



## CIRC Prover

Version 1.4

(DRAFT)

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

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

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

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

July 18, 2009

# Contents

<b>1</b>	<b>CIRC Tutorial</b>	<b>2</b>
1.1	Introduction . . . . .	2
1.2	Algebraic Behavioural Specifications . . . . .	2
1.3	Inductive Properties . . . . .	6
<b>2</b>	<b>CIRC User Manual</b>	<b>11</b>
2.1	Getting Started . . . . .	11
2.2	Running the Examples . . . . .	12
2.3	Commands . . . . .	13
2.4	Cautions . . . . .	20

# Chapter 1

## CIRC Tutorial

### 1.1 Introduction

CIRC [9] is an automated circular coinductive prover that is implemented as an extension of Maude. CIRC implements the *circularity principle*, which generalizes circular coinductive deduction [4] and can be expressed in plain English as follows. Assume that each equation of interest (to be proved)  $e$  admits a *frozen form*  $fr(e)$  (sometimes written as  $\boxed{e}$ ) and a set of derived equations, its *derivatives*,  $Der(e)$ . The circularity principle requires that the following rule be valid: if from the hypothesis  $\mathcal{H}$  together with  $fr(e)$  we can deduce  $Der(e)$ , then  $e$  is a consequence of  $\mathcal{H}$ . When  $fr(e)$  freezes the equation at the top as in [4], the circularity principle becomes circular coinduction. Interestingly, when the equation is frozen at the bottom on a variable, then it becomes a structural induction (on that variable) derivation rule. This way, CIRC supports both coinduction and induction as projections of a more general principle. In this paper, we concentrate on CIRC's coinductive capabilities.

**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 brilliant ideas.

### 1.2 Algebraic Behavioural Specifications

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., [3, 5]) *observational logic* in works by Hennicker, Bidoit and many others (e.g., [6]), *swinging types* in works by Padawitz [7], *coherent hidden algebra* in Diaconescu [2], *hidden logic* in Roşu [8], 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<sup>1</sup>, 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*.

Here is a very simple example. Let us consider a C++ program  $P$  that reads a file and then outputs all but the whitespace characters in the file:

---

<sup>1</sup>I.e., types whose values are infinite structures.

```

char c;
string fname;
cout << "Input file name: ";
cin >> fname;
fstream f(fname.c_str(), ios_base::in);
while (!f.eof())
{
    c = ' ';
    f >> c;
    cout << c;
}

```

The execution of  $P$  on the input file with content `a 123 bc<EOF>` and the execution of  $P$  on the input file with content `a<CR>1<CR>2<CR>3<CR>b<CR>c<EOF>` (`<CR>` denotes the new line character and `<EOF>` the end of file character) give the same output: `a123bc`. Even though the two files are not identical, they have the same behavior under the experiments made using  $P$ . Files can be modeled as streams. Streams with white spaces can be specified as the Maude [1] theory:

```

(fth STREAM is sorts Alpha Elt Stream .
  subsort Alpha < Elt .

  op nil : -> Stream .
  ops eof CR SP : -> Elt .

  ops isWhSp : Elt -> Bool .
  eq isWhSp(CR) = true .
  eq isWhSp(SP) = true .
  eq isWhSp(C:Alpha) = false .

  var S : Stream . var E : Elt .

  op getElt : Stream -> Elt .
  eq getElt(nil) = eof .

  op delElt : Stream -> Stream .
  eq delElt(nil) = nil .

  op hd : Stream -> Elt .
  ceq hd(S) = hd(delElt(S)) if isWhSp(getElt(S)) = true .
  ceq hd(S) = getElt(S) if isWhSp(getElt(S)) = false .

  op tl : Stream -> Stream .
  ceq tl(S) = tl(delElt(S)) if isWhSp(getElt(S)) = true .
  ceq tl(S) = delElt(S) if isWhSp(getElt(S)) = false .

endfth)

```

The sort `Elt` models all the characters composing a stream, the sort `Alpha` models the non-white space characters, and the sort `Stream` models the streams. The constant `nil` models the null (empty) stream. The constants `eof`, `CR`, and `SP` model the following special characters: end of file, new line, and space, respectively. The boolean operation `isWhSp` returns `true` if and only if the character given as parameter is a white space character. `getElt` and `delElt` are “low-level” operations which do not ignore white spaces, while `hd` and `tl` are “high-level” ignoring the white spaces. The later model the C++ operator `>>`. To make the theory above behavioral, we define `Alpha` and `Elt` as visible sorts, `Stream` as a hidden sort. If we consider  $\Delta$  to contain `hd(*:Stream)` and `tl(*:Stream)`, then  $\Delta$ -experiments have the form `hd(*:Stream)`, `hd(tl(*:Stream))`, `hd(tl(tl(*:Stream)))`, ..., and correspond to the operations included

in the reading loop of the program  $P$ . In this way we are interested only in experiments of the C++ operator  $\gg$ . Two streams  $s_1$  and  $s_2$  are behavioural equivalent iff  $\text{hd}(s_1) = \text{hd}(s_2)$ ,  $\text{hd}(\text{tl}(s_1)) = \text{hd}(\text{tl}(s_2))$ ,  $\text{hd}(\text{tl}(\text{tl}(s_1))) = \text{hd}(\text{tl}(\text{tl}(s_2)))$ , and so on.

Formally, a *behavioral specification* is a triple  $\mathcal{B} = (\Sigma, \Delta, E)$ , where  $\Sigma$  is an algebraic signature,  $\Delta$  is a set of behavioral operations, and  $E$  is a set of  $\Sigma$ -equations. We assume  $\text{Sorts}(\Sigma) = V \cup H$ , where  $V$  is a subset of *visible sorts*,  $H$  is a subset of *hidden sorts*, and  $V \cap H = \emptyset$ . Let  $=_E$  denote the standard equational derivability congruence over  $T_\Sigma(\mathcal{X})$  with the equations  $E$  (the  $E$ -equality).

A  $\Delta$ -*experiment* for the hidden sort  $h \in H$  is inductively defined as follows: each behavioral operation for the hidden sort  $h \in H$  with visible result sort is a  $\Delta$ -experiment for  $h$ ; if  $\gamma$  is a  $\Delta$ -experiment for  $h'$  and  $\delta$  a behavioral operation for  $h$  with result sort  $h'$ , then  $\gamma[\delta/*:h']$  is a  $\Delta$ -experiment for  $h$ .

The  $\Delta$ -*behavioral equivalence*  $\equiv_\Delta$  over  $T_\Sigma(\mathcal{X})$  is the  $E$ -equality (closure under equational deduction with  $E$ ) generated by the following: for each visible sort  $v \in V$ ,  $\equiv_{\Delta,v}$  is  $=_{E,v}$ ; if  $h \in H$  and  $t, t' \in T_\Sigma(\mathcal{X})_h$  then  $t \equiv_\Delta t'$  iff  $\gamma[t/*:h] =_E \gamma[t'/*:h]$  for each  $\Delta$ -experiment  $\gamma$  for  $h$ . (We here assume that operations are behaviorally congruent.) Note that the relation  $\equiv_\Delta$  is not algorithmic, because one needs an infinite number of experiments to decide it; the basis for the  $\Pi_2^0$  result in [8] is the observation that *for each experiment there is some proof* (thanks for the completeness of equational deduction).  $\mathcal{B}$  *behaviorally satisfies* the behavioral  $\Sigma$ -equation  $e$  of the form  $(\forall X) t = t'$  if  $t_1 = t'_1 \wedge \dots \wedge t_n = t'_n$ , written  $\mathcal{B} \models_\Delta e$ , iff  $t \equiv_\Delta t'$  whenever  $t_1 =_E t'_1, \dots, t_n =_E t'_n$  (note the implicit application of the lemma of constants).

CIRC can prove entirely automatically the behavioral equality

$$\text{a SP 1 2 3 SP b c nil} \equiv_\Delta \text{a CR 1 CR 2 CR 3 CR b CR c nil}.$$

Since a behavioral specification includes more information than a usual Full Maude specification, we designed an interface allowing the user to introduce behavioral operations, case sentences, goals, and CIRC commands. A typical scenario of using CIRC is as follows. The objects whose behavioral properties are investigated are specified using Maude. Here is an example defining the two streams (a stream constructor needs to also be added):

```
(fth STREAM-EX is
  inc STREAM .
  ops a b c 1 2 3 : -> Alpha .
  ops s1 s2 : -> Stream .
  op _ : Elt Stream -> Stream .
  var E : Elt . var S : Stream .
  eq getElt(E S) = E .
  eq delElt(E S) = S .
  eq s1 = a SP 1 2 3 SP b c nil .
  eq s2 = a CR 1 CR 2 CR 3 CR b CR c nil .
endfth)
```

Behavioral specifications are defined using `(cth ... endcth)` modules. These modules extend the `(fth ... endfth)` Maude functional theories allowing the declaration of behavioral operations (derivatives) :

```
(cth B-STREAM-EX is
  including STREAM-EX .
  der hd(*:Stream) .
  der tl(*:Stream) .
endcth)
```

We can now introduce CIRC proving commands. We first define the goal:

```

Maude> (add goal s1 = s2 .)
rewrites: 354 in 4348694219ms cpu (14ms real) (0 rewrites/second)
Goal s1 = s2 added.

```

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

```

Maude> (coinduction .)
rewrites: 7874 in 4341571417ms cpu (127ms real) (0 rewrites/second)
Proof succeeded.

```

As mentioned in [4], restricting the application of circularities to the top of proof goals using the operations  $fr(-)$  [implemented as  $[*_*]$ ] excludes many important situations. Consider, for instance, the following specification:

```

(fth STREAM is
  sorts Elt Stream .

  op hd : Stream -> Elt .
  op tl : Stream -> Stream .
  op zip : Stream Stream -> Stream .

  eq hd(zip(S:Stream, S':Stream)) = hd(S:Stream) .
  eq tl(zip(S:Stream, S':Stream)) = zip(S':Stream, tl(S:Stream)) .

  op 1 : -> Elt .
  ops f g : -> Stream .

  eq hd(f) = 1 .
  eq tl(f) = zip(f, f) .
  eq hd(g) = 1 .
  eq tl(g) = zip(g, g) .
endfth)

```

One may obviously expect that  $f = g$ . The circular coinduction hypothesis is  $f = g$ , and after derivation with  $tl$  it becomes  $zip(f, f) = zip(g, g)$ . If one was allowed to use the hypothesis under the  $zip$  then one would conclude that  $f = g$ . This can be accomplished by defining safe *special contexts* in which the circularities are allowed to be used.

A  $\Sigma - \Delta$  context  $C[*:h']$  with the result sort  $h$  is called *special* if for any  $\Delta$  experiment  $\gamma[*:h]$  there is a  $\Delta$  experiment  $\gamma'[*:h']$  such that  $\gamma[C[*:h']] = \gamma'[*:h']$  and the size of  $\gamma'[*:h']$  is no bigger than the size of  $\gamma[*:h]$  (see [4] for more details on special contexts).

The user can specify safe special contexts using the `scx` (`special-context`) command:

```

(cth B-STREAM is
  including STREAM .

  scx zip(*:Stream, S:Stream) .

  der hd(*:Stream) .
  der tl(*:Stream) .
endcth)

```

With the context  $zip(*, s)$ , the circular coinduction hypothesis ( $f = g$ ) can be used under the  $zip$ , so  $zip(f, f) = zip(g, g)$  is further reduced to  $zip(g, f) = zip(g, g)$ . After one more coinduction step and few reductions CIRC will be able to prove that indeed  $f = g$ .

In the current version of CIRC there is no internal check for the well-foundedness of the operators declared as special contexts. Consider the following example:

```

(cth F00 is
  protecting NAT .

  sort Stream .

  op hd : Stream -> Nat .
  op tl : Stream -> Stream .
  op odd : Stream -> Stream .

  var S : Stream .

  eq hd(odd(S:Stream)) = hd(S:Stream) .
  eq tl(odd(S:Stream)) = odd(tl(tl(S:Stream))) .

  ops s1 s2 : -> Stream .
  eq hd(s1) = hd(s2) .
  eq tl(s1) = odd(s1) .
  eq tl(tl(s2)) = odd(s2) .

  der hd(*:Stream) .
  der tl(*:Stream) .

  scx odd(*:Stream) .
endcth)

Maude> (add goal odd(s2) = s1 .)
Maude> (coinduction .)

```

```

Hypo odd(s2) = s1 added and coexpanded to
1. hd(odd(s2)) = hd(s1)
2. tl(odd(s2)) = tl(s1)
Goal hd(odd(s2)) = hd(s1) reduced to
  hd(s2) = hd(s2)
Goal hd(s2) = hd(s2) proved by reduction.
Goal tl(odd(s2)) = tl(s1) reduced to
  odd(s1) = odd(s1)
Goal odd(s1) = odd(s1) proved by reduction.

```

```

Proof succeeded.
Number of derived goals: 2
Number of proving steps performed: 17
Maximum number of proving steps is set to: 256

```

Even though the property  $odd(s_2) = s_1$  does not hold (take for instance  $s_1 = 001(2)^\infty$  and  $b = 01(0)^\infty$ ), CIRC will result in a false positive. Declaring special contexts should therefore be used with care for the time being.

### 1.3 Inductive Properties

CIRC includes a simple inductive prover based on the circularity principle.<sup>2</sup> The induction could also help the coinductive proofs. An example is when a derivative is of the form  $\delta(*:h, x:s)$  and  $s$  is an inductive sort, then we can use induction on  $x$  to prove a derived equation  $\delta(t, x) = \delta(t', x)$ .

Here is an example showing how the induction tactic is used. The following Maude theory specifies trees, where the nodes have arbitrary arities. The children of a node are specified as a list of trees. Since the definition of trees requires lists and the definition of lists requires trees, we have here an example of mutual inductive definition.

<sup>2</sup>The implementation of the induction tactic is mainly based on Andrei Popescu's contribution.

```

(cth TREE is
  including NAT .

  sorts Elt Tree TList .
  subsort Elt < Tree .
  subsort Tree < TList .

  var L L' : TList . var T T' : Tree . var E E' : Elt .

  op nil : -> TList [ctor] .
  op cons : Tree TList -> TList [ctor] .
  op tr : Elt TList -> Tree [ctor] .

  op swap : TList -> TList .
  eq swap(E) = E .
  eq swap(tr(E, L)) = tr(E, swap(L)) .
  eq swap(nil) = nil .
  eq swap(cons(T, cons(T', L))) = cons(swap(T'), cons(swap(T), swap(L))) .
  eq swap(cons(T, T')) = cons(T', T) .
  eq swap(cons(T, nil)) = swap(T) .

  op size : TList -> Nat .
  eq size(E) = 1 .
  eq size(tr(E, L)) = 1 + size(L) .
  eq size(nil) = 0 .
  eq size(cons(T, L)) = size(T) + size(L) .
endcth)

```

Over trees (and lists) we defined an operation `swap`, which changes the order of children (the first with the second, the third with the fourth, and so on), and an operation `size`, which returns the total number of nodes.

Now we are ready to prove that  $\text{size}(\text{swap}(T)) = \text{size}(T)$ :

```

Maude> (add goal size(swap(T:Tree)) = size(T:Tree) .)
Goal added: size(swap(T:Tree)) = size(T:Tree)

Maude> (induction .)
Induction started with the variables:
T:Tree
Proof succeeded.
  Number of derived goals: 20
  Number of proving steps performed: 69
  Maximum number of proving steps is set to: 256

```

It is worth to note that proof is completely automatic. To show the complexity of this proof, we use the `show proof .` command. We can see how the induction on trees triggers induction on lists, which triggers in turn induction on trees. This is also exhibited by the way the induction variables are encoded.

```

Maude> (show proof .)
rewrites: 34973 in 5981358573ms cpu (527ms real) (0 rewrites/second)

|- size(swap(T#3:Elt)) = size(T#3:Elt)
----- [Reduce]
|||- size(swap(T#3:Elt)) = size(T#3:Elt)

```

```

|- 1 + size(swap(T#7:Elt)) = 1 + size(T#7:Elt)
----- [Reduce]
|||- 1 + size(swap(T#7:Elt)) = 1 + size(T#7:Elt)

|- 1 + size(swap(T:Tree)) = 1 + size(T:Tree)
----- [Reduce]
|||- 1 + size(swap(T:Tree)) = 1 + size(T:Tree)

|- 1 + size(swap(cons(T#10:Elt,T#21:Elt))) = 2 + size(T#21:Elt)
----- [Reduce]
|||- 1 + size(swap(cons(T#10:Elt,T#21:Elt))) = 2 + size(T#21:Elt)

|- 1 + size(swap(cons(T#10:Elt,T:Tree))) = 2 + size(T:Tree)
----- [Reduce]
|||- 1 + size(swap(cons(T#10:Elt,T:Tree))) = 2 + size(T:Tree)

|- 1 + size(cons(swap(T#24:Elt),cons(T#10:Elt,swap(T:TList)))) =
  2 + size(T#24:Elt)+ size(T:TList)
----- [Reduce]
|||- 1 + size(cons(swap(T#24:Elt),cons(T#10:Elt,swap(T:TList)))) =
  2 + size(T#24:Elt)+ size(T:TList)

|- 1 + size(cons(swap(tr(T#22:Elt,T:TList)),cons(T#10:Elt,swap(T:TList)))) =
  2 + size(tr(T#22:Elt,T:TList)) + size(T:TList)
----- [Reduce]
|||- 1 + size(cons(swap(tr(T#22:Elt,T:TList)),cons(T#10:Elt,swap(T:TList)))) =
  2 + size(tr(T#22:Elt,T:TList)) + size(T:TList)

1. |||- 1 + size(cons(swap(tr(T#22:Elt,T:TList)),cons(T#10:Elt,swap(T:TList)))) =
  2 + size(tr(T#22:Elt, T:TList)) + size(T:TList)
2. |||- 1 + size(cons(swap(T#24:Elt),cons(T#10:Elt,swap(T:TList)))) =
  2 + size(T#24:Elt)+ size(T:TList)
----- [Derive-ind]
|||- 1 + size(cons(swap(T:Tree),cons(T#10:Elt,swap(T:TList)))) =
  2 + size(T:Tree)+ size(T:TList)

|- 1 + size(cons(swap(T:Tree),cons(T#10:Elt,swap(T:TList)))) =
  2 + size(T:Tree)+ size(T:TList)
----- [Normalize]
|- 1 + size(swap(cons(T#10:Elt,cons(T:Tree,T:TList)))) =
  2 + size(cons(T:Tree,T:TList))

|- 1 + size(swap(cons(T#10:Elt,nil))) = 2 + size(nil)
----- [Reduce]
|||- 1 + size(swap(cons(T#10:Elt,nil))) = 2 + size(nil)

1. |||- 1 + size(swap(cons(T#10:Elt,nil))) = 2 + size(nil)
2. |||- 1 + size(swap(cons(T#10:Elt,cons(T:Tree,T:TList)))) =
  2 + size(cons(T:Tree,T:TList))
3. |||- 1 + size(swap(cons(T#10:Elt,T:Tree))) = 2 + size(T:Tree)
4. |||- 1 + size(swap(cons(T#10:Elt,T#21:Elt))) = 2 + size(T#21:Elt)

```

```

----- [Derive-ind]
|||- 1 + size(swap(cons(T#10:Elt,T:TList))) = 2 + size(T:TList)

|- 1 + size(swap(cons(T#10:Elt,T:TList))) = 2 + size(T:TList)
----- [Normalize]
|- 1 + size(swap(cons(T#10:Elt,T:TList))) = 1 + size(T#10:Elt)+ size(T:TList)

|- 1 + size(swap(cons(tr(T#8:Elt,T:TList),T#14:Elt))) =
  2 + size(T#14:Elt)+ size(T:TList)
----- [Reduce]
|||- 1 + size(swap(cons(tr(T#8:Elt,T:TList),T#14:Elt))) =
  2 + size(T#14:Elt)+ size(T:TList)

|- 1 + size(swap(cons(tr(T#8:Elt,T:TList),T:Tree))) =
  2 + size(T:Tree)+ size(T:TList)
----- [Reduce]
|||- 1 + size(swap(cons(tr(T#8:Elt,T:TList),T:Tree))) =
  2 + size(T:Tree)+ size(T:TList)

|- 1 + size(cons(swap(T#17:Elt),cons(tr(T#8:Elt,swap(T:TList)),swap(T:TList)))) =
  2 + size(T#17:Elt)+ size(T:TList)+ size(T:TList)
----- [Reduce]
|||- 1 + size(cons(swap(T#17:Elt),cons(tr(T#8:Elt,swap(T:TList)),swap(T:TList)))) =
  2 + size(T#17:Elt)+ size(T:TList)+ size(T:TList)

|- 1 + size(cons(swap(tr(T#15:Elt,T:TList)),
  cons(tr(T#8:Elt,swap(T:TList)),swap(T:TList)))) =
  2 + size(tr(T#15:Elt,T:TList))+ size(T:TList)+ size(T:TList)
----- [Reduce]
|||- 1 + size(cons(swap(tr(T#15:Elt,T:TList)),
  cons(tr(T#8:Elt,swap(T:TList)),swap(T:TList)))) =
  2 + size(tr(T#15:Elt,T:TList))+ size(T:TList)+ size(T:TList)

1. |||- 1 + size(cons(swap(tr(T#15:Elt,T:TList)),
  cons(tr(T#8:Elt,swap(T:TList)),swap(T:TList)))) =
  2 + size(tr(T#15:Elt,T:TList))+ size(T:TList)+ size(T:TList)
2. |||- 1 + size(cons(swap(T#17:Elt),
  cons(tr(T#8:Elt,swap(T:TList)),swap(T:TList)))) =
  2 + size(T#17:Elt)+ size(T:TList)+ size(T:TList)
----- [Derive-ind]
|||- 1 + size(cons(swap(T:Tree),cons(tr(T#8:Elt,swap(T:TList)),swap(T:TList)))) =
  2 + size(T:Tree)+ size(T:TList)+ size(T:TList)

|- 1 + size(cons(swap(T:Tree),cons(tr(T#8:Elt,swap(T:TList)),swap(T:TList)))) =
  2 + size(T:Tree)+ size(T:TList) + size(T:TList)
----- [Normalize]
|- 1 + size(swap(cons(tr(T#8:Elt,T:TList),cons(T:Tree,T:TList)))) =
  2 + size(cons(T:Tree,T:TList))+ size(T:TList)

|- 1 + size(swap(cons(tr(T#8:Elt,T:TList),nil))) =
  2 + size(nil)+ size(T:TList)
----- [Reduce]

```

```

|||- 1 + size(swap(cons(tr(T#8:Elt,T:TList),nil))) =
      2 + size(nil)+ size(T:TList)

1. |||- 1 + size(swap(cons(tr(T#8:Elt,T:TList),nil))) =
      2 + size(nil)+ size(T:TList)
2. |||- 1 + size(swap(cons(tr(T#8:Elt,T:TList),cons(T:Tree,T:TList)))) =
      2 + size(cons(T:Tree,T:TList))+ size(T:TList)
3. |||- 1 + size(swap(cons(tr(T#8:Elt,T:TList),T:Tree))) =
      2 + size(T:Tree)+ size(T:TList)
4. |||- 1 + size(swap(cons(tr(T#8:Elt,T:TList),T#14:Elt))) =
      2 + size(T#14:Elt)+ size(T:TList)
----- [Derive-ind]
|||- 1 + size(swap(cons(tr(T#8:Elt,T:TList),T:TList))) =
      2 + size(T:TList)+ size(T:TList)

|- 1 + size(swap(cons(tr(T#8:Elt,T:TList),T:TList))) =
   2 + size(T:TList)+ size(T:TList)
----- [Normalize]
|- 1 + size(swap(cons(tr(T#8:Elt,T:TList),T:TList))) =
   1 + size(tr(T#8:Elt,T:TList))+ size(T:TList)

1. |||- 1 + size(swap(cons(tr(T#8:Elt,T:TList),T:TList))) =
      1 + size(tr(T#8:Elt,T:TList))+ size(T:TList)
2. |||- 1 + size(swap(cons(T#10:Elt,T:TList))) =
      1 + size(T#10:Elt)+ size(T:TList)
----- [Derive-ind]
|||- 1 + size(swap(cons(T:Tree,T:TList))) =
      1 + size(T:Tree)+ size(T:TList)

|- 1 + size(swap(cons(T:Tree,T:TList))) = 1 + size(T:Tree)+ size(T:TList)
----- [Normalize]
|- 1 + size(swap(cons(T:Tree,T:TList))) = 1 + size(cons(T:Tree,T:TList))

|- 1 + size(swap(nil)) = 1 + size(nil)
----- [Reduce]
|||- 1 + size(swap(nil)) = 1 + size(nil)

1. |||- 1 + size(swap(nil)) = 1 + size(nil)
2. |||- 1 + size(swap(cons(T:Tree,T:TList))) = 1 + size(cons(T:Tree,T:TList))
3. |||- 1 + size(swap(T:Tree)) = 1 + size(T:Tree)
4. |||- 1 + size(swap(T#7:Elt)) = 1 + size(T#7:Elt)
----- [Derive-ind]
|||- 1 + size(swap(T:TList)) = 1 + size(T:TList)

|- 1 + size(swap(T:TList)) = 1 + size(T:TList)
----- [Normalize]
|- size(swap(tr(T#1:Elt,T:TList))) = size(tr(T#1:Elt,T:TList))

1. |||- size(swap(tr(T#1:Elt,T:TList))) = size(tr(T#1:Elt,T:TList))
2. |||- size(swap(T#3:Elt)) = size(T#3:Elt)
----- [Derive-ind]
|||- size(swap(T:Tree)) = size(T:Tree)

```

# Chapter 2

## CIRC User Manual

### 2.1 Getting Started

#### 2.1.1 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 the CIRC home page <http://fsl.cs.uiuc.edu/index.php/Circ> or going directly to <http://fsl.cs.uiuc.edu/index.php/Special:CircOnline>.

#### 2.1.2 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 have Eclipse, 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 tool (`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 (see the file `index.htm` from the folder `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).

#### 2.1.3 Loading

1. Start Maude in Full-Maude mode. If Full-Maude is not loaded automatically, then introduce the command `in full-maude`.
2. Load the prover introducing the command `in circ`. 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 BASIC-PARSER
=====
mod CIRC-UNIT
=====
fmod PROOFSTATUS
=====
mod CIRC-DATABASE-HANDLING
=====
mod CIRC-PROVER
=====
mod CIRC-INTERFACE
rewrites: 5 in 3696189313ms cpu (2ms real) (0 rewrites/second)

          CIRC 1.4 (April 13th, 2008)

Maude>

```

3. You may introduce now any CIRC or Full-Maude command.

## 2.2 Running the Examples

The system is accompanied by a set of examples exhibiting the power of the prover. The usual scenario for running an example is:

- set the prover in the initial state introducing the command

```
Maude> loop init .
```

In many cases this is an optional step.

- introduce Maude specifications using Maude command `in file-name`. The files including specifications have the names of the form `base-name-spec.maude`. Some examples use more than one specification.

- introduce one by one the proving commands included in the files having names of the form *base-name-cmd.maude*. These can be merged with different CIRC commands or Maude commands, e.g., to see the intermediate results. The complete list of CIRC commands is included in the user manual.

## 2.3 Commands

The general syntax for a command is

$$\langle \text{command} \rangle$$

where

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

In what follows we use by default the following definition:

$$\langle \text{itemList} \rangle ::= \langle \text{item} \rangle \mid \langle \text{item} \rangle \langle \text{itemList} \rangle$$

where *item* is instantiated over different syntactical constructions.

### 2.3.1 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:

$$\begin{aligned} \langle specification \rangle ::= & \text{cth } \langle declarationList \rangle \text{ endcth} \mid \\ & \text{cthery } \langle declarationList \rangle \text{ endcttheory} \mid \\ \langle declaration \rangle ::= & \langle fthMaudeSpecificDeclaration \rangle \mid \\ & \langle derivative \rangle \mid \\ & \langle specialContext \rangle \mid \\ & \langle eqSimplification \rangle \end{aligned}$$

A derivative declaration has the syntax

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

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

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

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

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

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

### Example: Streams.

```
(cthery B-STREAM is
  including STREAM .
  der hd(*:Stream) .
  der tl(*:Stream) .
endcttheory)
```

The above command is equivalent to

```
(cth B-STREAM is
  including STREAM .
  der hd(*:Stream) tl(*:Stream) .
endcth)
```

### 2.3.2 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:

```

<addGoal> ::= add goal <equation> . |
           add goal <operatorDeclaration> . |
           add cgoal <condEquation> .
<equation> ::= <term> = <term>
<operatorDeclaration> ::= op <term> : <termList> -> <term> [ <attrList> ] .
<attr> ::= comm | assoc | idem |
        id: <term> | left id: <term> | right id: <term>
<attrList> ::= <attr> | <attr> <attrList>
<condEquation> ::= <term> = <term> if <condition>
<condition> ::= <term> = <term> | <condition> /\ <condition>

```

### 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 .)

```

### 2.3.3 Showing goals

Displays the goals to be proved. Its syntax is

```
<showGoals> ::= show goals .
```

### Example: Streams.

```

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

```

### 2.3.4 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

```
<focus> ::= focus <number> .
```

### 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>
```

### 2.3.5 Set show details

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

```
<setShowDetails> ::= set show details on . |
                    set show details off .
```

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

### 2.3.6 Adding a lemma

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

```
<addLemma> ::= add lemma <equation> . |
              add clemma <condEquation> .
```

#### 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.
```

### 2.3.7 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 considered to be proved. Its syntax is:

```
<eqReduction> ::= 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)
```

### 2.3.8 Coinduction

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

$$\langle \textit{coinduction} \rangle ::= \textit{coinduction} \ . \ | \ \textit{ccstep} \ .$$

#### 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>
```

### 2.3.9 Induction.

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

$$\langle \textit{induction} \rangle ::= \textit{induction} \ . \ | \ \textit{cistep} \ . \ | \ \textit{induction on} \ \langle \textit{variableSet} \rangle \ . \ | \ \textit{cistep} \ \langle \textit{variableSet} \rangle \ . \ | \ \textit{set induction vars} \ \langle \textit{variableSet} \rangle \ .$$

The structure of a variable set is :  $\langle \textit{variableSet} \rangle ::= \langle \textit{variable} \rangle \ | \ \langle \textit{variable} \rangle \ \langle \textit{variableSet} \rangle$ , 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>
```

### 2.3.10 Apply user-defined strategies

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

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

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

**Example: An induction strategy.**

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

### 2.3.11 The maximum number of prover steps allowed

Can be set and shown using the following commands:

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

**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 .

```

### 2.3.12 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:

```

⟨setAutoContexts⟩ ::= set auto contexts on. |
                    set auto contexts off. |

```

#### Example: Streams.

```

(set auto contexts on .)

(cttheory B-STREAM is
  including STREAM .
  der hd(*:Stream) .
  der tl(*:Stream) .
endcttheory)

```

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 ⟨termList⟩`

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

```

⟨checkScx⟩ ::= check scx ⟨term⟩ using ⟨termList⟩ .

```

### 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

### 2.3.13 Show proof

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

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

### 2.3.14 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

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

### 2.3.15 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 order 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 restored.

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

### 2.3.16 Undo

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

$$\langle \mathit{undo} \rangle ::= \mathbf{undo\ .}$$

## 2.4 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 `...&...#(number)...`.
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.

# Bibliography

- [1] M. Clavel et al. Maude: Specification and Programming in Rewriting Logic. *J. of TCS*, 285:187–243, 2002.
- [2] R. Diaconescu and K. Futatsugi. Behavioral coherence in object-oriented algebraic specification. *JUCS*, 6(1):74–96, 2000.
- [3] J. Goguen. Types as theories. In *Topology and Category Theory in Computer Science*, pages 357–390. Oxford, 1991.
- [4] J. Goguen, K. Lin, and G. Roşu. Conditional Circular Coinductive Rewriting with Case Analysis. In *WADT'02*, volume 2755 of *LNCS*, pages 216–232. Springer, 2003.
- [5] J. Goguen and G. Malcolm. A hidden agenda. *J. of TCS*, 245(1):55–101, 2000.
- [6] R. Hennicker and M. Bidoit. Observational logic. In *Proceedings of AMAST'98*, volume 1548 of *LNCS*, pages 263–277. Springer, 1999.
- [7] P. Padawitz. Swinging data types: Syntax, semantics, and theory. In *Proceedings, WADT'95*, volume 1130 of *LNCS*, pages 409–435. Springer, 1996.
- [8] G. Roşu. *Hidden Logic*. PhD thesis, University of California at San Diego, 2000.
- [9] G. Roşu and D. Lucanu. Circular Coinduction –A Proof Theoretical Foundation. Technical Report UIUCDCS-R-2009-3037, University of Illinois at Urbana-Champaign, 2009. Submitted.