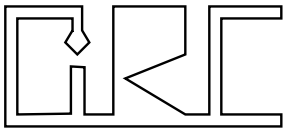

CIRC Prover

Version 1.5

(DRAFT)

Eugen-Ioan Goriac
Faculty of Computer Science
Alexandru Ioan Cuza University
Iaşi, Romania
egoriac[at]info[dot]uaic[dot]ro

Georgiana Caltais
Faculty of Computer Science
Alexandru Ioan Cuza University
Iaşi, Romania
gcaltais[at]info[dot]uaic[dot]ro

Dorel Lucanu
Faculty of Computer Science
Alexandru Ioan Cuza University
Iaşi, Romania
dlucanu[at]info[dot]uaic[dot]ro

Grigore Roşu
Department of Computer Science
University of Illinois
Urbana-Champaign, USA
grosu[at]cs[dot]uiuc[dot]edu

October 15, 2011

Contents

Chapter 1

Introduction

1.1 Motivation

Automated theorem proving is a subject of high interest in computer science, frequently used in industry for hardware and software verification. Coinduction [**?**] is a proof technique for properties over infinite data structures (which typically model behaviors of reactive systems) or for behavioral properties. Proving non-trivial properties by hand using coinduction is tedious due to its complexity and, therefore, using tools that automatize the process as much as possible is desired.

An important characteristic of such a tool is to implement algorithms that cover a wide range of practical cases. This is the idea we have bared in mind when developing the automated coinductive theorem prover CIRC [18]. The core of this prover is the *circular coinduction* principle [9, 26], which provides a simple, yet powerful algorithm for making proofs by coinduction. In time CIRC has been enriched with important features: special contexts [23], generalization and simplification rules [12], and case analysis [**?**].

In its current state, CIRC can successfully be used for proving properties over infinite streams and trees, processes, polynomial functors, and security and communication protocols. CIRC can also be used to prove properties using *circular induction* over inductively defined data structures.

Acknowledgment. We are grateful to Andrei Popescu for his essential contribution at the implementation of the first version of the tool. The current version of CIRC includes many of his ideas. We are also grateful to Radu Mereuţă for helping us in preparing this document.
The development of this tool iss supported in part by NSF grants CCF-0448501, CNS-0509321 and CNS-0720512, by NASA contract NNL08AA23C, CNCSIS grant PN-II-ID-393 (CIRC), and by ANCS 602/12516 (DAK).

1.2 CIRC Foundations Informal Presentation

Circular coinduction. Circular coinduction [23] is a coinductive proving technique for behavioral properties. Here, by a behavioral property we mean a property which can be experimentally evaluated. The soundness of the circular coinduction can be explained, among other interpretations, by a graph whose equalities from nodes can be regarded as lemmas inferred in order to prove the original task, and the graph itself as a dependence relation among these lemmas; one can take all these and produce a parallel proof. We exemplify this technique proving a very simple property over streams (infinite lists). Let *zeros* be the streams of 0's ($= 0 : 0 : 0 : \ldots$), *ones* the stream of 1's, *oz* the stream $0 : 1 : 0 : 1 \ldots$, *zo* the stream $1 : 0 : 1 : 0 \ldots$, and let $zip(S, S')$ be

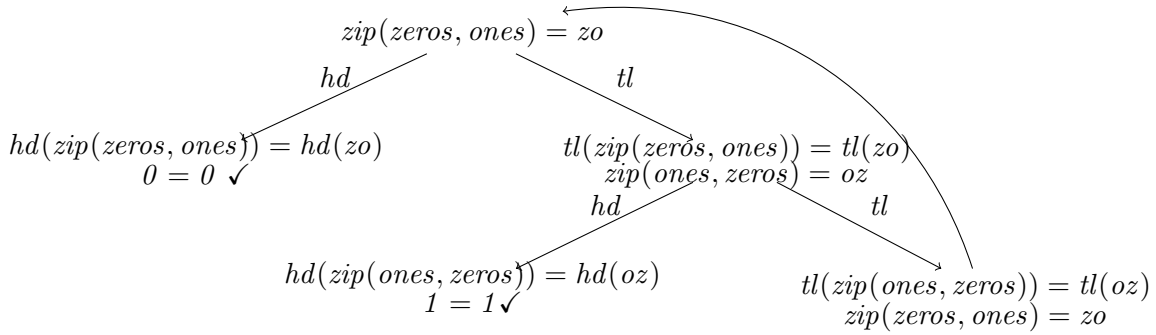

Figure 1.1: Intuitive proof by circular coinduction

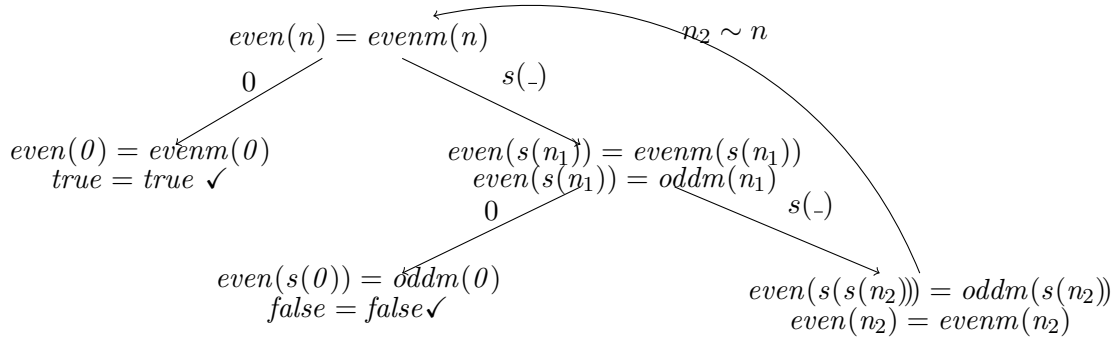

Figure 1.2: Intuitive proof by circular induction

the operation which is zipping two streams. The formal definitions for these streams are given by the following corecursive equations:

$$\begin{array}{l} zeros = 0 : zeros \\ ones = 1 : ones \end{array} \qquad zip(a : S, S') = a : zip(S', S) \qquad \begin{array}{l} zo = 0 : oz \\ oz = 1 : zo \end{array}$$

We consider the destructors hd (head) and tl (tail) defined by $hd(a : S) = a$ and $tl(a : S) = S$. The two destructors define two *derivatives* $hd(*{:}Stream)$ and $tl(*{:}Stream)$, which can be seen as equation transformers: an equation $S = S'$ over streams is transformed into $hd(S) = hd(S')$ and $tl(S) = tl(S')$, respectively. The proof tree of the property $zip(zeros, ones) = zo$ by circular coinduction is represented in Fig. 1.1. The head derivative produces a new property (equation), which can be showed to be equivalent to $0 = 0$ using the definition of the operations. The tail derivative produces a new lemma, which is derived in the same way. Again, the property produced by the head derivative is immediately proved. The tail derivative produces exactly the initial property, so there is no reason to continue because we ended into a circularity. For the general case, the circularity is modulo a substitution and/or special contexts.

Circular induction. In order to show the inductive proving technique, lets take a simple example, whose proof has a similar structure with the above one. Let us consider the following two definitions for even, one iterative and the other one mutual (using odd):[1]

$$\begin{array}{l} even(0) = true \\ even(s(0)) = false \\ even(s(s(N)) = even(N) \end{array} \qquad \begin{array}{ll} evenm(0) = true & \\ evenm(s(N)) = oddm(N) & oddm(s(N)) = evenm(N) \end{array} \qquad oddm(0) = false$$

The proof of the conjecture $(\forall N) even(N) = evenm(N)$ is intuitively represented by the graph

[1]This example is from Induction Challenge Problems site: `http://www.cs.nott.ac.uk/~lad/research/challenges/`

in Figure 1.2. The constructors of naturals, 0 and $s(\_)$, can be seen as properties (here equalities) transformers, by applying an appropriate substitution over a an inductive variable. These transformers obtained from constructors are called *derivatives*. The initial goal is transformed by derivatives into two new proof obligations: $even(0) = evenm(0)$ and $even(s(n_1)) = evenm(s(n_1))$.The former is reduced to $true = true$ using the axioms from the specification and the later is simplified to a simpler goal; this is then derived into two new proof obligations: $even(s(0)) = oddm(0)$ and $even(s(s(n_2))) = oddm(s(n_2))$. Again the former is reduced using the axioms; the latter is very "similar" to initial goal (the similarity is expressed by $n_2 \sim n$ meaning that "n_2 is an incarnation of n and therefore satisfies the same properties as n), so there is no reason to continue the proving process and we may conclude that the property holds. The composition of he derivatives produces *experiments*. We may say that the property above was proved using only two such experiments because we ended up into a cycle. It is easy to see now that this proof technique is very similar to the one applied for circular coinduction.

1.3 Getting Started

CIRC is available, free of charge, under the terms of the GNU General Public License as published by the Free Software Foundation, at the CIRC home page `http://fsl.cs.uiuc.edu/circ` .

1.3.1 Online execution

CIRC can be executed using an online interface, which provides a place to enter new proof scripts, view existing sample scripts, and evaluate new or existing scripts. You can reach the online interface from `http://fsl.cs.uiuc.edu/index.php/Special:CircOnline`.

1.3.2 Installing

1. If you already have Maude installed, then go to the next step. Otherwise,
   (a) if you work on a Linux platform, then go to Maude download page `http://maude.cs.uiuc.edu/download/` and follow the steps written there for downloading and installing Maude;
   (b) if you work on a Windows platform, then go to Moment project page `http://moment.dsic.upv.es/` and download Maude for Windows;
   (c) if you use the Eclipse environment, then you also may download Maude Development Tools from Moment project page `http://moment.dsic.upv.es/`; this include a set of plug-ins embedding the Maude system into the Eclipse environment.
2. Download CIRC (`circ.maude`) or the archive `circ.zip` from CIRC home page `http://fsl.cs.uiuc.edu/index.php/Circ` or its mirror `http://circ.info.uaic.ro/`. The archive includes documentation (this document), a set of examples, and the tool `circ.maude`.
3. Copy the file `circ.maude` in the folder including Maude tools (e.g., Full-Maude.maude, model-checker.maude and so on).

1.3.3 Loading

1. Start Maude in Full-Maude mode. If Full-Maude is not loaded automatically, then introduce the command `in full-maude`.

```
                \|||||||||||||||||/
              --- Welcome to Maude ---
                /|||||||||||||||||\
     Maude 2.4 built: Nov  6 2008 16:49:57
     Copyright 1997-2008 SRI International
            Mon Jun 28 17:11:59 2010

     Full Maude 2.4 February 6th 2009

Maude> _
```

2. Load the prover introducing the command `in circ.maude` . You will get the following output:

```
Maude> in circ.maude

==========================================
fmod CONTAINERS
==========================================
fmod BASIC-DATA-TYPES
==========================================
fmod DATA-TYPES
==========================================
fmod EQ-MANAGER
==========================================
fmod PRE-OUTPUT
==========================================
fmod CIRC-SPEC-LANG-SORTS
==========================================
fmod CIRC-SPEC-LANG-SIGN
==========================================
fmod CIRC-CMD-LANG-SIGN
==========================================
fmod CIRC-LANG-SIGN
==========================================
fmod META-CIRC-LANG-SIGN
==========================================
fmod CIRC-DECL
==========================================
fmod ALGORITHMS
==========================================
mod CIRC-UNIT
==========================================
fmod PROOFSTATUS
==========================================
mod CIRC-DATABASE-HANDLING
==========================================
mod CIRC-PROVER
==========================================
```

```
mod CIRC-INTERFACE

        CIRC 1.5 (April 20th, 2010)

Maude> _
```

3. At this point you may test any of the examples presented in this manual. It is worth noting that CIRC extends the Full Maude syntax of system theories, therefore any specification that does not include object oriented modules should be interpreted by our system.

1.3.4 CIRC at work

In this section we present a scenario showing 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 *zo* in Section 1.2.

This example illustrates how CIRC implements the proof system given in [26]. Supposing that the file including the c-theory `STREAM` has the name `stream.maude`, we present how the user can verify that $zip(zeros, ones) = zo$:

```
Maude> in stream.maude
Maude> (add goal zip(zeros, ones) = zo .)
Maude> (coinduction .)
Proof succeeded.
  Number of derived goals: 4
  Number of proving steps performed: 22
  Maximum number of proving steps is set to: 256
Proved properties:
  zip(ones,zeros) = tl(zo)
  zip(zeros,ones) = zo
```

From the output we conclude that CIRC needs to prove 4 extra derived subgoals in order to prove the initial property and that it performs 22 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(ones,zeros)) *] = [* tl(tl(zo)) *]
------------------------------------------------------------ [Reduce]
|||-  [* tl(zip(ones,zeros)) *] = [* tl(tl(zo)) *]


|-  [* hd(zip(ones,zeros)) *] = [* hd(tl(zo)) *]
------------------------------------------------------------ [Reduce]
|||-  [* hd(zip(ones,zeros)) *] = [* hd(tl(zo)) *]


1. |||-  [* hd(zip(ones,zeros)) *] = [* hd(tl(zo)) *]
2. |||-  [* tl(zip(ones,zeros)) *] = [* tl(tl(zo)) *]
------------------------------------------------------------ [Derive]
|||-  [* zip(ones,zeros) *] = [* tl(zo) *]


|-  [* zip(ones,zeros) *] = [* tl(zo) *]
------------------------------------------------------------ [Normalize]
|-  [* tl(zip(zeros,ones)) *] = [* tl(zo) *]
```

```
|-  [* hd(zip(zeros,ones)) *] = [* hd(zo) *]
------------------------------------------------------------ [Reduce]
|||-  [* hd(zip(zeros,ones)) *] = [* hd(zo) *]


1. |||-  [* hd(zip(zeros,ones)) *] = [* hd(zo) *]
2. |||-  [* tl(zip(zeros,ones)) *] = [* tl(zo) *]
------------------------------------------------------------ [Derive]
|||-  [* zip(zeros,ones) *] = [* zo *]
```

Comparing with the proof tree given in [26], we see that the [`Derive`] step is accompanied by a [`Normalize`] step.

Chapter 2

Basic Syntax and Commands

2.1 Commands

The general syntax for a command is

$$(\langle command \rangle)$$

where

$$\begin{aligned}
\langle command \rangle ::= & \langle specification \rangle \\
& \langle addGoal \rangle \\
& \langle focus \rangle \\
& \langle showGoals \rangle \\
& \langle setShowDetails \rangle \\
& \langle addLemma \rangle \\
& \langle eqReduction \rangle \\
& \langle circularCoinductionStep \rangle \\
& \langle circularInductionStep \rangle \\
& \langle coinduction \rangle \\
& \langle induction \rangle \\
& \langle applyStrategy \rangle \\
& \langle setMaxNoSteps \rangle \\
& \langle setAutoContexts \rangle \\
& \langle checkScx \rangle \\
& \langle showMaxNoSteps \rangle \\
& \langle showProof \rangle \\
& \langle saveProofState \rangle \\
& \langle undo \rangle \\
& \langle quitProof \rangle
\end{aligned}$$

In what follows we use by default the following definition:

$$\langle itemList \rangle ::= \langle item \rangle \mid \langle item \rangle \ \langle itemList \rangle$$

where *item* is instantiated over different syntactical constructions.

2.1.1 Introducing a specification

This is the first command which must be introduced when you want to use CIRC for proving coinductive/inductive properties. This command has a double function: 1. it includes information regarding the (behavioural) specification, derivatives, case analysis, and 2. it starts the prover by creating the initial configuration.

The syntax of this command is:

$$\begin{aligned}
\langle specification \rangle ::= & \texttt{cth } \langle declarationList \rangle \texttt{ endcth} \mid \\
& \texttt{theory } \langle declarationList \rangle \texttt{ endtheory} \mid \\
\langle declaration \rangle ::= & \langle fthMaudeSpecificDeclaration \rangle \mid \\
& \langle derivative \rangle \mid \\
& \langle specialContext \rangle \mid \\
& \langle eqSimplification \rangle \mid \\
& \langle enumeratedSort \rangle \mid \\
& \langle guardedEquation \rangle \mid
\end{aligned}$$

A derivative declaration has the syntax

$$\begin{aligned}
\langle derivative \rangle ::= & \texttt{der } \langle termList \rangle \texttt{ .} \mid \\
& \texttt{derivative } \langle termList \rangle \texttt{ .}
\end{aligned}$$

and declares a set of derivatives where the placeholder of the state is pointed by a star-variable $*{:}\langle sort \rangle$.

A special context declaration has the syntax

$$\begin{aligned}
\langle specialContext \rangle ::= & \texttt{scx } \langle termList \rangle \texttt{ .} \mid \\
& \texttt{special-context } \langle termList \rangle \texttt{ .}
\end{aligned}$$

and declares a set of special contexts. The variable where the coinduction hypothesis can be applied is pointed by $*{:}\langle sort \rangle$.

A simplification equation declaration has the syntax

$$\begin{aligned}
\langle eqSimplification \rangle ::= & \texttt{csrl } \langle equation \rangle \Rightarrow \langle equationList \rangle \texttt{ if } \langle conditionList \rangle \texttt{ .} \mid \\
& \texttt{srl } \langle equation \rangle \Rightarrow \langle equationList \rangle \texttt{ .}
\end{aligned}$$

and specifies that an equation can be proved by showing that other equations hold [when a certain set of conditions are satisfied.

An enumerated sort declaration has the syntax

$$\langle enumeratedSort \rangle ::= \texttt{enum } \langle identifier \rangle \texttt{ is } \langle identifierList \rangle \texttt{ .}$$

and specifies a sort composed only by constants.

A guarded equation has the syntax

$$\begin{aligned}
\langle guardedEquation \rangle ::= & \texttt{geq } \langle term \rangle = \langle choiceList \rangle \texttt{ .} \\
\langle choice \rangle ::= & \langle term \rangle \texttt{ if } \langle condition \rangle \texttt{ []}
\end{aligned}$$

and specifies a sort composed only by constants.

Example: Streams.

```
(theory B-STREAM is
   including STREAM .
   der hd(*:Stream) .
   der tl(*:Stream) .
endtheory)
```

The above command is equivalent to

```
(theory B-STREAM is
  including STREAM .
  der hd(*:Stream) tl(*:Stream) .
endtheory)
```

2.1.2 Adding a goal

Once the specification was introduced, it can be checked if it has the desired properties. A property to be checked is called *goal*. The command for introducing goals has the following syntax:

$$
\begin{aligned}
\langle addGoal \rangle &::= \texttt{add goal } \langle equation \rangle \texttt{ .} \mid \\
&\quad\ \texttt{add goal } \langle operatorDeclaration \rangle \texttt{ .} \mid \\
&\quad\ \texttt{add cgoal } \langle condEquation \rangle \texttt{ .} \\
\langle equation \rangle &::= \langle term \rangle \texttt{ = } \langle term \rangle \\
\langle operatorDeclaration \rangle &::= \texttt{ op } \langle term \rangle \texttt{ : } \langle termList \rangle \texttt{ -> } \langle term \rangle \texttt{ [} \langle attrList \rangle \texttt{] .} \\
\langle attr \rangle &::= \texttt{ comm } \mid \texttt{ assoc } \mid \texttt{ idem } \mid \\
&\quad\ \texttt{ id: } \langle term \rangle \mid \texttt{ left id: } \langle term \rangle \mid \texttt{ right id: } \langle term \rangle \\
\langle attrList \rangle &::= \langle attr \rangle \mid \langle attr \rangle\ \langle attrList \rangle \\
\langle condEquation \rangle &::= \langle term \rangle \texttt{ = } \langle term \rangle \texttt{ if } \langle condition \rangle \\
\langle condition \rangle &::= \langle term \rangle \texttt{ = } \langle term \rangle \mid \langle condition \rangle \texttt{ /\ } \langle condition \rangle
\end{aligned}
$$

Example: Streams.

```
--- equational unconditional goals
(add goal zip(odd(S:Stream), even(S:Stream)) = S:Stream .)

--- operational goal
(add goal (op _+_ : Stream Stream -> Stream [comm] .) .)

--- conditional goal
(add cgoal even? fib(N:Nat, N':Nat) = all-true
            if even? N:Nat = true /\ even? N':Nat = true .)
```

2.1.3 Showing goals

Displays the goals to be proved. Its syntax is

$$\langle showGoals \rangle ::= \texttt{show goals .}$$

Example: Streams.

```
Maude> (show goals .)
zip(odd(S:Stream),even(S:Stream))= S:Stream
Maude>
```

2.1.4 Setting the first goal

Changes the order of the goals by moving a given goal on the first position. To see the order of the goals, use `focus` command. The syntax is

$$\langle focus \rangle ::= \texttt{focus}\ \langle number \rangle\ \texttt{.}$$

Example: Streams.

```
Maude> (add goal odd(zip(S:Stream, S':Stream)) = S:Stream .)
Goal odd(zip(S:Stream,S':Stream))= S:Stream   added.

Maude> (add goal map-f(iter-f(E:Elt)) = iter-f(f(E:Elt)) .)
Goal map-f(iter-f(E:Elt))= iter-f(f(E:Elt))  added.

Maude> (show goals .)
1 .   map-f(iter-f(E:Elt))= iter-f(f(E:Elt))
2 .   odd(zip(S:Stream,S':Stream))= S:Stream
3 .   zip(odd(S:Stream),even(S:Stream))= S:Stream

Maude> (focus 3 .)

Maude> (show goals .)
1 .   zip(odd(S:Stream),even(S:Stream))= S:Stream
2 .   map-f(iter-f(E:Elt))= iter-f(f(E:Elt))
3 .   odd(zip(S:Stream,S':Stream))= S:Stream
Maude>
```

2.1.5 Set show details

There is a flag, `show details`, which controls the amount of information diplayed during the proving process. This flag can be set on or off:

$$\begin{aligned}\langle setShowDetails \rangle ::= {} & \texttt{set show details on .} \mid \\ & \texttt{set show details off .}\end{aligned}$$

If `show details` flag is on, then some additional information regarding the intermediate steps is displayed.

2.1.6 Adding a lemma

Sometimes a lemma is needed to prove a goal. This lemma could be proved priory, on-the-fly, or subsequently. The command for introducing lemmas has the following syntax:

$$\begin{aligned}\langle addLemma \rangle ::= {} & \texttt{add lemma}\ \langle equation \rangle\ \texttt{.} \mid \\ & \texttt{add clemma}\ \langle condEquation \rangle\ \texttt{.}\end{aligned}$$

Example: Finite Lists.

```
Maude> (add lemma E:Elt L:MyList = cons(E:Elt, L:MyList) .)
Lemma E:Elt L:MyList  =  cons(E:Elt,L:MyList) added.
```

2.1.7 Equation reduction

The operation reduces the first goal to its normal form. If the left hand side and the right hand side become equal, then the goal is condidered to be proved. Its syntax is:

$$\langle eqReduction \rangle ::= \texttt{reduce .}$$

Example: Streams.

```
Maude> (show goals .)
1 .   hd zip(odd(S:Stream),even(S:Stream)) = hd S:Stream
2 .   tl(zip(odd(S:Stream),even(S:Stream))) = tl(S:Stream)

Maude> (reduce .)
Goal hd zip(odd(S:Stream),even(S:Stream)) = hd S:Stream normalized to
hd S:Stream = hd S:Stream
Goal hd S:Stream = hd S:Stream proved by reduction.

Maude> (reduce .)
Goal tl(zip(odd(S:Stream),even(S:Stream))) = tl(S:Stream) normalized to
zip(even(S:Stream),even(tl(S:Stream))) = tl(S:Stream)
```

2.1.8 Coinduction

Starts the algorithm of circular coinduction over the set of goals using the derivatives included in the specification. Its syntax is

$$\langle coinduction \rangle ::= \texttt{coinduction .} \mid \texttt{ccstep .}$$

Example: Streams.

```
Maude> (show goals .)
zip(odd(S:Stream),even(S:Stream))= S:Stream

Maude> (ccstep .)
Hypo zip(odd(S:Stream),even(S:Stream)) = S:Stream added and coexpanded to
1 .   hd zip(odd(S:Stream),even(S:Stream)) = hd S:Stream
2 .   tl(zip(odd(S:Stream),even(S:Stream))) = tl(S:Stream)

Maude> (reduce .)
Goal hd zip(odd(S:Stream),even(S:Stream)) = hd S:Stream normalized to
hd S:Stream = hd S:Stream
Goal hd S:Stream = hd S:Stream proved by reduction.

Maude> (show goals .)
tl(zip(odd(S:Stream),even(S:Stream)))= tl(S:Stream)

Maude> (coinduction .)
Proof succeeded.

Maude>
```

2.1.9 Induction.

Starts the algorithm of circular induction over the set of goals. Its syntax is

$$\begin{aligned}\langle induction \rangle ::= {} & \texttt{induction .} \mid \texttt{cistep .} \mid \\ & \texttt{induction on } \langle variableSet \rangle \texttt{ .} \mid \texttt{cistep } \langle variableSet \rangle \texttt{ .} \mid \\ & \texttt{set induction vars } \langle variableSet \rangle \texttt{ .}\end{aligned}$$

The structure of a variable set is : $\langle variableSet \rangle ::= \langle variable \rangle \mid \langle variable \rangle \ \langle variableSet \rangle$, where a variable must be specified as a pair `name:Sort`.

Example: Finite trees.

```
Maude> in tree-spec .
Maude> (cth BTREE is
          including TREE .
        endcth)

Maude> (add goal size(app(L:TList, T:Tree)) = size(T:Tree) + size(L:TList) .)
Goal added: size(app(L:TList,T:Tree)) = size(T:Tree)+ size(L:TList)

Maude> (induction on L:TList .)
Proof succeeded.

Maude>
```

2.1.10 Apply user-defined strategies

Starts a user-defined algorithm (strategy) over the set of goals. The syntax of the command is:

$$\langle applyStrategy \rangle ::= \texttt{apply}\ \langle action \rangle\ \texttt{.}$$

$$\begin{aligned}\langle action \rangle ::= {} & \texttt{reduce} \mid \\ & \texttt{ccstep} \mid \\ & \texttt{cistep} \mid \\ & \texttt{simplify} \mid \\ & \texttt{(}\langle action \rangle\texttt{)}\ \texttt{|>}\ \texttt{(}\langle action \rangle\texttt{)} \mid \\ & \texttt{(}\langle action \rangle\texttt{)}\ \texttt{\#}\ \texttt{(}\langle action \rangle\texttt{)} \mid \\ & \texttt{(}\langle action \rangle\texttt{)}\ \texttt{!}\end{aligned}$$

Example: An induction strategy.

```
Maude> (apply (reduce |> cistep) ! .)
```

2.1.11 The maximum number of prover steps allowed

Can be set and shown using the following commands:

$$\begin{aligned}\langle setMaxNoSteps \rangle &::= \texttt{set max no steps}\ \langle variable : Int \rangle\ \texttt{.} \\ \langle showMaxNoSteps \rangle &::= \texttt{show max no steps .}\end{aligned}$$

Example: Streams.

```
Maude> (show goals .)
map-f(iter-f(E:Elt)) = iter-f(f(E:Elt))

Maude> (set max no steps 4 .)
The maximum number of proving steps was set to 4 .

Maude> (coinduction .)
Hypo map-f(iter-f(E:Elt)) = iter-f(f(E:Elt)) added and coexpanded to
1 .   hd map-f(iter-f(E:Elt)) = hd iter-f(f(E:Elt))
2 .   tl(map-f(iter-f(E:Elt))) = tl(iter-f(f(E:Elt)))
Stopped: the number of prover steps was exceeded.
```

```
Maude> (show max no steps .)
The number of proving steps allowed is 0 .

Maude> (set max no steps 100 .)
The maximum number of proving steps was set to 100 .

Maude> (coinduction .)
Goal hd map-f(iter-f(E:Elt)) = hd iter-f(f(E:Elt)) normalized to
    f(E:Elt) = f(E:Elt)
Goal f(E:Elt) = f(E:Elt) proved by reduction.
Goal tl(map-f(iter-f(E:Elt))) = tl(iter-f(f(E:Elt))) normalized to
    iter-f(f(f(E:Elt))) = iter-f(f(f(E:Elt)))
Goal iter-f(f(f(E:Elt))) = iter-f(f(f(E:Elt))) proved by reduction.

Proof succeeded.

Maude> (show max no steps .)
The number of proving steps allowed is 88 .
```

2.1.12 Computing special contexts

For the case of computing special contexts, there is a flag that controls the automated computation. This flag can be set on or off:

$$\langle setAutoContexts \rangle ::= \texttt{set auto contexts on.} \mid \texttt{set auto contexts off.} \mid$$

Example: Streams.

```
(set auto contexts on .)

(theory B-STREAM is
   including STREAM .
   der hd(*:Stream) .
   der tl(*:Stream) .
endtheory)

The special contexts are:
zip(*:BitStream,V#1:BitStream)
zip(V#2:BitStream,*:BitStream)
```

The special contexts are automatically computed only if the flag is set on. As an observation, no matter the flag is set on or off, if the specification contains explicit declaration of safe special contexts by use of `scx` command, then the prover checks whether the declarations are correct or not. If there is a `scx` declaration that could not be proved as correct, then a warning message will be displayed:

`The following specified contexts may not be special: scx` $\langle termList \rangle$

The user can specifically check if a context is special by providing a cobasis. The command for special contexts checking is:

$$\langle checkScx \rangle ::= \texttt{check scx } \langle term \rangle \texttt{ using } \langle termList \rangle \texttt{ .}$$

Example: Streams.

```
Maude> (check scx zip(*:BitStream, S:BitStream) using
         hd(*:BitStream)
         hd(tl(*:BitStream))
         tl(tl(*:BitStream))
       .)

zip(*:BitStream,S:BitStream) is a special context
```

2.1.13 Show proof

This command displays the main steps performed by the algorithm in the current proofs. Its syntax is

$$\langle showProof \rangle ::= \texttt{show proof .}$$

2.1.14 Quit proof

This command quits the current proof(s) and removes the information regarding the proof(s) including the hypothesis and lemmas added during the current proof(s). It is useful when the current proof fails or you do not know how to continue it and you wish to start a new proof with the initial specification. Its syntax is

$$\langle quitProof \rangle ::= \texttt{quit proof .}$$

2.1.15 Save proof state

When proving a goal in the assisted mode, the user may realize that they need to prove a separate lemma in order to complete the current proof. At this point, the `save proof state` command needs to be used in oder to save all the internal information gathered during the current proof and start a fresh proof. When the latter proof succeeds, the lemma is automatically added as a hypothesis and the state for the initial proof is resotred.

$$\langle saveProofState \rangle ::= \texttt{save proof state .}$$

2.1.16 Undo

This command is used in the assisted mode, when the user wants to rewind the proof state to a previous step.

$$\langle undo \rangle ::= \texttt{undo .}$$

2.2 Cautions

1. Avoid to use operation names starting with `[*`.
2. Avoid to use constant names starting with `CC-V2C`.
3. Avoid to use variable names of the form `...&...#`$\langle number \rangle$....
4. It is highly recommended not to declare constructors if the induction is not used in the proving process. Since the constructor variable are frozen, they cannot be properly instantiated when the coinductive hypothesis is applied. Note that the built-in modules NAT and INT include constructor declarations.

2.3 Strategies [5]

CIRC[19] is a Maude metalanguage application which implements the circular coinduction algorithm in order to prove *properties* (goals expressed as equations) for a given *equational behavioral specification*. CIRCis built using the patterns described in [13]. The procedure for proving a property uses a set of *rewrite rules* that can be applied according to some *proof strategy*. Three of the rules implemented for the CIRCprover are:

- [*comm*] processes a goal expressing the commutativity of an operator by adding a special operator and some equations in their frozen form to the specification. The rule fails when the current property to be proved is not a commutativity goal.
- [*eqRed*] removes an equational goal whenever it can be proved using ordinary equational reduction. The rule fails when the left hand side of an equation is different from the right hand side.
- [*ccstep*] implements the circularity principle: an equation's frozen normalized form is added to the specification and its derivatives are added to the set of goals. This rule fails whenever it finds a visible goal which cannot be proved using ordinary equational reduction.

The following steps describe a possible strategy for the prover:

1. Try to apply [*comm*]. If it fails, leave the system state unchanged.
2. Check whether the first goal is proved using [*eqRed*]. If the check fails try to apply [*ccstep*].
3. Repeat step 2 for as many times as possible.

The first problem is how to specify this kind of strategies. There are many ways to define rewriting strategies [1, 3, 7, 17, 27]. A previous version of CIRCuses regular strategies [21]. However, the union, concatenation and iteration operators of the regular expressions language are not proper for specifying a behavior like "apply a rule for as many times as possible" or "apply a rule only if some other rule fails".

The solution is to use strategy operators appropriate for specifying these kinds of behavior. We prefer to call the elements of this strategy language *actions*. A basic action is a rewrite rule. Generally, actions are combined by means of several operators: $\triangleright$ (orelse), $\circ$ (composition) and ! (repeat) resulting in other actions. For the given example, the action described by the steps 1 - 3 is:

$$([comm] \triangleright [id]) \circ ([eqRed] \triangleright [ccstep])!$$

where

- [*id*] leaves the list of goals unchanged. This rule never fails.
- $\triangleright$ has the semantics of an *orelse*-like operator. The system tries to apply [*comm*] in the first place, and only if the action does not succeed should it apply [*id*]. The same strategy is followed for the rules [*eqRed*] and [*ccstep*].
- $\circ$ has the semantics of a sequential composition operator. For the example above, the full action is successfully applied if both $([comm] \triangleright [id])$ and $([eqRed] \triangleright [ccstep])!$ are evaluated exactly in this order with success.
- ! imposes that an action is applied for as many times as possible. In our case, the system stops following the strategy only when both [*eqRed*] and [*ccstep*] fail.

It is obvious that once the set of rules is chosen and the semantics of the strategy operators is provided, complex proof strategies can easily be specified in the same manner previously described.

Here are the strategies implemented in CIRC:

```
reduce           = normalize # eqRed
coinduction      = ((normalize |> eqRed |> checkcond |> cases |> simplify |> ccstep) !)
coinduction-grlz = ((normalize |> eqRed |> simplify |> generalize |> ccstep) !)
induction        = ((normalize |> eqRed |> simplify |> checkcond |> cistep) !)
```

Chapter 3

Formal presentation

3.1 Behavioral Coinductive Specifications and CIRC [18]

Behavioral abstraction in algebraic specification appears under various names in the literature such as *hidden algebra* in works by Goguen and many others (see, e.g., [8, 11]) *observational logic* in works by Hennicker, Bidoit and many others (e.g., [16]), *swinging types* in works by Padawitz [24], *coherent hidden algebra* in Diaconescu [6], *hidden logic* in Roşu [25], 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[1], 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*.

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: $\mathcal{B} \Vdash e$ 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 [26] 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 [26] 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:

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

[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` $\mathcal{B} \cup \mathcal{F} \not\vdash_{\rightarrow,\leftarrow} e$
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. $\Delta(e)$ 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.

[Generalize]: $(\mathcal{B}, \mathcal{F}, \mathcal{G} \cup \{(\forall Y)\ \boxed{\theta(t)} = \boxed{\theta(t')}\}) \Rightarrow (\mathcal{B}, \mathcal{F}, \mathcal{G} \cup \{(\forall Y)\ \boxed{t} = \boxed{t'}\})$
where $\theta : X \to T_\Sigma(Y)$ is a substitution.
If the current goal can be generalized after identifying the substitution θ, then we replace it by its generalized form.

[Fail]: $(\mathcal{B}, \mathcal{F}, \mathcal{G} \cup \{\boxed{e}\}) \Rightarrow$ *fail* `if` $\mathcal{B} \cup \mathcal{F} \not\vdash_{\rightarrow,\leftarrow} e$ 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 [26]. 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. The rule [Generalize] is presented in [12]. [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 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:

```
(theory STREAM is including STREAM-EQ .
  derivative hd(*:Stream) .
  derivative tl(*:Stream) .
endtheory)
```

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.

3.2 Special Contexts [18, 23]

Another important part is the extension of the tool with *special contexts* [20], as explained below. An important technical aspect of our implementation of circular coinduction in CIRC [22] (see also [26]) is the 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.

CIRC implements the special contexts by the mean of a complex algorithm automatically computing the special contexts of a specification and using the following deriving rule:

[Derive]: $(\mathcal{B}, \mathcal{F}, \mathcal{G} \cup \{\boxed{e}\}) \Rightarrow (\mathcal{B}, \mathcal{F} \cup \{\boxed{e}\} \cup \boxed{\Gamma[e]}, \mathcal{G} \cup \{\boxed{\Delta(e)}\})$

if $\mathcal{B} \cup \mathcal{F} \not\vdash_{\rightarrow,\leftarrow} e$, e is hidden and Γ is a set of special contexts.

Since the special contexts cannot be always computed, the problem is Π^0_2-complete, CIRC allows the user to explicitly include in the specification special contexts proved by hand. See [18, 23] for more details.

Let us next discuss an example where the simple circular coinduction system fails to build a finite proof tree. A technique based on behavioral equivalence for checking well-definedness of stream operations is proposed in [28]. For instance, the well-definedness of *zip* follows by defining streams g and h by $hd(g) = hd(h) = 1$ (or any other constant), $tl(g) = zip(g, g)$, $tl(h) = zip(h, h)$, and then showing that $\texttt{STREAM} \Vdash g = h$.[2] Circular coinduction fails to find a proof for this property because the building process of the proof tree does not terminate. We show that a finite proof tree can be quickly obtained if the additional hypotheses defined by the special contexts are used. Two special contexts are needed, namely $\Gamma = \{zip(*{:}Stream, S{:}Stream), zip(S{:}Stream, *{:}Stream)\}$. These contexts together with the frozen hypothesis (corresponding to the initial goal) yield the following two special hypotheses:

$$\Gamma[g = h] = \{zip(g, S{:}Stream) = zip(h, S{:}Stream),\ zip(S{:}Stream, g) = zip(S{:}Stream, h)\}.$$

Here is the proof tree generated by the extended proof system:

$$\frac{\begin{array}{ll}\texttt{STREAM} \cup \{\boxed{g} = \boxed{h}\} \cup \boxed{\Gamma[g = h]} & \Vdash^{\circlearrowleft} \emptyset \\ \texttt{STREAM} \cup \{\boxed{g} = \boxed{h}\} \cup \boxed{\Gamma[g = h]} & \vdash \boxed{hd(g)} = \boxed{hd(h)}\end{array}}{\begin{array}{ll}\texttt{STREAM} \cup \{\boxed{g} = \boxed{h}\} \cup \boxed{\Gamma[g = h]} & \Vdash^{\circlearrowleft} \left\{\boxed{hd(g)} = \boxed{hd(h)}\right\} \\ \texttt{STREAM} \cup \{\boxed{g} = \boxed{h}\} \cup \boxed{\Gamma[g = h]} & \vdash \boxed{tl(g)} = \boxed{tl(h)}\end{array}}$$

$$\frac{\texttt{STREAM} \cup \{\boxed{g} = \boxed{h}\} \cup \boxed{\Gamma[g = h]} \quad \Vdash^{\circlearrowleft} \left\{\begin{array}{l}\boxed{hd(g)} = \boxed{hd(h)}, \\ \boxed{tl(g)} = \boxed{tl(h)}\end{array}\right\}}{\texttt{STREAM} \quad \Vdash^{\circlearrowleft} \boxed{g} = \boxed{h}}$$

The special hypotheses are used in the deduction of $\boxed{tl(g)} = \boxed{tl(h)}$ (fourth line of the proof tree above) as follows: we have $tl(g) = zip(g, g)$ and $tl(h) = zip(h, h)$ as defining axioms, and $\boxed{zip(g, g)} = \boxed{zip(h, g)} = \boxed{zip(h, h)}$ follow from $\boxed{\Gamma[g = h]}$.

[2] The authors warmly thank Hans Zantema for supplying this example.

Special contexts must satisfy a certain well-foundedness condition w.r.t. derivatives. Not all contexts are special. For example, if $odd(*{:}Stream)$ were special then our proof system with special contexts would be unsound, as shown by the following scenario inspired from [10]. Let a and b be specified by $hd(a) = hd(b)$, $tl(a) = odd(a)$ and $tl^2(b) = odd(b)$, and let $odd(b) = a$ be the goal we want to prove. Applying the third rule, this goal is added as frozen hypothesis $\boxed{odd(b)} = \boxed{a}$ and the following two new goals are generated: $hd(odd(b)) = hd(a)$ and $tl(odd(b)) = tl(a)$. The former is eliminated by the second rule, and the latter is reduced to $odd(odd(b)) = odd(a)$. If we assume that $odd(*{:}Stream)$ is special, and hence the hypothesis $\boxed{odd(odd(b))} = \boxed{odd(a)}$ is automatically added, then we would wrongly deduce that $odd(b) = a$. A counter-example is given by $a = 0 : 0 : 1 : 2^\infty$ and $b = 0 : 1 : 0^\infty$.

3.3 Simplification Rules [12]

In terms of trace equivalence, two processes are said to be equivalent if they execute the same sequences of actions [**?**]. Consider the two processes U and X illustrated in Figure 3.1. We use "+" to represent the *alternative composition* (non-deterministic choice) operator and ";" to denote the *sequential composition* operator. It is easy to see that the languages generated by the execution of these tow processes contain only words of the form: a, $a\,a\,a$, $a\,a\,a\,a\,a$, and so on (action a occurs for an odd number of times in each string).

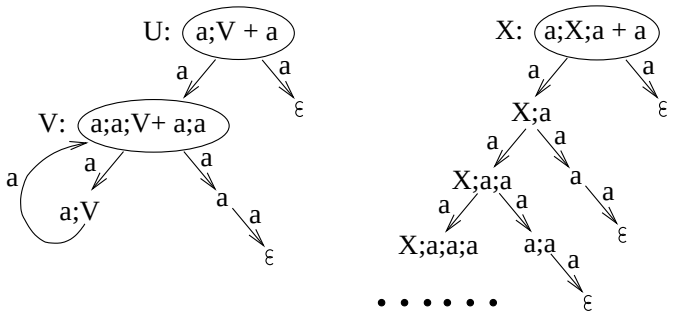

Figure 3.1: Two trace-equivalent processes

Our first aim was to use CIRC in order to prove that U and X are trace-equivalent. As an extension to the work presented in [21], this paper considers infinitary trace-equivalence between processes that are not normalized. We need an extra set of simplification rules used in order to avoid infinite executions of the prover. These rules replace the current goal with a simpler one. Since we do not impose any constraints over the new goal, we need to prove that using these rules is sound. For some of these simplification rules, the soundness can be easily proved by hand, but in many cases this approach is difficult and time consuming.

For example, in the case of processes, we proved that using the following rule is sound:

$$\frac{E_1 + E_2 = E_1' + E_2'}{E_1 = E_1'} \text{ if } E_2 = E_2'$$

Note that the equalities in the above rule are interpreted as behavioral equivalences (which imply the trace equivalences).

Our task will be to prove that $X \sim_{tr}^{\infty} U$, where $X =_{def} (a\,;X\,;a) + a$, $U =_{def} (a\,;V) + a$, and $V =_{def} (a\,;a\,;V) + (a\,;a)$. We provide the specification for these processes:

```
ctheory BPA is
  including BPA-EQ .

  ops a : -> Alph .
  ops X U V : -> Pid .

  eq pmain =
```

```
    ( X =def (a ; X ; a) + a ),
    ( U =def (a ; V) + a ),
    ( V =def (a ; a ; V) + (a ; a) ) .

  derivative *:Pexp { a } .
  derivative bot?(*:Pexp) .

  scx *:Pexp  + PX:Pexp .
  scx PX:Pexp + *:Pexp .
endctheory
```

Here `pmain` represents the specification of all processes. Also, we explicitly specify the operator "+" as special context [20].

We proceed by adding the goal $U = X$ in the usual manner and trying to prove it by coinduction. We also choose to view all the details of the steps performed during the proof session by using the command `(set show details on .)`:

```
Maude> (add goal U = X .)
Maude> (set show details on .)
Maude> (coinduction .)

Hypo [* U *] = [* X *] added and coexpanded to
1. [* U{a} *] = [* X{a} *]
2. [* bot?(U) *] = [* bot?(X) *]

Goal [* U{a} *] = [* X{a} *] normalized to
     [* V + epsilon *] = [* epsilon + X ; a *]

Hypo [* V + epsilon *] = [* epsilon + X ; a *] added and coexpanded to
1. [* (V + epsilon)a *] = [* (epsilon + X ; a)a *]
2. [* bot?(V + epsilon) *] = [* bot?(epsilon + X ; a) *]

Goal [* bot?(U) *] = [* bot?(X) *]  normalized to
     [* false *] = [* false *]
Goal [* false *] = [* false *]  proved by reduction.

Goal [* (V + epsilon)a *] = [* (epsilon + X ; a)a *]  normalized to
     [* a + a ; V *] = [* a + X ; a ; a *]

[...]

Stopped:
  the number of prover steps was exceeded.
```

The prover exceeds the maximum number of steps allowed. It is easy to see that the circular coinduction algorithm produces an infinite set of new goals. A solution to this problem is using simplification rules, as described in [21]. By analyzing the provided output, we see that instead of applying a derivation, the prover could use a simplification rule for the second step. The rule is the one specified in the motivating example:

$$\frac{E_1 + E_2 = E_1' + E_2'}{E_1 = E_1'} \text{ if } E_2 = E_2'$$

For instance, according to our example, this rule simplifies the goal $\boxed{a + a\,;V} = \boxed{a + X\,;a\,;a}$ to $\boxed{a\,;V} = \boxed{X\,;a\,;a}$.

After adding the corresponding simplification rule

```
  csrl (E1:Pexp + E2:Pexp) = (E1':Pexp + E2':Pexp) => (E1:Pexp = E1':Pexp) if (E2:Pexp = E2':Pexp) .
```

to the specification and running the example for the second time, we still encounter the problem of infinite rewriting. By using a similar technique, we identify another simplification rule:

$$\frac{E_1 = E_2 + (a\,;a\,;E_3) + (X\,;a\,;a\,;a\,;E_3)}{E_1 = E_2 + V\,;E_3}$$

After adding the corresponding rule

```
srl (E1:Pexp) = (E2:Pexp + (a ; a ; E3:Pexp) + (X ; a ; a ; a ; E3:Pexp)) => (E1:Pexp) = (E2:Pexp + (V ; E3:Pexp)) .
```

to the specification, the engine manages to prove our goal. Note that the prover prints all the intermediate properties proved during the session.

```
Maude> (add goal U = X .)
Maude> (coinduction .)
Proof succeeded.
  Number of derived goals: 14
  Number of proving steps performed: 86
  Max. number of proving steps is set to: 256
Proved properties:
  a ; V = a ; a ; a + a ; V ; a ; a
  V = a ; a + V ; a ; a
  a ; V = a ; a ; a + X ; a ; a ; a ; a
  V = a ; a + X ; a ; a ; a
  a ; V = X ; a ; a
  V = X ; a
  U = X
```

In order to illustrate the use of simplifying rules, we provide the key parts of the proof obtained by using the command `(show proof .)`. The proof is given in terms of the inference rules described in [26]:

```
[...]
[* V *] = [* a ; a + V ; a ; a *]
---------------------------------- [Simplify]
[* V *] = [* a ; a + a ; a ; a ; a + X ; a ; a ; a ; a ; a *]

[...]

1. |||-  [* V{a} *] = [* (X ; a){a} *]
2. |||-  [* bot?(V) *] = [* bot?(X ; a) *]
---------------------------------- [Derive]
|||-  [* V *] = [* X ; a *]


[* V *] = [* X ; a *]
---------------------------------- [Simplify]
[* V + epsilon *] = [* epsilon + X ; a *]


|-  [* V + epsilon *] = [* epsilon + X ; a *]
---------------------------------- [Normalize]
|-  [* U{a} *] = [* X{a} *]


1. |||-  [* U{a} *] = [* X{a} *]
2. |||-  [* bot?(U) *] = [* bot?(X) *]
---------------------------------- [Derive]
|||-  [* U *] = [* X *]
```

The problem that arises when using simplification rules during a proof is the soundness of the proof itself. One could, for instance, use a simplification rule that transforms the current goal into $true = true$, thus allowing CIRC to "prove" any goal.

In order to check the correctness of the proof, we need to consider the set $\mathcal{F}$ of all the lemmas obtained during the execution of the prover:

- the goals proved using [Derive], such as:
  $\boxed{U = X}$, $\boxed{V = X\,;a}$, etc.
- the goals before and after applying [Simplify], such as:
  $\boxed{V + epsilon} = \boxed{epsilon + X\,;a}$, etc.

If we manage to prove that all the properties in $\mathcal{F}$ hold without using the simplification rules, then the proof is correct. This follows directly from Theorem 2 in [15] for $(\mathcal{B}, \emptyset, \mathcal{F}) \Rightarrow^* (\mathcal{B}, \mathcal{F}', \emptyset)$.

Considering the language ROC! presented in [**?**], the strategy applied for proving the goals in $\mathcal{F}$ is:

$$(\text{[Normalize]} \triangleright \text{[Reduce]} \triangleright \text{[Derive]})!$$

We mention that $\triangleright$ has the semantics of an "orelse"-like operator, while ! imposes the prover to apply the indicated strategy for as many times as possible (*i.e.* until it succeeds or it fails).

The command that starts the execution of the prover for the set $\mathcal{F}$ automatically obtained from the last proof is `(check proof .)`. The user dialog for checking the correctness of the proof is:

```
Maude> (check proof .)
Check proof succeeded.
```

This message indicates that CIRC manages to successfully check the correctness of the proof that uses simplification rules.

An important note is that if CIRC is able to prove the current goal only by using equational reduction, then the simplification rule is not applied.

Let us assume that our task is to prove that $X \sim_{tr}^{\infty} U$, where $X =_{def} (a\,;X\,;a)+a$, $U =_{def} (a\,;V)+a$, and $V =_{def} (a\,;a\,;V)+(a\,;a)$. We provide the specification for these processes:

```
ctheory BPA is
  including BPA-EQ .

  ops a : -> Alph .
  ops X U V : -> Pid .

  eq pmain =
     ( X =def (a ; X ; a) + a ),
     ( U =def (a ; V) + a ),
     ( V =def (a ; a ; V) + (a ; a) ) .

  derivative *:Pexp { a } .
  derivative bot?(*:Pexp) .

  scx *:Pexp  + PX:Pexp .
  scx PX:Pexp + *:Pexp .
endctheory
```

Here `pmain` represents the specification of all processes. Also, we explicitly specify the operator "+" as special context [20].

We proceed by adding the goal $U = X$ in the usual manner and trying to prove it by coinduction. We also choose to view all the details of the steps performed during the proof session by using the command `(set show details on .)`:

```
Maude> (add goal U = X .)
Maude> (set show details on .)
Maude> (coinduction .)

Hypo [* U *] = [* X *]
  added and coexpanded to
1. [* U{a} *] = [* X{a} *]
2. [* bot?(U) *] = [* bot?(X) *]

Goal [* U{a} *] = [* X{a} *] normalized to
     [* V + epsilon *] = [* epsilon + X ; a *]

Hypo [* V + epsilon *] = [* epsilon + X ; a *]
  added and coexpanded to
1. [* (V + epsilon)a *] = [* (epsilon + X ; a)a *]
2. [* bot?(V + epsilon) *] = [* bot?(epsilon + X ; a) *]

Goal [* bot?(U) *] = [* bot?(X) *]
  normalized to
     [* false *] = [* false *]
Goal [* false *] = [* false *]
  proved by reduction.

Goal [* (V + epsilon)a *] = [* (epsilon + X ; a)a *]
```

```
  normalized to
     [* a + a ; V *] = [* a + X ; a ; a *]

[...]

Stopped:
  the number of prover steps was exceeded.
```

The prover exceeds the maximum number of steps allowed. It is easy to see that the circular coinduction algorithm produces an infinite set of new goals. A solution to this problem is using simplification rules, as described in [21]. By analyzing the provided output, we see that instead of applying a derivation, the prover could use a simplification rule for the second step. The rule is the one specified in the motivating example:

$$\frac{E_1 + E_2 = E_1' + E_2'}{E_1 = E_1'} \text{ if } E_2 = E_2'$$

For instance, according to our example, this rule simplifies the goal $\boxed{a + a\,;V} = \boxed{a + X\,;a\,;a}$ to $\boxed{a\,;V} = \boxed{X\,;a\,;a}$.

After adding the corresponding simplification rule

```
csrl (E1:Pexp + E2:Pexp) = (E1':Pexp + E2':Pexp) => (E1:Pexp = E1':Pexp) if (E2:Pexp = E2':Pexp) .
```

to the specification and running the example for the second time, we still encounter the problem of infinite rewriting. By using a similar technique, we identify another simplification rule:

$$\frac{E_1 = E_2 + (a\,;a\,;E_3) + (X\,;a\,;a\,;a\,;E_3)}{E_1 = E_2 + V\,;E_3}$$

After adding the corresponding rule

```
srl (E1:Pexp) = (E2:Pexp + (a ; a ; E3:Pexp) + (X ; a ; a ; a ; E3:Pexp)) => (E1:Pexp) = (E2:Pexp + (V ; E3:Pexp)) .
```

to the specification, the engine manages to prove our goal. Note that the prover prints all the intermediate properties proved during the session.

```
Maude> (add goal U = X .)
Maude> (coinduction .)
Proof succeeded.
  Number of derived goals: 14
  Number of proving steps performed: 86
  Max. number of proving steps is set to: 256
Proved properties:
  a ; V = a ; a ; a + a ; V ; a ; a
  V = a ; a + V ; a ; a
  a ; V = a ; a ; a + X ; a ; a ; a ; a
  V = a ; a + X ; a ; a ; a
  a ; V = X ; a ; a
  V = X ; a
  U = X
```

In order to illustrate the use of simplifying rules, we provide the key parts of the proof obtained by using the command `(show proof .)`. The proof is given in terms of the inference rules described in [26]:

```
[...]
[* V *] = [* a ; a + V ; a ; a *]
---------------------------------- [Simplify]
[* V *] = [* a ; a + a ; a ; a ; a + X ; a ; a ; a ; a ; a *]

[...]

1. |||-  [* V{a} *] = [* (X ; a){a} *]
2. |||-  [* bot?(V) *] = [* bot?(X ; a) *]
---------------------------------- [Derive]
|||-  [* V *] = [* X ; a *]
```

```
[* V *] = [* X ; a *]
---------------------------------- [Simplify]
[* V + epsilon *] = [* epsilon + X ; a *]


|-  [* V + epsilon *] = [* epsilon + X ; a *]
---------------------------------- [Normalize]
|-  [* U{a} *] = [* X{a} *]


1. |||-  [* U{a} *] = [* X{a} *]
2. |||-  [* bot?(U) *] = [* bot?(X) *]
---------------------------------- [Derive]
|||-  [* U *] = [* X *]
```

The problem that arises when using simplification rules during a proof is the soundness of the proof itself. One could, for instance, use a simplification rule that transforms the current goal into $true = true$, thus allowing CIRC to "prove" any goal.

In order to check the correctness of the proof, we need to consider the set $\mathcal{F}$ of all the lemmas obtained during the execution of the prover:

- the goals proved using [Derive], such as: $\boxed{U = X}$, $\boxed{V = X\,;a}$, etc.
- the goals before and after applying [Simplify], such as: $\boxed{V + epsilon} = \boxed{epsilon + X\,;a}$, etc.

If we manage to prove that all the properties in $\mathcal{F}$ hold without using the simplification rules, then the proof is correct. This follows directly from Theorem 2 in [15] for $(\mathcal{B}, \emptyset, \mathcal{F}) \Rightarrow^* (\mathcal{B}, \mathcal{F}', \emptyset)$.

Considering the language ROC! presented in [**?**], the strategy applied for proving the goals in $\mathcal{F}$ is:

$$([\text{Normalize}] \triangleright [\text{Reduce}] \triangleright [\text{Derive}])!$$

We mention that $\triangleright$ has the semantics of an "orelse"-like operator, while ! imposes the prover to apply the indicated strategy for as many times as possible (*i.e.* until it succeeds or it fails).

The command that starts the execution of the prover for the set $\mathcal{F}$ automatically obtained from the last proof is `(check proof .)`. The user dialog for checking the correctness of the proof is:

```
Maude> (check proof .)
Check proof succeeded.
```

This message indicates that CIRC manages to successfully check the correctness of the proof that uses simplification rules.

An important note is that if CIRC is able to prove the current goal only by using equational reduction, then the simplification rule is not applied.

3.4 Generalization [12]

Lets take another look at the generalization rule presented in Section 3.1:

[Generalize]: $(\mathcal{B}, \mathcal{F}, \mathcal{G} \cup \{\forall Y\ \boxed{\theta(t)} = \boxed{\theta(t')}\}) \Rightarrow$
$(\mathcal{B}, \mathcal{F}, \mathcal{G} \cup \{\forall Y\ \boxed{t} = \boxed{t'}\})$,
where $\theta : X \to T_\Sigma(Y)$ is a substitution.

We will now present in more detail this rule and prove that the system is sound even with this extension. Note that for every $i = \overline{0..n}$, there is some $\boxed{e} \in \mathcal{G}_i$ such that one of the following statements holds, each corresponding to the three rules:

[Reduce]: $\mathcal{B} \cup \mathcal{F}_i \vdash \boxed{e}$, $\mathcal{G}_{i+1} = \mathcal{G}_i - \{\boxed{e}\}$ and $\mathcal{F}_{i+1} = \mathcal{F}_i$

[Derive]: $\mathcal{G}_{i+1} = (\mathcal{G}_i - \{\boxed{e}\}) \cup \boxed{\Delta(e)}$ and $\mathcal{F}_{i+1} = \mathcal{F}_i \cup \boxed{e}$

[Generalize]: $\mathcal{G}_{i+1} = (\mathcal{G}_i - \{\boxed{e}\}) \cup \boxed{gen(e)}$ $\mathcal{F}_{i+1} = \mathcal{F}_i$

The function $gen(e)$ replaces each occurrence of a subterm that appears under both sides of e with a variable of the same sort as the subterm.

Consider the set $\mathcal{F} = \bigcup_{i=\overline{0..n}} \mathcal{F}_i$. Let us prove that $\forall i = \overline{0..n}$ $\mathcal{B} \cup \mathcal{F} \vdash \mathcal{G}_i$ by induction over $n - i$. The *base case*, $i = n$, follows directly because $\mathcal{B} \cup \mathcal{F} \vdash \emptyset = G_n$.

For the *inductive step* we assume that $0 \leq i < n$. Let $\boxed{e} \in \mathcal{G}_i$. If $\boxed{e} \in \mathcal{G}_{i+1}$ then $\mathcal{B} \cup \mathcal{F} \vdash \boxed{e}$ by the induction hypothesis. If $\boxed{e} \notin \mathcal{G}_{i+1}$ then we distinguish three cases:

[Reduce]: $\mathcal{B} \cup \mathcal{F}_i \vdash \boxed{e}$; but $\mathcal{F}_i \subseteq \mathcal{F}$, therefore $\mathcal{B} \cup \mathcal{F} \vdash \boxed{e}$

[Derive]: $\boxed{e} \in \mathcal{F}_{i+1}$; but $\mathcal{F}_{i+1} \subseteq \mathcal{F}$, therefore $\mathcal{B} \cup \mathcal{F} \vdash \boxed{e}$

[Generalize]: $\boxed{gen(e)} \in \mathcal{G}_{i+1}$; $\mathcal{B} \cup \mathcal{F} \vdash \boxed{gen(e)}$ (by the induction hypothesis) and $\boxed{gen(e)} \vdash \boxed{e}$ (by Theorem **??**), so $\mathcal{B} \cup \mathcal{F} \vdash \boxed{e}$

We proved that $\forall i = \overline{0..n}$ $\mathcal{B} \cup \mathcal{F} \vdash \mathcal{G}_i$, and therefore, by Theorem 2 in [26], $\mathcal{B} \Vdash G$.

The user needs to pay attention when using the generalization during coinductive proofs because CIRC does not detect over-generalizations. In this way, some goals that are proved without using this rule may not be proved when using it.

3.5 Case Analysis [14]

In this section we present how CIRC theories are specified and a few commands in order to automatically prove properties using case analysis. The coinduction proving engine is extended with the reduction rule [CaseAn]. This rule replaces a conditional equation $t = t'$ `if` $cond$ by a set of equations $\{t = t'$ `if` $cond \wedge \theta(case_i) \mid i \in I\}$ if the case sentence $(\forall Y)(p, \bigvee_{i \in I} case_i)$ is in $\widetilde{\mathcal{C}}$ and $\theta(p)$ is a subterm of t or t'. [CaseAn] is therefore an instance of the rule [itp] corresponding to $\langle t = t'$ `if` $cond, \{t = t'$ `if` $cond \wedge \theta(case_i) \mid i \in I\}\rangle$:

$$\begin{aligned}[\text{CaseAn}] :&(\widetilde{\mathcal{B}}, \mathcal{F}, \mathcal{G} \cup \{\boxed{t} = \boxed{t'} \texttt{ if } cond\}) \Rightarrow \\ &(\widetilde{\mathcal{B}}, \mathcal{F}, \mathcal{G} \cup \{\boxed{t} = \boxed{t'} \texttt{ if } cond \wedge \theta(case_i) \mid i \in I\} \\ &\quad \text{if } (\forall Y)(p, \bigvee\nolimits_{i \in I} case_i) \text{ is in } \widetilde{\mathcal{C}} \text{ and } \theta(p) \text{ is a subterm of } t \text{ or } t'\end{aligned}$$

where $\mathcal{F}$ is the set of frozen axions and $\mathcal{G} \cup \{\boxed{t} = \boxed{t'}$ `if` $cond\}$ is the current set of goals.

Let us use CIRC in order to prove that by merging two infinite sorted streams of natural numbers we obtain a sorted stream. This is not a trivial example; even the proof by hand requires a significant effort. The sorted property can be defined by $isSorted(S) = hd(S) < hd(tl(S)) \wedge isSorted(tl(S))$. The merge operation is defined by $hd(merge(S, S')) = hd(S)$ if $hd(S) < hd(S')$, $hd(merge(S, S')) = hd(S')$ if $hd(S) \geq hd(S')$, $tl(merge(S, S')) = merge(tl(S), S')$ if $hd(S) < hd(S')$, $tl(merge(S, S')) = merge(S, tl(S'))$ if $hd(S) \geq hd(S')$ and can be specified by two guarded equations. We further consider another operation, *toBits* that transforms the provided stream of natural numbers into a stream of bits in this manner: $hd(toBits(S)) = 1$ if $hd(S) < hd(tl(S))$, $hd(toBits(S)) = 0$ if $hd(S) \geq hd(tl(S))$, $tl(toBits(S)) = toBits(tl(S))$. The operation above can also be specified using guarded equations or two conditional equations together with a case sentence for the pattern *hd(S)*. If *ones* denotes the stream of 1's, then the property above is equivalent to:

$$\begin{aligned}&toBits(merge(S_1{:}Stream, S_2{:}Stream)) = ones \text{ if} \\ &\quad isSorted(S_1{:}Stream) = true \wedge isSorted(S_2{:}Stream) = true\end{aligned}$$

Even though *merge* is defined using guarded equations, the algorithm succeeds to find that $merge(*{:}Stream, S{:}Stream)$ and $merge(S{:}Stream, *{:}Stream)$ are special contexts. The context $toBits(*{:}Stream)$ is not found because the definition of *toBits* does not fulfill the criteria checked by the algorithm (the definition of $hd(toBits(S))$ depends on a bigger experiment, $hd(tl(S))$); this can be seen as a limitation of the algorithm. Recall that the problem of special contexts is Π^0_2-complete, so there is no an algorithm able to always find all special contexts.

We present the dialog needed to prove that by merging two sorted streams we obtain a sorted stream. After the tool and specification are loaded, three commands are need to prove this property:

- `(initialize .)`, which sets the initial state of the prover;
- add the property as an initial goal:

```
(add cgoal toBits(merge(S1:Stream,S2:Stream)) = ones
       if isSorted(S1:Stream) = true /\ isSorted(S2:Stream) = true .)
```

- `(coinduction .)`, which launches the circular coinduction engine.

Here is the full dialog with CIRC, where we can see that *merge* defines indeed special contexts.

```
> (initialize .)
Initializing ...
The special contexts are:
merge(*:Stream,V#2:Stream)
merge(V#1:Stream,*:Stream)

> (add cgoal toBits(merge(S1:Stream,S2:Stream)) = ones
           if isSorted(S1:Stream) = true /\ isSorted(S2:Stream) = true .)

> (coinduction .)
Proof succeeded.
  Number of derived goals: 10
  Number of proving steps performed: 39
  Maximum number of proving steps is set to: 256

Proved properties:
  toBits(merge(S1:Stream,S2:Stream)) = ones if
    isSorted(S1:Stream) = true /\ isSorted(S2:Stream) = true
```

The full proof for our property, given as inference rules, can be checked using the command `(show proof .)`. We present one of the rules in which we emphasize the application of [CaseAn]:

```
1. |||-  [* toBits(tl(merge(S1:Stream,S2:Stream))) *] = [* ones *] if
         isSorted(S1:Stream) = true /\ isSorted(S2:Stream) = true /\
         hd(S1:Stream)< hd(S2:Stream) = true
2. |||-  [* toBits(tl(merge(S1:Stream,S2:Stream))) *] = [* ones *] if
         isSorted(S1:Stream) = true /\ isSorted(S2:Stream) = true /\
         hd(S1:Stream)<= hd(S2:Stream) = false
------------------------------------------------------------ [Cases]
|||-  [* toBits(tl(merge(S1:Stream,S2:Stream))) *] = [* ones *] if
      isSorted(S1:Stream) = true /\ isSorted(S2:Stream) = true
```

3.6 Equational Interpolants [14]

The simplification rules, generalization, and the cases analysis are instances of a more general technique, namely the equational interpolants.

If Σ is a signature then a Σ-*equational interpolant* is a pair $\langle e, itp\rangle$, where e is a Σ-equation and itp is a finite set of Σ-equations. A *behavioral specification with interpolants* $\mathcal{B} = (S, (\Sigma, \Delta), (E, \mathcal{I}))$ is a behavioral specification $(S, (\Sigma, \Delta), E)$ together with a set $\mathcal{I}$ of interpolants. An entailment relation for E is extended to $(E, \mathcal{I})$ as follows: in the definition of $\vdash$ E is replaced with $(E, \mathcal{I})$ and a new rule is added:

$$\frac{(E, \mathcal{I}) \vdash itp}{(E, \mathcal{I}) \vdash e} \text{ if } \langle e, itp\rangle \in \mathcal{I} \tag{3.1}$$

If $\vdash$ is an entailment relation for E and $\mathcal{I}$ is a set of interpolants, then we say that $\mathcal{I}$ is $\vdash$*–preserving* if $E \vdash itp$ implies $E \vdash e$, for each $\langle e, itp\rangle \in \mathcal{I}$.

The CIRC engine associates a rewrite rule of the form:

$$\text{[itp]: } (\mathcal{B}, \mathcal{F}, \mathcal{G} \cup \boxed{e}) \Rightarrow (\mathcal{B}, \mathcal{F}, \mathcal{G} \cup \boxed{itp})$$

with each interpolant $\langle e, itp\rangle \in \mathcal{I}$. We write $[\mathsf{itp}] \in \mathcal{I}$ in order to denote that the rule $[\mathsf{itp}]$ is associated with an interpolant from $\mathcal{I}$. The following theorem states that if we enhance the proof system presented in [26] with interpolants, then it remains sound w.r.t. the new entailment relation.

This technique was successfully used to extend the circular coinduction to the case of the coalgebra of regular expressions defined over polynomial functors [2].

3.7 Behavioral Inductive Specifications

A *signature with constructors* is a pair $(S, (\Sigma, \Sigma^{ctor}))$, where (S, Σ) is a many sorted signature and $\Sigma^{ctor} \subseteq \Sigma$ is the subsignature of *constructors*. A sort s is called *inductive sort* if there is a constructor of sort s. An *inductive variable* is a variable of inductive sort. A *goal (conjecture)* is a conditional equation written as $(\forall Y)(\forall Z)t = t'$ `if` *cond*, where Y is a set of inductive variables. A **set of goals** G is a set of goals such that for any two different goals $(\forall Y)(\forall Z)t_1 = t_2$ `if` *cond* and $(\forall Y')(\forall Z')t'_1 = t'_2$ `if` *cond'* in G, we have $Y \cap Z' = \emptyset = Y' \cap Z$, i.e., the inductive variables and the non-inductive ones are not mixed. If $\theta : Y \to T_{\Sigma^{ctor}}$ is a constructor ground substitution, then $\theta[e]$ denotes the equation $(\forall \emptyset)(\forall Z)\theta(t) = \theta(t')$ `if` *cond*. A goal e is an (equational) **inductive consequence** of the specification $(S, (\Sigma, \Sigma^{ctor}), E)$, iff (S, Σ, E) satisfies all the constructor ground instances $\theta[e]$ in the equational deduction system. More details about inductive consequences can be found, e.g., in [4].

In order to exhibit the concepts and results presented in this paper, we consider the following examples.

Example 1 *(Natural numbers)*
The specification NAT, defining natural numbers, consists of a sort Nat, two constructors $0 : \to Nat$ *and* $s : Nat \to Nat$*, an operator* $sum : Nat\ Nat \to Nat$ *defined by the equations* $sum(M, 0) = 0$*,* $sum(M, s(N)) = s(sum(M, N))$*. We will exhibit how the commutativity and associativity of sum can be automatically proved with the basic circular induction proof system; two new lemmas are discovered during the proof process:* $sum(0, M) = M$ *and* $sum(s(M), N) = s(sum(M, N))$*. The associativity is expressed by the unconditional equation* $(\forall P)(\forall M, N)sum(M, sum(N, P)) = sum(sum(M, N), P)$*, where only* P *is viewed as an inductive variable for this goal. In order to exemplify how non-linear goals are handled, we use a max operator defined by* $max(M, 0) = M$*,* $max(0, N) = N$*,* $max(s(M), s(N)) = s(max(M, N))$*, and we show that the non-linear goal* $max(N, N) = N$ *is automatically proved in exactly one derivation step.*

Example 2 *(Generic trees)*
This example exhibits inductive properties over generic data types defined using the mutual recursion and the case when a constructor could have more than one inductive parameter. The specification TREE includes the inductive definition of trees built over a parameter data type Elt and where a node has a list of children. The lists are defined using a sort TList and the constructors

$$nil : \to TList \qquad [\_] : Tree \to TList \qquad \_;\_ : TList\ TList \to TList$$

Note that the concatenation is defined as a constructor; it replaces the usual constructor $cons : Tree\ TList \to TList$*. We made this choice because we want to exhibit how the nonlinear conjectures are handled in more complex examples. The trees are defined using the sort Tree and the constructor* $tr : Elt\ TList \to Tree$*. An example of operation over tree lists (and trees) is*

$$\begin{array}{ll} mirror(nil) = nil & mirror([T]) = [mirror(T)] \\ mirror(L_1; L_2) = mirror(L_2); mirror(L_1) & mirror(tr(E, L)) = tr(E, mirror(L)) \end{array}$$

We show how the goal $mirror(mirror(L{:}TList)) = L{:}TList$ *is automatically proved by circular induction. Note that the property can only be proved together with* $mirror(mirror(T{:}Tree)) = T{:}Tree$*, which is automatically discovered . This example requires an enriched set of special contexts.*

Example 3 *(Generic Lists)*
We consider nonempty lists defined over a generic sort Elt. The constructors for lists are different from those used for lists of trees in Example 2:

$$[\_] : Elt \to List \qquad \_;\_ : Elt\ List \to List$$

We assume a partial order $\leq$ *over Elt and the recursive algorithm computing the maximum of a list:*

$$max(X) = X \qquad max(E_1\,;E_2) = max(E_1, E_2)$$
$$max(E_1\,;E_2\,;L) = max(max(E_1, E_2), max(L))$$

where $max(E_1, E_2)$ returns the maximum between two elements Elt. We show that the correctness of the algorithm computing the maximum over lists can be proved with the circular induction proof system implemented in CIRC enhanced with the equational interpolants [14].

We show that the inductive consequences can be viewed as behavioral properties, i.e., we can define notions like experiments and derivatives, which satisfy the same properties as for the coinductive case, and define the inductive satisfaction similarly to behavioral coinductive satisfaction. We first define the notion of experiment.

Definition 1 *Let $(S, (\Sigma, \Sigma^{ctor}))$ be a signature with constructors.*
*1) A **substitution** $\theta : Y_0 \to \mathcal{T}_\Sigma(Y' \cup Z')$ **defines an equation transformer** $e \mapsto \theta[e]$, where $\theta[e]$ is given as follows: if e is $(\forall Y)(\forall Z)t = t'$ **if** $\wedge_{i\in I} u_i = v_i$, then*
- *$\theta[e]$ is $(\forall Y \backslash Y_0 \cup Y')(\forall Z \cup Z')\theta(t)=\theta(t')$ **if** $\wedge_{i\in I}\theta(u_i)=\theta(v_i)$ when $Y_0 \cap Y \neq \emptyset$,*
- *and $\theta[e]$ is not defined if $Y_0 \cap Y = \emptyset$.*

*2) An **experiment for** Y is a ground Σ^{ctor}-substitution $\theta : Y \to \mathcal{T}_\Sigma^{ctor}$.*
*3) If G is a set of goals, then an **experiment for** G is an experiment for Y, where Y is the set of all inductive variables in G.*

We then show that an experiment can be inductively defined using an appropriate notion of derivative.

Definition 2 *Let $(S, (\Sigma, \Sigma^{ctor}))$ be a signature with constructors. Each constructor $c \in \Sigma^{ctor}$ defines a **derivative** $\delta_c = \{\delta_{c,y}\}$ as follows:*
- *$\delta_{c,y}$ is the equation transformer $y \mapsto c$ for each $y \in Y$ and constructor constant c with the same sort;*
- *$\delta_{c,y}$ is the equation transformer $y \mapsto c(y_1, \ldots, y_n)$ for each $y \in Y$ of sort s and $c : s_1 \ldots s_n \to s$, where y_i is a fresh variable of sort s_i for $i = 1, \ldots, n$; in this case we write $y_i \sim y$ for each i such that $s_i = s$ (y_i can be seen as a new **incarnation** of y);*
- *if e is $(\forall Y)e'$, then $\delta_c[e] = \{\delta_{c,y}[e] \mid y \in Y\}$.*

Let Δ denote the set of derivatives $\{\delta_c \mid c \in \Sigma^{ctor}\}$.

Example 4 *Let us consider the specification TREE defined in Example 2. If e is $mirror(mirror(T{:}Tree)) = T{:}Tree$ and c is $tr(E{:}Elt, L{:}TList)$, then $\delta_c[\boxed{e}]$ consists of $mirror(mirror(tr(E,L)))=tr(E,L)$. If e is $mirror(mirror(L{:}TList))=L$ and c is $L_1{:}TList; L_2{:}TList$, then $\delta_c[e]$ consists of $mirror(mirror(L_1; L_2)) = L_1; L_2$ with $L \sim L_1$ and $L \sim L_2$.*

We will see later taht the relation $\sim$ between frozen variables plays also a key role in the definition of the circular induction proof system.

Proposition 1 *Let e be an equation and let θ be an experiment for e. Then there exist certain constructors (not necessarily distinct) $c_1, \ldots, c_n$ and inductive variables $y_1, \ldots, y_n$ such that $\theta[e] = \delta_{c_1,y_1}[\ldots\delta_{c_n,y_n}[e]\ldots]$.*

For instance, if e is $(\forall N)max(N, N) = N$ and $\theta(N) = s(0)$, then $\theta[e]$ is the same with $\delta_0[\delta_s[e]]$.

Proposition 1 says in fact that the experiments and can be inductively defined in a similar way to the coinductive case. It is worth noting to emphasize the duality between the two notions of experiment: for the coinductive case it is a context for a hidden sort, for the inductive case it is a substitution for a set of inductive sorts. The inductive definition of the experiments and their view as equation transformers are fully exploited by circular induction proof system we define later. From now on we often write $(S, (\Sigma, \Delta))$ for $(S, (\Sigma, \Sigma^{ctor}))$ in order to have an uniform notation and to stress that a signature with constructors defines a set of equational transformers. This double notation is not confusing because if we know Σ^{ctor} then we can compute Δ and, reciprocally, if we know Δ then we can compute Σ^{ctor}. Having defined Δ, it does make sense to consider Δ-contextual entailment systems $\vdash$ for the signature with constructors and define inductive consequences parameterized over these systems, similar it is done for coinductive behavioral consequences.

Definition 3 *Let $\mathcal{B} = (S, (\Sigma, \Delta), E)$ be a specification with constructors, $\vdash$ an Δ-contextual entailment, and let e be an equation. We say that $\mathcal{B}$ **inductively satisfies** e **w.r.t.** $\vdash$, written $\mathcal{B} \Vdash e$, if $\mathcal{B} \vdash \theta[e]$ for each experiment θ for Y.*

It is easy to see that e is an inductive consequence of $\mathcal{B}$ iff $\mathcal{B}$ inductively satisfies e w.r.t. the equational deduction entailment. If e has no inductive variables, then $\mathcal{B} \Vdash e$ iff $\mathcal{B} \vdash e$.

3.8 Behavioral Circular Induction

Similar to circular coinduction, a key notion for circular induction is that of frozen equation. If for the case of circular coinduction, the frozen equations inhibit the use of the frozen equations underneath proper contexts, for circular induction the frozen equation inhibit their use "over" proper substitution. The frozen equations help us to capture the informal notion of "circular inductive reasoning" elegantly, rigorously, and generally (modulo a restricted form of equational reasoning).

Definition 4 *1. A **frozen form of a variable** $y{:}s$ is a distinguished constant $y.s$ of sort s.*
*2. A **frozen form of a goal** $e \equiv (\forall Y)(\forall Z)t = t'$ if cond is the equation $\boxed{e} \equiv (\forall \emptyset)(\forall Z)\,(t = t'$ if cond$)[y.s/y{:}s \mid y{:}s \in Y]$, where the inductive variables $y{:}s \in Y$ are replaced with their frozen forms $y.s$.*
3. The set of frozen variables of a goal e is $FVar(\boxed{e}) = \{y.s \mid y{:}s \in Y\}$.
4. If F is a set of goals, then $\boxed{F} = \{\boxed{e} \mid e \in F\}$ and $FVar(\boxed{F}) = \cup_{e \in F} FVar(\boxed{e})$.

If e is an equations, F a set of equations, θ an experiment, and Θ a set of experiments, then we write $\theta[\![\boxed{e}]\!]$ for $\boxed{\theta[e]}$ and $\Theta[\![\boxed{F}]\!]$ for $\boxed{\Theta[F]}$.

Let $(S, \Sigma, E) \cup \boxed{F} \vdash \boxed{e}$ denote the entailment $(S, \Sigma(Y_F \cup Y_e), E \cup \boxed{F}) \vdash \boxed{e}$, where Y_F is the set of the all designated inductive variables of the equations F, Y_e is the set of the all designated inductive variables of e, and $\Sigma(Y)$ is the signature Σ enriched with the frozen forms of the variables Y (recall that these are constants). We often write $E \cup \boxed{F} \vdash \boxed{e}$ for $(S, \Sigma, E) \cup \boxed{F} \vdash \boxed{e}$.

The **circular induction proof system** implemented by CIRC consists of the following inference rules, represented as conditional reduction rules:

[Reduce]
$(\mathcal{B}, \mathcal{F}, \mathcal{G} \cup \{\boxed{e}\}) \Rightarrow (\mathcal{B}, \mathcal{F}, \mathcal{G})$ if $\mathcal{B} \cup \mathcal{F} \vdash \boxed{e}$

[Derive]
$(\mathcal{B}, \mathcal{F}, \mathcal{G} \cup \{\boxed{e}\}) \Rightarrow (\mathcal{B}, \mathcal{F} \cup \{\boxed{e}\} \cup \mathcal{H}(\boxed{e}, FVar(\Delta[\![\boxed{e}]\!])), \mathcal{G} \cup \Delta[\![\boxed{e}]\!])$ if $\mathcal{B} \cup \mathcal{F} \nvdash \boxed{e}$

where $\mathcal{H}(\boxed{e}, FV) = \{\boxed{e}[y'/y] \mid y' \sim y, y' \in FV, y \in FVar(\boxed{e})\}$.

The rule [Reduce] eliminates a goal whenever it can be deduced using the the basic entailment system. The key role is played by [Derive] rule: if a goal cannot be deduced with the initial entailment relation, it together with the equations "similar via an incarnation" to it are added as frozen hypotheses and its derived goals are added as new proof obligations.

Example 5 *(successful examples)*
Assume that $\mathcal{B}$ is the specification NAT defined in Example 1 and that $\mathcal{G}$ consists of the goal $sum(m, n) = sum(n, m)$, where m is the frozen form of $M{:}Nat$ and n is the frozen form of $N{:}Nat$. We also assume that the basic entailment relation $\vdash$ is the equational deduction. The initial $\mathcal{F}$ is empty. The derivation tree of the for proving the above goal is described by the following CIRC output:

```
==========================================

---> commutativity of sum

rewrites: 369 in 1ms cpu (2ms real) (184592 rewrites/second)

Goal added: sum(N:Nat,M:Nat) = sum(M:Nat,N:Nat)

rewrites: 13225 in 40ms cpu (41ms real) (322608 rewrites/second)

Induction started with the variables:

M:Nat N:Nat

Proof succeeded.

Number of derived goals: 6

Number of proving steps performed: 29

Maximum number of proving steps is set to: 256

Proved properties:
```

```
s(sum(M#2:Nat,N#0:Nat)) = sum(s(M#2:Nat),N#0:Nat)
sum(0,N#0:Nat) = N#0:Nat
sum(N#0:Nat,M#1:Nat) = sum(M#1:Nat,N#0:Nat)
sum(N:Nat,M:Nat) = sum(M:Nat,N:Nat)
rewrites: 3471 in 12ms cpu (13ms real) (267041 rewrites/second)
========================================
|- s(sum(M#2:Nat,s(N#4:Nat))) = sum(s(M#2:Nat),s(N#4:Nat))
---------------------------------------------------------- [Reduce]
|||- s(sum(M#2:Nat,s(N#4:Nat))) = sum(s(M#2:Nat),s(N#4:Nat))
|- s(sum(M#2:Nat,0)) = sum(s(M#2:Nat),0)
---------------------------------------------------------- [Reduce]
|||- s(sum(M#2:Nat,0)) = sum(s(M#2:Nat),0)
1. |||- s(sum(M#2:Nat,0)) = sum(s(M#2:Nat),0)
2. |||- s(sum(M#2:Nat,s(N#4:Nat))) = sum(s(M#2:Nat),s(N#4:Nat))
---------------------------------------------------------- [Derive-ind]
|||- s(sum(M#2:Nat,N#0:Nat)) = sum(s(M#2:Nat),N#0:Nat)
|- s(sum(M#2:Nat,N#0:Nat)) = sum(s(M#2:Nat),N#0:Nat)
---------------------------------------------------------- [Normalize]
|- sum(N#0:Nat,s(M#2:Nat)) = sum(s(M#2:Nat),N#0:Nat)
|- s(N#3:Nat) = sum(0,s(N#3:Nat))
---------------------------------------------------------- [Reduce]
|||- s(N#3:Nat) = sum(0,s(N#3:Nat))
|- 0 = sum(0,0)
---------------------------------------------------------- [Reduce]
|||- 0 = sum(0,0)
1. |||- 0 = sum(0,0)
2. |||- s(N#3:Nat) = sum(0,s(N#3:Nat))
---------------------------------------------------------- [Derive-ind]
|||- sum(0,N#0:Nat) = N#0:Nat
|- N#0:Nat = sum(0,N#0:Nat)
---------------------------------------------------------- [Normalize]
|- sum(N#0:Nat,0) = sum(0,N#0:Nat)
1. |||- sum(N#0:Nat,0) = sum(0,N#0:Nat)
2. |||- sum(N#0:Nat,s(M#2:Nat)) = sum(s(M#2:Nat),N#0:Nat)
---------------------------------------------------------- [Derive-ind]
|||- sum(N#0:Nat,M#1:Nat) = sum(M#1:Nat,N#0:Nat)
========================================
```

In what follows we present the way CIRC manages data during a proving session by induction. The inference rules are implemented using a similar mechanism to that used for coinductive proofs. The main difficulty raised by circular induction is the internal management of special hypotheses $\mathcal{H}(\boxed{e}, FV) \cup \mathcal{H}^*(\mathcal{F}, FV)$. The number of these hypotheses could exponentially increase and their computation become expensive. We use the subsort facility in order to drastically reduce the number of stored hypotheses. We store only generic hypotheses represented with variables of fresh subsorts and each new generated constant is declared as habitant to an appropriate such subsort. Then the set of all needed special hypotheses are obtained by instantiation. For that we need to distinguish between initial variables and their incarnations.

Definition 5 *A frozen inductive variable $y.s$ is a* ***source*** *iff* $(\nexists y'.s) y.s \sim^+ y'.s$.

For each frozen induction variable $y'.s$ we define the operation *source* such that $source(y'.s) = y.s$ if $y'.s \sim^+ y.s$ and $y.s$ is a source. Each time we add a new source $y.s$, either at the beginning or during a proof session, we enrich the specification with a fresh subsort $s\langle y \rangle$ such that $s\langle y \rangle < s$. CIRC implements a variant of the rule [Derive] such that instead of adding a specific hypothesis (equation) for each of the generated variables, it adds only one hypothesis that may be used by all the frozen variables with a common source. This leaves the set of hypotheses uncluttered and helps improving the speed of the matching engine:

$$\text{[Derive]}\quad (\mathcal{B}, \mathcal{F}, \mathcal{G} \cup \{\boxed{e}\}) \Rightarrow (\mathcal{B}', \mathcal{F} \cup \mathcal{H}'(\boxed{e}, FVar(\Delta[\![\boxed{e}]\!])), \mathcal{G} \cup \Delta[\![\boxed{e}]\!]) \quad \text{if } \mathcal{B} \cup \mathcal{F} \not\vdash \boxed{e}$$

where $\mathcal{H}'(\boxed{e}, FV) = \{\boxed{e}[y'{:}s\langle source(y)\rangle / y.s] \mid y'.s \in FV, y.s \in FVar(\boxed{e})\}$ and $\mathcal{B}'$ is $\mathcal{B}$ enriched with the constants $y'.s'$ of sort $s'\langle source(y')\rangle$ for each $y'.s' \in FVar(\Delta[\![\boxed{e}]\!])$. We have $\boxed{f} \in \mathcal{H}^*(\mathcal{F}, FV) \cup \mathcal{H}(\boxed{e}, FVar(\Delta[\![\boxed{e}]\!]))$ iff $\boxed{f}$ is an instance of a hypothesis in $\mathcal{F} \cup \mathcal{H}'(\boxed{e}, FVar(\Delta[\![\boxed{e}]\!]))$.

In order to understand the special hypothesis management algorithm better, let us analyse the derivation steps from Examples 2:

```
Introduced theory TREE

rewrites: 263 in 1ms cpu (1ms real) (263000 rewrites/second)

Goal added: mirror(mirror(L:TList)) = L:TList

rewrites: 7955 in 25ms cpu (25ms real) (306008 rewrites/second)

Induction started with the variables:

L:TList

Proof succeeded.

Number of derived goals: 4

Number of proving steps performed: 19

Maximum number of proving steps is set to: 256

Proved properties:

mirror(mirror(L#3:Tree)) = L#3:Tree

mirror(mirror(L#0:TList)) = L#0:TList

mirror(mirror(L:TList)) = L:TList

rewrites: 1772 in 8ms cpu (9ms real) (196910 rewrites/second)

========================================

|- mirror(mirror(tr(L#4:Elt,L#5:TList))) = tr(L#4:Elt,L#5:TList)

---------------------------------------------------------- [Reduce]

|||- mirror(mirror(tr(L#4:Elt,L#5:TList))) = tr(L#4:Elt,L#5:TList)
```

```
|||- mirror(mirror(tr(L#4:Elt,L#5:TList))) = tr(L#4:Elt,L#5:TList)
---------------------------------------------------------- [Derive-ind]
|||- mirror(mirror(L#3:Tree)) = L#3:Tree
|- mirror(mirror(L#1:TList ; L#2:TList)) = L#1:TList ; L#2:TList
---------------------------------------------------------- [Reduce]
|||- mirror(mirror(L#1:TList ; L#2:TList)) = L#1:TList ; L#2:TList
|- mirror(mirror(nil)) = nil
---------------------------------------------------------- [Reduce]
|||- mirror(mirror(nil)) = nil
1. |||- mirror(mirror(nil)) = nil
2. |||- mirror(mirror(L#1:TList ; L#2:TList)) = L#1:TList ; L#2:TList
3. |||- mirror(mirror(L#3:Tree)) = L#3:Tree
---------------------------------------------------------- [Derive-ind]
|||- mirror(mirror(L#0:TList)) = L#0:TList
==========================================
Maude> Bye.
```

Excepting the management of the special contexts and the meanings of the frozen equations, the engines implementing circular induction and circular coinduction, respectively, are similar. An immediate main advantage is that the mechanisms added to increase the power of one engine can be used for free by the other one. We exemplify this feature showing how the equational interpolants are used to prove the correctness of the maximum over lists defined in Example 3. To prove the correctness, the partial order $\leq$ is extended over lists: $L_1 \leq L_2$ iff each element of L_1 is less or equal than any element of L_2. The goal is $L \leq max(L)$. During the proof, the following intermediate goal is reached: $E \leq max(E\,;L)\ and\ L \leq max(E\,;L) = true$. This consists in fact of two subgoals which can be separately proved by induction. The splitting of such a goal is accomplished by adding to the specification the following simplification rule:

$$\frac{(B_1{:}Bool\ and\ B_2{:}Bool) = true}{B_1{:}Bool = true \qquad B_2{:}Bool = true}$$

Now the proof of the initial goal is achieved with a tactic where a simplification is tried before each derivation.

Here is the output displayed by CIRC:

```
Introduced theory MAXLIST
rewrites: 274 in 1ms cpu (1ms real) (137068 rewrites/second)
Goal added: L:List <= max(L:List) = true
rewrites: 298 in 1ms cpu (0ms real) (298000 rewrites/second)
Induction started with the variables:
L:List
rewrites: 20818 in 102ms cpu (103ms real) (202145 rewrites/second)
Proof succeeded.
Number of derived goals: 10
Number of proving steps performed: 43
```

```
Maximum number of proving steps is set to: 256
Proved properties:
L#11:List <= max(L#1:Elt,L#7:Elt,max(L#10:Elt ; L#11:List)) = true
L#8:List <= max(L#1:Elt,L#7:Elt,max(L#8:List)) = true
L#2:List <= max(L#1:Elt ; L#2:List) = true
L#1:Elt <= max(L#1:Elt ; L#2:List) = true
L#0:List <= max(L#0:List) = true
L:List <= max(L:List) = true
rewrites: 8972 in 22ms cpu (22ms real) (390137 rewrites/second)
========================================
|- L#3:Elt <= max(L#3:Elt) = true
------------------------------------------------------------ [Reduce]
|||- L#3:Elt <= max(L#3:Elt) = true
|- L#9:Elt <= max(L#1:Elt ; L#9:Elt) = true
------------------------------------------------------------ [Reduce]
|||- L#9:Elt <= max(L#1:Elt ; L#9:Elt) = true
|- L#12:Elt <= max(L#1:Elt,L#7:Elt,max(L#12:Elt)) = true
------------------------------------------------------------ [Reduce]
|||- L#12:Elt <= max(L#1:Elt,L#7:Elt,max(L#12:Elt)) = true
|- L#15:Elt <= max(L#1:Elt,L#7:Elt,max(L#10:Elt ; L#15:Elt)) = true
------------------------------------------------------------ [Reduce]
|||- L#15:Elt <= max(L#1:Elt,L#7:Elt,max(L#10:Elt ; L#15:Elt)) = true
|- (L#13:Elt ; L#14:List)<= max(L#1:Elt,L#7:Elt,max(L#10:Elt ; L#13:Elt ; L#14:List)) = true
------------------------------------------------------------ [Reduce]
|||- (L#13:Elt ; L#14:List)<= max(L#1:Elt,L#7:Elt,max(L#10:Elt ; L#13:Elt ; L#14:List)) = true
1. |||- (L#13:Elt ; L#14:List)<= max(L#1:Elt,L#7:Elt,max(L#10:Elt ; L#13:Elt ; L#14:List)) = true
2. |||- L#15:Elt <= max(L#1:Elt,L#7:Elt,max(L#10:Elt ; L#15:Elt)) = true
------------------------------------------------------------ [Derive-ind]
|||- L#11:List <= max(L#1:Elt,L#7:Elt,max(L#10:Elt ; L#11:List)) = true
|- L#11:List <= max(L#1:Elt,L#7:Elt,max(L#10:Elt ; L#11:List)) = true
------------------------------------------------------------ [Normalize]
|- (L#10:Elt ; L#11:List)<= max(L#1:Elt,L#7:Elt,max(L#10:Elt ; L#11:List)) = true
1. |||- (L#10:Elt ; L#11:List)<= max(L#1:Elt,L#7:Elt,max(L#10:Elt ; L#11:List)) = true
2. |||- L#12:Elt <= max(L#1:Elt,L#7:Elt,max(L#12:Elt)) = true
------------------------------------------------------------ [Derive-ind]
|||- L#8:List <= max(L#1:Elt,L#7:Elt,max(L#8:List)) = true
|- L#8:List <= max(L#1:Elt,L#7:Elt,max(L#8:List)) = true
```

```
------------------------------------------------------------ [Normalize]
|- (L#7:Elt ; L#8:List)<= max(L#1:Elt ; L#7:Elt ; L#8:List) = true
1. |||- (L#7:Elt ; L#8:List)<= max(L#1:Elt ; L#7:Elt ; L#8:List) = true
2. |||- L#9:Elt <= max(L#1:Elt ; L#9:Elt) = true
------------------------------------------------------------ [Derive-ind]
|||- L#2:List <= max(L#1:Elt ; L#2:List) = true
|- L#1:Elt <= max(L#1:Elt ; L#6:Elt) = true
------------------------------------------------------------ [Reduce]
|||- L#1:Elt <= max(L#1:Elt ; L#6:Elt) = true
|- L#1:Elt <= max(L#1:Elt ; L#4:Elt ; L#5:List) = true
------------------------------------------------------------ [Reduce]
|||- L#1:Elt <= max(L#1:Elt ; L#4:Elt ; L#5:List) = true
1. |||- L#1:Elt <= max(L#1:Elt ; L#4:Elt ; L#5:List) = true
2. |||- L#1:Elt <= max(L#1:Elt ; L#6:Elt) = true
------------------------------------------------------------ [Derive-ind]
|||- L#1:Elt <= max(L#1:Elt ; L#2:List) = true
|-
1. [* L#1:Elt <= max(L#1:Elt ; L#2:List) *] = [* true *]
2. [* L#2:List <= max(L#1:Elt ; L#2:List) *] = [* true *]
------------------------------------------------------------ [Simplify]
|- L#1:Elt <= max(L#1:Elt ; L#2:List<L>)and L#2:List<L> <= max(L#1:Elt ; L#2:List<L>) = true
|- L#1:Elt <= max(L#1:Elt ; L#2:List)and L#2:List <= max(L#1:Elt ; L#2:List) = true
------------------------------------------------------------ [Normalize]
|- (L#1:Elt ; L#2:List)<= max(L#1:Elt ; L#2:List) = true
1. |||- (L#1:Elt ; L#2:List)<= max(L#1:Elt ; L#2:List) = true
2. |||- L#3:Elt <= max(L#3:Elt) = true
------------------------------------------------------------ [Derive-ind]
|||- L#0:List <= max(L#0:List) = true
========================================
```

Bibliography

[1] E. Balland, P. Brauner, R. Kopetz, P.-E. Moreau, and A. Reilles. Tom: Piggybacking Rewriting on Java. In F. Baader, editor, *RTA*, volume 4533 of *Lecture Notes in Computer Science*, pages 36–47. Springer-Verlag, 2007.

[2] M. M. Bonsangue, G. Caltais, E.-I. Goriac, D. Lucanu, J. J. M. M. Rutten, and A. Silva. A decision procedure for bisimilarity of generalized regular expressions. In *SBMF*, volume 6527 of *Lecture Notes in Computer Science*, pages 226–241. Springer, 2010.

[3] P. Borovanský, C. Kirchner, H. Kirchner, and C. Ringeissen. Rewriting with Strategies in ELAN: A Functional Semantics. *Int. J. Found. Comput. Sci.*, 12(1):69–95, 2001.

[4] A. Bundy. The automation of proof by mathematical induction. In *Handbook of Automated Reasoning*, pages 845–911. Elsevier and MIT Press, 2001.

[5] G. Caltais, E. Goriac, D. Lucanu, and G. Grigoraş. A Rewrite Stack Machine for ROC! In *10th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*. IEEE Computer Society, 2008.

[6] R. Diaconescu and K. Futatsugi. Behavioral coherence in object-oriented algebraic specification. *JUCS*, 6(1):74–96, 2000.

[7] S. Eker, N. Martí-Oliet, J. Meseguer, and A. Verdejo. Deduction, strategies, and rewriting. *Electron. Notes Theor. Comput. Sci.*, 174(11):3–25, 2007.

[8] J. Goguen. Types as theories. In *Topology and Category Theory in Computer Science*, pages 357–390. Oxford, 1991.

[9] J. Goguen, K. Lin, and G. Roşu. Circular coinductive rewriting. In *ASE '00: Proceedings of the 15th IEEE international conference on Automated software engineering*, pages 123–132, Washington, DC, USA, 2000. IEEE.

[10] J. Goguen, K. Lin, and G. Roşu. Conditional circular coinductive rewriting with case analysis. In *WADT*, volume 2755 of *LNCS*, pages 216–232. Springer, 2002.

[11] J. Goguen and G. Malcolm. A hidden agenda. *J. of TCS*, 245(1):55–101, 2000.

[12] E. Goriac, G. Caltais, and D. Lucanu. Simplification and Generalization in CIRC. In *12th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*. IEEE Computer Society, 2009.

[13] E. Goriac, G. Caltais, D. Lucanu, O. Andrei, and G. Grigoraş. Patterns for Maude Metalanguage Applications. In *Proceedings of WRLA*, 2008.

[14] E. Goriac, D. Lucanu, and G. Roşu. Automating coinduction with case analysis. In *ICFEM*, volume 6447 of *Lecture Notes in Computer Science*, pages 220–236, 2010.

[15] 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.

[16] R. Hennicker and M. Bidoit. Observational logic. In *Proceedings of AMAST'98*, volume 1548 of *LNCS*, pages 263–277. Springer, 1999.

[17] C. Kirchner, H. Kirchner, and M. Vittek. Designing Constraint Logic Programming Languages using Computational Systems. In P. Van Hentenryck and V. Saraswat, editors, *Principles and Practice of Constraint Programming. The Newport Papers.*, chapter 8, pages 131–158. MIT Press, 1995.

[18] D. Lucanu, E.-I. Goriac, G. Caltais, and G. Roşu. CIRC : A behavioral verification tool based on circular coinduction. In *CALCO 2009*, volume 5728 of *LNCS*, pages 433–442. Springer, 2009.

[19] D. Lucanu and G. Roşu. Circ: A Circular Coinductive Prover. In T. Mossakowski and et al., editors, *2nd Conference on Algebra and Coalgebra in Computer Science (CALCO 2007)*, volume 4624 of *Lecture Notes in Computer Science*, pages 372–378. Springer, 2007.

[20] D. Lucanu and G. Roşu. Circular Coinduction with Special Contexts. Technical Report UIUCDCS-R-2009-3039, University of Illinois at Urbana-Champaign, 2009. Submitted.

[21] D. Lucanu, G. Roşu, and G. Grigoraş. Regular strategies as proof tactics for circ. *Electron. Notes Theor. Comput. Sci.*, 204:83–98, 2008.

[22] 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.

[23] D. Lucanu and G. Roşu. Circular coinduction with special contexts. In *ICFEM*, volume 5885 of *LNCS*, pages 639–659. Springer, 2009.

[24] P. Padawitz. Swinging data types: Syntax, semantics, and theory. In *Proceedings, WADT'95*, volume 1130 of *LNCS*, pages 409–435. Springer, 1996.

[25] G. Roşu. *Hidden Logic*. PhD thesis, University of California at San Diego, 2000.

[26] G. Roşu and D. Lucanu. Circular Coinduction – A Proof Theoretical Foundation. In *CALCO 2009*, volume 5728 of *LNCS*, pages 127–144. Springer, 2009.

[27] E. Visser. Stratego: A Language for Program Transformation based on Rewriting Strategies. System Description of Stratego 0.5. In A. Middeldorp, editor, *RTA*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–361. Springer-Verlag, 2001.

[28] H. Zantema. Well-definedness of streams by termination. Submitted, 2009.