CALCO'07, LNCS 4624, pp 372-378. 2007

CIRC: A Circular Coinductive Prover*

Dorel Lucanu[1] and Grigore Roşu[2]

[1] Faculty of Computer Science
Alexandru Ioan Cuza University, Iaşi, Romania, dlucanu@info.uaic.ro
[2] Department of Computer Science
University of Illinois at Urbana-Champaign, USA, grosu@cs.uiuc.edu

Abstract. CIRC is an automated circular coinductive prover implemented as an extension of Maude. The circular coinductive technique that forms the core of CIRC is discussed, together with a high-level implementation using metalevel capabilities of rewriting logic. To reflect the strength of CIRC in automatically proving behavioral properties, an example defining and proving properties about infinite streams of infinite binary trees is shown. CIRC also provides limited support for automated inductive proving, which can be used in combination with coinduction.

1 Introduction

Behavioral abstraction in algebraic specification appears under various names in the literature such as *hidden algebra* [5], *observational logic* [8], *swinging types* [11], *coherent hidden algebra* [2], *hidden logic* [12], and so on. Most of these approaches appeared as a need to extend algebraic specifications to ease the process of specifying and verifying designs of systems and also for various other reasons, such as, to naturally handle infinite types[3], to give semantics to the object paradigm, to specify finitely otherwise infinitely axiomatizable abstract data types, etc. The main characteristic of these approaches is that sorts are split into *visible* (or *observational*) for data and *hidden* for states, and the equality is behavioral, in the sense that two states are *behaviorally equivalent* if and only if they *appear* to be the same under any visible *experiment*.

Coalgebraic *bisimulation* [9] and *context induction* [7] are sound proof techniques for behavioral equivalence. Unfortunately, both need human intervention: coinduction to pick a "good" bisimulation relation, and context induction to invent and prove lemmas. *Circular coinduction* [3, 12] is an automatic proof technique for behavioral equivalence. By circular coinduction one can prove, for example, the equality $zip(zeros, ones) = blink$ on streams as follows (*zeros* is the stream 0^ω, *ones* is 1^ω, *blink* is $(01)^\omega$, *zip* merges two streams):

1. check that the two streams have the same head, 0;
2. take the tail of the two streams and generate the new goal $zip(ones, zeros) = 1{:}blink$; this becomes the next task;
3. check that the two new streams have the same head, 1;

* This work is partially supported by the CNCSIS project 1162/2007, by grants NSF CCF-0448501 and NSF CNS-0509321, and by Microsoft Research gifts.

[3] I.e., types whose values are infinite structures.

4. take the tail of the two new streams; after simplification one gets the new goal *zip*(*zeros*, *ones*) = *blink*, which is nothing but the original proof task;
5. conclude that *zip*(*zeros*, *ones*) = *blink* holds.

The intuition for the above "proof" is that the two streams have been exhaustively tried to be distinguished by iteratively checking their heads and taking their tails. Ending up in circles (we obtained the same new proof task as the original one) means that the two streams are indistinguishable, so equal.

Circular coinduction can be explained and proved correct by reducing it to either bisimulation or context induction: it iteratively constructs a bisimulation, but it also discovers all lemmas needed by a context induction proof. Since the behavioral equivalence problem is Π_2^0-complete even for streams [13], there is *no* algorithm complete for behavioral equality in general, as well as *no* algorithm complete for inequality of streams. Therefore, the best we can do is to focus our efforts on exploring heuristics or deduction rules to prove or disprove equalities of streams that *work well on examples of interest* rather than in general.

BOBJ [3, 12] was the first system supporting circular coinduction. Hausmann, Mossakowski, and Schröder [6] also developed circular coinductive techniques and tactics in the context of CoCASL. In this paper we propose CIRC, an automated circular coinductive prover implemented in Full Maude as a behavioral extension of the Maude system [1], making heavy use of meta-level and reflection capabilities of rewriting logic. Maude is by now a very mature system, with many uses, a high-performance rewrite engine, and a broad spectrum of analysis tools. Maude's current meta-level capabilities were not available when the BOBJ system was developed; consequently, BOBJ was a heavy system, with rather poor parsing and performance. By allowing the entire Maude system visible to the user, CIRC inherits all Maude's uses, performance and analysis tools.

CIRC implements the *circularity principle*, which generalizes circular coinductive deduction [4] and can be informally described as follows (a formalization of this principle, together with the related technical details will be discussed elsewhere). Assume that each equation of interest (to be proved) e admits a *frozen form* $fr(e)$, which is an equation associated to e defined possibly over an extended signature, and a set $Der(e)$ of equations, also defined possibly over an extended signature, called its *derivatives*. The circularity principle requires that the following rule be valid: if from the hypotheses $\mathcal{H}$ together with $fr(e)$ we can deduce $Der(e)$, then e is a consequence of $\mathcal{H}$. When $fr(e)$ freezes the equation at the top as in [4], the circularity principle becomes circular coinduction. Interestingly, when the equation is frozen at the bottom on a variable, then it becomes a structural induction (on that variable) derivation rule. This way, CIRC supports both coinduction and induction as projections of a more general principle. In this paper, we concentrate only on CIRC's coinductive capabilities.

2 Behavioral Algebraic Specifications

We assume the reader familiar with algebraic specification and Maude [1]. Here we intuitively present behavioral algebraic specification using an example. Infinite binary trees can be specified using three *behavioral operations* (observers): `node(`T`)` returning the information from the root of T, `left(`T`)` and `right(`T`)` returning the left child and the right child of T, respectively. Examples of experiments for the sort `BTree` of infinite binary trees are `node(*:BTree)`, `node(left(*:BTree))`, `node(right(*:BTree))`, and so on. The sort `BTree` is called *hidden*, and the sort `Elt` (i.e., the result sort of `node`) is called *visible* w.r.t. `BTree`. The result sort of the experiments is always visible. We next define a "mirror" operation over infinite binary trees. We further consider streams of infinite binary trees. A stream is specified by two behavioral operations: `hd(`S`)` returning the first element of stream S (in our case an infinite binary tree), and `tl(`S`)` returning the stream obtained by removing the first element of S. Examples of stream experiments are `hd(*:TStream)`, `hd(tl(*:TStream))`, `hd(tl(tl(*:TStream)))`, and so on. Note that `BTree` is visible w.r.t. `TStream`. We define the following stream operations: a constructor `cons(`T, S`)` inserting the tree T in front of S, *blink*(T_1, T_2) defining the stream $(T_1, T_2, T_1, T_2, \ldots)$, and *const*$(T)$ defining the constant stream $(T, T, T, \ldots)$. These infinite data structures can be defined in Full Maude as follows:

```
(th TSTREAM is
  sorts Elt BTree TStream .
  var S : TStream . var E : Elt .
  vars T T1 T2 : BTree .
  ops left right : BTree -> BTree .
  op node : BTree -> Elt .
  op mirror : BTree -> BTree .
  eq left(mirror(T)) =
     mirror(right(T)) .
  eq right(mirror(T)) =
     mirror(left(T)) .
  eq node(mirror(T)) = node(T) .
  op hd : TStream -> BTree .
  op tl : TStream -> TStream .
  op cons : BTree TStream -> TStream .
  eq hd(cons(T, S)) = T .
  eq tl(cons(T, S)) = S .
  op blink : BTree BTree -> TStream .
  eq hd(blink(T1, T2)) = T1 .
  eq tl(blink(T1, T2)) =
     cons(T2, blink(T1,T2)) .
  op const : BTree -> TStream .
  eq hd(const(T)) = T .
  eq tl(const(T)) = const(T) .
endth)
```

Trees T_1, T_2 are *behaviorally equivalent* iff `node(`T_1`)` $=$ `node(`T_2`)`, `node(left(`T_1`))` $=$ `node(left(`T_2`))`, `node(right(`T_1`))` $=$ `node(right(`T_2`))`, and so on. Similarly, streams S_1 and S_2 are behaviorally equivalent iff `hd(`S_1`)` $=$ `hd(`S_2`)`, `hd(tl(`S_1`))` $=$ `hd(tl(`S_2`))`, `hd(tl(tl(`S_1`)))` $=$ `hd(tl(tl(`S_2`)))`, and so on. For instance, `mirror(mirror(`T`))` and T are behaviorally equivalent. We write `TSTREAM` $\models\!\!\!\equiv$ `mirror(mirror(`T`))` $= T$. Also, `TSTREAM` $\models\!\!\!\equiv$ `blink(`S, S`)` $= S$.

Hence, a *behavioral specification* is an algebraic specification together with a specified subset of *behavioral operators*, which are used to define the crucial notion of *behavioral equivalence* as "indistinguishability under experiments."

3 CIRC

We here describe the underlying proving technique of CIRC and show how one can use the system. The CIRC prover can be downloaded from its website at [10].

As already mentioned, CIRC implements what we call "the principle of circularity," which generalizes both structural induction and circular coinduction, and which will be discussed in more detail elsewhere. We here only focus on its coinductive instance. Circular coinduction [3, 12, 4] is a sound proof calculus for $\equiv\!\!\!\!|$, which can be defined as an instance of the principle of circularity as follows:

- for any sort s, let us extend the signature with a new operation $\mathit{fr} : s \to \texttt{new}$, where `new` is a new sort;
- for each equation e of the form "$(\forall X)\ t = t'$ `if` c," let
  - $\mathit{fr}(e)$ be the equation "$(\forall X)\mathit{fr}(t) = \mathit{fr}(t')$ `if` c," which we call the "frozen" form of e; and
  - $\mathit{Der}(e)$ be the set of equations $\{(\forall X)\ \mathit{fr}(\delta[t/*{:}h]) = \mathit{fr}(\delta[t'/*{:}h])$ `if` $c \mid \delta$ behavioral for $h = \mathit{sort}(t)\}$, which we call the "derivatives" of e.

We say that $\mathit{fr}(e)$ "freezes e at the top" in the context of coinduction; this freezing operation ensures the sound use of the coinduction hypotheses in the procedure below, because it prevents the application of congruence inference steps [12, 4]. We take the liberty to also call $\mathit{fr}(e)$ *visible* when e is visible.

Any automatic proving procedure based on circular coinduction is parametric in a procedure for equational deduction. In CIRC we use the standard rewriting-based semi-decision procedure to derive equations "$(\forall X)\ t = t'$ `if` c": add the variables X as constants, then add the conditions in c to the set of equations, and then reduce t, t' to their normal forms $\mathit{nf}(t)$ and $\mathit{nf}(t')$, respectively, orienting all the equations into rewrite rules. In what follows we let $\mathcal{E} \vdash e$ denote the fact that e can be deduced from $\mathcal{E}$ using this standard approach ($\mathcal{E}$ is any set of equations). By $\mathcal{E} \not\vdash e$ we mean "knowingly incapable of proving it," that is, that the rewrite engine reduced the two terms to normal forms, but those were not equal. Obviously, this does not necessarily mean that the equation is not true.

Suppose that $\mathcal{B}$ is a behavioral specification whose equations form a set E and that e is an equation to prove. CIRC takes $\mathcal{B}$ and e as input and aims at proving $\mathcal{B} \equiv\!\!\!\!|\, e$. CIRC maintains and iteratively reduces a pair of sets of equations of the form $(\mathcal{E}, \mathcal{G})$, where $\mathcal{E}$ is the set of equations that are assumed to hold and $\mathcal{G}$ is the set of "goals," that is, the set of equations that still need to be proved. Therefore, CIRC aims at reducing the pair $(E, \mathit{fr}(e))$ to a pair of the form $(\mathcal{E}', \emptyset)$, i.e., one whose set of goals is empty. If successful, then $\mathcal{B} \equiv\!\!\!\!|\, e$; moreover, (the unfrozen variants of) all the equations in $\mathcal{E}'$ are behavioral consequences of $\mathcal{B}$. While trying to perform its task, CIRC's procedure can also fail, in which case we conclude that it could not prove $\mathcal{B} \equiv\!\!\!\!|\, e$, or it can run forever. Here are the reduction rules currently supported by CIRC:

[Equational Reduction] :

$$(\mathcal{E}, \mathcal{G} \cup \{\mathit{fr}(e)\}) \Rightarrow (\mathcal{E}, \mathcal{G}) \quad \text{if } \mathcal{E} \vdash \mathit{fr}(e)$$

[Coinduction Failure] :

$$(\mathcal{E}, \mathcal{G} \cup \{\mathit{fr}(e)\}) \Rightarrow \mathit{fail} \quad \text{if } \mathcal{E} \not\vdash \mathit{fr}(e) \text{ and } e \text{ is visible}$$

[Circular Coinduction] :

$$(\mathcal{E}, \mathcal{G} \cup \{\mathit{fr}(e)\}) \Rightarrow (\mathcal{E} \cup \{\mathit{nf}(\mathit{fr}(e))\}, \mathcal{G} \cup \mathit{Der}(e)) \text{ if } \mathcal{E} \not\vdash \mathit{fr}(e) \text{ and } e \text{ is hidden.}$$

$\mathit{nf}(e)$ denotes the equation e whose left-hand and right-hand sides are reduced to normal forms. [Equational Reduction] removes a goal if it can be proved using ordinary equational reduction. [Coinduction Failure] says that the procedure fails whenever it finds a visible goal which cannot be proved using ordinary equational reduction. Finally, [Circular Coinduction] implements the circularity principle: when a hidden equation cannot be proved using ordinary equational reduction, its frozen form (its normal form is an equivalent variant) is added to the specification and its derivatives are added to the set of goals.

The termination of the CIRC procedure above, i.e., reaching of a configuration of the form $(\mathcal{E}', \emptyset)$, is not guaranteed. Since the behavioral entailment problem is Π_2^0 [12, 13], we know that there can be no procedure to decide behavioral equalities or inequalities in general.

The remainder of this section shows, by means of our example with streams of infinite trees, how CIRC can be used in practice. Since a behavioral specification includes more information than a usual Full Maude specification, we designed an interface allowing the user to introduce behavioral operations, case sentences, goals, and CIRC commands. A typical scenario of using CIRC is as follows. The objects whose behavioral properties are investigated are described using Full Maude specifications, like the module `TSTREAM` given above. Note that a Full Maude specification does not explicitly say which are the behavioral operations. Therefore we consider a new type of module, delimited by the keywords `cmod` and `endcm`, used to describe the behavioral specifications. Such a module includes importing commands (for object specifications), behavioral operation (derivatives) declarations, and case sentences (not discussed here):

```
(cmod B-TSTREAM is
  importing TSTREAM .
  derivative left(*:BTree) right(*:BTree) node(*:BTree) .
  derivative hd(*:TStream) derivative tl(*:TStream) .
endcm)
```

To prove $blink(mirror(mirror(T)), T) = const(T)$, we introduce it as a new goal:

```
Maude>(add goal blink(mirror(mirror(T:BTree)),T:BTree)=const(T:BTree) .)
Goal blink(mirror(mirror(T:BTree)),T:BTree) =  const(T:BTree) added.
```

Then we can specify the tactic we want CIRC to use; for example:

```
Maude> (coinduction .)
Maude> rewrites: ... (ommited for space reasons)
Proof succeeded.
```

This command triggers the iterative execution of the three reduction rules. This property requires two coinduction proofs: one for streams and the other one for infinite trees. Note that the technique in [6] fails to automatically prove the above property because the user must explicitly tell the system where the second coinduction proof starts. Since the termination is not guaranteed, the "`coinduction`" tactic can also be given a depth (no depth needed here).

Several other examples have been experimented with in CIRC and can be found and run on CIRC's webpage at [10], including the following: proving behavioral equalities, i.e., language equivalence of regular expressions extended with complement; proving a series of known properties of infinite streams and operations on them; proving properties of powerlists; proving equivalent definitions of Fibonacci numbers equivalent; proving non-trivial properties about the Morse sequence; proving inductive and coinductive properties about finite or infinite trees. In all these examples, the important property to prove has been captured as an equation to be entailed by a specification extended with derivatives.

4 Conclusion

We presented CIRC, an automated prover supporting the principle of circularity and in particular circular coinduction. CIRC is implemented as an extension of Maude using its metalevel programming capabilities. An example was also discussed, reflecting the strength of CIRC in automatically proving behavioral properties. The technical details and proofs will be discussed elsewhere.

Acknowledgment. We warmly thank Andrei Popescu for help with the implementation of the first version of CIRC and for ideas on combining induction and coinduction via the principle of circularity, and to the anonymous referees for insightful comments.

References

1. M. Clavel, F. J. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and Programming in Rewriting Logic. *Theoretical Computer Science*, 285:187–243, 2002. Also, extended Maude manual available at `http://maude.csl.sri.com` and `http://maude.cs.uiuc.edu`.
2. R. Diaconescu and K. Futatsugi. Behavioral coherence in object-oriented algebraic specification. *JUCS*, 6(1):74–96, 2000.
3. J. Goguen, K. Lin, and G. Roşu. Circular coinductive rewriting. In *Proceedings of Automated Software Engineering 2000*, pages 123–131. IEEE, 2000.
4. J. Goguen, K. Lin, and G. Roşu. Conditional circular coinductive rewriting with case analysis. In *WADT'02*, volume 2755 of *LNCS*, pages 216–232. Springer, 2003.
5. J. Goguen and G. Malcolm. A hidden agenda. *J. of TCS*, 245(1):55–101, 2000.
6. D. Hausmann, T. Mossakowski, and L. Schröder. Iterative circular coinduction for CoCASL in Isabelle/HOL. In *Fundamental Approaches to Software Engineering 2005*, volume 3442 of *LNCS*, pages 341–356. Springer, 2005.
7. R. Hennicker. Context induction: a proof principle for behavioral abstractions. *Formal Aspects of Computing*, 3(4):326–345, 1991.
8. R. Hennicker and M. Bidoit. Observational logic. In *Proceedings of AMAST'98*, volume 1548 of *LNCS*, pages 263–277. Springer, 1999.
9. B. Jacobs and J. Rutten. A tutorial on (co)algebras and (co)induction. *Bulletin of the European Association for Theoretical Computer Science*, 62:222–259, 1997.
10. D. Lucanu and G. Roşu. CIRC prover. `http://fsl.cs.uiuc.edu/index.php/Circ`.
11. P. Padawitz. Swinging data types: Syntax, semantics, and theory. In *Proceedings, WADT'95*, volume 1130 of *LNCS*, pages 409–435. Springer, 1996.
12. G. Roşu. *Hidden Logic.* PhD thesis, University of California at San Diego, 2000.
13. G. Roşu. Equality of streams is a Π_2^0-complete problem. In *the 11th ACM SIGPLAN Int. Conf. on Functional Programming (ICFP'06).* ACM, 2006.