$		17750
map3iter	$map(*(s(s(s(0)))), jump(0, s(0))) =$ $iter(+(s(s(s(0)))), 0)$		Time out
mapapp	$map3(W, X <> Y :: Z) =$ $map3(W, X) <> map3(W, Y :: nil <> Z)$		17730
mapidnat	$lconst(M) = map(idnat, lconst(M))$		14448
mapdble	$map(double, L) = map(\lambda X + X, L)$		68616
mapfinfl1	$map(H, iter(T, A)) = inflist(A, H, T)$		6843
mapflip	$map(flip_{bv}, map(flip_{bv}, L)) = L$		18777
mapfold	$map(F, L) =$ $foldr(\lambda x.\lambda t.F(x) :: t, L, nil)$		9306
mapid	$map(L, idnat) = L$		15505
mapiter	$iter(F, F(M)) =$ $map(F, iter(F, M))$		7083
mapiter2	$iter(F, X) =$ $X :: map(F, iter(F, X))$		16302
mapjump	$map(lconst, jump(0, 1)) =$ $iter(map(s), lconst(0))$		38131
mapthm	$map(F, map(G, L)) = map(F \circ G, L)$	cmpfun	10331
mergezz	$merge(merge(oddl(A), oddl(B)),$ $merge(evenl(A), evenl(B))) =$ $zigzag(nil, A :: nil, B :: nil)$		E2
nat1	$iter(s, s(s(0))) = jump(s(s(0)), s(0))$		23764
nat2	$jump(0, s(0)) =$ $merge(jump(0, s(s(0))), jump(s(0), s(s(0))))$		163129
oneslc	$ones = lconst(s(0))$		4192
parity0	$parity(T, L) = numparity(0, T)$	evlem oddlem evlem2 oddlem2	117864
plus2right	$X + s(Y) = s(X + Y)$		6901
pluslem2	$X + s(0) = s(X)$		5231
plusxx	$X + s(X) = s(X + X)$	ssid	494681
tbrswap	$tswap(A, B) = brswap(tswap2(B, A))$		6852
zeroplus	$X = 0 \wedge Y = 0 \Rightarrow X + Y = 0$		7991

B.3.1. Test Set

Name	Theorem	Lemmata	Result
appatend	$X <> Y :: Z = atend(Y, X) <> Z$		11952
appiter	$iter(F, X) <> N = iter(F, X)$		6984
applapp	$L_1 <> L_2 = L_1 <>_l L_2$		26969
appnlconst	$:: (N)^{\infty}(nil) = lconst(N)$		6142
assconsapp	$P <> V_0 :: L = (P <> V_0 :: nil) <> L$		12659
assm	$A * (B * C) = (A * B) * C$		Time out
brswap	$T = brswap(brswap(T))$		6086
comaddl	$addl(M, N) = addl(N, M)$		543042
comlplus	$M +_l N = N +_l M$		462749
comm	$X * Y = Y * X$		Time out
dist	$A * (B + C) = A * B + A * C$		Time out
disttwo	$(B + C) * A = (B * A) + (C * A)$		Time out
dbleplus	$double(X) = X + X$		42689
dbletimes1	$double(N) = s(s(0)) * N$		289416
dbletimes2	$double(N) = N * s(s(0))$		19853
dpsearchlc	$dpsearch(node(M, lconst(node(M, nil))), nil)$ $= lconst(M)$		16246
dptconst	$dpsearch(tconstinf(M), nil) = lconst(M)$		4578
dupinf	$dup(M, \infty) = lconst(M)$		3974
flatfold	$flattenA(map(\lambda a.map(\lambda b.a :: b :: nil),$ $iter(F, N), lconst(M))) =$ $foldr(\lambda x.\lambda p.foldr(\lambda y.\lambda l.(x :: y :: nil) :: l,$ $iter(F, N), p), lconst(M), nil)$	Time out	
gordon1	$map(\lambda x.idnat(x), Y) = idlist(Y)$		17199
halfdble	$half(double(N)) = N$		4785
halflapp1	$half(length(A <> B)) =$ $half(length(B <> A))$	ssid	Time out
halfplus2	$half(X + Y) = half(Y + X)$	ssid	8899600
identrm	$X * s(0) = X$		4204
infl1lswap	$lswap(0, 1) = inflist(0, idnat, flip)$		15284
lappnill	$nil <>_l L = L$		3995
lconsta	$map(F, lconst(M)) = lconst(F(M))$		4884
lconstaddl	$addl(lconst(M), lconst(N)) =$ $lconst(M + N)$		5174
lconstapp	$lconst(M) = lconst(M) <> L$		4538
lconsteven	$lconst(T) = map(even, jump(0, s(s(0))))$	evlem	18986
lconstiter	$lconst(iter(F, F(M))) =$ $lconst(map(F, iter(F, M)))$		17608
lconstzz	$lconst(N) = zigzag(lconst(N) :: nil, nil, nil)$		16140
lenlcit	$length(lconst(M)) = length(iter(F, N))$		4924
lenlconst	$\infty = length(lconst(M))$		3603
lenmapcar	$length(map(F, L)) = length(L)$		6619
lplus0r	$N +_l 0 = N$		3861
ltms2right	$X *_l s(Y) = X +_l X *_l Y$		77335
mergedrop	$merge(merge(evenl(A), evenl(B)),$ $drop(s(s(0)), merge(oddl(A), oddl(B)))) =$ $drop(s(s(0)), zigzag(A :: B :: nil, nil, nil))$		E2
mergeolel	$merge(oddl(L), evenl(L)) = L$		48124
minusp	$pred(X -_l Y) = pred(X) -_l Y$		34936

Name	Theorem	Lemmata	Result
map2thm	$map2(F, map2(G, L)) = map2(F \circ G, L)$		31843
mapaddl	$addl(L, L) = map(double, L)$		88834
mapcarapp	$map(F, L_1 <> L_2) =$ $map(F, L_1) <> map(F, L_2)$		16732
maplconst	$map(\lambda m.L <> m, lconst(M)) =$ $lconst(L <> M)$		6747
natmap	$jump(0, s(0)) = 0 :: map(s, jump(0, s(0)))$		34637
parityeven	$even(N) \Rightarrow parity(T, L) = numparity(N, L)$		43767
pluslplus	$A +_l B = B + A$		246904
plus1right	$X + 0 = X$		3260
replacels	$replace(A, B, lswap(A, B)) = lconst(A)$		19930
tconstts	$tconst(M) = tswap(M, M)$		4090
ticktock	$tick = map(\mathit{flip}, tock)$		11172
tms2right	$X * s(Y) = X + X * Y$		175642
timesltimes	$X * Y = X *_l Y$		Time out
zerotimes2	$X = 0 \vee Y = 0 \Rightarrow X * Y = 0$		E1

B.4. Lemmata

B.4.1. 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.

Name	Lemmata
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	$s^0(X) = X$ $s^{s(N)}(X) = s(s^N(X))$

Name	Lemmata
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. etc. However this does highlight the need for a lemma speculation critic where accurate lemmata could be speculated as needed rather than having to be supplied by hand.

B.4.2. Other Lemmata

Name	Lemma	Name	Lemma
cmpfun	$\langle F(G(X)),\, F \circ G(X)\rangle \in\sim$	evlem	$\langle T,\, even((s \circ s)^N(0))\rangle \in\sim$
evlem2	$\langle \bot,\, even(s((s \circ s)^N(0)))\rangle \in\sim$	oddlem	$\langle \bot,\, odd((s \circ s)^N(0))\rangle \in\sim$
oddlem2	$\langle T,\, odd(s((s \circ s)^N(0)))\rangle \in\sim$	ssid	$s(s(X)) = s(s(X))$