CycleQ: An Efficient Basis for Cyclic Equational Reasoning

Eddie Jones
Department of Computer Science
University of Bristol
Bristol, United Kingdom

C.-H. Luke Ong
Department of Computer Science
University of Oxford
Oxford, United Kingdom

Steven Ramsay
Department of Computer Science
University of Bristol
Bristol, United Kingdom

Abstract

We propose a new cyclic proof system for automated, equational reasoning about the behaviour of pure functional programs. The key to the system is the way in which cyclic proofs and equational reasoning are mediated by the use of contextual substitution as a cut rule. We show that our system, although simple, already subsumes several of the approaches to implicit induction variously known as "inductionless induction", "rewriting induction", and "proof by consistency". By restricting the form of the traces, we show that global correctness in our system can be verified incrementally, taking advantage of the well-known size-change principle, which leads to an efficient implementation of proof search. Our CycleQ tool, implemented as a GHC plugin, shows promising results on a number of standard benchmarks.

CCS Concepts: • **Theory of computation → Equational logic and rewriting**; **Logic and verification**.

Keywords: Cyclic proofs, equational reasoning, rewriting induction, inductionless induction.

ACM Reference Format:
Eddie Jones, C.-H. Luke Ong, and Steven Ramsay. 2022. CycleQ: An Efficient Basis for Cyclic Equational Reasoning. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '22), June 13–17, 2022, San Diego, CA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3519939.3523731

1 Introduction

An advantage of pure functional programming is the ease with which one can reason about the behaviour of programs. Interesting properties can often be proven using only a combination of induction and equational reasoning.

However, as is well known, inductive theorem proving is challenging. The incompleteness of typical inductive theories and the non-analyticity of their induction rules excludes a general algorithmic solution [12]. Moreover, even if we restrict our attention to what we might loosely imagine are the cases we care about, namely those functional programs that occur in practice, the situation is still extremely complex.

Functional programmers employ a variety of inductive and mutually inductive datatypes and rarely restrict themselves to functions defined by structured recursion schemes. Hence, not only do we need induction principles for each datatype, but we should expect that these schemes can be nested, combined for mutual induction, or generalised to account for non-structural recursion.

Despite this, there are already several tools that have shown success at automatically proving equational properties of functional programs. However, to the best of our knowledge, none have a smooth treatment of the more complicated induction schemes that are frequently required in practice. For example, proofs that require mutual induction are not supported by default in either HipSpec [14], IsaPlanner [20] or Zeno [45], and reasoning about mutually recursive functions is described in the ACL2 manual as being "a bit awkward" [32]. Moreover, in several of these tools, mutually inductive datatypes are simply not supported.

Using induction schemes that are tailored to specific conjectures is important; although automatic lemma discovery techniques can sometimes compensate, they have a number of weaknesses such as limited applicability, over generalisation, and scalability for complex formulas [26]. Hence, although lemma discovery is crucial in all but the simplest inductive proofs, any improvement to the underlying inductive proof system, reducing the burden on lemma generation heuristics, is worthwhile.

In this paper, we propose a novel *cyclic proof system for equational reasoning* and an accompanying algorithm for efficient proof search, which we have implemented as a plugin for GHC — CycleQ. The system seamlessly supports complex forms of inductive argument, such as nested or mutual induction, and is agnostic about lemma discovery techniques (which we leave aside as an orthogonal concern).

PLDI '22, June 13–17, 2022, San Diego, CA, USA

ACM ISBN 978-1-4503-9265-5/22/06…$15.00
https://doi.org/10.1145/3519939.3523731

$$\dfrac{\dfrac{}{n \doteq n} \quad \dfrac{\dfrac{}{\mathsf{Var}\ v \doteq \mathsf{Var}\ v}}{\mathsf{mapT}\ \mathsf{id}\ (\mathsf{Var}\ v) \doteq (\mathsf{Var}\ v)} \quad \dfrac{}{\mathsf{Cst}\ c \doteq \mathsf{Cst}\ c} \quad \dfrac{\dfrac{(0)\quad \overline{e_1 \doteq e_1}}{\mathsf{mapE}\ \mathsf{id}\ e_1 \doteq e_1} \quad \dfrac{(0)\quad \overline{e_2 \doteq e_2}}{\mathsf{mapE}\ \mathsf{id}\ e_2 \doteq e_2}}{\mathsf{App}\ (\mathsf{mapE}\ \mathsf{id}\ e_1)\ (\mathsf{mapE}\ \mathsf{id}\ e_2) \doteq \mathsf{App}\ e_1\ e_2}}{\mathsf{mapT}\ \mathsf{id}\ (\mathsf{Cst}\ c) \doteq (\mathsf{Cst}\ c) \qquad \mathsf{mapT}\ \mathsf{id}\ (\mathsf{App}\ e_1\ e_2) \doteq \mathsf{App}\ e_1\ e_2}}{\mathsf{mapT}\ \mathsf{id}\ t \doteq t}$$

$$\dfrac{\mathsf{MkE}\ (\mathsf{mapT}\ \mathsf{id}\ t)\ n \doteq \mathsf{MkE}\ t\ n}{\mathsf{mapE}\ \mathsf{id}\ (\mathsf{MkE}\ t\ n) \doteq \mathsf{MkE}\ t\ n}$$

$$0\colon \mathsf{mapE}\ \mathsf{id}\ e \doteq e$$

Figure 1. A cyclic proof of $\mathsf{mapE}\ \mathsf{id}\ e \doteq e$.

1.1 Cyclic Proofs and Equational Reasoning

Cyclic proofs occupy a part of non-wellfounded proof theory, in which infinite proof trees are required to be regular, e.g. representable as a finite graph [6, 46]. Unsound arguments are excluded by requiring a *global correctness condition* on the infinite paths, such as inclusion in a particular ω-regular language. The regularity restriction makes cyclic proofs quite well behaved, and there has recently been a number of works exploring the theoretical and practical advantages of this form of circular reasoning [6, 8, 9, 16–18, 25, 34, 47, 49, 52].

On the practical side, one of the major driving forces has been the potential to improve state-of-the-art automated reasoning. Cyclic proof systems appear to better capture the exploratory nature of goal-directed proof search, especially with respect to "inductive" reasoning. A particular advantage is the ability to avoid committing to either a fixed menu of induction schemes or a fixed choice of induction variables in advance. Rather, systems can justify a circular argument post hoc through an appeal to infinite descent bespoke to the proof structure discovered by the search.

For example, consider a mutually inductive definition of two types comprising annotated syntax trees[1] in Haskell:

```
data Term a          data Expr a
  = Var a              = MkE (Term a) Nat
  | Cst Nat
  | App (Expr a) (Expr a)
```

We can define two functions: mapT and mapE, that express the functoriality of these type constructors and conjecture that the relevant laws hold. Fig. 1 shows the cyclic proof obtained by our system for the identity law for mapE. Here, and elsewhere, the cycle is presented by labelling a node in the proof tree with a number, e.g. labelling the root 0, and using this label elsewhere as a premise without further justification. Equations are given using the symbol $\doteq$ to emphasize that they are regarded as unordered (i.e. the left- and right-hand sides are interchangeable). See Example 3.2 and the following remark for a fuller description of this notation.

[1] A typical annotation is the line and column numbers marking their provenance in some source code, but here we use a single natural number for simplicity.

Without a proper treatment of mutual induction, an inductive theorem prover would have to guess, heuristically, a strengthening of the inductive property, e.g. adding the conjunct $\mathsf{mapT}\ \mathsf{id}\ t \doteq t$ to the original goal. In our cyclic system, however, the two cycles depicted using label 0 fall out naturally from equational reasoning, and the fact that each involves a decrease, i.e. that the proof is globally correct, is easily verified.

A key consideration in the design of an automated cyclic proof system is how to control the formation of cycles in the proof. Technically, it is sound to form cycles whenever proof search discovers a node of the proof tree that is logically stronger than an ancestor and for which the newly formed cycle would satisfy the global correctness condition. Since, in general, we cannot expect there to be any syntactical relationship between the node and its ancestor, the formation of cycles is closely related to the use of cuts in the proof. Indeed Tsukada and Unno [52] have demonstrated that many techniques developed for efficient software model checking can be viewed as the introduction of cuts into cyclic proofs to discharge proof obligations earlier.

In Brotherston, Gorogiannis and Petersen's state-of-the-art Cyclist$_{\mathrm{FO}}$ prover, cycles are formed by a restricted kind of cut in which the node follows from its ancestor by a combination of weakening and instantiation [9]. However, the authors note that the lack of a more general cut rule and the lack of native support for equational reasoning causes their system to have difficulty with heavily-equational goals, such as the commutativity of addition: $x + y = y + x$. They conjecture that a cyclic proof could be obtained if the lemma $x + \mathsf{S}\ y = \mathsf{S}\ (x + y)$ were supplied as a hint.

In fact, the cyclic proof system that we develop in this paper can prove the commutativity of addition automatically, without any externally supplied lemma such as the one above. The synthesized proof is given in Fig. 4, but we defer its discussion until later.

Our system consists of four rules: the reflexivity of equality, evaluation of program expressions, reasoning by cases, and substitution of equals for equals. The key is the way that cyclic proof and equational reasoning are mediated through

$$\dfrac{\dfrac{}{\texttt{Nil} \doteq \texttt{Nil}} \quad \dfrac{\dfrac{}{y \doteq y} \quad \dfrac{(0)\quad \texttt{take}\ (\texttt{len}\ zs)\ (\texttt{Cons}\ z\ zs) \doteq \texttt{take}\ (\texttt{len}\ zs)\ (\texttt{Cons}\ z\ zs)}{\texttt{butLast}\ (\texttt{Cons}\ z\ zs) \doteq \texttt{take}\ (\texttt{len}\ zs)\ (\texttt{Cons}\ z\ zs)}}{\texttt{Cons}\ y\ (\texttt{butLast}\ (\texttt{Cons}\ z\ zs)) \doteq \texttt{Cons}\ y\ (\texttt{take}\ (\texttt{len}\ zs)\ (\texttt{Cons}\ z\ zs))}}{0\colon \texttt{butLast}\ (\texttt{Cons}\ y\ ys) \doteq \texttt{take}\ (\texttt{len}\ ys)\ (\texttt{Cons}\ y\ ys)}$$

$$\dfrac{\dfrac{}{\texttt{Nil} \doteq \texttt{Nil}} \quad \dfrac{\dfrac{}{x \doteq x}\quad (0)}{\texttt{Cons}\ x\ (\texttt{butLast}\ (\texttt{Cons}\ y\ ys)) \doteq \texttt{Cons}\ x\ (\texttt{take}\ (\texttt{len}\ ys)\ (\texttt{Cons}\ y\ ys))}}{\texttt{butLast}\ (\texttt{Cons}\ x\ xs) \doteq \texttt{take}\ (\texttt{len}\ xs)\ (\texttt{Cons}\ x\ xs)}}{\texttt{butLast}\ xs \doteq \texttt{take}\ (\texttt{len}\ xs - \texttt{S}\ \texttt{Z})\ xs}$$

Figure 2. A cyclic proof of $\texttt{butLast}\ xs \doteq \texttt{take}\ (\texttt{len}\ xs - \texttt{S}\ \texttt{Z}) \doteq xs$.

the use of (contextual) substitution as a cut rule.

$$(\text{Subst})\ \dfrac{M \doteq N \quad C[N\theta] \doteq P}{C[M\theta] \doteq P}$$

We refer to the left-hand premise of this rule as *the lemma* and the right-hand premise as *the continuation*. This rule says that given a lemma $M \doteq N$ and a goal $C[M\theta] \doteq P$ containing an instance of M, the proof can be continued by solving $C[N\theta] \doteq P$ in which the instance of M has been replaced by a matching instance of N.

Although, in principle, the choice of equation comprising the lemma may be completely unrelated to the rest of the proof tree (e.g. it may be supplied by a human or conjectured by a theory exploration tool), our proof search algorithm is able to synthesize proofs for 61% of the relevant problems from the IsaPlanner benchmark suite whilst only choosing lemmas $M \doteq N$ that already occur as nodes within the same proof tree, i.e. without the need to invoke any potentially costly lemma discovery technology. For example, our system can prove $\texttt{butLast}\ xs \doteq \texttt{take}\ (\texttt{len}\ xs - \texttt{S}\ \texttt{Z}) \doteq xs$ in ~40 ms. A proof can be seen in Fig. 2. By comparison, HipSpec fails to prove the same result after ~40 s, an attempt that involved 22 synthesised lemmas, 12 of which failed [42].

The substitution rule can be seen in the two rule applications of Fig. 1 that have (0) as a premise. Here, the lemma is chosen to be the node labelled 0 at the root of the tree and the continuation is simply discharged by reflexivity. The usage in Fig. 2 is similar. In the proof of the commutativity of addition Fig. 4, the continuation labelled (2) is much more complex and contains a nested inductive argument.

1.2 Simulation of Inductionless Induction

It is well known that cyclic proof systems can already simulate explicit structural induction schemes, and we additionally show that our system subsumes various kinds of implicit induction based on Knuth-Bendix completion, such as "inductionless induction" and "proof by consistency", that were intensively studied in the 1980s and 1990s, e.g. [10, 19, 21, 23, 29–31, 39].

On the surface, these approaches seem quite distinct from cyclic proofs; rather than proving a conjecture by induction, they posit it as an axiom and attempt to show that the resulting theory is consistent. In order to connect the approaches, we use *term rewriting induction*, in the sense of Reddy [40], as a stepping stone, which is already known to subsume proof by consistency. The key is to observe that the unconstrained use of hypotheses in Reddy's system gives rise to the structure of cyclic preproofs and that global correctness is guaranteed by construction as progress proceeds by rewriting and equations are orientated by a fixed (well-founded) reduction order.

Term rewriting approaches to induction share some of the advantages of cyclic proofs. They support mutual induction, for example, and do not require a fixed induction scheme in advance. However, our analysis also highlights a disadvantage by comparison with our cyclic system: rewriting approaches require that any equations discovered by the proof search be orientable with respect to the fixed reduction order. Systems are not only very sensitive to the choice of the order in practice, but this requirement also precludes theorems like the above commutativity of addition, the symmetry of which is inherently unorientable. For a critique see the 1988 POPL paper of Garland and Guttag [22].

1.3 The CycleQ Theorem Prover

Since our proof system is quite simple, it is straightforwardly amenable to a goal-directed proof-search algorithm. However, a naïve implementation will quickly run into performance difficulties.

One source is in the number of nodes that are candidates for cycles. As mentioned previously, the formation of cycles is enabled by the (Subst) rule restricted to employ only existing nodes of the current proof tree as the lemma $M \doteq N$. However, the number of eligible lemmas to consider will consequently grow with the size of the proof.

The number of eligible lemmas can be drastically reduced by using a number of further restrictions motivated by redundancies we identify in the structure of proofs. For example, if a lemma is itself justified by the (Subst) rule, we can

use its premise directly as contexts and substitutions are composable.

Another bottleneck is in the verification of the global correctness condition. This source of inefficiency was already identified for the Cyclist prover, where a large proportion of the overall proof time is spent checking the global correctness of proof trees that turn out to be unsound [47]. In this work, we avoid a similar problem by restricting our attention to variable-based traces and exploiting the incremental nature of goal-directed proof search. We annotate the proof graph with an abstract domain representing the ω-regular language of paths — Lee, Jones and Ben-Amram's size-change graphs [35]. Encoding the information directly in the proof graph allows the global correctness argument to be updated as each node is uncovered, and thus, unsound cycles are represented explicitly so that proof search can be terminated as soon as one is detected. Furthermore, there is no recomputation of soundness for shared proof fragments.

We implemented our approach as a plugin for GHC called CycleQ. It currently supports a small subset of Haskell, including top-level recursive functions, algebraic datatypes, and polymorphism. Our evaluation on a number of benchmarks shows that it performs well on standard and mutual induction problems and can sometimes prove more complex goals that would typically require lemmas in other systems.

Contributions. Our main contributions are as follows:

- We identify contextual substitution as the appropriate means for introducing cycles into an equational proof, presenting a simple proof system based on this mechanism.
- We show that, when targeting proofs about functional programs, our system subsumes approaches to implicit induction, known variously as "inductionless induction", "proof by consistency" and "rewriting induction".
- We show that, by restricting to variable traces, the global correctness condition of cyclic proof reduces to Lee, Jones and Ben-Amram's size-change principle. This approach leads directly to an efficient and *incremental* procedure for detecting and verifying cycles based on size-change graphs.
- We identify several natural restrictions on contextual substitution that allow it to play the role of an efficient matching function for detecting potential cycles. Our evaluation on a number of benchmarks shows that it performs well on standard and mutual induction problems and can sometimes prove more complex goals that would typically require lemmas in other systems.

Outline. The remainder of the paper is structured as follows. In Section 2, we introduce necessary preliminaries and, in Section 3, we present our simple cyclic proof system for equational reasoning. In Section 4, we show that the system already subsumes Reddy's system of rewriting induction. In Section 5, we develop the heuristics necessary for making the formation of cycles efficient, and we show that size-change termination can be used to enable an incremental approach to checking the global correctness condition. A description of our implementation and its evaluation comprise Section 6, and we conclude in Section 7 with a discussion of related work.

2 Preliminaries

For the purpose of this formalism, we will consider a higher-order rewriting system and its induced equational theory. Although the intended application of our work is functional programs, this setting is more general and facilitates direct comparison with rewriting induction (Section 4).

In this section, we will cover some definitions from term rewriting used throughout the paper.

We assume a fixed ***signature*** consisting of a finite set of algebraic datatypes D and function symbols Σ.

For the types of our formal system, we use simple types built over D, i.e. $\tau, \sigma \coloneqq d \in D \mid \tau \to \sigma$. The *order* of a type is defined as follows:

$$\mathrm{ord}(d) \coloneqq 0$$
$$\mathrm{ord}(\tau \to \sigma) \coloneqq \max\{\mathrm{ord}(\tau) + 1,\ \mathrm{ord}(\sigma)\}$$

Each function symbol is assigned a type, written $f : \tau \in \Sigma$. Furthermore, function symbols are partitioned into a set of ***constructors*** Σ_{con} (e.g. `Cons`, `Nil`, `Zero`, `Succ`), which are required to be at most first-order, and ***defined functions*** Σ_{def} (e.g. `map`, `add`). We write $\Sigma_{\mathrm{con}}(d)$ for the set of constructors whose return type is d.

Terms are generated from application, function symbols Σ, and variables drawn from some countable set.

$$M, N ::= x \mid f \in \Sigma \mid M\ N$$

As usual, we associate applications to the left.

A ***type environment***, typically Γ or Δ, is a set of variable-type pairs, written $x : \tau$. We will write Γ, Δ (or $\Gamma, x_0 : \tau_0, \ldots, x_n : \tau_n$) for the disjoint union of two environments. The judgement $\Gamma \vdash M : \tau$, defined by usual typing rules, asserts that M is a well-typed term of type τ for the environment Γ.

In what follows, we will restrict our attention to well-typed terms, but we omit the rules for simple typing, which are standard.

Terms give rise to a natural set of ***(one-hole) contexts***, generically written $C[\cdot]$, which we define as:

$$C[\cdot] ::= \cdot \mid C[\cdot]\ M \mid M\ C[\cdot]$$

where M ranges over terms. We write $C \circ D$ for their composition, where $(C \circ D)[X] \coloneqq C[D[X]]$ for all terms X.

A term M is ***subterm*** of N, written $M \trianglelefteq N$, if there exists a context C such that $C[M] = N$. When the witness C is non-trivial, i.e. not $\cdot$, we write $M \triangleleft N$.

Lemma 2.1. *$\trianglelefteq$ is a well-founded, partial order.*

Lemma 2.2. *The relation on contexts $D \sqsubseteq C$ defined as the existence of some context E such that $C = D \circ E$ is a partial order. Furthermore, if two unrelated contexts C and D are equal for terms M and N, i.e. $C[M] = D[N]$, then $M \trianglelefteq D[X]$ for any term X.*

Substitutions, typically θ, are partial functions from variables to terms with the usual action $M\theta$ on terms M. We write $\theta_1 \circ \theta_0$ for the composition of substitutions, defined as $x \mapsto (\theta_0(x))\theta_1$.

A ***stable order*** on terms $\leq$ is a partial order such that $M\theta \leq N\theta$ follows from $M \leq N$ for any substitution θ.

A ***rewrite rule*** is a pair of terms, written $M \to N$ such that M is of the form $f\ M_0\ \cdots\ M_n$ where $f \in \Sigma_{\text{def}}$, each M_i doesn't contain any defined function symbols, and both $\Gamma \vdash M : d$ and $\Gamma \vdash N : d$ for some type environment Γ and a datatype d.

For a set of rewrite rules R, we define the ***one-step reduction*** as $C[M\theta] \to_R C[N\theta]$ whenever $M \to N \in R$. We write $M \to_R^* N$ for the reflexive-transitive closure of this relation.

A term M is in R-***normal form*** when there does not exist a term N such that $M \to_R N$. We write $M{\downarrow_R}$ for the term N that is a normal form such that $M \to_R^* N$.

Remark 2.1 (Assumptions). We will assume some fixed set of rules R such that the induced relation $\to_R^*$ is:

- *Complete*, in the sense that, no closed first-order term headed by a defined function symbol is in normal form. That is, for any term $\emptyset \vdash f\ M_0\ \cdots\ M_n : d$ with $f \in \Sigma_{\text{def}}$, there exists some N such that $f\ M_0\ \cdots\ M_n \to_R N$.
- And both weakly normalising and confluent so that $[\cdot]{\downarrow_R}$ is a well-defined function on terms.

It is easy to ensure that the rewriting system corresponding to a functional program is complete and is often guaranteed by compilers. Pure functional programs are also confluent. On the other hand, the assumption that the program is weakly normalising is not without loss of generality. However, it has been observed that problems of non-termination are relatively rare in comparison to those of functional correctness. It is also worth noting that although undecidable, practical algorithms exist for verifying this property.

Example 2.1. The reduction relation $\to_R$ induced by the following program clearly satisfies the assumptions of Remark 2.1.

```
add Zero     y = y
add (Succ x) y = Succ (add x y)

map f Nil         = Nil
map f (Cons x xs) = Cons (f x) (map f xs)
```

An ***equation***, generically ϕ or ψ, is an *unordered* pair of terms M and N such that $\Gamma \vdash M, N : d$ for a type environment Γ and datatype d. Equations are written as $\Gamma \vdash M \doteq N$, or equivalently $\Gamma \vdash N \doteq M$. When clear from the context, we will omit the type enironment.

A ***(ground) instance*** of an equation $\Gamma \vdash M \doteq N$ is a subsitution α, such that, $\emptyset \vdash \alpha(x) : \tau$ for all $x : \tau \in \Gamma$. An equation $\Gamma \vdash M \doteq N$ is ***satisfied*** by an instance, written $\alpha \vDash M \doteq N$, if $M\alpha{\downarrow_R} = N\alpha{\downarrow_R}$. If it is satisfied by all such instances, then we say it is ***valid*** and write $\vDash M \doteq N$.

Note that the satisfaction relation, $\cdot \vDash \cdot$, and by extension validity, is well-defined as normalisation is a function and syntactic equality is a symmetric relation.

3 Cyclic Proofs

An infinitary proof generalises traditional finite derivation trees to possibly infinite ones. Such proofs are not necessarily sound; the standard approach is, therefore, to first define *preproofs*, which are later refined by a global condition to ensure the argument is well-founded [7].

Cyclic proofs are a subclass of infinitary proofs whose derivation trees are regular, i.e. there are only finitely many distinct subtrees. Such proofs can be represented as finite but incomplete derivation trees where unjustified premises called "buds" refer to other vertices called "companions" [6]. We will, however, present the cycles of a preproof directly.

Definition 3.1. A ***(cyclic) preproof*** is a tuple $P = (V, e, r, p)$ where V is a finite set of vertices, typically an initial segment of the natural numbers, such that, for each vertex $v \in V$:

- There is an associated equation $e(v)$, inference rule from Fig. 3 $r(v)$, and a finite sequence of vertices $p(v) \in V^*$ called the premises. We write $p_i(v)$ for the i^{th} element of $p(v)$ starting with p_0
- And
$$\frac{e(p_0(v)) \quad \dots \quad e(p_n(v))}{e(v)}$$
is a well-formed instance of the rule $r(v)$.

The ***underlying graph*** of a preproof $P = (V, e, r, p)$ is a graph $G(P) = (V, E)$, over the same set of vertices, where:

$$E := \{(v, p_i(v)) \mid v \in V,\ p_i(v) \text{ is defined}\}$$

Note that when a premise appears as part of a cycle, it needn't be a direct ancestor. In particular, a cousin node may be used as a lemma by the (Subst) rule.

Remark 3.1 (Rules in Fig. 3 defining equational preproofs).

1. The rules are named according to their goal-orientated use. Hence (Reduce) refers to the reduction of a goal to the premise.
2. In this light, we will refer to the left- and right-hand premises the (Subst) rule as the ***lemma*** and ***continuation*** respectively. The reason behind this convention will later become apparent when discussing our proof search algorithm Section 6.
3. As the usual rules of transitivity, congruence, and instantiation are instances of (Subst), they are trivially

$$\text{(Refl)}\ \frac{}{\Gamma \vdash M \doteq M} \qquad \text{(Reduce)}\ \frac{\Gamma \vdash M' \doteq N'}{\Gamma \vdash M \doteq N}\ (M \to_R^* M',\ N \to_R^* N')$$

$$\text{(Subst)}\ \frac{\Delta \vdash M \doteq N \quad \Gamma \vdash C[N\theta] \doteq P}{\Gamma \vdash C[M\theta] \doteq P} \qquad \text{(Case)}\ \frac{\forall k \in \Sigma_{\text{cons}}(d) \quad \Gamma, \Delta \vdash M[k\ x_0\ \cdots\ x_n/x] \doteq N[k\ x_0\ \cdots\ x_n/x]}{\Gamma,\ x : d \vdash M \doteq N}$$

Figure 3. The inference rules for preproofs

derivable. Symmetry follows immediately from the use of unordered equations. In particular, any combination of these rules can be used to form cycles.

4. In the rule (Case), there are as many equations in the premises as there are constructors of the datatype d.

A trivial example of a preproof can be constructed by using substitution to rewrite any equation according to itself, thus assuming the exact equation which is to be proved.

Example 3.2. Let $V = \{0, 1\}$, let $e(0)$ and $e(1)$ be the equations $\texttt{Cons}\ x\ xs \doteq \texttt{Nil}$ and $\texttt{Nil} \doteq \texttt{Nil}$, let $r(0)$ and $r(1)$ be the rules (Subst) and (Refl), and let $p(0) = [0, 1]$ and $p(1) = []$. Then (V, e, r, p) is a preproof satisfying Definition 3.1.

Remark 3.2 (Representing preproofs).

1. Here, and in what follows, we will depict preproofs as a set of finite trees with labelled vertices and "back edges" that reference those labels. For example, the preproof of Example 3.2 would be presented as follows:

$$\text{(Subst)}\ \frac{(0) \qquad \text{(Refl)}\ \dfrac{}{\texttt{Nil} \doteq \texttt{Nil}}}{0\text{:}\ \texttt{Cons}\ x\ xs \doteq \texttt{Nil}}$$

2. To keep proofs compact, we shall also omit vertices justified by (Reduce).

Although this example clearly illustrates that preproofs are not necessarily sound arguments, they are, however, *locally* sound in the sense that the premises of an inference rule justify its conclusion. This property is witnessed by relating instances of a vertex to those of its premises, which is sufficient for concluding that the vertex's equation is satisfied for that instance.

Definition 3.3. Let (V, e, r, p) be a cyclic preproof with vertex $v \in V$. Then for any instance of $e(v)$, α, a ***preceding instance*** is a pair (i, β), where $p_i(v)$ is a premise and β is an instance of $e(p_i(v))$ such that one of the following conditions is met depending on the rule $r(v)$:

- (Case) where $x : d$ is the variable upon which case analysis is performed. In this case, if $(\alpha(x))\downarrow_R$ is of the form $k\ M_0\ \cdots\ M_n$ and $p_i(v)$ is the premise associated with the constructor k using fresh variables $x_0, \ldots, x_n$, then (i, β) is a preceding instance where β is defined as follows:

$$\beta(y) \coloneqq \begin{cases} M_i & y = x_i \\ \alpha(y) & y \neq x \end{cases}$$

- (Subst) with substitution θ. In this case, $(0, \alpha \circ \theta)$ and $(1, \alpha)$ are preceding instances for the lemma and continuation respectively.
- Otherwise, there is a unique premise, for which $(0, \alpha)$ is a preceding instance.

The following lemma states the important property that preceding instances must witness — the contrapositive of local soundness (i.e. from an invalid conclusion, one can derive an invalid premise).

Lemma 3.1. *Let (V, e, r, p) be a cyclic preproof with vertex $v \in V$. If α is an instance of $e(v)$ such that $\alpha \not\vDash e(v)$, then there exist a preceding instance (i, β) where $\beta \not\vDash e(p_i(v))$.*

There are two direct corollaries of this lemma. The first is that the equation of each vertex in a cyclic preproof is valid if all of its premises are:

Corollary 3.2 (Local soundness)**.** *Let (V, e, r, p) be a cyclic preproof with vertex $v \in V$. If the equation of each premise is valid, i.e. $\vDash e(p_i(v))$ when $p_i(v)$ is defined, then $\vDash e(v)$.*

However, Lemma 3.1 also implies that we can extract an infinite sequence of invalid equations from any invalid equation in a cyclic preproof. This process also gives us the corresponding instances that are not satisfied.

If a parallel sequence of terms can be constructed from these instances that is infinitely decreasing according to some well-founded order, we have shown there are no invalid equations. To this end, a global condition is placed upon cyclic preproofs based on the notion of a *trace*. A trace is another sequence of terms intuitively capturing any dependency between the instances of a conclusion and its premise.

Definition 3.4. A ***path*** through a preproof $P = (V, e, r, p)$ is a finite or infinite sequence of vertices (v_i) such that, for each $i \in \mathbb{N}$, v_{i+1} is a premise of v_i, i.e. there is some j such that $v_{i+1} = p_j(v_i)$.

Definition 3.5. Let $\leq$ be a stable, well-founded order. A ***$\leq$-trace*** along a path (v_i) is a finite or infinite sequence of terms (T_i) where T_{i+1} is constrained according to the rule $r(v_i)$:

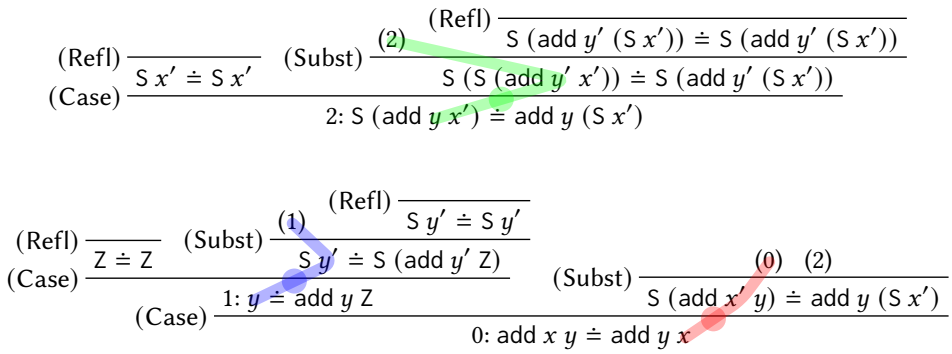

Figure 4. A cyclic proof that addition is commutative.

- (Case) where $x : d$ is the variable upon which case analysis is performed. If v_{i+1} is the premise associated with constructor k using fresh variables $x_0, \ldots, x_n$, then $T_{i+1} \leq T_i[k\ x_0\ \cdots\ x_n/x]$.
- (Subst) with substitution θ. If v_{i+1} is the lemma, then $T_{i+1}\theta \leq T_i$ and if v_{i+1} is the continuation, then $T_{i+1} \leq T_i$.
- Otherwise, $T_{i+1} \leq T_i$

When there is a strict inequality in the above definition, we say that v_i is a ***progress point***.

Remark 3.3 (Progress points). Part of our intention with this work is to relate rewriting induction to cyclic proofs. It is, therefore, not possible to build a specific relationship between derivations and progress points, as is done in e.g. Brotherston's work [7], because different rules will entail a progress point for different orderings. For example, our implementation is based on the substructural order where progress points are marked by the (Case) rule, but a reduction order would also use (Reduce) and (Subst) as in Section 4.

Lemma 3.3. *Let (V, e, r, p) be a preproof with vertex $v \in V$, let α be an instance of $e(v)$, and let (i, β) be a preceding instance. If T_0, T_1 is a trace for the path v, $p_i(v)$, then $T_1\beta \leq T_0\alpha$ and, in particular, $T_1\beta < T_0\alpha$, if v is a progress point.*

The aforementioned lemma shows how a trace, as previously defined, leads to a monotonic sequence of terms by closing those terms according to a sequence of preceding instances that emerges as a consequence of Lemma 3.1. If every path has a trace, then we can construct an infinitely decreasing sequence of terms for any sequence of invalid equations generated by Lemma 3.1. Thus we define a proof as a preproof that satisfies the following *global correctness condition.*

Definition 3.6. A $\leq$-***(cyclic) proof*** is a preproof such that, for every infinite path (v_i), there is a suffix, i.e. (v_{i+k}) for some $k \in \mathbb{N}$, which has a $\leq$-trace with infinitely many progress points.

Because our definition of a trace and its progress points is highly generic, it is undecidable if a sufficient set of traces exists or not. This is a significant difference from proofs by structural induction, whose validity is effectively a syntactic well-formedness condition. Although undesirable, we will, in practice, restrict the space of traces according to a particular application as in Section 5. In particular, our implementation only checks for traces composed solely of variables, which is decidable. However, we chose not to overfit the declarative system as alternative restrictions are equally valid. This point is discussed further under related work.

Example 3.7 (Commutativity of addition)**.** Fig. 4 displays a preproof for the commutativity of addition. Note that there are implicit applications of the (Reduce) rule in the S-case of the root node, i.e. the rewriting of add (S x) y to S (add x' y) in it's left parent. And similarly throughout.

To show that this is also a $\trianglelefteq$-proof, we must consider every infinite path and show they each have a suffix with an infinitely progressing trace. There are three cycles we must consider:

- One passing through 0 by following the continuation in case associated with the S constructor, for which the trace x, x', x, x, ... is sufficient. The decrease $x' \triangleleft x[S\ x'/x]$ marks a progress point.
- One passing through 1 by following the lemma in the case associated with the Z constructor, for which the trace y, y', y, y, ... is similarly sufficient.
- And finally, one passing through 2 by following the lemma in case associated with the S constructor, for which the trace y, y', y, y, ... is also sufficient.

These traces are informally depicted as coloured lines in the preproof diagram with progress points marked by circles, following [34].

Theorem 3.4 (Global soundness)**.** *Let $\leq$ be a stable well-founded order. If (V, e, r, p) is a $\leq$-proof with some vertex $v \in V$, then $\vDash e(v)$.*

Remark 3.4 (A refinement of global correctness). Allowing for traces that only cover a certain suffix of a path is particularly useful in the context of cyclic proofs, as paths must be ultimately periodic. It is, therefore, only necessary to find a trace for every cycle. Note, however, cycles may overlap, and so it is *not* sufficient to assign a single term to each vertex.

4 Rewriting Induction

As discussed in the introduction, automating traditional inductive proofs is highly non-trivial, and many alternatives have thus been proposed. One such well-developed line of work is proof by consistency or inductionless induction [23, 30, 39]. Musser observed that if an equation can be consistently added to a strongly complete theory, it is true of its least model. The consistency of an equational theory, in this case that $\not\vdash \texttt{False} = \texttt{True}$, can then be verified by converting the theory into a confluent and terminating rewrite system by using the Knuth-Bendix algorithm [33].

Rewriting induction, due to Reddy, highlights its core mechanisms [40]. The principal idea behind rewriting induction is to perform induction using a well-founded ordering that includes the reduction relation — a "reduction" order. A reduction order is more flexible than the substructural order in that more terms are related. Furthermore, unlike the use of a structural induction scheme, this approach can be easily extended to mutually inductive datatypes as there is no need to invent a complementary induction hypothesis.

For this section, we shall assume $\leq$ is a ***reduction order***. That is, a well-founded stable order such that each rewrite rule $M \to N \in R$ is strictly decreasing, i.e. $N < M$.

The ***decreasing order*** $\prec$ is defined as the transitive closure of the relation $< \cup \lhd$.

Lemma 4.1. *$\prec$ is a reduction order.*

A serious complication of rewriting induction, however, is that all lemmas (including equations that play the role of induction hypotheses) must also be orientated according to the reduction order. Properties such as the commutativity of addition are, therefore, difficult to prove. Although there are extensions that allow for unoriented equations, the increase in complexity detracts from the advantage of rewriting induction — its simplicity [2].

Furthermore, rewriting induction is highly sensitive to the choice of order and choosing an order in advance is a non-trivial task. For example, if the term $\mathsf{add}\ (\mathsf{add}\ x\ y)\ z$ is less than $\mathsf{add}\ x\ (\mathsf{add}\ y\ z)$, then it is impossible to prove addition is associative without externally supplied lemmas.

Our cyclic proof system allows for both unoriented equations and is ambivalent to the choice of order, overcoming these limitations. However, it is worth reiterating that we have not provided a method for verifying the global condition, which is required in the general case.

Definition 4.1. The most significant inference rule concerns the ***expansion*** of an equation:

$$\mathsf{Expand}_C(C[f\ M_0\ \ldots\ M_n] = N) \coloneqq \{C[L]\theta = N\theta \mid f\ N_0 \cdots N_n \to L \in R,\ \theta = \mathsf{mgu}(\overline{M}, \overline{N})\}$$

following the presentation used in [2].

This operator is used to perform case analysis of the variables, which are instantiated with constructors. However, a critical part of this definition is that a reduction step has occurred, and the left-hand side is, therefore, strictly smaller. In other words, it marks a progress point for a reduction order.

Definition 4.2 (Rewriting induction)**.** The inference rules of rewriting induction manipulate pairs (E, H) of oriented equations E (denoted $M = N$, in contrast to $M \doteq N$) to be proven, and rewrite rules H that supplement the original set R. The judgement $\vdash (E, H)$ is inductively defined by the rules of Fig. 5.

Note that although the rules from H must comply with the reduction order, they needn't be orthogonal to R or behave like a functional program. For example, $\mathsf{add}\ (\mathsf{add}\ x\ y)\ z \to \mathsf{add}\ x\ (\mathsf{add}\ y\ z)$ is valid despite there already being rules that govern the reduction of add.

Theorem 4.2 (Soundness)**.** *If $\vdash (E, \emptyset)$ is a rewriting induction derivation, then every equation in E is valid [40].*

4.1 Translation to Cyclic Proof

Rewriting induction allows for previously seen equations to be used as hypotheses. This circularity is not unsound as hypotheses are only introduced through a strict decrease.

We will show that rewriting induction proofs can be translated into our cyclic proof system and, therefore, can be seen as a form of cyclic proof search, see Theorem 4.3. Furthermore, as rewriting induction subsumes inductionless induction, a line of work that adapts the Knuth-Bendix completion procedure to perform saturation based proofs by consistency, our system also subsumes that approach [40].

We will construct a cyclic proof by induction over a rewriting induction derivation. Cyclic proofs discharge their hypothesis globally rather than locally, and thus we need to allow for undischarged hypotheses when reasoning locally in this manner. We first define this generalisation of cyclic proofs as follows:

Definition 4.3. A ***partial proof*** is a tuple (V, H, e, r, p) where V and H are disjoint finite sets of vertices such that:

- For each $v \in V \cup H$, there is an associated equation $e(v)$.
- For each $v \in V$, there is inference rule from Fig. 3 $r(v)$, and list of vertices $p(v) \in (V \cup H)^*$ called the premises. We write $p_i(v)$ for the i^{th} element of $p(v)$ starting with p_0.

$$\text{(End)}\ \frac{}{\vdash (\emptyset,\ H)} \qquad \text{(Delete)}\ \frac{\vdash (E,\ H)}{\vdash (E \cup \{M = M\},\ H)}$$

$$\text{(Simplify)}\ \frac{\vdash (E \cup \{M' = N\},\ H)}{\vdash (E \cup \{M = N\},\ H)}\ (M \to^*_{R \cup H} M') \qquad \text{(Expand)}\ \frac{\vdash (E \cup \mathsf{Expand}_C(M = N),\ H \cup \{M \to N\})}{\vdash (E \cup \{M = N\},\ H)}\ (N < M)$$

Figure 5. Inference rules of rewriting induction.

- And

$$\frac{e(p_0(v)) \quad \ldots \quad e(p_n(v))}{e(v)}$$

is a well-formed instance of the rule $r(v)$.

Furthermore, partial proofs must also satisfy the global condition that for every path (v_i), there is a suffix, i.e. (v_{i+k}) for some $k \in \mathbb{N}$, which has a $\preceq$-trace with infinitely many progress points.

We will refer to the elements of H as ***hypotheses***.

Intuitively, a partial proof is a proof where the hypotheses H may be used as premises but needn't be justified by an instance of an inference rule themselves. Note that when H is empty, we have a cyclic proof.

Theorem 4.3. *If* $\vdash (E, H)$ *is rewriting induction derivation, then there exists a partial proof* (V', H', e, r, p) *where* $E \subseteq \{e(v) \mid v \in V\}$ *and* $H = \{e_1 \to e_2 \mid e_1 \doteq e_2 \in H',\ e_2 \preceq e_1\}$, *i.e. the rewrite rules of* H *are orientations of equations in* H'.

Remark 4.1. Rewriting induction and related approaches do not typically require a confluent rewrite system. Therefore, this theorem only shows that our system subsumes rewriting induction when confluence is taken as an assumption, which is the case for our intended application. It is also worth noting that we only require confluence when defining the semantics of terms; we are confident that the proof system could be made sound for non-confluent rewrite systems.

Cyclic proofs, for a generic sequent calculus, have been shown to subsume traditional structural induction [6]. Although this result is not directly applicable to our system, which is specialised to unconditional equational reasoning, we conjecture that an analogous argument could be made with unconstrained usage of the (Subst) rule. For examples of the translation from structural induction to proofs in our calculus, see the long version of this paper [28].

5 Detecting and Verifying Cycles

Our proof system is designed to be used in a goal-orientated manner. It is necessary to form cycles to produce a finite proof, and subsequently, verify that the global condition has been met. In this section, we discuss these high-level aspects of our proof search algorithm.

There exists a generic cyclic theorem prover for sequent calculi — the Cyclist system [9]. It is generic in that it supports an arbitrary set of inference rules. Given this setup, it would be possible to naïvely enumerate derivations of the goal formula and create cycles just when formulas are repeated.

Sequents discovered earlier in proof search are intuitively simpler in that they apply to a more general instance. For example, consider the (Case) rule where only certain instances of the conclusion will be relevant to each premise. Therefore, it will often be necessary to generalise an equation to relate it to an ancestor.

However, it is desirable to avoid generalisation as part of normal proof search, as the space is often intractable and not guaranteed to lead to a cycle [37]. The Cyclist framework is thus parameterised by a "matching function" that detects when cycles can be formed. The matching function for first-order logic is a combination of weakening and substitution. For separation logic, the matching function is the frame rule.

The capacity of the Cyclist framework for equational reasoning is known to be limited. For example, the commutativity of addition cannot be automatically proven without the lemma $\mathsf{add}\ x\ (\mathsf{S}\ y) \doteq \mathsf{S}\ (\mathsf{add}\ x\ y)$. Our observation is that the existing matching functions are too restrictive for equational reasoning.

The usual rules for equational reasoning, i.e. the congruence axioms, are intractable due to the vast number of intermediate equations they create [3]. However, we cannot simply avoid such equational reasoning in cyclic proofs as they are often needed to form cycles. We thus propose the substitution of equals as an alternative matching function, appearing in our proof system as the (Subst) rule. This way of closing cycles resembles the use of hypotheses as rewrite rules in rewriting induction.

Algorithmically, (Subst) is only used as a matching function in a goal-directed manner. The task of generating useful lemmas is a non-trivial and orthogonal concern [26]. For a given goal equation $\Gamma \vdash M \doteq N$, any subterms that are instances of the left- or right-hand side of an existing node are considered and the goal is rewritten accordingly, leaving the continuation premise as a new subgoal. Thus the lemma, i.e. the first premise of (Subst), is always an equation that has already appeared in the tree, acting somewhat like an induction hypothesis.

There is a significant novelty in using (Subst) as a matching function not present in the Cyclist system — it doesn't completely close a branch of the derivation tree into a cycle but leaves a new subgoal, i.e. the continuation, which must also be solved.

5.1 Refining Substitution

While substitution is an appropriate technique for detecting cycles, it can also create many redundancies when searching for proofs that lead to performance issues. Therefore, we only consider a subset of available lemmas for substitution in our implementation. These are determined by the rule used to justify the lemma:

- (Refl) Clearly, no useful lemma is justified by reflexivity as the continuation is identical to the goal.
- (Reduce) We also do not consider lemmas justified by reduction. This restriction follows naturally from the reasonable strategy that we ought to reduce a goal as far as possible before further reasoning. Suppose we have a goal $C[M\theta] \doteq P$ and a candidate lemma $M \doteq N$ that is justified by (Reduce). As the goal is assumed to be in normal form, we know that M is also in normal form. Thus there is a premise $M \doteq N'$ where $N \to_R^* N'$. We can apply this lemma directly, to leave the continuation $C[N'\theta] \doteq P$. Of course, this is distinct from the continuation that we would arrive at if we used the unreduced lemma, i.e. $C[N\theta] \doteq P$. However, if we normalise this original continuation to $Q \doteq P'$, then, by confluence, the new continuation must also normalise to $Q \doteq P'$, and we can proceed as normal. The comparison between these proofs can be seen in Fig. 6.
- (Subst) If a lemma is itself justified by (Subst), we can use the secondary lemma directly as contexts and substitutions are composable. Here we are observing that the order in which lemmas are applied is associative. However, choosing one of these as the canonical form, i.e. associating nested instances into the continuation, increases performance because the roles of the lemma and continuation are not symmetric — we wish to reduce the number of choices for the former. This argument can also be seen in Fig. 6.
- Therefore, only those lemmas justified by (Case) are considered for substitution.

In the proof that addition is commutative, for example, there are 16 vertices but only 3 instances of the (Case), a significant reduction that mitigates the cost of verifying cycles.

5.2 Verifying Cycles

Our global condition on paths is undecidable in general. If we restrict our attention to traces comprising variables and the substructural order, it becomes decidable. Informally, this captures the space of typical proofs where induction concerns an explicit variable.

A comparable result was first shown by reduction to Büchi automata in the original work on cyclic proofs for first-order logic with inductive definitions [6]. Two ω-regular languages are extracted from a preproof: the path language and the trace language. It can then be checked whether the path language is included in the trace language, i.e. every path has an infinitely progressing trace.

Unfortunately, checking the inclusion of Büchi automata is doubly exponential in the number of vertices, as it involves complementing the automata [38]. This procedure becomes onerous if several candidate proofs, the majority of which may be unsound, need to be checked throughout the proof search. In the Cyclist theorem prover, soundness checking could take a significant proportion of the proof time [47].

This approach to verifying cyclic proofs fails to take advantage of the incremental nature of the goal-orientated proof search, where proofs share a common prefix. Furthermore, as soon as a cycle that does not satisfy the global condition is detected, there is no advantage to completing the proof. Instead, we annotate the proof graph with an abstract domain representing the ω-regular language of paths — size-change graphs, originally developed for termination analysis [35]. The workload is performed as each node is uncovered so that the soundness condition is represented explicitly.

Definition 5.1. Let $e(v) = \Gamma \vdash \phi$ and $e(v') = \Gamma' \vdash \phi'$ be two vertices in a preproof (V, e, r, p). A ***size-change graph*** between v and v' is a labelled bipartite graph between Γ and Γ', i.e. a set of triples $(x, y, l) \in \Gamma \times \Gamma' \times \{\simeq, \lesssim\}$ where the labels mark equality or a decrease which are possible progress points.

We write $G : v \to v'$ for such a size-change graph, $x \simeq y \in G$ if $(x, y, l) \in G$ for any l, and $x \lesssim y \in G$ if, specifically, $(x, y, \lesssim) \in G$. Labels from a simple lattice with $\lesssim > \simeq$.

Definition 5.2 (Composition of size-change graphs)**.** Given two size-change graphs $G : v \to v'$ and $G' : v' \to v''$, then there is a size-change graph $G' \circ G : v \to v''$ defined as:

$$G' \circ G := \{(x, z, l \sqcup l') \mid (x, y, l) \in G, (y, z, l') \in G'\}$$

That is, there is an edge $x \simeq z$ whenever there exists a variable y and edges $x \simeq y \in G$ and $y \simeq y \in G'$. It is decreasing if either edge is decreasing.

The following definition associates a canonical size-change graph with each edge in a preproof. Intuitively, an edge $x \simeq y \in G_{(v,v')}$ indicates that x, y is a valid trace passing from v to v', and that it is a progress point if $x \lesssim y \in G_{(v,v')}$.

Definition 5.3 (The size-change graph of an edge)**.** Let (V, e, r, p) be a preproof. For each edge $(v, v') \in E$ of the underlying graph, $G_{(v,v')} : v \to v'$ is defined as follows:

- If $r(v)$ is an instance of (Subst) with substitution θ and v' is the lemma, then there is a non-decreasing edge $x \simeq y \in G_{(v,v')}$ for all other variables such that $x = \theta(y)$.

$$\dfrac{\dfrac{\Delta \vdash M \doteq N'}{\Delta \vdash M \doteq N}\,(\text{Reduce}) \quad \dfrac{\Gamma \vdash Q \doteq P'}{\Gamma \vdash C[N\theta] \doteq P}\,(\text{Reduce})}{\Gamma \vdash C[M\theta] \doteq P}\,(\text{Subst}) \rightsquigarrow \dfrac{\Delta \vdash M \doteq N' \quad \dfrac{\Gamma \vdash Q \doteq P'}{\Gamma \vdash C[N'\theta] \doteq P}\,(\text{Reduce})}{\Gamma \vdash C[M\theta] \doteq P}\,(\text{Subst})$$

$$\dfrac{\dfrac{\Lambda \vdash M \doteq N \quad \Delta \vdash D[N\theta] \doteq P'}{\Delta \vdash D[M\theta] \doteq P'}\,(\text{Subst}) \quad \Gamma \vdash C[P'\sigma] \doteq P}{\Gamma \vdash C[(D[M\theta])\sigma] \doteq P}\,(\text{Subst}) \rightsquigarrow \dfrac{\Lambda \vdash M \doteq N \quad \dfrac{\Delta \vdash D[N\theta] \doteq P' \quad \Gamma \vdash C[P'\sigma] \doteq P}{\Gamma \vdash C[(D[N\theta])\sigma] \doteq P}\,(\text{Subst})}{\Gamma \vdash C[(D[M\theta])\sigma] \doteq P}\,(\text{Subst})$$

Figure 6. Redundancy of unreduced lemmas & Reassociation of nested substitution.

- If $r(v)$ is an instance of (Case) and the variable being analysed is x, then there is a decreasing edge $x \lesssim y$ for each fresh variable y introduced into v' and a non-decreasing edge $z \simeq z$ for all variables.
- Otherwise, the size-change graph is simply the identity: $z \simeq z$ for all variables in both environments.

The composition of these size-change graphs provides traces for general paths. And, by taking a generalisation of the transitive closure, we represent the space of possible infinite traces.

Definition 5.4. The ***closure of a preproof*** $P = (V, e, r, p)$ is a set of size-change graphs, $\mathrm{cl}(P)$, such that

- For each edge $(v, v') \in E$, $G_{(v,v')} \in \mathrm{cl}(P)$
- If $G : v \to v' \in \mathrm{cl}(P)$ and $G' : v' \to v'' \in \mathrm{cl}(P)$, then $G' \circ G \in \mathrm{cl}(P)$

Lemma 5.1. *Suppose $P = (V, e, r, p)$ is a preproof with a path $v_0, \ldots, v_n$ for some $n > 0$, then there is a size-change graph $G : v_0 \to v_n \in cl(P)$ such that whenever $x \simeq y \in G$ there is a trace $x, \ldots, y$ for this path, which has a progress point if $x \lesssim y \in G$.*

As any infinite sequence of nodes is ultimately periodic, it is sufficient to look for decreasing edges in the size-change graphs representing cycles in the closure.

Theorem 5.2. *A cyclic preproof $P = (V, e, r, p)$ is a proof if every $G : v \to v \in \mathrm{cl}(P)$ such that $G = G \circ G$ has a decreasing edge $x \lesssim x$ for some variable x.*

6 Implementation and Empirical Evaluation

We implemented a prototype cyclic equational reasoning tool as a plugin for GHC 9.0.2 — CycleQ. It currently supports a small subset of Haskell, including top-level recursive functions, algebraic datatypes, and polymorphism. The user adds equations to their program using the following syntax, and the plugin will attempt to prove them at compile-time, optionally outputting a cyclic proof graph if successful.

```
mapId :: List a -> Equation
mapId xs = map id xs ≡ xs
```

The tool performs a bounded depth-first search using the inference rules from Fig. 3, in addition to a rule for function extensionality and decomposition of datatype constructors:

$$\dfrac{\forall i \leq n \quad M_i \doteq N_i}{k\ M_1\ \cdots\ M_n \doteq k\ N_1\ \cdots\ N_n}$$

Although this rule is derivable from (Subst), we distinguish it because it is not intended as a mechanism for creating cycles and can be applied eagerly, in a goal-directed manner, without incurring the cost associated with lemma generation. Where more than one rule is applicable to a goal, the rules are prioritised as follows: reduction, reflexivity, congruence, function extensionality, subst, case analysis. Once applied, the tool never backtracks past the first three as they always simplify the goal without loss of generality. Furthermore, case analysis always selects a variable preventing further (non-strict) reduction, much like needed narrowing [1].

6.1 Evaluation

There are very few implementations of cyclic proof systems, and their performance with equational goals is not well understood. The Cyclist system [9], which is certainly the most developed, is known to have difficulty with equational reasoning and has issues with the verification of cycles [47]. The primary objective of this evaluation is to demonstrate that our system, although simple, is reasonably efficient, avoiding a bottleneck in cycle verification.

We tested the tool against a standard benchmark suite of 85 induction problems concerning natural numbers, lists, and trees, originally used to test the IsaPlanner tool [20]. Since none of these concern mutual induction explicitly, we also designed a small number of problems around the representation of annotated, mutually recursive syntax trees, as shown in the introduction. The results were obtained as an average of 10 runs on a 2.20GHz Intel® Core™ i5–5200U with 4 cores and 7.5 GB RAM.

The number of IsaPlanner benchmark problems solved in a given time bound is plotted in Fig. 7. The tool was able to solve 44 of the problems (13 were not in scope as they concerned conditional equations), with 40 of those solvable in under 100 ms. The average time for solvable IsaPlanner benchmarks was 129 ms. All the mutual induction problems were solved in 5.3 ms on average.

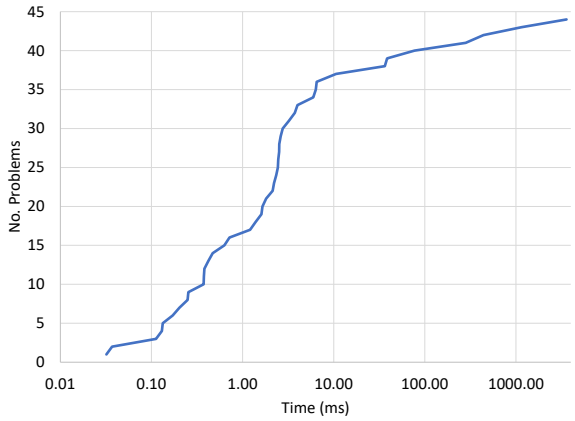

Figure 7. Summary of IsaPlanner benchmarks

6.2 Limitations

Although the tool performs efficiently on those 44 benchmark problems that it is able to solve, this number is relatively small. By comparison: HipSpec proved 80, Zeno 82, CVC4 80, ACL2 74, Inductive Horn Clause Solving 68, IsaPlanner 47, and Dafny 45 (as reported by [14, 53]).

However, the following analysis shows that the problems CycleQ could not solve are attributable to two features that it lacks: *conditional equations* and *lemma discovery*, both of which are essentially orthogonal to cyclic reasoning. Hence, we expect that our tool can incorporate these features in future work, which would allow for a more meaningful comparison.

First, many of the benchmark problems are themselves conditional equations. Hence they simply fall outside of the scope of our system.

Some further 23 benchmarks, although not themselves conditional equations, require conditional equations internally in their proof. Problem 4 is a typical example — $\texttt{S}\ (\texttt{count}\ x\ xs) \doteq \texttt{count}\ x\ (x : xs)$. This can be solved by performing case analysis on the equality predicate, by assuming $x == y \doteq \texttt{False}$ or $x == y \doteq \texttt{True}$. Our system does not currently have a mechanism for hypothetical reasoning of this form. Since x may take infinitely many values in each case, and none of these enable a cycle, there is no other way to progress with the proof.

As far as we are aware, there is no reason why our system could not be extended in this direction (for example, by formulating the proof system using a judgement with an antecedent), but we felt it would overcomplicate the paper without adding any interesting new ideas. These problems account for all those that Dafny solved that our tool did not.

The second reason some problems were unsolved is that they require lemmas, which was the case for the 4 remaining problems [42]. Specifically, property 47 *is* provable by our system when it is given the commutativity of `max` and 54, 65, and 69 when given the commutativity of add. Most comparable tools incorporate some form of lemma discovery, which is very powerful but orthogonal to this work. It is worth noting, however, that CycleQ solved a number of the benchmark problems designed to test strengthening and lemma discovery, despite not having a specialised tactic for either. We look to incorporate a theory exploration system into our solver as future work, after which a direct comparison will be more insightful.

A couple of problems took significantly longer to solve. And, unsurprisingly, these required the construction of larger proofs. There are several factors to which this could be attributed: the branching factor of proof search, the increased number of lemmas, or the cost of verifying the global correctness condition. In any case, we believe theory exploration could be used to mitigate this by allowing for smaller, more compositional proofs.

7 Related Work and Conclusion

As inductive definitions are ubiquitous in computer science, and functional programming in particular, a lot of work has been dedicated to developing tools that automate or aid equational reasoning over these structures. However, as proof systems for inductive definitions don't admit cut elimination, most research is aimed at "lemma" discovery [12, 26].

One technique for generating lemmas is to generalise the current goal by identifying common subterms, as implemented by ACL2 [5]. The heuristic was later refined by the Zeno tool that checks for counterexamples to prevent over generalisation, a common problem with the original method [45]. It seems likely that the exploratory nature of cyclic proofs could be used to suggest generalisations from failed proofs without over-generalisation.

Proof planning was developed as a way to better control heuristics in automated reasoning tools [11]. It gave rise to Clam system and IsaPlanner [13, 20, 27]. A lemma discovery strategy based on "rippling", a form of rewriting used in proof planning, was to construct a lemma from a failed proof [24]. However, the required higher-order unification became a bottleneck to the technique's success [13, 20, 27].

A radically different approach to lemma discovery is theory exploration [14]. Instead of attempting to construct suitable lemmas analytically, theory exploration generates random lemmas and attempts to prove them in an incremental manner. It is currently the state-of-the-art lemma discovery strategy, although it is hampered by scalability [26].

HipSpec is a tool that couples theory exploration with a traditional first-order theorem prover [15, 43]. As with the other approaches discussed so far, it ultimately relies on induction schema and thus cannot handle mutual induction. We plan to integrate a theory exploration strategy into our

tool, thus combining powerful lemma discovery with mutual induction.

The difficulties with induction has motivated a long line of work in inductionless induction [23, 30, 39]. While initially popular, as it can take advantage of general equational reasoning and rewriting techniques, the development of practical tools was limited [54]. However, the SPIKE theorem prover was based on this work and has since adopted a form of cyclic proof, but the relationship with our system is not completely clear [4, 48].

Circular coinduction is a similar technique but for equations about coinductive structure [44]. Analogous to the "expand" operator in rewriting induction, the "derivative" of an equation is taken before it can be used as a hypothesis, where encodes the possible coinductive observations.

Although originally ill-suited to inductive theorem proving, many tools have successfully been built upon SMT solvers [36, 41, 50]. More recently, induction has been incorporated into Horn clause solvers which, historically, struggle with domains such as non-linear arithmetic or some complex kinds of algebraic datatype [53].

Cyclic proofs have previously received attention for their application to program verification. Specifically, a cyclic proof system for separation logic has been given that automatically verifies that a program terminates [8, 51]. Cyclic proof systems have recently been shown to subsume generic model-checking algorithms such as: lazy-abstraction with interpolants, property-directed reachability, and maximal conservativity for infinite game solving [52]. As with the generic cyclic theorem prover Cyclist, it is the choice of "matching-function" or "cut" that determines exactly how the verification algorithm operates outside of the usual reasoning on the abstract domain. Cyclic proofs have also been applied to program synthesis for pointer manipulating programs [25].

The cost of verifying cycles has long been identified as a bottleneck of any tool based on cyclic proofs. In Brotherston's thesis, he proposed an alternative approach — "trace manifolds" [7]. A trace manifold is a set of trace segments that can be stitched together to construct a trace for any given path. The trace segments are uniquely assigned, simplifying the space of traces significantly but excluding some complex cycles that might require different traces for the same path segment. An alternative approach based on normalising the forms of cycles has been proposed and shown to be significantly more efficient [47, 49]. The algorithm is polynomial. However, there is no characterisation of exactly what patterns of cycles it is able to verify.

Acknowledgments

We gratefully acknowledge the support of the Engineering and Physical Sciences Research Council (EP/T006579/1, EP/T006595/1) and the National Centre for Cyber Security via the UK Research Institute in Verified Trustworthy Software Systems.

References

[1] Sergio Antoy, Rachid Echahed, and Michael Hanus. 2000. A needed narrowing strategy. *Journal of the ACM (JACM)* 47, 4 (2000), 776–822. https://doi.org/10.1145/347476.347484

[2] Takahito Aoto. 2006. Dealing with non-orientable equations in rewriting induction. In *International Conference on Rewriting Techniques and Applications*. Springer Berlin Heidelberg, Berlin, Heidelberg, 242–256. https://doi.org/10.1007/11805618_18

[3] Leo Bachmair and Harald Ganzinger. 1998. Equational reasoning in saturation-based theorem proving. *Automated deduction — a basis for applications* 1 (1998), 353–397. https://doi.org/10.1007/978-94-017-0435-9

[4] Adel Bouhoula, Emmanuel Kounalis, and Michaël Rusinowitch. 1992. SPIKE, an automatic theorem prover. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer Berlin Heidelberg, Berlin, Heidelberg, 460–462. https://doi.org/10.1007/BFb0013087

[5] Robert S Boyer and J Strother Moore. 2014. A computational logic. *Journal of Symbolic Logic* 55, 3 (2014), 1302–1304. https://doi.org/10.2307/2274490

[6] James Brotherston. 2005. Cyclic proofs for first-order logic with inductive definitions. In *International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*. Springer Berlin Heidelberg, Berlin, Heidelberg, 78–92. https://doi.org/10.1007/11554554_8

[7] James Brotherston. 2006. *Sequent calculus proof systems for inductive definitions*. Ph. D. Dissertation. University of Edinburgh. College of Science and Engineering. School of Informatics.

[8] James Brotherston, Richard Bornat, and Cristiano Calcagno. 2008. Cyclic proofs of program termination in separation logic. *ACM SIGPLAN Notices* 43, 1 (2008), 101–112. https://doi.org/10.1145/1328897.1328453

[9] James Brotherston, Nikos Gorogiannis, and Rasmus L Petersen. 2012. A generic cyclic theorem prover. In *Asian Symposium on Programming Languages and Systems*. Springer Berlin Heidelberg, Berlin, Heidelberg, 350–367. https://doi.org/10.1007/978-3-642-35182-2_25

[10] Reinhard Bündgen and Wolfgang Küchlin. 1989. Computing ground reducibility and inductively complete positions. In *Rewriting Techniques and Applications*. Springer Berlin Heidelberg, Berlin, Heidelberg, 59–75. https://doi.org/10.1007/3-540-51081-8_100

[11] Alan Bundy. 1988. The use of explicit plans to guide inductive proofs. In *International conference on automated deduction*. Springer Berlin Heidelberg, Berlin, Heidelberg, 111–120. https://doi.org/10.1007/BFb0012826

[12] Alan Bundy. 2001. The Automation of Proof by Mathematical Induction. In *Handbook of Automated Reasoning*. North-Holland, Amsterdam, 845–911. https://doi.org/10.1016/B978-044450813-3/50015-1

[13] Alan Bundy, Frank Van Harmelen, Christian Horn, and Alan Smaill. 1990. The OYSTER-CLAM system. In *International Conference on Automated Deduction*. Springer Berlin Heidelberg, Berlin, Heidelberg, 647–648. https://doi.org/10.1007/3-540-52885-7_123

[14] Koen Claessen, Moa Johansson, Dan Rosén, and Nicholas Smallbone. 2013. Automating inductive proofs using theory exploration. In *International Conference on Automated Deduction*. Springer Berlin Heidelberg, Berlin, Heidelberg, 392–406. https://doi.org/10.1007/978-3-642-38574-2_27

[15] Koen Claessen, Nicholas Smallbone, and John Hughes. 2010. QuickSpec: Guessing formal specifications using testing. In *International Conference on Tests and Proofs*. Springer Berlin Heidelberg, Berlin, Heidelberg, 6–21. https://doi.org/10.1007/978-3-642-13977-2_3

[16] Anupam Das. 2020. On the logical complexity of cyclic arithmetic. *Logical Methods in Computer Science* 16, 1 (2020). https://doi.org/10.23638/LMCS-16(1:1)2020

[17] Anupam Das. 2021. On the Logical Strength of Confluence and Normalisation for Cyclic Proofs. In *6th International Conference on Formal Structures for Computation and Deduction, FSCD 2021, July 17-24, 2021, Buenos Aires, Argentina (Virtual Conference) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 195)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 29:1–29:23. https://doi.org/10.4230/LIPIcs.FSCD.2021.29

[18] Anupam Das, Amina Doumane, and Damien Pous. 2018. Left-handed completeness for Kleene algebra, via cyclic proofs. In *LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning (EPiC Series in Computing)*. EasyChair Publications, 271–289. https://doi.org/10.29007/hzq3

[19] Nachum Dershowitz. 1982. Applications of the Knuth-Bendix Completion Procedure. In *Proceedings of the Seminaire d'Informatique Theorique*. Paris, 95–111.

[20] Lucas Dixon and Jacques Fleuriot. 2003. IsaPlanner: A prototype proof planner in Isabelle. In *International Conference on Automated Deduction*. Springer Berlin Heidelberg, Berlin, Heidelberg, 279–283. https://doi.org/10.1007/978-3-540-45085-6_22

[21] Laurent Fribourg. 1986. A Strong restriction of the inductive completion procedure. In *Automata, Languages and Programming*. Springer Berlin Heidelberg, Berlin, Heidelberg, 105–115. https://doi.org/10.1007/3-540-16761-7_60

[22] Stephen J. Garland and John V. Guttag. 1988. Inductive Methods for Reasoning about Abstract Data Types. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '88)*. Association for Computing Machinery, New York, NY, USA, 219–228. https://doi.org/10.1145/73560.73579

[23] Gérard Huet and Jean-Marie Hullot. 1982. Proofs by induction in equational theories with constructors. *Journal of computer and system sciences* 25, 2 (1982), 239–266. https://doi.org/10.1016/0022-0000(82)90006-X

[24] Andrew Ireland and Alan Bundy. 1996. Productive Use of Failure in Inductive Proof. In *Automated Mathematical Induction*. Springer Netherlands, Dordrecht, 79–111. https://doi.org/10.1007/978-94-009-1675-3_3

[25] Shachar Itzhaky, Hila Peleg, Nadia Polikarpova, Reuben N. S. Rowe, and Ilya Sergey. 2021. Cyclic Program Synthesis. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. Association for Computing Machinery, New York, NY, USA, 944–959. https://doi.org/10.1145/3453483.3454087

[26] Moa Johansson. 2019. Lemma discovery for induction. In *International Conference on Intelligent Computer Mathematics*. Springer International Publishing, Cham, 125–139. https://doi.org/10.1007/978-3-030-23250-4_9

[27] Moa Johansson, Lucas Dixon, and Alan Bundy. 2010. Dynamic rippling, middle-out reasoning and lemma discovery. In *Verification, Induction, Termination Analysis*. Springer Berlin Heidelberg, Berlin, Heidelberg, 102–116. https://doi.org/10.1007/978-3-642-17172-7_6

[28] Eddie Jones, C Ong, H Luke, and Steven Ramsay. 2021. CycleQ: An Efficient Basis for Cyclic Equational Reasoning. (2021). https://doi.org/10.48550/arXiv.2111.12553

[29] Jean-Pierre Jouannaud and Emmanuel Kounalis. 1989. Automatic proofs by induction in theories without constructors. *Information and Computation* 82, 1 (1989), 1–33. https://doi.org/10.1016/0890-5401(89)90062-X

[30] Deepak Kapur and David R Musser. 1987. Proof by consistency. *Artificial Intelligence* 31, 2 (1987), 125–157. https://doi.org/10.1016/0004-3702(87)90017-8

[31] Deepak Kapur, Paliath Narendran, and Hantao Zhang. 1986. Proof by induction using test sets. In *8th International Conference on Automated Deduction*. Springer Berlin Heidelberg, Berlin, Heidelberg, 99–117. https://doi.org/10.1007/3-540-16780-3_83

[32] Matt Kaufmann and J Strother Moore. 2021. *ACL2 User's Manual*.

[33] Donald E. Knuth and Peter B. Bendix. 1983. Simple word problems in universal algebras. In *Automation of Reasoning*. Springer Berlin Heidelberg, Berlin, Heidelberg, 342–376. https://doi.org/10.1007/978-3-642-81955-1_23

[34] Denis Kuperberg, Laureline Pinault, and Damien Pous. 2021. Cyclic proofs, system T, and the power of contraction. *Proceedings of the ACM on Programming Languages* 5, POPL (2021), 1–28. https://doi.org/10.1145/3434282

[35] Chin Soon Lee, Neil D Jones, and Amir M Ben-Amram. 2001. The size-change principle for program termination. In *Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. Association for Computing Machinery, New York, NY, USA, 81–92. https://doi.org/10.1145/360204.360210

[36] K. Rustan M. Leino. 2012. Automating induction with an SMT solver. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer Berlin Heidelberg, Berlin, Heidelberg, 315–331. https://doi.org/10.1007/978-3-642-27940-9_21

[37] Ewen Maclean. 1999. *Generalisation as a critic to the induction strategy*. Master's thesis. Department of Artificial Intelligence, University of Edinburgh.

[38] Max Michel. 1988. Complementation is more difficult with automata on infinite words. *CNET, Paris* 15 (1988).

[39] David R. Musser. 1980. On Proving Inductive Properties of Abstract Data Types. In *POPL '80* (Las Vegas, Nevada). Association for Computing Machinery, New York, NY, USA, 154–162. https://doi.org/10.1145/567446.567461

[40] Uday S. Reddy. 1990. Term rewriting induction. In *10th International Conference on Automated Deduction*. Springer Berlin Heidelberg, Berlin, Heidelberg, 162–177. https://doi.org/10.1007/3-540-52885-7_86

[41] Andrew Reynolds and Viktor Kuncak. 2015. Induction for SMT Solvers. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer Berlin Heidelberg, Berlin, Heidelberg, 80–98. https://doi.org/10.1007/978-3-662-46081-8_5

[42] Dan Rosén. [n. d.]. HipSpec Evaluation Results. http://danr.github.io/hipspec/

[43] Dan Rosén. 2012. *Proving equational Haskell properties using automated theorem provers*. Master's thesis. University of Gothenburg, Sweden.

[44] Grigore Roşu and Dorel Lucanu. 2009. Circular coinduction: A proof theoretical foundation. In *International Conference on Algebra and Coalgebra in Computer Science*. Springer Berlin Heidelberg, Berlin, Heidelberg, 127–144. https://doi.org/10.1007/978-3-642-03741-2_10

[45] William Sonnex, Sophia Drossopoulou, and Susan Eisenbach. 2012. Zeno: An automated prover for properties of recursive data structures. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer Berlin Heidelberg, Berlin, Heidelberg, 407–421. https://doi.org/10.1007/978-3-642-28756-5_28

[46] Christoph Sprenger and Mads Dam. 2003. On the Structure of Inductive Reasoning: Circular and Tree-Shaped Proofs in the μ-Calculus. In *Proceedings of the 6th International Conference on Foundations of Software Science and Computation Structures and Joint European Conference on Theory and Practice of Software* (Warsaw, Poland) *(FOSSACS'03/ETAPS'03)*. Springer Berlin Heidelberg, Berlin, Heidelberg, 425–440. https://doi.org/10.1007/3-540-36576-1_27

[47] Sorin Stratulat. 2019. Efficient Validation of FOL ID Cyclic Induction Reasoning.

[48] Sorin Stratulat. 2020. SPIKE, an automatic theorem prover — revisited. In *2020 22nd International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*. IEEE, 93–96. https://doi.org/https:/doi.org/10.1109/SYNASC51798.2020.00025

[49] Sorin Stratulat. 2021. E-Cyclist: Implementation of an Efficient Validation of FOLID Cyclic Induction Reasoning. In *SCSS 2021 - 9th International Symposium on Symbolic Computation in Software Science*, Vol. 342. 129–135.

[50] Philippe Suter, Ali Sinan Köksal, and Viktor Kuncak. 2011. Satisfiability modulo recursive programs. In *International Static Analysis Symposium*. Springer Berlin Heidelberg, Berlin, Heidelberg, 298–315. https://doi.org/10.1007/978-3-642-23702-7_23

[51] Gadi Tellez and James Brotherston. 2017. Automatically verifying temporal properties of pointer programs with cyclic proof. In *International Conference on Automated Deduction*. Springer International Publishing, Cham, 491–508. https://doi.org/10.1007/978-3-319-63046-5_30

[52] Takeshi Tsukada and Hiroshi Unno. 2022. Software Model-Checking as Cyclic-Proof Search. *POPL* 6, Article 63 (2022), 29 pages. https://doi.org/10.1145/3498725

[53] Hiroshi Unno, Sho Torii, and Hiroki Sakamoto. 2017. Automating induction for solving horn clauses. In *International Conference on Computer Aided Verification*. Springer International Publishing, Cham, 571–591. https://doi.org/10.1007/978-3-319-63390-9_30

[54] Claus-Peter Wirth. 2005. History and Future of Implicit and Inductionless Induction: Beware the Old Jade and the Zombie! In *Mechanizing Mathematical Reasoning: Essays in Honor of Jörg H. Siekmann on the Occasion of His 60th Birthday*. Springer Berlin Heidelberg, Berlin, Heidelberg, 192–203. https://doi.org/10.1007/978-3-540-32254-2_12