CIRC: A Behavioral Verification Tool based on Circular Coinduction

Dorel Lucanu[1], Eugen-Ioan Goriac[1], Georgiana Caltais[1], and Grigore Roşu[2]

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

Abstract. CIRC is a tool for automated inductive and coinductive theorem proving. It includes an engine based on circular coinduction, which makes CIRC particularly well-suited for proving behavioral properties of infinite data-structures. This paper presents the current status of the coinductive features of the CIRC prover, focusing on new features added over the last two years. The presentation is by examples, showing how CIRC can automatically prove behavioral properties.

1 Introduction

The first version of the coinductive engine of CIRC was presented in [7]. Meanwhile, important contributions have been made to the tool. One of the contributions is the implementation of the deterministic language ROC! [3] used for specifying proof strategies. This language provides a flexible way to combine basic proof actions, such as *reduction* and *coinduction expansion*, into coinductive proof schemata. ROC! was designed so that the combination of different proving techniques is possible.

Another contribution was the extension of the tool with *special contexts* [6], as explained below. An important technical aspect of our previous implementation of circular coinduction in CIRC [7] (see also [8]) was a freezing operator $\boxed{\cdot}$ used to add frozen versions $\boxed{e}$ of dynamically discovered coinductive hypotheses e to the behavioral specification. The role of the freezing operator is to enforce the use of coinductive hypotheses in a sound manner, forbidding their use in contextual reasoning. However, many experiments with CIRC have shown that this constraint to completely forbid the use of coinductive hypotheses in all contexts is too strong, in that some contexts are actually safe. The novel special context extension of CIRC allows the user to specify contexts in which the frozen coinductive hypotheses can be used. CIRC can be used to check that those contexts are indeed safe. Moreover, the current version of CIRC includes an algorithm for automatically computing a set of special contexts, which are then, also automatically, used in circular coinductive proofs.

The CIRC tool, user manual and many examples can be downloaded from `http://fsl.cs.uiuc.edu/CIRC`. CIRC can also be used online through its web interface at `http://fsl.cs.uiuc.edu/index.php/Special:CircOnline`.

The theoretical foundation for the coinductive engine included in CIRC is presented in [8]. So, many concepts and notations used in this paper are presented in detail there. In this paper we present a set of examples aiming to illustrate the main features of CIRC. These include non-trivial properties of streams, bisimulation of automata, how to handle goals when a possible infinite proof is detected, the use of special contexts.

2 Behavioral Specifications and CIRC

Behavioral specifications are pairs of the form $(\mathcal{B}, \Delta)$, where $\mathcal{B} = (S, \Sigma, E)$ is a many sorted algebraic specification and Δ is a set of Σ-contexts, called *derivatives*. A derivative in Δ is written as $\delta[*{:}h]$, where $*{:}h$ is a special variable of sort h designating the place-holder in the context δ. The sorts S are split in two classes: *hidden sorts*, $H = \{h \mid \delta[*{:}h] \in \Delta\}$, and *visible sorts*, $V = S \setminus H$. A Δ-*context* is inductively defined as follows: 1) each $\delta[*{:}h] \in \Delta$ is a context for h; and 2) if $C[*{:}h']$ is a a context for h' and $\delta[*{:}h] \in \Delta_{h'}$, then $C[\delta[*{:}h]]$ is a context for h, where $\Delta_{h'}$ is the subset of derivatives of sort h'. A Δ-*experiment* is a Δ-context of visible sort. If e is an equation of the form $(\forall X)t = t'$ and C a Δ-context appropriate for t and t', then $C[e]$ denotes the equation $(\forall X)C[t] = C[t']$. If $\delta \in \Delta$, then $\delta[e]$ is called a *derivative* of e. Given an entailment relation $\vdash$ over $\mathcal{B}$, the *behavioral entailment* relation is defined as follows: $\mathcal{B} \Vvdash e$ iff $\mathcal{B} \vdash C[e]$ for each Δ-context C appropriate for the equation e. In this case, we say that $\mathcal{B}$ *behaviorally satisfies* the equation e. The reader is referred to [8] for more rigorous definitions, properties and other technical details.

Several of the examples we present in this paper are based on infinite streams. To specify the streams, we consider two sorts: a hidden sort *Stream* for the streams and a visible sort *Data* for the stream elements. The streams are defined in terms of head and tail, i.e., $hd(*{:}Stream)$ and $tl(*{:}Stream)$ are derivatives. For instance, the stream $(1:0:1)^\infty = 1:0:1:1:0:1\ldots$, which is mathematically defined by the equation $ozo = 1{:}0{:}1{:}ozo$, is behaviorally specified by the following equations written in terms of head and tail: $hd(ozo) = 1$, $hd(tl(ozo)) = 0$, $hd(tl^2(ozo)) = 1$ and $tl^3(ozo) = ozo$. The specifications of other hidden structures are obtained in a similar manner, by defining their behavior in terms of the derivatives.

CIRC is developed as an extension of the Full Maude language. The underlying entailment relation used in CIRC is $\mathcal{E} \vdash_{\rightarrow\leftarrow} (\forall X)t = t'$ `if` $\wedge_{i \in I}(u_i = v_i)$ iff $\mathrm{nf}(t) = \mathrm{nf}(t')$, where $\mathrm{nf}(t)$ is computed as follows:

– the variables X of the equations are turned into fresh constants;

– the condition equalities $u_i = v_i$ are added as equations to the specification;

– the equations in the specification are oriented and used as rewrite rules.

The circular coinduction engine implements the proof system given in [8] by a set of reduction rules $(\mathcal{B}, \mathcal{F}, \mathcal{G}) \Rightarrow (\mathcal{B}, \mathcal{F}', \mathcal{G}')$, where $\mathcal{B}$ represents the (original) algebraic specification, $\mathcal{F}$ is the set of frozen axioms and $\mathcal{G}$ is the current set of proof obligations. Here is a brief description of the reduction rules:

[Done]: $(\mathcal{B}, \mathcal{F}, \emptyset) \Rightarrow \cdot$
This rule is applied whenever the set of proof obligations is empty and indicates the termination of the reduction process.
[Reduce]: $(\mathcal{B}, \mathcal{F}, \mathcal{G} \cup \{\boxed{e}\}) \Rightarrow (\mathcal{B}, \mathcal{F}, \mathcal{G})$ `if` $\mathcal{B} \cup \mathcal{F} \vdash_{\rightarrow\leftarrow} e$
This rule is applied whenever the current goal is a $\vdash_{\rightarrow\leftarrow}$-consequence of $\mathcal{B} \cup \mathcal{F}$ and operates by removing $\boxed{e}$ from the set of goals.
[Derive]: $(\mathcal{B}, \mathcal{F}, \mathcal{G} \cup \{\boxed{e}\}) \Rightarrow (\mathcal{B}, \mathcal{F} \cup \{\boxed{e}\}, \mathcal{G} \cup \{\boxed{\Delta(e)}\})$ `if` $\mathcal{B} \cup \mathcal{F} \nvdash_{\rightarrow\leftarrow} e$
This rule is applied when the current goal e is hidden and it is not a $\vdash_{\rightarrow\leftarrow}$-consequence. The current goal is added to the specification and its derivatives to the set of goals. $\boxed{\Delta(e)}$ denotes the set $\{\delta[e] \mid \delta \in \Delta\}$.
[Normalize]: $(\mathcal{B}, \mathcal{F}, \mathcal{G} \cup \{\boxed{e}\}) \Rightarrow (\mathcal{B}, \mathcal{F}, \mathcal{G} \cup \{\boxed{\mathrm{nf}(e)}\})$
This rule removes the current goal from the set of proof obligations and adds its normal form as a new goal. The normal form $\mathrm{nf}(e)$ of an equation e of the form $(\forall X)t = t'$ `if` $\wedge_{i \in I}(u_i = v_i)$ is $(\forall X)\mathrm{nf}(t) = \mathrm{nf}(t')$ `if` $\wedge_{i \in I}(u_i = v_i)$, where the constants from the normal forms are turned back into the corresponding variables.
[Fail]: $(\mathcal{B}, \mathcal{F}, \mathcal{G} \cup \{\boxed{e}\}) \Rightarrow$ *fail* `if` $\mathcal{B} \cup \mathcal{F} \nvdash_{\rightarrow\leftarrow} e$ and e is visible
This rule stops the reduction process with failure whenever the current goal e is visible and the corresponding normal forms are different.

It is easy to see that the reduction rules [Done], [Reduce], and [Derive] implement the proof rules with the same names given in [8]. The reduction rules [Normalize] and [Fail] have no correspondent in the proof system. [Normalize] is directly related to the particular definition for the basic entailment relation used in CIRC. [Fail] signals a failing stop of the reduction process and such a case needs (human) analysis in order to know the source of the failure. The soundness of CIRC follows by showing that the proof system in [8] is equivalent to its extension with the rule:

$$\frac{\mathcal{B} \cup \mathcal{F} \vdash \mathcal{G} \cup \boxed{\mathrm{nf}(e)}}{\mathcal{B} \cup \mathcal{F} \vdash \mathcal{G} \cup \boxed{e}} \quad \text{[Normalize]}$$

The use of CIRC is very simple. First the user must define (or load if already defined) the equational specification $\mathcal{B}$ using a Full Maude-like syntax. For instance, the equational description of streams can be given as follows:

```
(theory STREAM-EQ is
  sorts Data Stream .          op hd : Stream -> Data .
  ops 0 1 : -> Data .          op tl : Stream -> Stream .
  op not : Data -> Data .
  eq not(0) = 1 .              eq not(1) = 0 .
  op ozo : -> Stream  .
  eq hd(ozo) = 0 .             eq hd(tl(tl(ozo))) = 0 .
  eq hd(tl(ozo)) = 1 .         eq tl(tl(tl(ozo))) = ozo .
  ...
endtheory)
```

Since Full Maude has no support for behavioral specifications, CIRC uses a new kind of modules, called *c-theories*, where the specific syntactic constructs are included. Here is a c-theory specifying the derivatives Δ for streams:

```
  (ctheory STREAM is including STREAM-EQ .
    derivative hd(*:Stream) .
    derivative tl(*:Stream) .
  endctheory)
```

As c-theories extend Full-Maude theories, the whole specification $(\mathcal{B}, \Delta)$ may be included into a single c-theory.

The user continues by loading several goals, expressed as (conditional) equations, using the command `(add goal _ .)` and then launches the coinductive proving engine using the command `(coinduction .)`. The coinductive engine of CIRC has three ways to terminate:

- *successful termination*: the initial goals are behavioral consequences of the specification;
- *failing termination*: the system fails to prove a visible equation (in this case we do not know if the initial goals hold or not);
- *the maximum number of steps was exceeded*: either the execution does not terminate or the maximum number of steps is set to a too small value and should be increased.

However, the termination of CIRC is conditioned by the terminating property of the equational specification $\mathcal{B}$. For instance, CIRC does not terminate if Maude falls into a infinite rewriting when it computes a certain normal form.

3 CIRC at work

In this section we present several scenarios on how to interact with the tool in order to prove behavioral properties. The definitions of the stream operations are given by mathematical specifications. The behavioral variants, expressed in terms of head and tail derivatives, are obtained in a similar way to that used for the definition of the stream *ozo* in Section 2.

Example 1. This example illustrates how CIRC implements the proof system given in [8]. The operations *even* and *odd* over streams are defined as follows: $odd(a:s) = a:even(s)$ and $even(a:s) = odd(s)$. Supposing that the file including the c-theory `STREAM` has the name `stream.maude`, we present how the user can verify that $zip(odd(S), even(S)) = S$:

```
Maude> in stream.maude
Maude> (add goal zip(odd(S:Stream), even(S:Stream)) = S:Stream .)
Maude> (coinduction .)
Proof succeeded.
  Number of derived goals: 2
  Number of proving steps performed: 13
  Maximum number of proving steps is set to: 256
Proved properties:
  zip(odd(S:Stream),odd(tl(S:Stream))) = S:Stream
```

From the output we conclude that CIRC needs to prove 2 extra derived subgoals in order to prove the initial property and that it performs 13 basic steps ([Reduce], [Derive], etc). Note that the superior limit for the number of basic steps is set to 256. The command `(set max no steps _ .)` is used to change this number. Exceeding this limit is a good clue for possible infinite computations.

The command `(show proof .)` can be used in order to visualize the applied rules from the proof system:

```
Maude> (show proof .)

|-  [* tl(zip(odd(S),odd(tl(S)))) *] = [* tl(S) *]
---------------------------------------------------------- [Reduce]
|||-  [* tl(zip(odd(S),odd(tl(S)))) *] = [* tl(S) *]


|-  [* hd zip(odd(S),odd(tl(S))) *] = [* hd S *]
---------------------------------------------------------- [Reduce]
|||-  [* hd zip(odd(S),odd(tl(S))) *] = [* hd S *]


1. |||-  [* hd zip(odd(S),odd(tl(S))) *] = [* hd S *]
2. |||-  [* tl(zip(odd(S),odd(tl(S)))) *] = [* tl(S) *]
---------------------------------------------------------- [Derive]
|||-  [* zip(odd(S),odd(tl(S))) *] = [* S *]


|-  [* zip(odd(S),odd(tl(S))) *] = [* S *]
---------------------------------------------------------- [Normalize]
|-  [* zip(odd(S),even(S)) *] = [* S *]
```

Comparing with the proof tree given in [8], we see that the [`Derive`] step is accompanied by a [`Normalize`] step.

Example 2. This is a non-trivial example of coinductive property over streams automatically proved with the circular coinduction. Let s denote the First Feigenbaum sequence: $(1\!:\!0\!:\!1\!:\!1\!:\!1\!:\!0\!:\!1\!:\!0\!:\!1\!:\!0\!:\!1\!:\!1)^\infty$ and *31zip* an operation defined by the equation $\mathit{31zip}(a_0\!:\!a_1\!:\!\ldots, b_0\!:\!b_1\!:\!\ldots) = a_0\!:\!a_1\!:\!a_2\!:\!b_0\!:\!a_3\!:\!a_4\!:\!a_5\!:\!b_1\!:\!\ldots$. We present below how the user can verify that $\mathit{31zip}(\mathit{ozo}, \mathit{ozo}) = s$:

```
Maude> (add goal 31zip(ozo, ozo) = s .)
Maude> (coinduction .)
Proof succeeded.
  Number of derived goals: 24
  Number of proving steps performed: 102
  Maximum number of proving steps is set to: 256
Proved properties:
  tl(tl(tl(31zip(ozo,tl(tl(ozo)))))) =
    tl(tl(tl(tl(tl(tl(tl(tl(tl(tl(tl(s)))))))))))
  tl(tl(31zip(ozo,tl(tl(ozo))))) =
    tl(tl(tl(tl(tl(tl(tl(tl(tl(tl(s))))))))))
  tl(31zip(ozo,tl(tl(ozo)))) = tl(tl(tl(tl(tl(tl(tl(tl(tl(s)))))))))
  31zip(ozo,tl(tl(ozo))) = tl(tl(tl(tl(tl(tl(tl(tl(s))))))))
  tl(tl(tl(31zip(ozo,tl(ozo))))) = tl(tl(tl(tl(tl(tl(tl(s)))))))
  tl(tl(31zip(ozo,tl(ozo)))) = tl(tl(tl(tl(tl(tl(s))))))
  tl(31zip(ozo,tl(ozo))) = tl(tl(tl(tl(tl(s)))))
  31zip(ozo,tl(ozo)) = tl(tl(tl(tl(s))))
  tl(tl(tl(31zip(ozo,ozo)))) = tl(tl(tl(s)))
  tl(tl(31zip(ozo,ozo))) = tl(tl(s))
  tl(31zip(ozo,ozo)) = tl(s)
  31zip(ozo,ozo) = s
```

The "proved properties" constitute the set F of the intermediate lemmas the circular coinduction discovered during the derivation process. The set $\boxed{F}$ of their frozen forms satisfies the Circularity Principle [8]: $\mathtt{STREAM} \cup \boxed{F} \vdash \boxed{\Delta[F]}$, which implies $\mathtt{STREAM} \Vvdash F$.

Example 3. In this example we show how CIRC can be used for proving simultaneous several properties related to the famous Thue-Morse sequence. We first introduce the operations *not*, *zip*, and f, mathematically described by the following equations:

$$\begin{aligned} \mathit{not}(a\!:\!s) &= \mathit{not}(a)\!:\!\mathit{not}(s) \\ \mathit{zip}(a\!:\!s, s') &= a\!:\!\mathit{zip}(s', s) \\ f(a\!:\!s) &= a\!:\!\mathit{not}(a)\!:\!f(s) \end{aligned}$$

Note that the operation *not* is overloaded over data and streams, the right meaning resulting from the type of the argument. The Thue-Morse sequence is $\mathit{morse} = 0\!:\!\mathit{zip}(\mathit{not}(\mathit{morse}), \mathit{tl}(\mathit{morse}))$. It is known that *morse* and its complement are the only fixed points of the function f. If we try to prove this property with CIRC, then we obtain the following output:

```
Maude> (add goal f(morse) = morse .)
Maude> (coinduction .)
Stopped: the number of prover steps was exceeded.
```

Often, this message indicates that the execution of the circular coinductive engine does not terminate for the given goal(s). Analyzing the derived goals (these are obtained either by calling the command `(set show details on .)` before starting the coinduction engine or the command `(show proof .)` at the end), we observe that the fourth derived goal is normalized to $f(\mathit{tl}(\mathit{morse})) = \mathit{zip}(\mathit{tl}(\mathit{morse}), \mathit{not}(\mathit{tl}(\mathit{morse})))$. If we replace the subterm $\mathit{tl}(\mathit{morse})$ with a free variable S, then we obtain a more general lemma: $f(S) = \mathit{zip}(S, \mathit{not}(S))$. CIRC can prove by circular coinduction simultaneously a set of goals. So, we try to simultaneously prove the two properties and we see that this time CIRC successfully terminates:

```
Maude> in stream.maude
Maude> (add goal f(morse) = morse .)
Maude> (add goal f(S:Stream) = zip(S:Stream, not(S:Stream)) .)
Maude> (coinduction .)
Proof succeeded.
  Number of derived goals: 8
  Number of proving steps performed: 41
  Maximum number of proving steps is set to: 256
Proved properties:
  tl(f(morse)) = tl(morse)
  tl(f(S:Stream)) = zip(not(S:Stream),tl(S:Stream))
  f(morse) = morse
  f(S:Stream) = zip(S:Stream,not(S:Stream))
```

A similar reasoning is used for proving the following properties:

$$\mathit{31zip}(\mathit{ozo}, g(\mathit{altMorse})) = g(\mathit{altMorse}) \text{ and } \mathit{31zip}(\mathit{ozo}, g(S)) = g(f(f(S)))$$

where *altMorse* is Thue-Morse sequence but defined using the alternative definition $\mathit{altMorse} = f(\mathit{0}\!:\!\mathit{tl}(\mathit{altMorse}))$ and g is the function over streams defined by $g(a_0\!:\!a_1\!:\!a_2\!:\!\ldots) = (a_0 + a_1)\!:\!g(a_1\!:\!a_2\!:\!\ldots)$. Here is the dialog with CIRC:

```
Maude> (add goal 31zip(ozo, g(altMorse)) = g(altMorse) .)
Maude> (add goal 31zip(ozo, g(S:Stream)) = g(f(f(S:Stream))) .)
Maude> (coinduction .)
Proof succeeded.
  Number of derived goals: 16
  Number of proving steps performed: 79
  Maximum number of proving steps is set to: 256
Proved properties:
  g(tl(f(tl(f(altMorse))))) = g(tl(f(tl(altMorse))))
  tl(tl(tl(31zip(ozo,g(S:Stream))))) = g(tl(f(tl(f(S:Stream)))))
  g(f(tl(f(altMorse)))) = g(f(tl(altMorse)))
  tl(tl(31zip(ozo,g(S:Stream)))) = g(f(tl(f(S:Stream))))
  g(tl(f(f(altMorse)))) = g(tl(altMorse))
  tl(31zip(ozo,g(S:Stream))) = g(tl(f(f(S:Stream))))
  g(f(f(altMorse))) = g(altMorse)
  31zip(ozo,g(S:Stream)) = g(f(f(S:Stream)))
```

The stream $g(\mathit{altMorse})$ is studied in [9] and its equivalent definition with *31zip* and g is given in [1].

Example 4. This example illustrates CIRC capabilities of proving fairness properties. Let g_1, g_2 and g_3 be three infinite streams defined in a mutually-recursive manner by the following equations: $g_1 = 0 : \mathit{not}(g_3)$, $g_2 = 0 : \mathit{not}(g_1)$ and $g_3 = 0 : \mathit{not}(g_2)$. The three streams model a ring of three gates, where the output of each gate is the negation of its current input, except for the initial output which is 0. We introduce the infinite stream *ones* representing the sequence 1^∞, defined by the equation $\mathit{ones} = 1 : \mathit{ones}$. We also consider the operator *extractOnes* that filters all the occurrences of the element 1 when provided a certain stream:

$$\mathit{extractOnes}(b_0 : b_1 : b_2 \ldots) = \begin{cases} 1 : \mathit{extractOnes}(b_1 : b_2 \ldots), & \mathit{if}\ \ b_0 = 1 \\ \mathit{extractOnes}(b_1 : b_2 \ldots), & \mathit{otherwise} \end{cases}$$

We want to use CIRC in order to prove the property that g_1 includes an infinite number of occurrences of the element 1. For proving this property the tool needs the lemma $\mathit{not}(\mathit{not}(a_0 : a_1 \ldots)) = a_0 : a_1 \ldots$ that can be also proved with CIRC in one derivation step and several equational reductions. Finally, the goal we want to prove is $\mathit{extractOnes}(g_1) = \mathit{ones}$:

```
  Maude> (add goal extractOnes(g1) = ones .)
  Maude> (coinduction .)
  Proof succeeded.
    Number of derived goals: 6
    Number of proving steps performed: 30
    Maximum number of proving steps is set to: 256
  Proved properties:
    extractOnes(g3) = ones
    extractOnes(g2) = ones
    extractOnes(g1) = ones
```

From the provided output, one can see that CIRC needed also to automatically prove two other similar fairness properties for the streams g_2 and g_3. However, the specification of *extractOnes* should be used with care because is not always terminating.

Example 5. CIRC is also used for proving automata bisimilarity. In our example, we consider the two deterministic automata presented in Fig. 1.

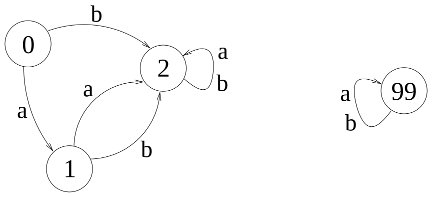

Fig. 1. Two bisimilar automata

The transition function *delta* is defined over states modeled as elements from the hidden sort *State*, and respectively over the alphabet elements (a and b) modeled as elements from the visible sort *Alph*. The function is defined in terms of equations such as: $delta(0, a) = 1$, $delta(0, b) = 2$, $delta(99, a) = 99$, $delta(99, b) = 99$ and so on. In this case *delta* is the only observer, so the derivative set consists of *delta*-derivatives defined for every element from the alphabet: `delta(*:State, a)` and `delta(*:State, b)`. CIRC manages to prove that the state 0 of the first automaton is bisimilar to state 99 of the second automaton:

```
  Maude> (add goal 0 = 99 .)
  Maude> (coinduction .)
  Proof succeeded.
    Number of derived goals: 6
    Number of proving steps performed: 30
    Maximum number of proving steps is set to: 256
  Proved properties:
    2 = 99
    1 = 99
    0 = 99
```

The tool needs in this case to automatically prove two other bisimilarity relations between the states 2 and 99 and respectively 1 and 99.

Example 6. In this scenario we exhibit the use of the special contexts. A particular definition for special contexts was given for the first time in [4]. A more detailed presentation of special contexts and of how they are used to extend circular coinduction is given in [6].

We assume that we want to prove simultaneously the following goals: $morse = altMorse$ and $f(S) = zip(S, not(S))$. Here is dialog with CIRC:

```
Maude> in streams.maude
Maude> (add goal morse = altMorse .)
Maude> (add goal f(S:Stream) = zip(S:Stream, not(S:Stream)) .)
Maude> (coinduction .)
Stopped: the number of prover steps was exceeded.
```

At this moment, the command `(show proof .)` lists all the steps it managed to perform before reaching the limit. Two of the frozen hypotheses added to the specification are: $\boxed{zip(S, not(S))} = \boxed{f(S)}$ and $\boxed{tl(morse)} = \boxed{tl(altMorse)}$. In order to see what is happening, let us consider the complete definitions for *morse* and *altMorse*:

$$\begin{array}{ll}
hd(morse) = 0 & hd(altMorse) = 0 \\
hd(tl(morse)) = 1 & hd(tl(altMorse)) = 1 \\
tl^2(morse) = zip(tl(morse), not(tl(morse))) & tl^2(altMorse) = f(tl(altMorse))
\end{array}$$

CIRC cannot reduce $\boxed{tl^2(morse)} = \boxed{tl^2(altMorse)}$ because the above frozen hypotheses cannot be applied under *zip* and *not*. However, for this case, the deductions

$$\frac{\vdash \boxed{tl(morse)} = \boxed{tl(altMorse)}}{\vdash \boxed{not(tl(morse))} = \boxed{not(tl(altMorse))}}$$

$$\frac{\vdash \boxed{tl(morse)} = \boxed{tl(altMorse)},\ \boxed{not(tl(morse))} = \boxed{not(tl(altMorse))}}{\vdash \boxed{zip(tl(morse), not(tl(morse)))} = \boxed{zip(tl(altMorse), not(tl(altMorse)))}}$$

are sound. We say that the contexts $not(*{:}Stream)$, $zip(*{:}Stream, S{:}Stream)$, and $zip(S{:}Stream, *{:}Stream)$ are *special*, because they allow to use frozen hypotheses under them in the deduction process.

The current version of CIRC includes an algorithm for computing special contexts for a given specification. To activate this algorithm, we have to set on the switch `auto contexts` before loading the c-theory, using the command `(set auto contexts on .)`. After the c-theory is loaded, CIRC outputs the computed special contexts:

```
  The special contexts are:
  not *:Stream
  zip(V#1:Stream,*:Stream)
  zip(*:Stream,V#2:Stream)
```

Having the special contexts computed, CIRC successfully terminates the proof computation for the above property:

```
Maude> (add goal morse = altMorse .)
Maude> (add goal f(S:Stream) = zip(S:Stream, not(S:Stream)) .)
Maude> (coinduction .)
Proof succeeded.
  Number of derived goals: 8
  Number of proving steps performed: 41
  Maximum number of proving steps is set to: 256
Proved properties:
  tl(morse) = tl(altMorse)
  tl(f(S:Stream)) = zip(not(S:Stream),tl(S:Stream))
  morse = altMorse
  f(S:Stream) = zip(S:Stream,not(S:Stream))
```

We see above that CIRC does not report the context $f(*{:}Stream)$ as being special. The problem of computing the special contexts is not decidable, so the algorithm implemented in CIRC is not able to always find all the special contexts. CIRC includes the facility to declare a context as special, provided that the user guarantees for that.

Other examples of properties which can be proved using the special contexts are: $S_1 \times (S_2 + S_3) = S_1 \times S_2 + S_1 \times S_3$ and $(S_1 + S_2) \times S_3 = S_1 \times S_3 + S_2 \times S_3$ for streams, similar properties for infinite binary trees [5], the equivalence for basic process algebra, and well-definedness of stream definitions [10]. All these examples are available on the web site of the tool.

4 Conclusions

The development of CIRC started about three years ago. The main focus up to now was to implement the circular coinduction engine and to use it on many examples. The experience we gained in this period helped us better understand how circular coinduction and its extension with special contexts can be recast as a formal proof system [8, 6].

The special contexts were useful for proving many properties of streams and infinite binary trees. They were also useful for proving the well-definedness of several stream definitions inspired from [10].

The potential of the ideas included in [4] is not exhausted. For instance, the case analysis is not included in the current version. A former version of CIRC included some facilities in this respect, but it needed manual assistance. One of our future aim is to extend CIRC with automated case analysis.

Acknowledgment. The paper is supported in part by NSF grants CCF-0448501, CNS-0509321 and CNS-0720512, by NASA contract NNL08AA23C, by the Microsoft/Intel funded Universal Parallel Computing Research Center at UIUC, and by CNCSIS grant PN-II-ID-393.

References

1. J.-P. Allouche, A. Arnold, J. Berstel, S. Brlek, W. Jockusch, S. Plouffe, and B. E. Sagan. A sequence related to that of Thue-Morse. *Discrete Mathematics*, 139:455–461, 1995.
2. J.-P. Allouche and J. Shallit. The ubiquitous prouhet-thue-morse sequence. In C. Ding, T. Helleseth, and H. Niederreiter, editors, *Sequences and Their applications (Proc. SETA'98)*, pages 1–16. Springer-Verlag, 1999.
3. G. Caltais, E.-I. Goriac, D. Lucanu, and G. Grigoraş. A Rewrite Stack Machine for ROC! Technical Report TR 08-02, "Al.I.Cuza" University of Iaşi, Faculty of Computer Science, 2008. URL:http://www.infoiasi.ro/ tr/tr.pl.cgi.
4. J. Goguen, K. Lin, and G. Rosu. Conditional circular coinductive rewriting with case analysis. In M. Wirsing, D. Pattinson, and R. Hennicker, editors, *WADT*, volume 2755 of *Lecture Notes in Computer Science*, pages 216–232. Springer, 2002.
5. G. Grigoraş and D. L. G. Caltais, E. Goriac. Automated proving of the behavioral attributes. Accepted for the 4th Balkan Conference in Informatics (BCI'09), 2009.
6. D. Lucanu and G. Roşu. Circular Coinduction with Special Contexts. Technical Report UIUCDCS-R-2009-3039, University of Illinois at Urbana-Champaign, 2009. Submitted.
7. D. Lucanu and G. Rosu. Circ : A circular coinductive prover. In T. Mossakowski, U. Montanari, and M. Haveraaen, editors, *CALCO*, volume 4624 of *Lecture Notes in Computer Science*, pages 372–378. Springer, 2007.
8. G. Roşu and D. Lucanu. Circular Coinduction –A Proof Theoretical Foundation. In *CALCO'09*, LNCS, 2009. This volume.
9. M. R. Schroeder. *Fractals, Chaos, Power Laws.* W. H. Freeman, 1991.
10. H. Zantema. Well-definedness of streams by termination. Submitted, 2009.