Automatic Cyclic Termination Proofs for Recursive Procedures in Separation Logic

Reuben N. S. Rowe
Department of Computer Science
University College London
r.rowe@ucl.ac.uk

James Brotherston
Department of Computer Science
University College London
j.brotherston@ucl.ac.uk

In recent years, it has been shown that the principle of *cyclic proof* (see, e.g., [12, 13, 3]) can be used to verify the (safe) termination of heap-manipulating, pointer-based programs: Brotherston et al. [4] present a cyclic Hoare-style proof system for deriving pre-conditions sufficient for termination of programs in a simple imperative language with `while` loops, expressed in the well-known *symbolic heap* fragment of *separation logic* [11], where user-defined inductive predicates are used to describe data structures in memory (see e.g. [2]). In this system, termination measures are always obtained from (combinations of) semantic *approximations* of the inductive predicates used in the proof. The potential benefits of using cyclic proof for verification include the automatic discovery of termination measures and inductive loop invariants.

Our aim is to demonstrate that such cyclic proof-based verification techniques can be scaled to compete with other existing approaches to verifying termination of pointer-based code, such as those implemented by tools such as AProVE [8] and HipTNT+ [9]. To that end, we provide a tool implementation for automatically proving the termination of procedural, heap-manipulating programs, such as the following benchmark from the Termination Problems Database.

```
void shuffle(Node x) {
  if x != NULL {
    y := x.nxt; reverse(y); shuffle(y);
    x.nxt := y;
  }
}
```

This procedure rearranges the elements of a null-terminated linked list in memory referenced by the program variable `x`. Intuitively, we can infer termination of `shuffle` for two reasons: the call to `reverse` acts on the local variable `y` which references a smaller linked list; and, assuming (as might be expected) that `reverse` does not change the number of elements in the list, the recursive call to `shuffle` also acts on a smaller list. Thus, this example presents non-trivial challenges for automatic termination provers, since the termination of `shuffle` depends on the `reverse` procedure not increasing the termination measure.

The core of our approach is a *cyclic* Hoare-style proof system for total correctness, which extends the aforementioned cyclic proof system for simple `while` programs [4]. We continue to obtain termination measures from semantic *approximations* of the inductive predicates, however to handle procedures we extend the proof system in order to be able to track how these approximations are affected by procedure calls. For this, our formalism uses explicit ordinal variables, e.g., specifying `reverse` as follows:

$$\{\mathsf{List}_\alpha(\mathtt{x})\}\ \mathtt{reverse(x)}\ \{\mathsf{List}_\alpha(\mathtt{x})\}$$

Here $\mathsf{List}(x)$ is an inductively defined predicate of separation logic describing null-terminated linked lists with head pointer x, and α is an ordinal variable referring to an (under-)approximation of this predicate – unchanged by the procedure – that in this context can intuitively be read simply as the length of the list. In other cases, we might write, e.g., $\{P_\alpha(x)\}\,\mathtt{myproc(x)}\,\{\exists \beta < \alpha.\ Q_\beta(x)\}$ if it happens that `myproc` actually decreases the measure referred to by α.

Proofs in our system are *cyclic proofs*, which are standard finite derivation trees, but with some leaves possibly closed by *back-links* to identical interior nodes; to ensure soundness of such proofs, a *global soundness condition* (decidable by automata-theoretic methods) is imposed on the proof structure, which amounts to ensuring that all infinite paths in the proof correspond to valid arguments by *infinite descent*, cf. [6]. In our case, the global soundness condition ensures that some ordinal termination measure decreases infinitely often on every infinite path in the proof structure. In this sense, our technique is closely related to *size-change termination* [10], which attempts to extract similarly well-founded measures directly from data manipulated by the program.

Our tool [1], given as input a set of inductive predicate definitions and a Hoare triple, searches for a cyclic termination proof, implementing a fully-automatic procedure. The tool is built on top of Cyclist [7], a general framework for implementing cyclic theorem provers which provides a general search procedure for cyclic proofs (details may be found in [5]). Logic-specific theorem provers can be obtained by instantiating this general procedure with the appropriate syntax, proof rules and tactics implementing a particular proof system. Cyclist produces a cyclic proof object as output.

The proof-search strategy that we implement in our tool is largely guided by the syntax of the input program: we attempt to apply as many symbolic execution rules as possible; when no symbolic execution rules can be applied, we perform

TAPAS 2016, September 7th 2016, Edinburgh, UK
ACM ISBN .
DOI:

predicate unfoldings until sufficient structure is revealed in the current symbolic state to allow further symbolic execution. When all commands have been symbolically executed, we must decide whether the post-condition is entailed by the current symbolic state. To decide such entailment questions our tool uses a separate instantiation of Cyclist with an entailment proof system for our logic. Thus it is capable of proving entailments that require inductive reasoning.

Loops and recursion are handled naturally in cyclic proof by back-linking. When a procedure call or loop is first encountered during symbolic execution, it is *unrolled*. On subsequent symbolic iterations of the loop or recursions/calls of the procedure, our tactics will typically attempt to form a back-link to a node created by a previous encounter with the loop/procedure. Global soundness of the generated proof is checked incrementally as new back-links are formed.

Our implementation currently relies on procedures being annotated with pre-/post-conditions, which is necessary for the *compositional* treatment of procedure calls: it allows proofs verifying procedures to be reused at multiple call-sites. Unfolding a procedure call, then, requires us to determine whether the current state contains a portion which is 'left over' from that described by the procedure specification, i.e. *frame inference*. We perform frame inference by unfolding predicates in the call site or back-linked leaf precondition, up to some pre-specified limit, until syntactic matching of atomic formulas is possible. A candidate frame can then be computed by subtracting the matching atomic formulas from the current symbolic state. To verify that this is indeed a frame, we then conjoin it to the post-condition of the procedure specification or back-link companion, and perform a similar unfold-and-match procedure. Frame inference may fail, not only in the case that no frame exists but also if we do not unfold sufficiently many times or because inductive reasoning is required. Since our tool encounters many potential back-link candidates during proof search, this simple unfold-and-match approach provides a relatively cheap if somewhat weak solution to frame inference. Empirically we have observed this is an effective trade-off when combined with a more powerful procedure for deciding entailments at axioms as well as other locations specified by the user.

We tested our tool against two state-of-the-art termination provers for heap-manipulating code. HipTNT+ [9] extends a Hoare-style separation logic system with temporal operators expressing termination and both possible and definite non-termination. Like our work, it requires pre/post-condition annotations for heap-manipulating procedures and arithmetic parameters (comparable to our ordinal-valued labels) must be incorporated into inductive predicates. The AProVE tool [8] handles termination of C code and Java bytecode by transforming it into a term rewriting system and applying existing techniques for proving termination of these. As such, it does not require procedures to be annotated, but it does consider termination due to the raising of a checked exception as safe termination.

The results of our experiments are shown in Figures 1 and 2; the AProVE benchmarks are from the Recursive Java Bytecode suite in the latest version (10.3) of the Termination Problems Database [14], and include non-trivial recursion schemes. For example, the `Alternate` benchmark contains a procedure that creates a binary tree using the left and right subtrees of its two inputs in an alternating fashion; the recursive calls also swap the arguments and so proving termination requires a lexicographic measure, which is discovered by our tool. Other examples require complex reasoning about the shape of the heap resulting from procedure calls; these include the `SharingAnalysisRec`, `CyclicAnalysisRec`, and `TwoWay` benchmarks, with the latter two requiring to prove that a cyclic heap structure and non-terminating loop, respectively, is unreachable. Since our tool does not currently support numeric features we focussed on examples containing solely or predominantly heap-manipulating features, however we did model some arithmetic-based control flow using linked-lists to stand for natural numbers; this includes an implementation of the Ackermann function. Compared with HipTNT+ and AProVE, our tool displayed much shorter execution times on almost all of the examples in the benchmark suite. The annotation burden in our system is similar to that required by HipTNT+.

Benchmark Suite	Test	Time (seconds) AProVE	Cyclist
Costa_Julia_09-Recursive	Ackermann	3.82	0.14
	BinarySearchTree	1.41	0.95
	BTree	1.77	0.03
	List	1.43	1.74
Julia_10-Recursive	AckR	3.22	0.14
	BTreeR	2.68	0.03
	Test8	2.95	0.97
AProVE_11_Recursive	CyclicAnalysisRec	2.61	5.21
	RotateTree	5.86	0.32
	SharingAnalysisRec	2.47	4.72
	UnionFind	TIMEOUT	1.21
BOG_RTA_11	Alternate	5.47	1.47
	AppE	2.19	0.09
	BinTreeChanger	3.38	3.33
	CAppE	2.04	1.78
	ConvertRec	3.72	0.06
	DupTreeRec	4.18	0.03
	GrowTreeR	3.53	0.05
	MirrorBinTreeRec	4.96	0.02
	MirrorMultiTreeRec	5.16	0.63
	SearchTreeR	2.74	0.34
	Shuffle	11.72	0.21
	TwoWay	1.94	0.02

Figure 1: Comparison with AProVE Benchmarks

Benchmark	Time (seconds) HipTNT+	Cyclist
traverse acyclic linked list	0.31	0.02
traverse cyclic linked list	0.52	0.02
append acyclic linked lists	0.36	0.03
TPDB Shuffle	1.79	0.21
TPDB Alternate	6.33	1.47
TPDB UnionFind	4.03	1.21

Figure 2: Comparison with HipTNT+ Benchmarks

Our results demonstrate that cyclic proof is indeed a viable approach for verifying termination of heap-manipulating programs. Key to its practicability is the ability to treat procedures directly (as opposed to in-lining) and compositionally; we show how this can be achieved within the cyclic proof framework. Our tool puts cyclic proof-based termination reasoning roughly on a par with the current state-of-the-art, and provides a platform for developing the technique further. In future work, we intend to investigate the possibility of inferring the ordinal labels and corresponding constraints; we believe that this information can be extracted from the structure of a cyclic proof. We also intend to investigate the possibility of inferring entire pre-/post-condition specifications, most probably using *bi-abduction*.

1. REFERENCES

[1] https://github.com/ngorogiannis/cyclist/tree/ordinal_labels.

[2] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Symbolic execution with separation logic. In *Proc. APLAS-3*, volume 3780 of *LNCS*, pages 52–68. Springer, 2005.

[3] James Brotherston. Cyclic proofs for first-order logic with inductive definitions. In *Proceedings of TABLEAUX-14*, volume 3702 of *LNAI*, pages 78–92. Springer-Verlag, 2005.

[4] James Brotherston, Richard Bornat, and Cristiano Calcagno. Cyclic proofs of program termination in separation logic. In *Proceedings of POPL-35*, pages 101–112. ACM, 2008.

[5] James Brotherston, Nikos Gorogiannis, and Rasmus L. Petersen. A generic cyclic theorem prover. In *Proceedings of APLAS-10*, LNCS, pages 350–367. Springer, 2012.

[6] James Brotherston and Alex Simpson. Sequent calculi for induction and infinite descent. *Journal of Logic and Computation*, 21(6):1177–1216, December 2011.

[7] Cyclist: software distribution. https://github.com/ngorogiannis/cyclist/.

[8] Jürgen Giesl, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, Thomas Ströder, Stephanie Swiderski, and René Thiemann. Proving termination of programs automatically with AProVE. In *IJCAR-7*, pages 184–191, 2014.

[9] Ton Chanh Le, Shengchao Qin, and Wei-Ngan Chin. Termination and non-termination specification inference. In *Proceedings of PLDI-15*, pages 489–498, 2015.

[10] Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. In *Proceedings of POPL*, pages 81–92. ACM, 2001.

[11] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of LICS-17*, pages 55–74. IEEE Computer Society, 2002.

[12] Luigi Santocanale. A calculus of circular proofs and its categorical semantics. In *Foundations of Software Science and Conputation Structures*, volume 2303 of *LNCS*, pages 357–371. Springer-Verlag, 2002.

[13] Christoph Sprenger and Mads Dam. On the structure of inductive reasoning: circular and tree-shaped proofs in the μ-calculus. In *Proceedings of FOSSACS-6*, volume 2620 of *LNCS*, pages 425–440. Springer-Verlag, 2003.

[14] Termination problems database. http://termination-portal.org/wiki/TPDB.