

Circular Coinductive Rewriting

Joseph Goguen, Kai Lin, Grigore Roşu*
Department of Computer Science & Engineering
University of California at San Diego
{goguen,klin,grosu}@cs.ucsd.edu

Abstract

Circular coinductive rewriting is a new method for proving behavioral properties, that combines behavioral rewriting with circular coinduction. This method is implemented in our new BOBJ behavioral specification and computation system, which is used in examples throughout this paper. These examples demonstrate the surprising power of circular coinductive rewriting. The paper also sketches the underlying hidden algebraic theory and briefly describes BOBJ and some of its algorithms.

1 Introduction

Circular coinductive rewriting is an algorithm for proving behavioral equalities that integrates behavioral rewriting [6, 17] with circular coinduction [19, 11, 12]. We give examples showing that this algorithm is surprisingly powerful in practice, even though a recent result of Buss and Roşu [3] shows that no such algorithm can be complete. Of course, incompleteness is more the rule than the exception for non-trivial theorem proving problem classes.

We work in a general framework for equational behavioral specification that distinguishes between *visible* and *hidden* sorts, with equality being *strict* on visible sorts and *behavioral* on hidden sorts, in the sense of “indistinguishability under experiments;” special cases include hidden algebra [7, 8, 10, 9], coherent hidden algebra [5, 6], observational logic [15, 2], and our recent generalizations of hidden algebra [19, 18]. A related approach is coalgebra [16, 1].

Because of space constraints, we omit all proofs and most definitions, instead giving explanations and examples. We assume the reader familiar with algebraic specification and theorem proving, especially many sorted and order sorted algebra and term rewriting; familiarity with OBJ would be helpful. We also introduce and illustrate the newly developed BOBJ system for behavioral specification and verification, using it for all our examples. These examples include: several equivalences of streams of values, such as

*Also Fundamentals of Computing, Faculty of Mathematics, University of Bucharest, Romania.

might arise in verifying lazy functional programs; a behavioral refinement example, that stack can be implemented with a pointer into an array; and the equivalence of regular languages. In addition, we discuss the concurrent connection of systems, and the existence of multiple cobases for behavioral specs.

2 Behavioral Specification and Reasoning

A **hidden signature** Σ is an order sorted signature distinguishing **visible sorts** V from **hidden sorts** H . An **implementation** or **model** of Σ is a Σ -algebra. The elements of visible sort in models are called **data** and those of hidden sorts are called **states**. A **behavioral specification** or **theory** is a hidden signature Σ , a hidden subsignature Γ , and a (finite) set E of Σ -equations. The operations in Γ are called **behavioral** and are used to form **experiments** or **contexts**, which are Γ -terms of visible result having a distinguished hidden variable. These generate a **behavioral equivalence** relation on any model, under which two data elements are equivalent iff they are equal, and two states are equivalent iff they cannot be distinguished by experiments, i.e., iff any experiment produces the same value when applied to them. A model **behaviorally satisfies** a Σ -equation iff the two terms evaluate to behaviorally equivalent elements under all assignments of values to variables. The equations in E restrict the class of models to those that behaviorally satisfy each equation. Unrestricted equational reasoning is not generally sound in this setting, but it can be adapted [19], as can term rewriting [6, 17]. This framework captures all situation of practical interest of which we are aware.

It often happens that not all experiments are needed, because the equations imply that some experiments are equivalent to others. A similar but dual situation occurs in abstract data type theory when all the elements can be generated from a subset of operations, called the constructors, generators, or basis (when induction is involved). The dual notion of *cobasis* first appeared in [19], and was later simplified [11, 12, 17, 18]; for the purposes of this paper, read-

ers may consider a cobasis Δ to be a subset of operations in Γ that generates enough experiments, in the sense that no other experiment can distinguish two states that cannot be distinguished by these experiments. Finding a minimal cobasis seems to be undecidable, but there are *cobasis criteria* that work well in practice [19, 2, 18], and are implemented in BOBJ.

2.1 BOBJ

BOBJ supports behavioral specification and verification based on recent developments in hidden algebra; in addition to ordinary rewriting and behavioral rewriting over order sorted equational logic, modulo attributes that can be any combination of associative, commutative and identity, it also implements circular coinductive rewriting over theories where behavioral operations may have multiple hidden arguments. Because the concurrent connection of systems can be represented with tupling, BOBJ supports tupling of behavioral theories with syntax given by the binary associative infix operator “|” which takes two or more modules as its arguments; see Section 4.1 for more details. BOBJ automatically computes cobases for specifications using recently discovered congruence criteria, with the notable new feature of simplifying cobases that involve tupling.

BOBJ syntax is almost the same as that of OBJ3 [13], with exceptions that include the following: term syntax is more flexible, due to using JavaCC for parsing; in addition to the keywords “th” and “obj” from OBJ3, BOBJ modules can also begin with the keywords “dth” for data theories (equivalent to the keyword “obj”) with initial semantics, and “bth” for behavioral theories with hidden semantics; any module can be closed with the keyword “end”; operations within behavioral theories are considered congruent (i.e., “coherent” in the original terminology of Diaconescu [5]) unless they are given the attribute “ncong”; the automatically generated cobasis for the currently selected module can be seen with the command “show cobasis .”, and circular coinductive rewriting is invoked with the command “cred”, whereas “red” invokes behavioral rewriting without circularity (which is ordinary rewriting for visible sorts). The command “set trace on .” causes BOBJ to output information about its computations.

Several data types and many equations are “built-ins,” i.e., are available without having to be declared by the user. In particular, the data types `BOOL` of Booleans and `NAT` of natural numbers are built in. Users can see what a module `MOD` provides with the command “show MOD .”; this works for built in as well as user supplied modules. For example, “show `TRUTH` .” will reveal the equations for the built-in `if_then_else_fi` conditional.

The BOBJ implementation is due to Kai Lin, and includes

innovative techniques for speeding up rewriting and for implementing behavioral and circular coinductive rewriting. Many of the finer details of OBJ3 are not yet implemented, and the system should still be considered experimental. We were largely inspired to proceed with BOBJ by the success of the CafeOBJ system, which embodies rewriting logic as well as a somewhat more restricted version of hidden algebra [6].

The specifications `STREAM` and `ZIP` below are much used in our examples and should be considered to have been pre-loaded. The first declares infinite streams parameterized by `TRIV`, a built in visible theory containing only the visible sort `Elt`.

```
bth STREAM[X :: TRIV] is sort Stream .
  op head : Stream -> Elt .
  op tail : Stream -> Stream .
  op _&_ : Elt Stream -> Stream .
  var E : Elt . var S : Stream .
  eq head(E & S) = E .
  eq tail(E & S) = S .
end
```

BOBJ’s cobasis algorithm discovers that `_&_` is not needed:

```
The cobasis for sort Stream is:
  op head : Stream -> Elt
  op tail : Stream -> Stream
```

The second specification adds an operation which “zips” two streams together by taking elements from them alternately; BOBJ can find its cobasis to be `{head, tail}`.

```
bth ZIP[X :: TRIV] is pr STREAM[X] .
  op zip : Stream Stream -> Stream .
  vars S S' : Stream .
  eq head(zip(S,S')) = head(S) .
  eq tail(zip(S,S')) = zip(S',tail(S)) .
end
```

2.2 Circular Coinductive Rewriting in BOBJ

Few behavioral properties can be proved with just equational behavioral reasoning. Context induction [14] and coinduction are more powerful proof techniques, but unfortunately both need human intervention. Circular coinductive rewriting integrates behavioral rewriting [6, 17] with circular coinduction [18], where the latter is a powerful generalization of coinduction that brings out the duality with ordinary induction more clearly by allowing coinductive hypotheses to be used in proofs. The circular coinductive rewriting algorithm has as input a pair of terms, and returns true whenever it can prove the terms behaviorally equivalent, and otherwise returns false or fails to terminate, much as with proving term equality by rewriting. The detailed correctness proof is difficult and will appear later; here we describe the BOBJ implementation.

Given a behavioral specification $\mathcal{B} = (\Sigma, \Gamma, \mathcal{E})$ with a cobasis $\Delta \subseteq \Gamma$, a set of pairs of terms¹ \mathcal{C} , and a Σ -term s , let $bnf_{\mathcal{C}}(s)$ be the term derived from s by rewriting with the equations in $\mathcal{E} \cup \mathcal{C}$ under the usual restrictions for behavioral rewriting [17] with \mathcal{E} , and then applying the equations in \mathcal{C} at a term position if all (zero or more) operations on the path to that position are in $\Gamma - \Delta$.

Given a pair of Σ -terms (t, t') , the circular coinductive rewriting algorithm, hereafter denoted CCRW, can be described as follows:

1. let $\mathcal{C} = \emptyset$ and $\mathcal{G} = \{(t, t')\}$
2. for each (s, s') in \mathcal{G}
3. move (s, s') from \mathcal{G} to \mathcal{C}
4. for each $\delta \in \Delta$
5. let $u = bnf_{\mathcal{C}}(\delta[s, x])$ and $u' = bnf_{\mathcal{C}}(\delta[s', x])$
6. if $u \neq u'$ then add (u, u') to \mathcal{G}

\mathcal{G} contains the still unproved goals and \mathcal{C} contains the ‘‘circularities’’ to be used in proofs. By definition of cobasis, to prove $t \equiv t'$, it suffices to prove $\delta[t, x] \equiv \delta[t', x]$ for all $\delta \in \Delta$ and all appropriate x . Note that an equation (t, t') in \mathcal{C} can be used in the special way described above in proving an equation $(\delta[t, x], \delta[t', x])$, which is then used in proving (t, t') ; this explains the word ‘‘circular.’’ Note also that the algorithm may fail to terminate. Justification for the choice and use of equations in \mathcal{C} is rather technical, and must be omitted here.

3 Lazy Functional Programming Examples

We give some simple examples from lazy functional programming. Many similar examples were done by Louise Dennis using a system called CoClam with a complex heuristic planning algorithm [4]; all her examples that we tried can be done algorithmically by CCRW without human intervention or machine heuristics. We thank Wolfram Schulte for getting us started in this direction.

Example 1 We define a function `rev` taking an infinite stream of boolean values and returning a stream where each value is reversed:

```
bth REV is pr STREAM[BOOL] .
  op rev : Stream -> Stream .
  var S : Stream .
  eq head(rev(S)) = not head(S) .
  eq tail(rev(S)) = rev(tail(S)) .
end
```

We show that $rev(rev(S))$ is equivalent to S :

¹BOBJ may reorder these to reduce the possibility of non-termination.

```
BOBJ> set trace on
BOBJ> cred rev(rev(S)) == S .
=====
c-reduce in REV : rev(rev(S)) == S
using cobasis for REV:
  op head : Stream -> Bool
  op tail : Stream -> Stream
-----
reduced to: rev(rev(S)) == S
-----
add rule (C1) : rev(rev(S)) = S
-----
target is: rev(rev(S)) == S
expand by: op head : Stream -> Bool
reduced to: true
          nf: head(S)
-----
target is: rev(rev(S)) == S
expand by: op tail : Stream -> Stream
deduced using (C1) : true
          nf: tail(S)
-----
result: true
c-rewrite time: 82ms      parse time: 6ms
```

The first command, `set trace on`, tells BOBJ to display some high level information about the execution of CCRW; this often helps find errors in the spec, and suggest needed lemmas. BOBJ’s cobasis algorithm discovers that `rev` is congruent, so the cobasis is just $\{\text{head}, \text{tail}\}$. These operations are applied to the two terms, giving the subgoals $\text{head}(\text{rev}(\text{rev}(S))) == \text{head}(S)$ and $\text{tail}(\text{rev}(\text{rev}(S))) == \text{tail}(S)$. The first follows directly by behavioral rewriting (the built in module `BOOL` contains the fact `not not B == B` for any boolean `B`). The second subgoal needs the circularity, since it is reduced to $\text{rev}(\text{rev}(\text{tail}(S))) == \text{tail}(S)$, which is an instance of its initial goal, where S is replaced by $\text{tail}(S)$. BOBJ reports circularity by displaying the keyword ‘‘deduced’’ instead of ‘‘reduced’’ with the normal form $\text{tail}(S)$ of $\text{rev}(\text{rev}(\text{tail}(S)))$. ■

Example 2 We define infinite streams `zero`, `one`, and `blink`, containing only 0’s, only 1’s, and alternations of 0 and 1, as follows:

```
bth BLINK is pr ZIP[NAT] .
  ops zero one blink : -> Stream .
  eq head(zero) = 0 .
  eq tail(zero) = zero .
  eq head(one) = 1 .
  eq tail(one) = one .
  eq head(blink) = 0 .
  eq head(tail(blink)) = 1 .
  eq tail(tail(blink)) = blink .
end
```

`BLINK` imports `ZIP` instantiated with the built in module `NAT` of natural numbers. The property that `blink =`

zip(zero, one) is proved by circular coinduction as follows:

```
BOBJ> cred blink == zip(zero, one) .
```

producing the output

```
c-reduce in BLINK : blink == zip(zero,one)
using cobasis for BLINK:
  op head : Stream -> Nat
  op tail : Stream -> Stream
-----
reduced to: blink == zip(zero , one)
-----
add rule (C1) : blink = zip(zero , one)
-----
target is: blink == zip(zero , one)
expand by: op head : Stream -> Nat
reduced to: true
          nf: 0
-----
target is: blink == zip(zero , one)
expand by: op tail : Stream -> Stream
reduced to: tail(blink) == zip(one , zero)
-----
add rule (C2): tail(blink) = zip(one,zero)
-----
target is: tail(blink) == zip(one , zero)
expand by: op head : Stream -> Nat
reduced to: true
          nf: 1
-----
target is: tail(blink) == zip(one , zero)
expand by: op tail : Stream -> Stream
deduced using (C1) : true
          nf: zip(zero , one)
-----
result: true
```

The cobasis algorithm found {head, tail} for BLINK, and the next four steps used this for circular coinductive rewriting. First head is used for circular coinductive rewriting. First head is applied to the two streams, giving 0 in each case. Then tail is applied, giving the new goal tail(blink) == zip(one, zero). Again head is applied to the two new terms, giving 1, and then tail is applied, giving a circularity, namely the subgoal blink == zip(zero, one), so that the result is true. ■

Example 3 One obvious way to define the stream of natural numbers, that is, 0 1 2 3 4 ... is to define a function nat by nat(N) = N & nat(N + 1) for all N, and then consider nat(0). Less obvious is to define a function succ incrementing all elements in a stream by succ(S) = (head(S) + 1) & succ(tail(S)), and then define nat' by nat'(N) = N & succ(nat'(N)).

```
bth NAT-STREAM is pr STREAM[NAT] .
  op nat : Nat -> Stream .
```

```
var N : Nat . var S : Stream .
eq head(nat(N)) = N .
eq tail(nat(N)) = nat(N+1) .
op succ : Stream -> Stream .
eq head(succ(S)) = head(S) + 1 .
eq tail(succ(S)) = succ(tail(S)) .
op nat' : Nat -> Stream .
eq head(nat'(N)) = N .
eq tail(nat'(N)) = succ(nat'(N)) .
end
```

We show these definitions equivalent, i.e., nat(0) == nat'(0), by proving the more general result nat(N) == nat'(N) for all N; we omit some BOBJ output.

```
BOBJ> cred nat(N) == nat'(N) .
=====
c-reduce in NAT-STREAM : nat(N) == nat'(N)
reduced to: nat(N) == nat'(N)
-----
add rule (C1) : nat(N) = nat'(N)
-----
.....
add rule (C2) : nat'(1+N) = succ(nat'(N))
-----
.....
result: true
```

The new rule (C1) was generated and used for (C2), and (C2) for the final subgoal. ■

Example 4 We show that two definitions of the stream of Fibonacci numbers are equivalent.

```
dth FIBO-NAT is ex NAT .
  var N : Nat .
  op f : Nat -> Nat .
  eq f(0) = 0 . eq f(1) = 1 .
  eq f(N + 2) = f(N + 1) + f(N) .
end

bth FIBO-STREAM is pr ZIP[FIBO-NAT] .
  var N : Nat . var S : Stream .
  op nat : Nat -> Stream .
  eq head(nat(N)) = N .
  eq tail(nat(N)) = nat(N + 1) .
  op f : Stream -> Stream .
  eq head(f(S)) = f(head(S)) .
  eq tail(f(S)) = f(tail(S)) .
  op fib : Nat -> Stream .
  eq fib(N) = f(nat(N)) .
  op add : Stream -> Stream .
  eq head(add(S)) = head(S) +
                    head(tail(S)) .
  eq tail(add(S)) = add(tail(tail(S))) .
  op fib' : Nat -> Stream .
  eq head(fib'(N)) = f(N) .
  eq head(tail(fib'(N))) = f(N + 1) .
  eq tail(tail(fib'(N))) =
```

```

      add(zip(fib'(N),tail(fib'(N)))) .
end

```

Here is the BOBJ output:

```

BOBJ> cred fib(N) == fib'(N) .
=====
c-reduce in FIBO-STREAM: fib(N) == fib'(N)
using cobasis for FIBO-STREAM:
  op head : Stream -> NzNat
  op tail : Stream -> Stream
-----
reduced to: f(nat(N)) == fib'(N)
-----
add rule (C1) : f(nat(N)) = fib'(N)
-----
target is: f(nat(N)) == fib'(N)
expand by: op head : Stream -> NzNat
reduced to: true
          nf: f(N)
-----
target is: f(nat(N)) == fib'(N)
expand by: op tail : Stream -> Stream
deduced using (C1) : fib'(1 + N) ==
                    tail(fib'(N))
-----
add rule (C2): fib'(1 + N) = tail(fib'(N))
-----
target is: fib'(1 + N) == tail(fib'(N))
expand by: op head : Stream -> NzNat
reduced to: true
          nf: f(1 + N)
-----
target is: fib'(1 + N) == tail(fib'(N))
expand by: op tail : Stream -> Stream
reduced to: tail(fib'(1 + N)) ==
            add(zip(fib'(N) , tail(fib'(N))))
-----
add rule (C3) : tail(fib'(1 + N)) =
              add(zip(fib'(N) , tail(fib'(N))))
-----
target is: tail(fib'(1 + N)) ==
          add(zip(fib'(N) , tail(fib'(N))))
expand by: op head : Stream -> NzNat
reduced to: true
          nf: f(N) + f(1 + N)
-----
target is: tail(fib'(1 + N)) ==
          add(zip(fib'(N) , tail(fib'(N))))
expand by: op tail : Stream -> Stream
deduced using (C2) : true
          nf: add(zip(tail(fib'(N)) ,
                    add(zip(fib'(N) , tail(fib'(N))))))
-----
result: true

```

■

Example 5 We show that two behavioral specifications of

stream are equivalent. First we declare their common signature:

```

bth SIGMA[X :: TRIV] is sort Stream .
  op head : Stream -> Elt .
  op tail : Stream -> Stream .
  op _&_   : Elt Stream -> Stream .
  op odd  : Stream -> Stream .
  op even : Stream -> Stream .
  op zip  : Stream Stream -> Stream .
end

```

The first three operations and the sixth are as usual, while odd and even give the streams formed by the elements in the odd and even positions, respectively. For example, $\text{odd}(1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ \dots)$ is $1\ 3\ 5\ 7\ 9\ \dots$, while $\text{even}(1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ \dots)$ is $2\ 4\ 6\ 8\ \dots$. Notice that all operations are behavioral, because they preserve the intended behavioral equivalence, which is ‘two streams are equivalent iff they have the same elements in the same order.’

```

bth STREAM1[X :: TRIV] is using SIGMA[X] .
  var E : Elt . var S S' : Stream .
  eq head(E & S) = E .
  eq tail(E & S) = S .
  eq head(odd(S)) = head(S) .
  eq tail(odd(S)) = even(tail(S)) .
  eq head(even(S)) = head(tail(S)) .
  eq tail(even(S)) = even(tail(tail(S))) .
  eq head(zip(S, S')) = head(S) .
  eq tail(zip(S, S')) = zip(S', tail(S)) .
end

```

This has the expected cobasis $\{\text{head}, \text{tail}\}$, but given any model, there is another interesting way to generate its behavioral equivalence, namely using observations built with operations head, odd and even. For example, the term $\text{head}(\text{even}(\text{odd}(\text{odd}(S))))$ ‘observes’ the fifth element of S , while the term $\text{head}(\text{even}(\text{even}(\text{odd}(\text{even}(\text{odd}(S)))))$ ‘observes’ the 27th element. The following specification is in this spirit:

```

bth STREAM2[X :: TRIV] is using SIGMA[X] .
  var E : Elt . var S S' : Stream .
  eq head(tail(S)) = head(even(S)) .
  eq odd(tail(S)) = even(S) .
  eq even(tail(S)) = tail(odd(S)) .
  eq head(E & S) = E .
  eq odd(E & S) = E & even(S) .
  eq even(E & S) = odd(S) .
  eq head(zip(S, S')) = head(S) .
  eq odd(zip(S, S')) = S .
  eq even(zip(S, S')) = S' .
  eq head(odd(S)) = head(S) .
end

```

BOBJ finds the cobasis $\{\text{head}, \text{odd}, \text{even}\}$, and also easily proves the two specifications behaviorally equivalent, in

the sense of having the same models with the same behavioral equivalences [19, 11], by proving all equalities in each spec from those in the other (see also [18]). We encourage the reader investigate why the last equation in `STREAM2` is needed (hint: without it there could be models of `STREAM2` that are not models of `STREAM1`).

There are reasons to consider the second cobasis better than the first. For example, a stream's elements can be reached more quickly (e.g., the 27th element can be observed in 6 steps instead of 27), so that less computation is needed for testing. Also properties like $\text{zip}(\text{odd}(S), \text{even}(S)) = S$ have much easier proofs by $\{\text{head}, \text{odd}, \text{even}\}$ -coinduction. On the other hand, the equality $\text{head}(S) \ \& \ \text{tail}(S) == S$ is easy to prove with `CCRW` over $\{\text{head}, \text{tail}\}$ in `STREAM1`, but cannot be proved with `CCRW` over $\{\text{head}, \text{odd}, \text{even}\}$ in `STREAM2`. Thus neither cobasis is the best for all purposes, and so proving their equivalence has a practical value. ■

4 Behavioral Refinement

Some researchers have complained that stack examples have been overused, but they provide a very useful benchmark just because they have been used in so many different settings, and they often reveal something new and interesting. Here, we illustrate our approach to behavioral refinement with the familiar implementation of stack as a pointer into an array, with the indicated array element as the top, the pointer incremented and the element inserted into the array for `push`, and the pointer decremented when for `pop`.

```
bth STACK[X :: TRIV] is sort Stack .
  op top_ : Stack -> Elt .
  op pop_ : Stack -> Stack .
  op push : Elt Stack -> Stack .
  op empty : -> Stack .
  var E : Elt . var S : Stack .
  eq top push(E,S) = E .
  eq pop push(E,S) = S .
end
```

`BOBJ` find the cobasis $\{\text{top}, \text{pop}\}$. We have left `top` and `pop` undefined on `empty` to allow a larger class of models. The stack operations are implemented on pointer array pairs as expected:

```
bth ARRAY[X :: TRIV] is sort Arr .
  protecting NAT .
  op nil : -> Arr .
  op put : Elt Nat Arr -> Arr .
  op _[_] : Arr Nat -> Elt .
  vars I J : Nat . var A : Arr .
  var E : Elt .
  eq nil [I] = 0 .
  cq put(E, I, A) [J] = E if eq(I, J) .
```

```
    cq put(E, I, A) [J] = A [J]
      if not eq(I, J) .
end

bth STACKIMP[X :: TRIV] is
  pr (NAT || ARRAY[X])
    * (sort Tuple to Stack) .
  op top_ : Stack -> Elt .
  op pop_ : Stack -> Stack .
  op push : Elt Stack -> Stack .
  op empty : -> Stack .
  vars I J : Nat . var A : Arr .
  var E : Elt .
  eq empty = <0, nil>.
  eq push(E, <I, A>) =
    <s I, put(E, I, A)> .
  eq top <s I, A> = A [I] .
  eq top <0, A> = 0 .
  eq pop <s I, A> = <I, A> .
  eq pop <0, A> = <0, A> .
end
```

The operation `eq` used in `ARRAY` belongs to `NAT`, and returns `true` when its arguments can be proved equal, `false` when they can be proved unequal, and a normal form when `BOBJ` cannot prove them equal or unequal. The operation “`||`” on modules is also built in; it generates states which are parallel connections of the states of its components, adding a new sort called `Tuple`, a tupling operation

$$\langle _, _, \dots, _ \rangle : S_1 S_2 \dots S_n \rightarrow \text{Tuple}$$

and projection operations $i^* : \text{Tuple} \rightarrow S_i$ where i ranges from 1 to the number of modules connected, plus the “tupling equation”

$$\text{eq} \langle 1^*(T), 2^*(T), \dots, n^*(T) \rangle = T$$

which says that all tuple states are tuples of component states.

To show refinement, we first prove a lemma, that

$$\langle I, \text{put}(E, J, A) \rangle = \langle I, A \rangle \text{ if } I \leq J$$

for any I, J, E, A , by induction on I , using circular coinductive rewriting:

```
open .
  set cobasis of STACK
  ***> base case
  cred <0, put(E, J, A)> == <0, A> .
  ***> induction step
  ops i j : -> Nat . eq i < j = true .
  cq <i, put(E, J, A)> = <i, A> if i <= J .
  cred <s i, put(E, j, A)> == <s i, A> .
close
```

Because this is a refinement proof, we want to use the cobasis $\{\text{top}, \text{pop}\}$ of `STACK`, not that of `STACKIMP`, which `BOBJ` correctly finds to include `push`. For this reason, `BOBJ` has a command “set cobasis of `<modname>`” to tell the system to use the cobasis of `<modname>` (there are other commands for setting cobases). Since both reductions give `true`, we can add the conditional equation:

```
bth STACKIMP is pr STACKIMP .
  vars I J : Nat .   var A : Arr .
  var E : Elt .
  cq <I, put(E,J,A)> = <I, A> if I <= J .
end
```

The two equations of `STACK` now follow by just behavioral rewriting:

```
red pop push(E, <I, A>) == <I, A> .
red top push(E, <I, A>) == E .
```

4.1 Concurrent Connection

Because our so-called “concurrent connection” operation is implemented in `BOBJ` as tupling, some readers might question whether it really captures the concurrent operation of its components. However, this is a place where our behavioral framework makes a significant contribution, because simple tupling is behaviorally equivalent to a more sophisticated way of combining components for which concurrency is obviously satisfied, the universal notion of concurrent connection given in [8]. Therefore tupling is justified as a computational simplification for theorem proving purposes. (Much more could be said here concerning the underlying mathematics, but there is not sufficient space for these details.)

The code below for the case of two behavioral theories is equivalent to what `BOBJ` provides; note that the projection operations are noncongruent, that the n -tuple sort is named “`Tuple`” for all n , and that the untupling equation is the last one. The sort `Tuple` is always hidden, even if all of its component sorts are visible; `BOBJ` provides a different syntax for purely visible tupling.

```
bth 2[1 2 :: TRIV] is sort Tuple .
  op < _,_ > : Elt.1 Elt.2 -> Tuple .
  op 1* _ : Tuple -> Elt.1 [ncong].
  op 2* _ : Tuple -> Elt.2 [ncong].
  var E1 : Elt.1 .   var E2 : Elt.2 .
  var T : Tuple .
  eq 1* < E1, E2 > = E1 .
  eq 2* < E1, E2 > = E2 .
  eq < 1* T, 2* T > = T .
end
```

Tupling is a prime example of a congruent operation with multiple hidden sorts. `BOBJ`’s cobasis algorithm augments our previous congruence criteria by using the untupling equation to simplify cobases.

5 Equivalent Regular Languages

We give a behavioral specification for regular expressions with several examples of their equivalence. It is perhaps surprising that no human intervention is required – `BOBJ` does all the work, using circular coinductive rewriting. We thank to Bogdan Warinschi, whose work inspired us. The behavioral spec for regular expressions contains constants `empty` and `nil`, for the empty language and the language consisting of just the empty string, plus concatenation, union, Kleene star, and an operation `nil-in`, for asking if a language contains the empty string.

```
bth REGEXP is sort RegExp .
  ops empty nil : -> RegExp .
  op ___ : RegExp RegExp -> RegExp
  [assoc id: nil prec 20] .
  op _+_ : RegExp RegExp -> RegExp
  [assoc id: empty comm idem prec 30] .
  op _* : RegExp -> RegExp [prec 10] .
  op nil-in _ : RegExp -> Bool [prec 40] .
  vars L L' : RegExp .
  eq empty L = empty .
  eq nil-in L L' =
    nil-in L and nil-in L' .
  eq nil-in L + L' =
    nil-in L or nil-in L' .
  eq nil-in L* = true .
  eq nil-in nil = true .
  eq nil-in empty = false .
end
```

Notice the attributes `assoc`, `comm`, `idem`, `id:`, `prec`, which declare an operation associative, commutative, idempotent, having an identity, and having a precedence, respectively; `BOBJ` can parse and rewrite modulo these attributes.

The following adds a letter `x` to the alphabet, where `x?(L)` is true iff there are words that start with `x` in `L`, and `-x(L)` gives the language obtained from `L` keeping only those words that start with `x` and then removing the leading `x`:

```
bth LETTER is pr REGEXP .
  sort Letter .   subsort Letter < RegExp .
  op x : -> Letter .
  eq nil-in x = false .
  var Y : Letter .
  op x? : RegExp -> Bool .
  vars L L' : RegExp .   var X : Letter .
  eq x?(L L') =
    x?(L) or nil-in L and x?(L') .
  eq x?(L + L') = x?(L) or x?(L') .
  eq x?(L*) = x?(L) .
  eq x?(X) = x == X .
  eq x?(nil) = false .
  eq x?(empty) = false .
  op -x : RegExp -> RegExp [prec 1] .
  eq -x(L L') = -x(L) L' +
    if (nil-in L) then -x(L')
```

```

else empty fi .
eq -x(L + L') = -x(L) + -x(L') .
eq -x(L*) = -x(L) L* .
eq -x(X) = if x == X then nil
else empty fi .
eq -x(nil) = empty .
eq -x(empty) = empty .
end

```

We can now create regular languages with as many letters as we want:

```

bth LANGUAGE is
  pr LETTER * (op x to a ,
               op x? to a? ,
               op -x to -a) .
  pr LETTER * (op x to b ,
               op x? to b? ,
               op -x to -b) .
end

```

BOBJ finds the correct cobasis, $\{a?, b?, \text{nil-in}, -a, -b\}$, and can prove equivalences of regular languages by circular coinductive rewriting:

```

cred a* + nil == a* .
cred aa* + nil == a* .
cred a*a* == a* .
cred (a+nil)a* == a* .
cred (a+nil)* == a* .
cred a*a == aa* .
cred a(ba)* == (ab)*a .
cred (a*b)*a* == a*(ba*)* .
cred (a+b)* == (a*b*)* .

open .
  op L : -> RegExp .
***> suppose L = aL + b
  eq a?(L) = true . eq -a(L) = L .
  eq b?(L) = true . eq -b(L) = nil .
  eq nil-in L = false .
***> then we can prove that
  cred L == a*b .
close

```

Of course, no one should be surprised that the equivalence of regular expressions is decidable; what we found intriguing is that it can be decided using only circular coinductive rewriting. The correctness of this method is justified by the following easy to check result:

Theorem 6 *Two regular languages are equal iff they are behaviorally equivalent under the above specification, that is, iff they cannot be distinguished by experiments involving the operations nil-in, a?, -a, b?, -b, etc., as above.*

6 Summary and Future Work

This paper has introduced and illustrated circular coinductive rewriting, a new method for proving behavioral prop-

erties which combines behavioral rewriting with circular coinduction. The paper has also sketched the underlying hidden algebraic specification theory and the BOBJ behavioral specification and computation system. A number of examples have demonstrated the surprising power of circular coinductive reasoning.

Future research will investigate other examples, such as the equivalence of automata, finite state machines, processes in process algebra and similar formalisms, context-free grammars, lambda-expressions, linear temporal logic, concurrent connections of buffers and streams, and eventually, more complex examples like communication protocols.

References

- [1] Jon Barwise and Lawrence Moss. *Vicious Circles: On the Mathematics of Non-wellfounded Phenomena*. Cambridge University Press, 1996. Originally appeared as CSLI Lecture Notes 60.
- [2] Michael Bidoit and Rolf Hennicker. Observer complete definitions are behaviourally coherent. In Kokichi Futatsugi, Joseph Goguen, and José Meseguer, editors, *OBJ/CafeOBJ/Maude at Formal Methods '99*, pages 83–94. Theta, 1999. Proceedings of a workshop held in Toulouse, France, 20th and 22nd September 1999.
- [3] Samuel Buss and Grigore Roşu. Incompleteness of behavioral logics. In Horst Reichel, editor, *Proceedings, Coalgebraic Methods in Computer Science (CMCS'00)*, volume 33 of *Electronic Notes in Theoretical Computer Science*, pages 61–79. Elsevier Science, March 2000.
- [4] Louise Dennis, Alan Bundy, and Ian Green. Using a generalisation critic to find bisimulations for coinductive proofs. In *Proceedings, 14th Conference on Automated Deduction*, pages 276–290. Springer, 1996. Lecture Notes in Computer Science, Volume 1249.
- [5] Răzvan Diaconescu. Behavioural coherence in object-oriented algebraic specification. Technical Report IS-RR-98-0017F, Japan Advanced Institute for Science and Technology, June 1998. Submitted for publication.
- [6] Răzvan Diaconescu and Kokichi Futatsugi. *CafeOBJ Report: The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification*. World Scientific, 1998. AMAST Series in Computing, Volume 6.
- [7] Joseph Goguen. Types as theories. In George Michael Reed, Andrew William Roscoe, and Ralph F. Wachter, editors, *Topology and Category Theory in Computer Science*, pages 357–390. Oxford, 1991. Proceedings of a Conference held at Oxford, June 1989.
- [8] Joseph Goguen and Răzvan Diaconescu. Towards an algebraic semantics for the object paradigm. In Hartmut Ehrig and Fernando Orejas, editors, *Proceedings, Tenth Workshop on Abstract Data Types*, pages 1–29. Springer, 1994. Lecture Notes in Computer Science, Volume 785.

- [9] Joseph Goguen and Grant Malcolm. Hidden coinduction: Behavioral correctness proofs for objects. *Mathematical Structures in Computer Science*, 9(3):287–319, June 1999.
- [10] Joseph Goguen and Grant Malcolm. A hidden agenda. *Theoretical Computer Science*, to appear. Also UCSD Dept. Computer Science & Eng. Technical Report CS97–538, May 1997.
- [11] Joseph Goguen and Grigore Roşu. Hiding more of hidden algebra. In Jeannette Wing, Jim Woodcock, and Jim Davies, editors, *FM'99 – Formal Methods*, pages 1704–1719. Springer, 1999. Lecture Notes in Computer Sciences, Volume 1709, Proceedings of World Congress on Formal Methods, Toulouse, France.
- [12] Joseph Goguen and Grigore Roşu. A protocol for distributed cooperative work. In Gheorghe Stefanescu, editor, *Proceedings, FCT'99, Workshop on Distributed Systems*, pages 1–22. Elsevier, 1999. (Iaşi, Romania). Also, Electronic Lecture Notes in Theoretical Computer Science, Elsevier Volume 28.
- [13] Joseph Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ. In Joseph Goguen and Grant Malcolm, editors, *Software Engineering with OBJ: Algebraic Specification in Action*. Kluwer, 2000. Also Technical Report SRI-CSL-88-9, August 1988, SRI International.
- [14] Rolf Hennicker. Context induction: a proof principle for behavioural abstractions. *Formal Aspects of Computing*, 3(4):326–345, 1991.
- [15] Rolf Hennicker and Michel Bidoit. Observational logic. In *Algebraic Methodology and Software Technology (AMAST'98)*, volume 1548 of *Lecture Notes in Computer Science*, pages 263–277. Springer, 1999.
- [16] Bart Jacobs and Jan Rutten. A tutorial on (co)algebras and (co)induction. *Bulletin of the European Association for Theoretical Computer Science*, 62:222–259, 1997.
- [17] Grigore Roşu. Behavioral coinductive rewriting. In Kokichi Futatsugi, Joseph Goguen, and José Meseguer, editors, *OBJ/CafeOBJ/Maude at Formal Methods '99*, pages 179–196. Theta (Bucharest), 1999. Proceedings of a workshop in Toulouse, 20 and 22 September 1999.
- [18] Grigore Roşu and Joseph Goguen. Circular coinduction. Technical Report CSE2000–0647, Dept. Computer Science & Engineering, Univ. California at San Diego, February 2000. Written October 1999.
- [19] Grigore Roşu and Joseph Goguen. Hidden congruent deduction. In Ricardo Caferra and Gernot Salzer, editors, *Automated Deduction in Classical and Non-Classical Logics*, pages 252–267. Springer, 2000. Lecture Notes in Artificial Intelligence, Volume 1761; papers from a conference held in Vienna, November 1998.