A Generic Cyclic Theorem Prover

James Brotherston[1], Nikos Gorogiannis[1], and Rasmus L. Petersen[2]

[1] Dept. of Computer Science, University College London
[2] Microsoft Research Cambridge

Abstract. We describe the design and implementation of an automated theorem prover realising a fully general notion of *cyclic proof*. Our tool, called Cyclist, is able to construct proofs obeying a very general cycle scheme in which leaves may be linked to any other matching node in the proof, and to verify the general, global infinitary condition on such proof objects ensuring their soundness. Cyclist is based on a new, generic theory of cyclic proofs that can be instantiated to a wide variety of logics. We have developed three such concrete instantiations, based on: (a) first-order logic with inductive definitions; (b) entailments of pure separation logic; and (c) Hoare-style termination proofs for pointer programs. Experiments run on these instantiations indicate that Cyclist offers significant potential as a future platform for inductive theorem proving.

1 Introduction

In program analysis, *inductive definitions* are essential for specifying the shape of complex data structures held in memory. Thus automated reasoning about such definitions, a.k.a. *inductive theorem proving*, is a key activity supporting program verification. Unfortunately, the explicit induction rules employed in standard inductive proofs pose considerable problems for proof search [10]. *Cyclic proof* has been recently proposed as an alternative to traditional proof by explicit induction for fixed point logics. In contrast to standard proofs, which are simply derivation trees, a cyclic proof is a derivation tree with "back-links" (see Figure 1), subject to a global soundness condition ensuring that the proof can be read as a proof by *infinite descent* à la Fermat [5]. This allows explicit induction rules to be dropped in favour of simple unfolding or "case split" rules.

Cyclic proof systems seem to have first arisen in computer science as tableaux for the propositional modal μ-calculus [23]. Since then, cyclic proof systems have been proposed for a number of applications, including: first-order μ-calculus [22]; verifying properties of concurrent processes [20]; first-order logic with inductive definitions [4, 9], bunched logic [6]; and termination of pointer programs [7]. However, despite the fairly rich variety of formal cyclic proof systems, automated tools implementing these formal systems remain very thin on the ground.

In this paper we describe the design and implementation of a new cyclic theorem prover, called Cyclist, based on a generic theory of cyclic proofs and instantiable to a wide variety of logical systems. We have implemented three concrete instantiations of Cyclist: (a) a system for a fragment of first-order

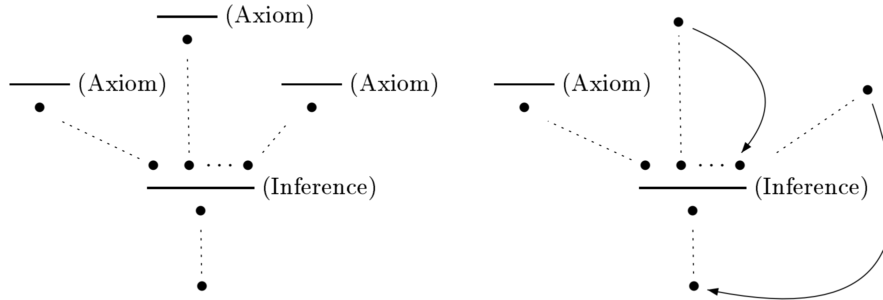

Fig. 1. Left: a typical proof structured as a finite tree, with the parent-child relation between nodes (•) given by a set of inference rules. Right: a typical *cyclic pre-proof*, structured as a tree proof with "back-links" between nodes (shown as arrows).

logic with inductive definitions, based on the formal system in [4]; (b) a system for entailment in separation logic, extending the one in [8]; and (c) a Hoare-style system for termination of pointer programs using separation logic, based on [7].

In Section 2, we give our general cyclic proof framework and, in parallel, discuss our implementation of CYCLIST using its instantiation (a) to first-order logic as a running example.

As above, cyclic proofs can be generally characterised as derivation trees with back-links ("pre-proofs") obeying a global, infinitary soundness condition qualifying them as *bona fide* cyclic proofs. The soundness condition states that every infinite path in the pre-proof must possess a syntactic *trace* that "progresses" infinitely often; informally, a trace can be thought of as a well-founded measure and its progress points to strict decreases in this measure. Our generic theory formalises this characterisation of cyclic proofs, which is entirely independent of the choice of any particular logical formalism.

There are two main technical obstacles to implementation of cyclic proof, both stemming from the structural complexity of cyclic proofs compared to standard proofs. First, the prover must be able to form back-links in the derivation tree. This inevitably leads to a *global* view of proofs, rather than one localised to the current subgoal as in most theorem provers. Second, the prover must be able to check whether or not a given pre-proof satisfies the general soundness condition. Our approach to both difficulties is described in Section 2.

Section 3 briefly describes our instantiations (b) and (c) of CYCLIST to separation logic frameworks. Then, in Section 4, we examine some of the issues pertaining to automated proof search in CYCLIST, and report on our experimental evaluation of the prover's performance in all three instantiations. Particular issues for cyclic proof search include looking for potential back-links in the proof, and deciding when to invoke the (potentially expensive) soundness check. Other issues, such as the priority ordering of rules and the conjecture / application of appropriate lemmas during a proof, are features of inductive theorem proving in general. Finally, Section 5 compares our contribution with related work, and Section 6 outlines directions for future work.

The theoretical framework on cyclic proofs in Section 2 is based on its earlier presentation in the first author's PhD thesis [5].

2 Cyclic proofs and the CYCLIST prover

In this section we develop a general notion of a *cyclic proof system*, which generalises the concrete formal systems given in, e.g., [4, 6–8, 20–23]. In an interleaved fashion we also describe CYCLIST, a mechanised theorem proving framework for cyclic proof. We use its instantiation $\text{CYCLIST}_{\text{FO}}$ to first-order logic with inductive predicates as a running example to illustrate our ideas and design choices. However, most of the issues we discuss here are relevant to *any* logical instantiation of the prover, and in particular to the two other instantiations we have also implemented: pure entailment in separation logic (cf. [8]) and termination of pointer programs based on separation logic (cf. [7]). In order to avoid overwhelming the reader with technical details, we intentionally elide some finer points of our implementation to begin with, and introduce these gradually as we go along.

2.1 Implementation platform. The core of CYCLIST is an OCaml functor parameterised over a user-defined datatype that describes the desired logic and its basic rules of inference. The functor provides functions for proof search and basic manipulation of cyclic proofs. CYCLIST also provides an OCaml interface to a custom model checker in C++ that checks the soundness of cyclic proofs.

In contrast to e.g. [8], we decided against implementing CYCLIST inside a theorem prover such as Isabelle or HOL Light. This is because the structural machinery of cyclic proof cannot be straightforwardly represented inside a tool employing a standard, tree-like internal notion of proof. Consequently, as in [8], a *deep embedding* of cyclic proof systems would be necessary, whereby cyclic proofs are represented as explicit datatypes, and reasoned about using functions defined over those datatypes. In addition to its technical difficulty, this approach negates most of the advantages of using a trusted theorem prover, as correctness depends fundamentally on the (unproven) correctness of the deep embedding as well as the correctness of the external soundness checker. Thus we gain implementation efficiency at relatively little expense of confidence by working directly in OCaml.

2.2 Sequents and inference rules. First, we assume a set $\mathcal{S}$ of objects, corresponding to the '•'s in Figure 1 and called *sequents*, from which our proofs will be built. Next we assume a set $\mathcal{R}$ of *proof rules* which are each of the form:

$$\frac{S_1 \dots S_n}{S} \, (\text{R})$$

where $n \geq 0$ and $S, S_1, \dots, S_n$ are sequents. S is called the *conclusion* of rule (R) and $S_1, \dots, S_n$ its *premises*; a rule with no premises is called an *axiom*. (Strictly speaking, the rules are usually understood as rule schemata, where parts of sequents may act as parameters.) A *derivation tree* is then, as usual, a tree each of whose nodes v is labelled with a sequent $S(v)$, and the rule $R(v)$ of which it is the conclusion, in such a way as to respect the proof rules.

CYCLIST expects a user-defined type for sequents, and for inference rules, each of which reduces a given conclusion sequent to a list of premises. However, because rules are really *rule schemata*, a rule may have multiple different applications to a sequent. To support this, rules in CYCLIST return a *list* of lists of

premises, corresponding to the results of all possible applications of the rule to the sequent. In particular, axioms always return a list of empty lists of premises.

In $\text{Cyclist}_{\text{FO}}$, we define a type for negation-free $\forall\exists$-sequents in disjunctive normal form, built from first-order formulas with equality and disequality, function terms and inductively defined predicates. It is straightforward to define standard rules and axioms for handling equalities, contradiction, simplification, quantifiers and conjunction / disjunction.

2.3 Inductive predicates and unfolding rules. Inductive predicates are specified by a set of *inductive rules* each of the form $F \Rightarrow P\mathbf{t}$, where P is an inductive predicate, $\mathbf{t}$ is a tuple of appropriately many terms and F is a formula (subject to certain restrictions to ensure monotonicity of the definitions).

Example 1. We can define a "natural number" predicate N, mutually defined "even / odd" predicates E and O, and a ternary "addition" predicate Add (where $Add(x, y, z)$ should be read as "$x + y = z$") via the following inductive rules:

$$\begin{array}{ccc} \Rightarrow N0 & \Rightarrow E0 & Ny \Rightarrow Add(0, y, y) \\ Nx \Rightarrow Nsx & Ox \Rightarrow Esx & Add(x, y, z) \Rightarrow Add(sx, y, sz) \\ & Ex \Rightarrow Osx & \end{array}$$

where 0 is a constant symbol and s is a unary function symbol, understood as the usual zero and "successor" function in Peano arithmetic respectively. □

Given a set of inductive rules, $\text{Cyclist}_{\text{FO}}$ generates rules for unfolding inductive predicates on the right and left of sequents. As usual, the right-unfolding rules for a predicate P are just sequent versions of the inductive rules introducing P.

Example 2. Applying the right-unfolding rule corresponding to $Nx \Rightarrow Nsx$ from Example 1 to the conclusion sequent $F \vdash Nsy, Nssz$[3] yields:

$$\frac{\big[\,[F \vdash Ny, Nssz]\ ;\ [F \vdash Nsy, Nsz]\,\big]}{F \vdash Nsy, Nssz}\,(R_N)$$

Note the bracketing indicating the two possible applications of this rule (we use ';' to separate list items), each resulting in a single premise sequent. □

The left-unfolding rule for an inductive predicate can be seen as a *case distinction* principle that replaces an inductive predicate in the left of a conclusion sequent with a premise for every clause of its definition.

Example 3. Applying the left-unfolding rule for the predicate E given in Example 1 to the conclusion sequent $Ey \vdash G$ yields the following:

$$\frac{\big[\,[\,y = 0 \vdash G\ ;\ y = sz, Oz \vdash G\,]\,\big]}{Ey \vdash G}\,(L_E)$$

where z is a fresh variable. Observe that, in this case, there is only one possible application of the rule which results in two premises. □

[3] As usual in sequent calculi, comma corresponds to $\wedge$ in the LHS and $\vee$ in the RHS.

2.4 Cyclic proofs and the forming of back-links. We first define cyclic pre-proofs below. Here, a leaf of a derivation tree is called *open* if it is not the conclusion of an axiom, i.e. if $R(v)$ is undefined.

Definition 1 (Pre-proof). A *pre-proof* of a sequent S is a pair $(\mathcal{D}, \mathcal{L})$, where $\mathcal{D}$ is a finite derivation tree whose root is labelled by S, and $\mathcal{L}$ is a *back-link function* assigning to every open leaf ℓ of $\mathcal{D}$ a node $\mathcal{L}(\ell)$ of $\mathcal{D}$ such that $S(\mathcal{L}(\ell)) = S(\ell)$.

Any pre-proof $\mathcal{P} = (\mathcal{D}, \mathcal{L})$ can be understood as a graph by identifying each open leaf ℓ of $\mathcal{D}$ with $\mathcal{L}(\ell)$. A *path* in $\mathcal{P}$ is an infinite sequence v_i of nodes of $\mathcal{P}$ such that for every i, either (y_i, y_{i+1}) is an edge in $\mathcal{D}$, or $\mathcal{L}(v_i) = v_{i+1}$.

According to Definition 1, a back-link in a cyclic pre-proof is formed by assigning to a leaf ℓ in the derivation tree another node $\mathcal{L}(\ell)$ such that $S(\ell) = S(\mathcal{L}(\ell))$. CYCLIST relaxes this strict requirement slightly and permits back-links between a leaf node S_1 and any other node S_2 such that a user-defined *matching function* returns true, given S_1, S_2 as arguments.[4]

In $\text{CYCLIST}_{\text{FO}}$, we use the following matching function: S_1 matches S_2 if S_1 is derivable from S_2 using only weakening and substitution principles.

Example 4. In $\text{CYCLIST}_{\text{FO}}$, the sequent S_1 below matches S_2 because there is a derivation of S_1 from S_2 using weakening and substitution principles, as follows:

$$\cfrac{\cfrac{\mathbf{S_2}: \quad Oy \vdash Ny}{Osz \vdash Nsz}\ \text{(Subst)}}{\mathbf{S_1}: \quad Osz, Essz, Ez \vdash Nsz, Ny}\ \text{(Weak)}$$

Thus a leaf labelled by S_1 can be back-linked to any node labelled by S_2. □

2.5 Defining the trace pair function. To qualify as a *bona fide* proof, a cyclic pre-proof must satisfy a global soundness condition, defined using the notion of a *trace* along a path in a pre-proof.

Definition 2 (Trace). Let $\mathcal{T}$ be a set of *trace values*. A *trace pair function* is a function $\delta : (\mathcal{S} \times \mathcal{R} \times \mathcal{S}) \to \text{Pow}((\mathcal{T} \times \mathcal{T} \times \{0, 1\}))$ (where $\text{Pow}(-)$ is powerset) such that for any $S, S' \in \mathcal{S}$ and $R \in \mathcal{R}$, the set $\delta(S, R, S')$ is finite (and computable). If $(\alpha, \alpha', n) \in \delta(S, R, S')$ for some $n \in \{0, 1\}$ then (α, α') is said to be a *trace pair* for (S, R, S'), and if $n = 1$ then (α, α') is said to be a *progressing trace pair*.

Now let $\pi = (v_i)_{i \geq 0}$ be a path in a pre-proof $\mathcal{P}$. A *trace following* π is a sequence $\tau = (\alpha_i)_{i \geq 0}$ such that, for all $i \geq 0$, (α_i, α_{i+1}) is a trace pair for $(S(v_i), R(v_i), S(v_{i+1}))$. If infinitely many of these (α_i, α_{i+1}) are progressing trace pairs, then τ is said to be *infinitely progressing*.

Since we are only interested in traces following paths in a pre-proof, we may assume for simplicity that the domain of a trace pair function δ, written $\text{dom}(\delta)$,

[4] One could also simply include a rule allowing one to conclude S_1 from S_2 whenever S_1 matches S_2, but our treatment is typically more convenient for proof search.

is restricted to triples (S, R, S') such that S is the conclusion of an instance of the rule R and S' is one of the premises of that instance. Given such a δ, the tuple $(\mathcal{S}, \mathcal{R}, \mathcal{T}, \delta)$ is then called a *cyclic proof system*.

In order to facilitate checking the global soundness condition, CYCLIST requires pre-proofs to carry information about trace pairs. According to Defn. 2, a *trace pair function* δ takes as input a sequent S, the rule R applied to it and one of the premises S' obtained as a result, and returns the sets of associated *progressing* and *non-progressing* trace pairs. Intuitively, a progressing trace pair identifies a measure that becomes strictly smaller when moving from S to S' under the application of R, while a non-progressing trace pair identifies a measure that at least does not increase. (Defn. 5 below will make precise the correspondence between trace pairs and measures.)

In CYCLIST$_{\text{FO}}$, we adopt the notion of trace from [4, 5]. There, trace values are atomic formulas of the form $P\mathbf{t}$ occurring on the left of sequents, where P is an inductive predicate. Then $(P\mathbf{t}, Q\mathbf{t}')$ is a progressing trace pair on (S, R, S') if R is a left-unfolding rule, $P\mathbf{t}$ is the formula in S being unfolded and $Q\mathbf{t}'$ is obtained in S' by unfolding $P\mathbf{t}$. $(P\mathbf{t}, Q\mathbf{t}')$ is a non-progressing trace pair if $P\mathbf{t}$ and $Q\mathbf{t}'$ occur on the left of S and S' respectively and $P\mathbf{t} \equiv Q\mathbf{t}'$, where the equivalence is equality modulo any substitution applied by the rule R.

To implement this notion in CYCLIST$_{\text{FO}}$, each atomic formula $P\mathbf{t}$ in the left of a sequent is annotated with a natural number, called its *tag*. Then for any conclusion sequent S and rule R we use these tags to attach to each premise S' the lists of progressing and non-progressing trace pairs associated with (S, R, S'). Similarly, matching functions are also required to return lists of (usually non-progressing) trace pairs for matching sequents.

Example 5. The following example shows how the premises of an instance of (L_E) are extended with lists of progressing and non-progressing trace pairs (in that order), where the numeric subscripts on atomic formulas are tags:

$$\frac{[\,[\,(N_1x, y = 0 \vdash G, [], [(1,1)])\ ;\ (N_1x, y = sz, O_3z \vdash G, [(2,3)], [(1,1)])\,]\,]}{N_1x, E_2y \vdash G}\,(L_E)$$

Thus, in the right hand premise, the first list indicates that $(2, 3)$, denoting the formulas E_2y in the conclusion and O_3y in the premise, is a progressing trace pair, and the second list indicates that $(1, 1)$, denoting the formulas N_1x occurring in both conclusion and premise, is a non-progressing trace pair. The left hand premise is similar, except that there are no progressing trace pairs. □

2.6 Soundness of cyclic proofs and decision procedures. It is clear that a pre-proof may not be sound, e.g., a sequent back-linked to itself. The following definition captures a sufficient condition of soundness.

Definition 3 (Cyclic proof). A pre-proof $\mathcal{P}$ in a cyclic proof system is said to be a *(cyclic) proof* if, for every infinite path $(v_i)_{i\geq 0}$ in $\mathcal{P}$, there is a tail of the path, $\pi = (v_i)_{i\geq n}$, such that there is an infinitely progressing trace following π.

Our trace-based condition qualifying pre-proofs as proofs follows the one by Sprenger and Dam [21], who showed that their trace condition for the first-order μ-calculus subsumed a number of previous formulations by others. Analogous trace conditions were adopted for other logics in [4, 6, 7]. Sprenger and Dam also established that their trace condition was decidable, a result we extend to the generic notion of trace given by Defn. 2.

Theorem 4 (Decidability of soundness condition). *In any cyclic proof system* $(\mathcal{S}, \mathcal{R}, \mathcal{T}, \delta)$ *it is decidable whether or not a pre-proof is a cyclic proof.*

Proof. (Sketch) From a given pre-proof $\mathcal{P}$ we construct two Büchi automata over strings of nodes of $\mathcal{P}$. The *path automaton* $\mathcal{A}_{Path}$ simply accepts all infinite paths in $\mathcal{P}$. The *trace automaton* $\mathcal{A}_{Trace}$ accepts all infinite paths in $\mathcal{P}$ such that an infinitely progressing trace exists on some tail of the path. $\mathcal{P}$ is then a proof if and only if $\mathcal{A}_{Trace}$ accepts all strings accepted by $\mathcal{A}_{Path}$. We are then done since inclusion between the languages of Büchi automata is known to be decidable. The full details appear as Appendix A of [5]. □

Checking that a pre-proof $\mathcal{P}$ satisfies the soundness condition on cyclic proofs (Defn. 3) amounts to checking language inclusion between two Büchi automata $\mathcal{A}_{Path}$ and $\mathcal{A}_{Trace}$ constructed from $\mathcal{P}$ (see the proof of Theorem 4). We implement this check as a function that, given a Cyclist pre-proof, constructs the two automata and then uses a model checker to decide language inclusion.

We use *transition-labelled* Büchi automata [11] in constructing $\mathcal{A}_{Path}$ and $\mathcal{A}_{Trace}$, as they allow the most succinct representation. We represent such an automaton as a directed graph with labelled edges, where (u, v, l, n) with $n \in \{0, 1\}$ describes an edge from u to v accepting the label l. The automaton accepts any infinite string of labels such that edges with $n = 1$ are visited infinitely often. The path automaton $\mathcal{A}_{Path}$ accepts all infinite paths in $\mathcal{P}$, and thus it has an edge $(u, v, v, 1)$ for every edge (u, v) of $\mathcal{P}$ (viewing $\mathcal{P}$ as a graph in the obvious way). The trace automaton $\mathcal{A}_{Trace}$ is more complicated, and built using both the node identifiers of $\mathcal{P}$ and the trace pair information attached to rule instances as described above. Essentially, $\mathcal{A}_{Trace}$ accepts any infinite path through $\mathcal{P}$ that eventually (a) is decorated with trace values that agree with the trace pair function and (b) goes through a progressing trace pair infinitely often. Thus, in particular, $\mathcal{A}_{Trace}$ contains an edge $((u, \alpha_1), (v, \alpha_2), v, n)$ whenever (u, v) is an edge of $\mathcal{P}$ and (α_1, α_2) is a trace pair annotating the corresponding rule instance in $\mathcal{P}$, with $n = 1$ if (α_1, α_2) is progressing and $n = 0$ otherwise. For full details of the construction, see Appendix A of [5].

Our model checker is built using Spot [13], an open-source C++ library for building custom, on-the-fly model checkers. We also provide an OCaml interface between Cyclist and the model-checking C++ code.

Checking inclusion between Büchi automata is computationally expensive, as it entails complementing one of the automata, which can lead to an explosion in the number of states [15]. Thus readers may wonder whether the general infinitary soundness condition on cyclic proofs ought to be discarded in favour

of a stronger but simpler condition. The following (admittedly artificial) example is intended to show that a fairly complex proof condition is in fact needed.

Example 6. Define a binary predicate R via the following inductive rules:

$$\Rightarrow R(0,y) \qquad R(x,0) \Rightarrow R(sx,0) \qquad R(ssx,y) \Rightarrow R(sx,sy)$$

The following is a cyclic proof of the sequent $Nx, Ny \vdash R(x,y)$, where N is the natural number predicate defined in Example 1 (for brevity, we omit applications of equality rules, contraction and weakening):

$$\dfrac{\dfrac{}{\vdash R(0,y)}(R_R) \qquad \dfrac{\dfrac{\dfrac{(*)\ \overline{Nx}, \underline{Ny} \vdash R(x,y)}{\overline{Nx'}, \underline{N0} \vdash R(x',0)}\text{(Subst)}}{\overline{Nx'}, \underline{N0} \vdash R(sx',0)}(R_R) \qquad \dfrac{\dfrac{\dfrac{(*)\ Nx, \underline{Ny} \vdash R(x,y)}{Nssx', \underline{Ny'} \vdash R(ssx',y')}\text{(Subst)}}{Nx', \underline{Ny'} \vdash R(ssx',y')}\text{(Cut)}}{Nx', \underline{Ny'} \vdash R(sx',sy')}(R_R)}{\overline{Nx'}, \underline{Ny} \vdash R(sx',y)}(L_N)}{(*)\ \overline{Nx}, \underline{Ny} \vdash R(x,y)}(L_N)$$

where we suppress the easy proof that $Nx' \vdash Nssx'$ in the instance of (Cut) on the right hand branch. The leaves marked $(*)$ are both back-linked to the root.

To see that this pre-proof is in fact a cyclic proof, we must show that any infinite path π has a tail on which an infinitely progressing trace exists. There are two cases to consider. First, if π has a tail consisting entirely of repetitions of the left-hand loop, then we can form a trace following this tail given by the overlined formulas, which progresses (infinitely often) at the first application of (L_N). Otherwise, π must traverse the right-hand loop infinitely often (and might also traverse the left-hand loop infinitely often). In that case, we can form a trace following π given by the underlined formulas, which progresses (infinitely often) at the second application of (L_N). □

Cyclist$_{\text{FO}}$ is in fact capable of proving the above example. We note that the overlapping of cycles in this example is essentially unavoidable, and that we are forced to select different traces for the left-hand cycle depending on the order in which these overlapping cycles are traversed. Thus, the proof condition cannot be restated in this case as a simpler property to be satisfied by each cycle individually. However, this proof does satisfy Brotherston's condition of having a "trace manifold", which is stated in terms of connected sets of cycles [4, 5].

2.7 Soundness of cyclic proof systems. Although our implementation of cyclic proof naturally deals only with the syntactic notion of provability given by Defn. 3, we shall nevertheless outline here how soundness of a cyclic proof system may be established. We assume a set $\mathcal{I}$ of *interpretations* of sequents, which are functions from $\mathcal{S}$ into $\{\text{true}, \text{false}\}$; we write $I \models S$ to mean $I(S) = \text{true}$. S is called *valid* if $I \models S$ for all $I \in \mathcal{I}$.

Definition 5 (Ordinal trace function). An *ordinal trace function* for a cyclic proof system $(\mathcal{S}, \mathcal{R}, \mathcal{T}, \delta)$ and interpretations $\mathcal{I}$ is a function $\sigma : (\mathcal{T} \times \mathcal{I}) \to \mathcal{O}$,

```
applyrule(rule,proof,node) :
begin
    result := [];
    applications := rule(node);
    foreach subgoallist in applications do
        (proof',subgoalnodes) :=
           replacenode(proof, node, subgoallist, rule);
        result := (proof',subgoalnodes) :: result;
    end
    return result;
end

backlink (matchfun,proof,node) :
begin
    result := [];
    foreach node' in proof do
        if matchfun node node' then
            proof' := linknode(proof,node,node',matchfun);
            if sound(proof') then result := (proof', []) :: result;
        end
    end
    return result;
end

proofsearch(bound,proof,node) :
begin
    if closed(node) then return proof;
    if bound=0 then return nil;
    foreach rule in ruleset do
        if rule is a matching function then
            results := backlink(rule, proof, node);
        else
            results := applyrule(rule, proof, node);
        end
        foreach (proof', subgoalnodes) in results do
            p' := proof';
            foreach node' in subgoalnodes do
                p' := proofsearch(bound-1,p',node');
                if p'=nil then break;
            end
            if p'=nil then return nil else return p';
        end
    end
end
```

Fig. 2. Pseudocode for proof search in CYCLIST.

where $\mathcal{O}$ is an initial segment of the ordinals, satisfying the following conditions for all $I \in \mathcal{I}$ and $S \in \mathcal{S}$:

$$\begin{array}{l} \text{if } I \not\models S \text{ then } \exists S' \in \mathcal{S}, R \in \mathcal{R}, I' \in \mathcal{I}. \\ \qquad I' \not\models S' \text{ and } (S, R, S') \in \text{dom}(\delta) \text{ and} \\ \qquad\qquad \text{if } (\alpha, \alpha', n) \in \delta(S, R, S') \text{ then } \begin{cases} \sigma(\alpha', I') \le \sigma(\alpha, I) & \text{if } n = 0 \\ \sigma(\alpha', I') < \sigma(\alpha, I) & \text{if } n = 1 \end{cases} \end{array}$$

We note that the existence of an ordinal trace function subsumes local soundness of the proof rules, because of the requirement in Definition 5 that falsifiability of the conclusion of a rule implies falsifiability of one of its premises.

In the case of first-order logic, it is well known that an inductive predicate P can be generated semantically via a chain of ordinal-indexed *approximants* $(P^\gamma)_{\gamma \ge 0}$. Here, given a suitable interpretation I the ordinal trace function $\sigma(P\mathbf{t}, I)$ returns the smallest γ such that $I \models P^\gamma \mathbf{t}$. See e.g. [5, 4, 9] for details.

Theorem 6 (Soundness). *Suppose there exists an ordinal trace function for* $(\mathcal{S}, \mathcal{R}, \mathcal{T}, \delta)$ *and* $\mathcal{I}$*. Then, if* S *has a cyclic proof, then* S *is valid.*

Proof. (Sketch) Let $\mathcal{P}$ be a cyclic proof of S, and suppose for contradiction that $I \not\models S$. Using local soundness of the rules, we can construct an infinite path $\pi = (v_j)_{j \ge 0}$ in $\mathcal{P}$ and an infinite sequence $(I_j)_{j \ge 0}$ of interpretations such that $I_j \not\models S(v_j)$ for all $j \ge 0$. Since $\mathcal{P}$ is a cyclic proof, there exists an infinitely progressing trace $(\alpha_j)_{j \ge n}$ following some tail $(v_j)_{j \ge n}$ of π. It follows from Definition 5 that the sequence $(\sigma(\alpha_j, I_j))_{æ \ge n}$ is monotonically decreasing, and strictly decreases infinitely often. This contradicts the well-foundedness of $\mathcal{O}$. $\square$

2.8 Proof search. Provided with the appropriate descriptions of sequents, inductive definitions and inference rules, CYCLIST instantiates a proof search function, `proofsearch()`, shown in pseudo-code in Figure 2. This function, given

a proof, a node within that proof and a maximum recursion depth, performs an iterative depth-first search aiming at closing open nodes in the proof. The global variable "ruleset" provides the ordered list of inference rules and matching functions defined by the user; the functions `replacenode()` and `linknode()` do the requisite graph surgery in order to replace an open node in the proof with either the application of an inference rule or a back-link, respectively. Finally, the function `sound()` checks the global soundness of a cyclic proof. The design and trade-offs regarding this algorithm will be further discussed in Section 4.

3 Separation logic instantiations of Cyclist

We briefly present the two instantiations of Cyclist based on separation logic.

3.1 Separation logic entailment prover. $\textsc{Cyclist}_{\mathrm{SL}}$ is a prover for separation logic similar to the prover in [8]. The syntax (left) and semantics (right) of the $\forall\exists$ DNF-like fragment of separation logic the prover accepts appear below.

$$\begin{array}{rl} t ::= & x \mid \mathsf{nil} \\ \alpha ::= & t = t \\ & \mid t \neq t \\ & \mid \mathsf{emp} \\ & \mid t \mapsto \langle t, \ldots, t \rangle \\ & \mid P(t, \ldots, t) \\ H ::= & \alpha \mid H * H \\ F ::= & H \\ & \mid F \vee F \\ & \mid \exists x.F \end{array}$$

$$\begin{array}{l} s(\mathsf{nil}) \notin \mathrm{dom}(h),\ \text{for all } s, h \\ s, h \models x = y \text{ iff } s(x) = s(y) \\ s, h \models x \neq y \text{ iff } s, h \not\models x = y \\ s, h \models \mathsf{emp} \text{ iff } h = \emptyset \\ s, h \models a_0 \mapsto \langle a_1, \ldots, a_n \rangle \text{ iff } h = \{s(a_0) \mapsto (s(a_1), \ldots, s(a_n))\} \\ s, h \models H_1 * H_2 \text{ iff } \exists \text{ domain-disjoint } h_1, h_2,\ \text{s.t.} \\ \qquad s, h_1 \models H_1 \text{ and } s, h_2 \models H_2 \text{ and } h = h_1 \circ h_2 \\ s, h \models F_1 \vee F_2 \text{ iff } s, h \models F_1 \text{ or } s, h \models F_2 \\ s, h \models \exists x.F \text{ iff } \exists v.\, s[x \mapsto v], h \models H \end{array}$$

where stacks s are functions from variables to values, heaps h are finite partial maps from addresses to value tuples (where addresses are a subset of values) and $\circ$ is disjoint union. The semantics of inductive predicates are standard [6, 7].

Inductive predicates are defined in a manner similar to that in $\textsc{Cyclist}_{\mathrm{FO}}$. For example, an acyclic, possibly empty, singly-linked list segment is defined as:

$$(a_1 = a_2) \Rightarrow ls(a_1, a_2) \qquad (a_1 \neq a_2) * a_1 \mapsto \langle e_3 \rangle * ls(e_3, a_2) \Rightarrow ls(a_1, a_2)$$

Left- and right-unfolding rules are generated as in $\textsc{Cyclist}_{\mathrm{FO}}$. Back-linking is also as in $\textsc{Cyclist}_{\mathrm{FO}}$, except that classical weakening is replaced by the spatial weakening of separation logic, captured by the rule $B \vdash C \Longrightarrow A * B \vdash A * C$.

3.2 Separation logic termination prover. $\textsc{Cyclist}_{\mathrm{Term}}$ implements a termination prover for heap-manipulating programs in a simple imperative language, the theory of which was presented in [7]. By way of illustrating the programming language, a program that traverses a linked list is as follows.

```
0: if a1=nil goto 3;   1: a1 := a1→next;   2: goto 0;   3: stop.
```

Sequents are of the form $F \vdash_i \downarrow$, where F is a precondition in separation logic as in $\textsc{Cyclist}_{\mathrm{SL}}$, and i is the line of the program to which the sequent applies. Such

a sequent expresses the fact that if execution starts with the program counter set to i at a state satisfying F, then the program will (safely) terminate. For example, the sequent $ls(a_1, \mathsf{nil}) \vdash_0 \downarrow$ means that the above program will terminate if started at line 0 with a heap satisfying $ls(a_1, \mathsf{nil})$.

Cyclist$_{\text{Term}}$ builds on Cyclist$_{\text{SL}}$. Additional are rules for the symbolic execution of commands, derived via weakest preconditions. Unfolding rules for inductive predicates are generated in a manner similar to that in Cyclist$_{\text{SL}}$ apart from the fact that there are no right-unfold rules. Back-linking is also similar to that in Cyclist$_{\text{SL}}$, except that in Cyclist$_{\text{Term}}$ the program counters in the sequents must also match (exactly). We note that Cyclist$_{\text{Term}}$ is not a program analysis as it lacks abstraction capability.

4 Proof search issues and experimental results

Designing a proof search procedure for a cyclic theorem prover poses some design challenges distinct to those of standard proof search. Here we discuss the main issues, and report on our tests of Cyclist's proof search performance.

4.1 Global search strategy. Non-ancestral back-links, i.e. back-links that point to a sequent which is not an ancestor of the back-link, can significantly reduce the depth of a proof [4]. Thus it is reasonable to conjecture that a breadth-first search might find these shorter proofs, and consequently yield a faster search algorithm than depth-first. Our early experiments overwhelmingly favoured the latter. We conjecture that the high fan-out degree of the search space makes breadth-first search impractical, even though shorter proofs may be found. Also, employing a depth-first strategy will allow some non-ancestral back-links 'to the left' of the current subgoal but also to open subgoals 'to the right' of the current subgoal, thus representing a reasonable compromise. A best-first strategy might perform better and we intend to pursue this question in future work.

4.2 Soundness checking. Invoking a model checker to check the soundness of a pre-proof can be a costly step during proof search. To mitigate this we employ an abstraction/minimisation heuristic that reduces the size of the proof graphs to be checked by pruning leaf subgoals and composing certain types of successive arcs. In the context of iterative depth-first search we also memoise the results of these checks so as to avoid duplication of effort. This led to an order of magnitude of reduction in the cost of the soundness check, and is reflected in the low proportion of time spent checking soundness in our tests (see Table 4).

4.3 Forming back-links. When a partial pre-proof is found to be unsound then we know that it can never form part of a sound, closed proof. Thus we have the choice of either checking soundness once when the proof is closed, or to apply the check eagerly, i.e. every time a back-link is formed. Our tests showed a clear advantage in the eager soundness checking strategy under both depth- and breadth-first search schemes. We conjecture that early elimination of an unsound proof leads to a major reduction of the size of the search space outweighing the cost of frequent soundness checking, especially after our optimisations.

It is known that the set of sequents provable with the use of non-ancestral back-links is equal to that with back-linking restricted to ancestor nodes [4]. This raises the question whether using only ancestral back-links improves performance, due to a smaller number of calls to matching functions and soundness checks. Restricting back-links to ancestral nodes does not speed up the instantiations we provide, but makes some proofs impractical. It seems that the matching functions we use will not fire significantly more often when allowed access to non-ancestral nodes, and thus will not lead to excessive soundness checking.

4.4 Order of rule applications. As in most theorem provers, the order in which inference rules/tactics are attempted directly impacts performance. We list here two points specific to cyclic theorem proving. First, when matching functions are computationally cheap, they can be prioritised and attempted early and often, eagerly creating back-links. Used within tactics such as fold-then-match, they can entail a higher computational cost and are thus placed last in the priority order. Second, unfolding rules generally increase the size of sequents, thus have lower priority than other inference rules. In particular, left-unfolding precedes right-unfolding as it introduces progressing trace pairs in the cyclic proof, and, it may (after simplification) enable right-unfolding rules to fire.

4.5 Predicate folding/lemma application. It seems certain that Cut elimination does not hold, in general, for cyclic proof systems. Thus the ability to conjecture and apply lemmas can be crucial to a successful proof, as is the case, e.g., in our Example 6 above. Our instantiations of CYCLIST do not yet permit the application of arbitrary lemmas. Instead, we currently permit only *predicate foldings*, where the lemma applied is essentially an inductive rule. For example, the inductive rule $Add(x, y, z) \Rightarrow Add(sx, y, sx)$ from Example 1 becomes the "folding" lemma $Add(x, y, z) \vdash Add(sx, y, sx)$. We found empirically that this very limited form of lemma application is very useful in quite a number of proofs.

4.6 Limitations. CYCLIST is a young framework aimed at proving theorems with a complex inductive structure. As such, it does not yet utilise the totality of existing know-how on theorem proving, and this entails some limitations.

Focussing on inductive predicates means that function declaration and related equational reasoning facilities are lacking. As a result $\text{CYCLIST}_{\text{FO}}$ has difficulty dealing with heavily-equational goals, since such goals have to be translated into a predicate-based language resulting in loss of structural information.

Another limitation is that, although we do provide a predicate folding facility as explained above, we have no functionality currently for applying general lemmas, and this restricts the ability of CYCLIST instantiations to prove theorems that must rely on the use of Cut in their proofs.

A well-known example that is unprovable as yet in $\text{CYCLIST}_{\text{FO}}$ and demonstrates both limitations is the commutativity of addition. In $\text{CYCLIST}_{\text{FO}}$ this goal can be expressed relationally as $Nx, Ny, Add(x, y, z) \vdash Add(y, x, z)$. This form discourages the use of rewriting techniques guided by the structure of terms. In addition, the cyclic proof of the theorem requires essentially the same lemma,

$x + sy = s(x+y)$, as is needed for the standard inductive proof (relationally, this lemma can be stated as $Nx, Ny, Add(x,y,z) \vdash Add(x,sy,sz)$). In standard inductive theorem provers, this lemma would be supplied as a "hint" to the prover, or would be found by an appropriate conjecture mechanism (cf. [16]).

4.7 Experimental results. The results of tests run on the three instantiations of CYCLIST are summarised in Table 4. All tests were run on a x64 Linux system with an Intel i5 3.33GHz. CYCLIST and all tests are available online at [1].

CYCLIST$_{\text{FO}}$. We ran a number of tests with the first-order prover, mainly involving natural number induction. The two most interesting theorems we managed to prove are "the P & Q example" [24], and the sequent appearing in Example 6. Both proofs have a complex inductive structure, multiple cycles and require the use of predicate folding. They are both found in under a second. It is notable that Example 6 uses a lemma ($Nx \vdash Nssx$) that is not an instance of folding (it represents a "double fold"). CYCLIST$_{\text{FO}}$ proves this theorem by finding a deeper proof that requires only single folds.

CYCLIST$_{\text{SL}}$. The prover was run on the test cases from [8]. Proving time is nearly zero for most, suggesting that CYCLIST$_{\text{SL}}$ could be used as a backend for program analysis that automatically handles arbitrary inductive datatypes.

CYCLIST$_{\text{Term}}$. We ran the termination checker on a number of small programs including the programs in [7]. Notable are an iterative binary-tree search (program B in Table 4) and the reversal of a frying-pan list (program C, last theorem in Table 4). The authors of [3] report that the MUTANT tool for separation logic, which deals only with lists, fails to prove the latter theorem (under an appropriate precondition). A cyclic termination proof was later presented in [7] where it was painstakingly constructed by hand. CYCLIST$_{\text{Term}}$ proves this in under a second. Its proof contains five cycles, all requiring predicate folding.

5 Related work

There are a few theorem provers employing cyclic proof in some form. The QUODLIBET tool [2], based on first-order logic with inductive datatypes, uses a version of infinite descent to prove inductive theorems whereby a proof node is annotated with a *weight*, which must strictly decrease at back-link sites. Compared to CYCLIST, which is fully automatic, QUODLIBET is intended for semi-interactive proof. An automated cyclic prover for entailments of separation logic, implemented in HOL Light, appeared in [8]. Compared to CYCLIST$_{\text{SL}}$, the prover in [8] disallows non-ancestral back-links and uses a restricted soundness condition, which in particular rules out the use of predicate folding. Nguyen and Chin [19] provide a separation logic entailment prover using cyclic proof, but which appears to be restricted in at least as many ways as [8].

In summary, the main restrictions on previous cyclic provers are: (a) a single logical setting; (b) ancestral cycle schemes; (c) strong soundness conditions that rule out many proofs; and (d) automated search limited to cut-free proofs. CYCLIST lifts all of these restrictions, albeit only partially in the case of (d).

	Theorem	Time	SC%	Depth	Nodes	Uns./All
	$O_1 x \vdash Nx$	16	0	5	7	0/1
	$E_1 x \vee O_2 x \vdash Nx$	20	0	6	15	4/6
	$E_1 x \vee O_1 x \vdash Nx$	16	25	4	9	2/4
	$N_1 x \vdash Ox \vee Ex$	12	0	4	6	0/1
	$N_1 x \wedge N_2 y \vdash Q(x, y)$	512	31	7	13	171/181
	$N_1 x \vdash Add(x, 0, x)$	4	0	3	5	0/1
	$N_1 x \wedge N_2 y \wedge Add_3(x, y, z) \vdash Nz$	24	0	4	6	3/4
	$N_1 x \wedge N_2 y \wedge Add_3(x, y, z) \vdash Add(x, s(y), s(z))$	40	20	5	12	8/9
	$N_1 x \wedge N_2 y \vdash R(x, y)$	560	44	7	26	176/183
	$x \mapsto y * RList_1(y, z) \vdash RList(x, z)$	16	0	5	8	0/1
	$RList_1(x, y) * RList_2(y, z) \vdash RList(x, z)$	16	0	4	7	0/1
	$List_1(x, y) * y \mapsto z \vdash List(x, z)$	8	0	4	6	0/1
	$List_1(x, y) * List_2(y, z) \vdash List(x, z)$	8	0	3	5	0/1
	$PeList_1(x, y) * y \mapsto z \vdash PeList(x, z)$	12	0	4	6	0/1
	$PeList_1(x, y) * PeList_2(y, z) \vdash PeList(x, z)$	12	0	3	4	0/1
	$DLL_1(x, y, z, w) \vdash SLL(x, y)$	12	0	3	5	0/1
	$DLL_1(x, y, z, w) \vdash BSLL(z, w)$	12	0	4	6	0/1
	$DLL_1(x, y, z, w) * DLL_2(a, x, w, b) \vdash DLL(a, y, z, b)$	8	0	3	4	0/1
	$ListO_1(x, y) * ListO_2(y, z) \vdash ListE(x, z)$	12	0	5	12	0/1
	$ListE_1(x, y) * ListE_2(y, z) \vdash ListE(x, z)$	20	0	5	8	0/1
	$ListE_1(x, y) * ListO_2(y, z) \vdash ListO(x, z)$	24	0	5	8	0/1
	$BinListFirst_1 x \vdash BinTreex$	8	0	4	6	0/1
	$BinListSecond_1 x \vdash BinTreex$	20	0	4	6	0/1
	$BinPath_1(x, z) * BinPath_2(z, y) \vdash BinPath(x, y)$	24	0	3	6	0/2
	$BinPath_1(x, y) \vdash BinTreeSeg(x, y)$	16	0	4	8	0/2
	$BinTreeSeg_1(x, z) * BinTreeSeg_2(z, y) \vdash BinTreeSeg(x, y)$	12	0	3	6	0/2
	$BinTreeSeg_1(x, y) * BinTreey \vdash BinTree(x)$	12	0	3	6	0/2
	$x \neq z * x \mapsto y * ls_1(y, z) \vdash ls(x, z)$	0	0	2	2	0/0
	$ls_1(x, y) * ls_2(y, \mathsf{nil}) \vdash ls(x, \mathsf{nil})$	16	0	3	4	0/1
	$ListE_1(x, y) \vee ListO_1(x, y) \vdash List(x, y)$	16	0	4	9	2/4
A:	$ls_1(x, \mathsf{nil}) \vdash_0 \downarrow$	16	0	5	7	0/1
B:	$btx \vdash_0 \downarrow$	12	0	6	13	0/2
C:	$ls_1(x, \mathsf{nil}) * ls_2(y, \mathsf{nil}) \vdash_1 \downarrow$	52	8	8	10	13/14
D:	$y \neq \mathsf{nil} * ls_1(x, \mathsf{nil}) * ls_2(y, \mathsf{nil}) \vdash_0 \downarrow$	2036	16	12	24	197/233
	$ls(x, z) * ls(y, \mathsf{nil}) * z \mapsto a * ls(a, z)$					
	$\vee\ ls(b, \mathsf{nil}) * z \mapsto b * ls(x, z) * ls(y, z)$					
C:	$\vee\ ls(x, \mathsf{nil}) * ls(y, z) * z \mapsto c * ls(c, z) \ \vdash_1 \downarrow$	124	0	9	39	19/23

A

```
// List traversal
0: if x=nil goto 3;
1: x := x→next;
2: goto 0;
3: stop
```

B

```
// Bin. tree search
0: if x=nil goto 6;
1: if * goto 4;
2: x := x→left ;
3: goto 0 ;
4: x := x→right ;
5: goto 0 ;
6: stop
```

C

```
// List reversal
0: y := nil;
1: if x=nil goto 7;
2: z := x;
3: x := x→next;
4: z→next := y;
5: y := z;
6: goto 1;
7: stop
```

D

```
// List append
// (one-at-a-time)
 0: if x=nil goto 10;
 1: z := y→next;
 2: if z≠nil goto 8;
 3: y→next := x;
 4: x := x→next;
 5: y := y→next;
 6: y→next := nil;
 7: goto 0;
 8: y := y→next;
 9: goto 0;
10: stop
```

Table 1. *Upper:* Theorems proved by the instantiations. The column labelled 'Time' is the time taken in milliseconds, 'SC%' is the percentage of time taken by the soundness checks, 'Depth' is the depth of the proof found, 'Nodes' is the number of nodes in the proof and the last column shows the number of calls to the model checker as (calls on unsound proof)/(total calls). *Lower:* The input programs to the termination prover. NB the formulas used for program **C** are loop invariants and as such the program counter in the judgment is set to 1, i.e., a statement in the loop.

The "size change principle" for program termination by Lee et al [18] is based on a condition similar to the soundness condition for cyclic proofs: a program terminates if every possible infinite execution in the control flow graph would result in an infinite descent of some well-founded data value. It is plausible that the approach of [18] to termination checking, empirically shown to often be more efficient in practice than a Büchi automata construction [14], would also benefit the soundness checking in CYCLIST. However, in contrast to size-change termination problems, the main problem we face is not in checking the soundness condition, but in discovering the correct candidate pre-proofs.

Finally, there are a number of mature, automated theorem provers employing explicit induction, including ACL2 [17], IsaPlanner [12], LambdaOtter and many others. Unfortunately, most test suites for these provers are largely based on equational reasoning about functions over inductive datatypes, whereas our instantiations of CYCLIST currently only cater for inductively defined predicates, making a direct comparison difficult. These tools will most probably outperform ours on problems requiring extensive rewriting, generalisation or the application of non-trivial lemmas. On the other hand, CYCLIST performs well on small problems requiring complex induction schemes, which are typically problematic for explicit induction (cf. Example 6). Thus we believe that integrating the sophisticated non-inductive features of explicit-induction provers into CYCLIST might yield significant benefits. For example, conjecturing appropriate lemmas (cf. [16]) seems extremely useful in forming back-links during proof search.

6 Conclusions and future work

The main contributions of this paper are our generic theory of cyclic proof, its unrestricted implementation in our theorem prover CYCLIST, and the application of CYCLIST to three concrete logical systems, including automated proof search procedures. In particular, we provide the first implementation of the cyclic proof system for program termination proposed in [7]. We believe that CYCLIST represents the first fully general implementation of cyclic proof.

Although CYCLIST is by no means an industrial-strength theorem prover, the results of our experiments to date are nevertheless encouraging. In its various instantiations, the prover is capable of automatically proving theorems with a complex inductive structure, notable Wirth's "P&Q" example, the proof of in-place reversal of a "frying-pan" list from [7], and our own Example 6.

There are obvious directions in which CYCLIST could be improved, both at the generic level (e.g. function definition over datatypes, rewriting support, lemma application and generalisation mechanisms) and in its various instantiations (e.g. more advanced search strategies for particular logics). There is also the potential for developing new instantiations of CYCLIST to other fixed-point logics, such as the μ-calculus or temporal logics. We warmly encourage the development of such instantiations by interested readers.

References

1. Cyclist framework download. `http://www.cs.ucl.ac.uk/staff/ngorogia/`
2. Avenhaus, J., Kühler, U., Schmidt-Samoa, T., Wirth, C.P.: How to prove inductive theorems? QuodLibet! In: CADE-19. LNAI 2741, pp. 328–333. Springer (2003)
3. Berdine, J., Cook, B., Distefano, D., O'Hearn, P.W.: Automatic termination proofs for programs with shape-shifting heaps. In: CAV-18. LNCS 4144, pp. 386–400. Springer (2006)
4. Brotherston, J.: Cyclic proofs for first-order logic with inductive definitions. In: TABLEAUX-14. LNAI 3702, pp. 78–92. Springer-Verlag (2005)
5. Brotherston, J.: Sequent Calculus Proof Systems for Inductive Definitions. Ph.D. thesis, University of Edinburgh (November 2006)
6. Brotherston, J.: Formalised inductive reasoning in the logic of bunched implications. In: SAS-14. LNCS, vol. 4634, pp. 87–103. Springer-Verlag (2007)
7. Brotherston, J., Bornat, R., Calcagno, C.: Cyclic proofs of program termination in separation logic. In: POPL-35, pp. 101–112. ACM (2008)
8. Brotherston, J., Distefano, D., Petersen, R.L.: Automated cyclic entailment proofs in separation logic. In: CADE-23. LNAI 6803, pp. 131–146. Springer (2011)
9. Brotherston, J., Simpson, A.: Sequent calculi for induction and infinite descent. Journal of Logic and Computation 21(6), pp. 1177–1216, (Dec 2011).
10. Bundy, A.: The automation of proof by mathematical induction. In: Handbook of Automated Reasoning, vol. I, chap. 13, pp. 845–911. Elsevier Science (2001)
11. Couvreur, J.M.: On-the-fly verification of linear temporal logic. In: FM. pp. 253–271. Springer-Verlag (1999)
12. Dixon, L., Fleuriot, J.: IsaPlanner: A prototype proof planner in Isabelle. In: CADE'03. pp. 279–283 (2003)
13. Duret-Lutz, A., Poitrenaud, D.: Spot: An extensible model checking library using transition-based generalized Büchi automata. In: MASCOTS. pp. 76–83. (2004)
14. Fogarty, S., Vardi, M.: Büchi complementation and size-change termination. In: TACAS-15. LNCS 5505, pp. 16–30 (2009)
15. Friedgut, E., Kupferman, O., Vardi, M.Y.: Büchi complementation made tighter. In: 2nd Int. Symp. on Automated Technology for Verification and Analysis (2004)
16. Johansson, M., Dixon, L., Bundy, A.: Conjecture synthesis for inductive theories. Journal of Automated Reasoning 47(3) (Oct 2011)
17. Kaufmann, M., Manolios, P., Moore, J.S.: Computer-Aided Reasoning: An Approach. Kluwer (2000)
18. Lee, C.S., Jones, N.D., Ben-Amram, A.M.: The size-change principle for program termination. In: POPL-28, pp. 81–92. ACM (2001)
19. Nguyen, H.H., Chin, W.N.: Enhancing program verification with lemmas. In: CAV-20. LNCS 5123, pp. 355–369. Springer (2008)
20. Schöpp, U., Simpson, A.: Verifying temporal properties using explicit approximants: Completeness for context-free processes. In: FOSSACS-5. LNCS 2303, pp. 372–386. Springer (2002)
21. Sprenger, C., Dam, M.: A note on global induction mechanisms in a μ-calculus with explicit approximations. Theor. Informatics and Applications 37, 365–399 (2003)
22. Sprenger, C., Dam, M.: On the structure of inductive reasoning: circular and tree-shaped proofs in the μ-calculus. In: FOSSACS-6. LNCS 2620, pp. 425–440. Springer (2003)
23. Stirling, C., Walker, D.: Local model checking in the modal μ-calculus. Theoretical Computer Science 89, 161–177 (1991)
24. Wirth, C.P.: Descente infinie + Deduction. Logic J. of the IGPL 12(1), 1–96 (2004)