ICFEM'10, LNCS 6447, pp. 200–216, 2010

Automating Coinduction with Case Analysis

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

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

Abstract. Coinduction is a major technique employed to prove behavioral properties of systems, such as behavioral equivalence. Its automation is highly desirable, despite the fact that most behavioral problems are Π^0_2-complete. Circular coinduction, which is at the core of the CIRC prover, automates coinduction by systematically deriving new goals and proving existing ones until, hopefully, all goals are proved. Motivated by practical examples, circular coinduction and CIRC have been recently extended with several features, such as special contexts, generalization and simplification. Unfortunately, none of these extensions eliminates the need for case analysis and, consequently, there are still many natural behavioral properties that CIRC cannot prove automatically. This paper presents an extension of circular coinduction with case analysis constructs and reasoning, as well as its implementation in CIRC. To uniformly prove the soundness of this extension, as well as of past and future extensions of circular coinduction and CIRC, this paper also proposes a general correct-extension technique based on equational interpolants.

1 Introduction

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. Since coinduction is too complex to be automated in its full generality, existing tools attempt to implement simpler, algorithmic variants which work in many practical cases. Such a tool is CIRC [7], which implements circular coinduction [4, 10]. Circular coinduction has been recently extended with special contexts [8], generalization and simplification rules [6]. Many computer experiments with CIRC led us to the necessity of introducing and automating case reasoning.

Case analysis is a fundamental algebraic/coalgebraic reasoning technique whose importance has been early noticed and which has been partly investigated (see, e.g., [1, 5]). Automating case analysis in its full generality is a difficult task, which would certainly lead to expensive, non-terminating procedures. In this paper we investigate practical means to incorporate limited but effective support for automatic case analysis in coinductive provers, such as CIRC.

We start by discussing two motivating examples that emphasize the importance of case analysis when proving properties by coinduction.

A *stream* is an infinite-list data structure $a_1 : a_2 : a_3 \ldots$ which can be used to model infinite behaviors. Considering the stream observers hd and tl, defined by $hd(a : s) = a$ and $tl(a : s) = s$, to prove a stream equality $s = s'$ by coinduction one needs to find a set of pairs $R = \{u_i \equiv v_i \mid i \in I\}$, which contains the pair $s \equiv s'$ and which is a congruence with respect to hd and tl, i.e., $u \equiv v \in R$ implies $hd(u) = hd(v)$ and $tl(u) \equiv tl(v) \in R$ [11, 10].

Example 1. We present a situation where case analysis over a term of enumerable sort is needed. We assume that the bitwise negation operator, *not*, is defined over streams of bits using the auxiliary bit-complement operation $\overline{\cdot}$. The function f creates an infinite alternating bit stream, starting with a given first element:

$$\begin{array}{ll} \overline{0} = 1 \quad \overline{1} = 0 & hd(f(a)) = a \\ hd(not(s)) = \overline{hd(s)} & hd(tl(f(a))) = \overline{a} \\ tl(not(s)) = not(tl(s)) & tl(tl(f(a))) = f(a) \end{array}$$

Above, s is a variable of sort stream and a is a variable of sort bit. Let us prove $f(\overline{a}) = not(f(a))$ by coinduction. Take $R = \{f(\overline{a}) \equiv not(f(a)), tl(f(\overline{a})) \equiv tl(not(f(a)))\}$. For the first pair, $hd(f(\overline{a})) = hd(not(f(a)))$ holds. This property can only be checked by making a case analysis over a, which takes values from $\{0, 1\}$. Further, $tl(f(\overline{a})) \equiv tl(not(f(a)))$ is reduced to $tl(f(\overline{a})) \equiv not(tl(f(a))) \in R$. For the second pair, the reasoning is similar. When checking for congruence w.r.t. hd, another case analysis over a is needed.

We show in this paper that, when the system knows that certain terms are of enumerable sort, case analysis can be done automatically. Case analysis based on enumerated sorts can be seen as a particular case of induction. However, we prefer to treat it separately because its integration with the circular coinduction engine is much simpler and, consequently, more efficient.

Example 2. The second example shows how to prove a property over streams of integers. We define the operator *sign* which, when provided a stream of integers, returns another stream with elements from the set $\{-1, 0, 1\}$:

$$hd(sign(s)) = \begin{cases} -1 & \texttt{if } hd(s) < 0 \\ 0 & \texttt{if } hd(s) = 0 \\ 1 & \texttt{if } hd(s) > 0 \end{cases}$$
$$tl(sign(s)) = sign(tl(s))$$

Let us prove that $sign(s) = sign(sign(s))$. We consider the set $R = \{sign(s) \equiv sign(sign(s)) \mid s \text{ is a stream}\}$.

Checking that $hd(sign(s)) = hd(sign(sign(s)))$ can only be done by making a case analysis over $hd(s)$. If, for instance, $hd(s) < 0$, then $hd(sign(s)) = -1$. Therefore $hd(sign(s)) < 0$, so $hd(sign(sign(s))) = -1$. Both the left hand side and the right hand side of the initial equation are reduced to -1 in this case. The other two cases, $hd(s) = 0$ and $hd(s) > 0$, are handled similarly.

In the end, we need to prove that $tl(sign(s)) \equiv tl(sign(sign(s))) \in R$, which is equivalent to $sign(tl(s)) \equiv sign(sign(tl(s))) \in R$. The property holds because

$tl(s)$ is a stream s' and $sign(s') \equiv sign(sign(s')) \in R$ for any s'. By the coinduction principle, $sign(s) = sign(sign(s))$.

To automate case analysis for this situation, we "encapsulate" the definition of *sign* in a single syntactic construct, named *guarded equation*. The tool is able to extract from such equations the information it needs to perform case analysis.

The general goal of this paper is to present our approach to automating coinduction with case analysis, and more concretely to present our extension of CIRC with case analysis statement constructs, to describe our automated implementation of case analysis, and to prove the soundness of our technique and implementation. A secondary but equally important goal is to propose a generic technique to deal with extensions of coinductive proof systems[3], based on what we call equational interpolants. An equational interpolant is a new sentence of the form $\langle e, itp \rangle$, where e is an equation and itp is a set of equations; each of the equations involved in an interpolant can be conditional and can have its own quantifiers, which can be different from the others'. The intuition for $\langle e, itp \rangle$ is simple: e holds whenever itp holds. In other words, to prove $E \vdash e$ one can chose to instead prove $E \vdash itp$. This is somewhat similar to the homonimous notion in Craig interpolation, though the later also imposes restrictions over the signature of the interpolant; we here only take over the intuition that the interpolant interposes between the hypotheses and the task to prove.

Equational interpolants can be used in two ways: 1) to extend the ability of the prover to automatically find new lemmas by preserving the initial entailment relation, and 2) to extend the initial entailment relation in a consistent way with the specification enriched with new constructs. All previous extensions of circular coinduction consist of adding new types of statements together with new proof rules for them to the proof system. Interestingly, all these can be captured as special instances of a similar but more general process involving equational interpolants: the specific statements can be regarded as particular interpolants and the specific proof rules can be regarded as corresponding instances of general interpolant rules. Case analysis is no different. We borrow the general definition of a case statement from [5], but we here capture its semantics as a particular case of specification with equational interpolants. An immediate advantage of our new approach is that we can use case analysis in both ways mentioned above.

In short, the solution adopted in CIRC for automated case analysis is:

- enrich the specification syntax with new constructs, named *CIRC case statements*, for declaring enumerated sorts and guarded equations;
- transform the new constructs above uniformly into *annotated case sentences* (which can be regarded as interpolants);
- extend the coinduction proof engine with a new rule, [CaseAn]; and
- redesign the algorithm for automatic detection of special contexts.

Section 2 introduces notions and notations used throughout the paper, and the derivation rules of CIRC. Section 3 shows how the prover may be extended

[3] The technique appears to be more general, but we have not experimented with it outside our framework discussed here.

with new rules associated to interpolants. Section 4 presents the formal representation of case sentences and their corresponding entailment relation. Section 5 introduces the new syntactic constructs for specifying CIRC case statements. Subsections 5.1 and 5.2 describe how we extend the coinduction engine with a new rule for case analysis, and, how we improve the algorithm for detecting special contexts using case sentences, respectively. Section 5.3 shows how to write behavioral specifications and prove properties using CIRC.

Acknowledgment. The paper is supported in part by NSF grants CCF-0448501, CNS-0509321 and CNS-0720512, by NASA contract NNL08AA23C, CNCSIS grant PN-II-ID-393, and by ANCS 602/12516 (DAK).

2 Behavioral Specification and Circular Coinduction

An *algebraic specification* is a triple $\mathcal{E} = (S, \Sigma, E)$, where S is a set of *sorts*, Σ is a *many-sorted signature* and E is a set of conditional equations of the form $(\forall X)\, t = t'$ `if` *cond*, where $cond = (\bigwedge_{i \in I} u_i = v_i)$, I is a set of indexes; t, t', u_i, and v_i $(i \in I)$ are Σ-terms with variables in X. If $I = \emptyset$ then the equation is *unconditional* and may be written as $(\forall X)\, t = t'$.

A *Σ-context* C is a Σ-term with one occurrence of a distinguished variable $*{:}s$ of sort s. The context is written more explicitly as $C[*{:}s]$ instead of just C. When Σ is understood, a Σ-context may be referred to as a *context*. If $C[*{:}s]$ is a context of sort s' and t is a term of sort s, then $C[t]$ is the term of sort s' obtained by replacing t for $*{:}s$ in C. Consider an equation $e : (\forall X)\, t = t'$ `if` *cond*. By $C[e]$ we denote the equation $(\forall X \cup Y)\, C[t] = C[t']$ `if` *cond*, where Y is the set of non-star variables occurring in $C[*{:}s]$.

A *behavioral specification* (e.g., [10]) is a triple $\mathcal{B} = (S, (\Sigma, \Delta), E)$, where S, Σ and E are the sets composing an algebraic specification $\mathcal{E}$, and Δ is a set of Σ-contexts, called *derivatives*. A derivative in Δ is written as $\delta[*{:}h]$. 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 Δ-context; and 2) if $C[*{:}h']$ is a Δ-context and $\delta[*{:}h]$ is a term of sort h' from Δ, then $C[\delta[*{:}h]]$ is a Δ-context. A *Δ-experiment* is a Δ-context of visible sort.

If $\delta \in \Delta$ and e is an equation, then $\delta[e]$ is called a *derivative* of e. Given an entailment relation $\vdash$ over $\mathcal{E}$, the *behavioral entailment* relation is defined as follows: $\mathcal{B} \Vdash e$ iff $\mathcal{E} \vdash C[e]$ for each Δ-experiment C appropriate for the equation e. In this case, we say that $\mathcal{B}$ *behaviorally satisfies* e. For the streams defined in Section 1, the derivatives are $hd[*{:}Stream]$ and $tl[*{:}Stream]$. If $\mathcal{B} = (S, (\Sigma, \Delta), E)$, then we often write $\mathcal{B} \vdash e$ for $(S, \Sigma, E) \vdash e$ and $\mathcal{B} \cup F$ for $(S, (\Sigma, \Delta), E \cup F)$. We assume that $\vdash$ satisfies properties like reflexivity, monotonicity, transitivity, and Δ-congruence (see [10] for more details).

Circular coinduction [4, 10] is a coinductive proving technique for behavioral properties which can be defined as a proof system (see [10]). To prevent the use of coinductive hypotheses in contextual reasoning, circular coinduction uses a *freezing* operator $\boxed{-} : s \to \mathit{Frozen}$, defined for each sort s; *Frozen* is a new sort. A *frozen equation* is an equation of the form $(\forall X)\ \boxed{t} = \boxed{t'}$ if *cond*.

CIRC implements a circular coinduction engine for the proof system given in [10] using a set of reduction rules of the form $(\mathcal{B}, \mathcal{F}, \mathcal{G}) \Rightarrow (\mathcal{B}, \mathcal{F}', \mathcal{G}')$, where $\mathcal{B}$ represents the behavioral specification, $\mathcal{F}$ is the set of coinductive hypotheses (a set of frozen equations) and $\mathcal{G}$ is the current set of goals. An equational *goal* (proof obligation) is a conditional equation g of the form $(\forall X)t = t'$ `if` *cond*. For the sake of the presentation, the goals are also represented as frozen equations.

Here is a brief description of the reduction rules underlying CIRC:

[Done]: $(\mathcal{B}, \mathcal{F}, \emptyset) \Rightarrow \cdot$
Whenever the set of goals is empty, the system terminates with success.

[Reduce]: $(\mathcal{B}, \mathcal{F}, \mathcal{G} \cup \{\boxed{e}\}) \Rightarrow (\mathcal{B}, \mathcal{F}, \mathcal{G})$ `if` $\mathcal{B} \cup \mathcal{F} \vdash \boxed{e}$
If the current goal is a $\vdash$–consequence of $\mathcal{B} \cup \mathcal{F}$ then $\boxed{e}$ is dropped.

[Derive]: $(\mathcal{B}, \mathcal{F}, \mathcal{G} \cup \{\boxed{e}\}) \Rightarrow (\mathcal{B}, \mathcal{F} \cup \{\boxed{e}\}, \mathcal{G} \cup \{\boxed{\Delta[e]}\})$
`if` $\mathcal{B} \cup \mathcal{F} \nvdash \boxed{e} \wedge e$ is hidden
When the current goal e is hidden and it is not a $\vdash$–consequence, it is added to the specification and its derivatives to the set of goals. $\boxed{\Delta[e]}$ denotes the set $\{\boxed{\delta[e]} \mid \delta \in \Delta\}$.

[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]: $(\mathcal{B}, \mathcal{F}, \mathcal{G} \cup \{\boxed{e}\}) \Rightarrow$ *failure* `if` $\mathcal{B} \cup \mathcal{F} \nvdash \boxed{e} \wedge e$ is visible
This rule stops the reduction process with failure whenever the current goal e is visible and cannot be proved using $\vdash$.

The entailment relation used in CIRC is $\vdash_{\rightarrow,\leftarrow} : \mathcal{E} \vdash_{\rightarrow,\leftarrow} (\forall X)t = t'$ `if` $\bigwedge_{i \in I} u_i = v_i$ iff $\mathrm{nf}(t) = \mathrm{nf}(t')$, where $\mathrm{nf}(t)$, the *normal form* of t, is computed using an enhanced version of the initial specification:

– 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 on t.

The rules [Done], [Reduce], and [Derive] implement the proof rules with the same names given in [10]. The rule [Generalize] is presented in [6]. After a failing stop signaled by [Fail], further human intervention is required in order to identify the source of the failure. There are also some additional rules used only for optimization purposes. An example of such a rule is [Normalize], which computes the normal form of an equation and can be used, for instance, in combination with the rule [Derive].

The next result is a variant of the soundness theorem given in [10].

Theorem 1 (Soundness). *Let $\mathcal{B}$ be a behavioral specification, and $\vdash$ an entailment relation. If $(\mathcal{B}, \mathcal{F}_0 = \emptyset, \mathcal{G}_0 = \boxed{G}) \Rightarrow^* (\mathcal{B}, \mathcal{F}_n, \mathcal{G}_n = \emptyset)$, using [Reduce] and [Derive], then $\mathcal{B} \Vdash G$.*

We use this result in the next section. The proof of correctness for the rule [Generalize] is given in [6]; in the next section we show that it can be regarded as a particular case of a more general technique.

3 Extending CIRC with equational interpolants

In this section we present a technique that allows us to easily extend the proof system with new reduction rules.

Definition 1. 1) *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.*
2) *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:*

$$\frac{(E, \mathcal{I}) \vdash itp}{(E, \mathcal{I}) \vdash e} \text{ if } \langle e, itp\rangle \in \mathcal{I} \tag{1}$$

3) *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 $\langle e, itp\rangle \in \mathcal{I}$.*

Theorem 2. *Let $\mathcal{B} = (S, (\Sigma, \Delta), (E, \mathcal{I}))$ and $\vdash$ be an entailment relation such that $\mathcal{I}$ is $\vdash$–preserving. If e is a Σ–equation then $(S, (\Sigma, \Delta), E) \Vdash e$ if and only if $(S, (\Sigma, \Delta), (E, \mathcal{I})) \Vdash e$.*

The CIRC engine associates a rewrite rule of the form:

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

with each interpolant $\langle e, itp\rangle \in \mathcal{I}$. We write $[\mathsf{itp}] \in \mathcal{I}$ in order to denote that the rule [itp] is associated with an interpolant from $\mathcal{I}$. The following theorem states that if we enhance the proof system presented in [10] with interpolants, then it remains sound w.r.t. the new entailment relation.

Theorem 3. *Let $\mathcal{B}$ be a behavioral specification, $\vdash$ an entailment relation, and $\mathcal{I}$ a set of interpolants. If $(\mathcal{B}, \mathcal{F}_0 = \emptyset, \mathcal{G}_0 = \boxed{G}) \Rightarrow^* (\mathcal{B}, \mathcal{F}_n, \mathcal{G}_n = \emptyset)$, using [Reduce], [Derive] and the rules associated to $\mathcal{I}$, then $\mathcal{B} \Vdash G$.*

A consequence of the proof of Theorem 3 is that the equational interpolants preserve the circular coinduction principle [10]:

Corollary 1. *In the hypothesis of Theorem 3, there is a set of frozen equations $\boxed{F}$ such that $\mathcal{B} \cup \boxed{F} \vdash \boxed{\Delta[F]}$.*

Notice that in the conclusion of Theorem 3, $\Vdash$ is built on the extended $\vdash$ (the extension of the initial entailment to specifications with interpolants). If $\mathcal{I}$ is $\vdash$-preserving, then $\Vdash$ is the behavioral extension of the initial $\vdash$. Therefore there are two ways of using interpolants within a coinductive proof:
Implicit use. In this case the equational interpolants must be $\vdash$–preserving and are used to justify why other rules are sound; the soundness of their use is given by Theorem 2. An example of implicit using of equational interpolants is that of the generalization rule [6]. By this rule, a concrete equation $u = u'$ is replaced by a more general one $t = t'$. This means that there is a substitution θ such that

$\theta(t) = u$ and $\theta(t') = u'$. It is easy to see that [Generalize] for $u = u'$ and θ is equivalent with the rule [itp] corresponding to the interpolant $\langle u = u', \{t = t'\}\rangle$. Theorem 4 in [6] becomes a direct consequence of Theorem 3.

Explicit use. The specification includes special syntactical constructs for denoting equational interpolants. For instance, the equational interpolants may be explicitly included in a CIRC theory using simplification rules [6] or case sentences (see Section 5). In this case the user is the one who decides whether the equational interpolants included in the specification are $\vdash$–preserving or that they really extend $\vdash$. In other words, the explicit use of equational interpolants is a part of the specification design.

4 Specifications with Cases

In this section we recall from [5] the definitions for case sentences and we show that their semantics can be given by means of equational interpolants.

Let (S, Σ) be an algebraic signature. A *Σ-case sentence* over the set of variables Y is a nonempty set $\{case_i \mid i \in I\}$ written as $(\forall Y)\,(\bigvee_{i\in I} case_i)$, where $case_i = (\bigwedge_{j\in J_i} u_j^i = v_j^i)$, and $u_j^i, v_j^i \in \mathcal{T}_\Sigma(Y)$, for each $i \in I$ and $j \in J_i$. Hence, cases and conditions have the same syntax. If $\theta : Y \to T_\Sigma(X)$ is a substitution, then $\theta(case_i)$ denotes the case $\theta(\bigwedge_{j\in J_i} u_j^i = v_j^i) = \bigwedge_{j\in J_i} \theta(u_j^i) = \theta(v_j^i)$. A *specification with cases* is a triple $\mathcal{E} = (S, \Sigma, (E, \mathcal{C}))$, where (S, Σ, E) is an algebraic specification and $\mathcal{C}$ is a set of case sentences.

An entailment relation $(S, \Sigma, E) \vdash e$ can be extended to specifications with cases $(S, \Sigma, (E, \mathcal{C})) \vdash e$ by means of equational interpolants. Each specification with cases $(S, \Sigma, (E, \mathcal{C}))$ is associated with a specification with equational interpolants $(S, \Sigma, (E, \mathcal{I}_\mathcal{C}))$, where $\mathcal{I}_\mathcal{C}$ is the set of pairs $\langle e, itp_{case,\theta}(e)\rangle$ with e an equation $(\forall X)t = t'$ if *cond*, *case* a case sentence $(\forall Y)(\bigvee_{i\in I} case_i)$ in $\mathcal{C}$, $\theta : Y \to T_\Sigma(X)$ a substitution, and $itp_{case,\theta}(e)$ the set $\{(\forall X)t = t' \text{ if } cond \wedge \theta(case_i) \mid i \in I\}$. In other words, each triple that consists of an equation, a case sentence, and a substitution uniquely defines an equational interpolant. The second inference rule from Definition 1 interpreted for the interpolant defined by a case sentence becomes similar to the one given in [5]:

$$\frac{\begin{array}{l}(\forall Y)\,(\bigvee_{i\in I} case_i) \in \mathcal{C},\\ \theta : Y \to \mathcal{T}_\Sigma(X),\\ (\forall i \in I)(S, \Sigma, (E, \mathcal{C})) \vdash (\forall X)\, t = t' \text{ if } cond \ \wedge\ \theta(case_i)\end{array}}{(S, \Sigma, (E, \mathcal{C})) \vdash (\forall X)\, t = t' \text{ if } cond} \qquad (2)$$

However, we do not impose a "completeness" condition for case sentences, which in our terms is equivalent to saying that the interpolants $\mathcal{I}_\mathcal{C}$ are $\vdash$–preserving. For instance, we may have a specification as follows:

$$\begin{array}{ll} even(0) = true & f(x) = x \text{ if } even(x)\\ even(s(0)) = false & f(x) = s(x) \text{ if } not\ even(x)\\ even(s(s(x))) = even(x) & \end{array}$$

If we consider the case sentence c given by $(\forall x) even(x) = true \vee even(x) =$

false and $\vdash$ denotes the equational deduction, then we have $(S, \Sigma, (E, \{c\})) \vdash (\forall x) even(f(x)) = true$. Obviously, we cannot infer from the specification that $(S, \Sigma, E) \vdash (\forall x)(even(x) == true$ or $even(x) == false)$ (and hence $(S, \Sigma, E) \nvdash (\forall x) even(f(x)) = true$) because this property is an inductive consequence. Therefore the rule (2) is a real extension of the initial entailment relation (here the equational deduction). The explicit use of case sentences in specifications may directly influence the definition of the entailment relation. In this way the user has a larger freedom in using case analysis.

As it is noted in [5], the use of case analysis at this level of generality is very expensive and finding an appropriate substitution θ is a difficult task which cannot be easily automatized. In [4] the following method is proposed: each case sentence comes with a pattern, usually denoted by p, which is just a Σ-term with variables. The case analysis rule is enabled only if the pattern p matches a subterm of t or t', and then the substitution also comes for free. The user must specify the pair (pattern, case sentence) to be used for a particular task.

We want to exploit the same idea but in an automated way. For this, we introduce *annotated case sentences*, which are pairs of the form $(p, \{case_i \mid i \in I\})$. Here, $p \in T_\Sigma(Y)$ is the *pattern* and $(\forall Y)\,(\bigvee_{i \in I} case_i)$ is a case sentence.

As we said above, the case analysis is a proper component of the specification. Therefore the solution we propose here is to include in the specification language special syntactical constructs from which annotated case sentences can be automatically computed. Knowing the set of annotated case sentences included in the specification, a prover can supervise the case analysis making use of proof tactics. In the next section we introduce three such syntactical constructs.

5 Implementation in CIRC

CIRC theories extend the syntax of Full-Maude theories by allowing the user to specify *derivatives* [7], *special contexts* [8] and *simplification rules* [6]. Here we present how one can use new syntactic constructs in CIRC theories that enable the prover to automatically use case analysis. These new syntactic constructs are named *CIRC case statements*. We introduce three types of CIRC case statements: enumerated sorts, guarded equations and annotated case sentences.

Enumerated sorts are declared using the syntax "`enum` s `is` $ct_1 \ \ldots \ ct_n$.", where s is the name of the sort and $ct_i, i = \overline{1..n}$, are the constants that define it. The *guarded equations* syntax is: "`geq` $t =\ t_1$ `if` $case_1$ `[]` $\ldots\ t_n$ `if` $case_n$ `[]` .", where t and $t_i, i = \overline{1..n}$ are $\mathcal{T}_\Sigma(Y)$-terms and $case_i, i = \overline{1..n}$ are disjoint conditions. The notation for guarded equations is inspired from Dijkstra's command language [3], except that the syntax of the guards is inspired from the Maude convention for the conditional equations. *Annotated case sentences* are directly declared using the syntax "`cases pattern` $= p$ `if` $case_1$ `\/` $\ldots$ `\/` $case_n$.".

The syntactic constructs presented above are not disjoint. For instance, the enumerated sorts and the guarded equations can be seen as particular instances of the annotated case sentence (see below). We found these syntactic constructs adequate in most of the case studies we considered. We believe they are more intuitive and familiar to a programmer than a syntactic construct that allows

the direct definition of an annotated case sentence. Moreover, the solution we present here can easily be extended with other syntactic constructs if they are considered to be useful in practice.

A *CIRC specification with cases* is a triple $\mathcal{E} = ((S, S^e), \Sigma, (E, E^g, \mathcal{C}))$, where S, Σ, E have similar meanings to those from the equational many sorted specifications, S^e is a set of enumerated sorts, E^g is a set of guarded equations, and $\mathcal{C}$ are the annotated case sentences. We associate each $\mathcal{E} = ((S, S^e), \Sigma, (E, E^g, \mathcal{C}))$ with a "compiled" specification with cases $\widetilde{\mathcal{E}} = (\widetilde{S}, \widetilde{\Sigma}, (\widetilde{E}, \widetilde{\mathcal{C}}))$, where:

- $\widetilde{S}$ is S together with the names of the enumerated sorts;
- $\widetilde{\Sigma}$ is Σ together with the constants of the enumerated sorts;
- $\widetilde{E}$ is E together with the conditional equations $(\forall Y)t = t_i$ if $case_i$, $i = \overline{1..n}$, for each guarded equation in E^g;
- $\widetilde{\mathcal{C}}$ is the set of annotated case sentences obtained in the following manner:
  - any enumerated sort "`enum` s `is` ct_1 ... ct_n ." defines the annotated case sentence $(\forall\{y\})\,(y, y = ct_1 \vee \cdots \vee y = ct_n)$, where y is of sort s;
  - any guarded equation "`geq` $t = t_1$ `if` $case_1$ `[]` ... t_n `if` $case_n$ `[]` ." defines the annotated case sentence $(\forall Y)\,(t, case_1 \vee \ldots \vee case_n)$, where Y is the set of the variables occurring in the guarded equation;
  - any sentence "`cases pattern` $= p$ `if` $case_1$ `\/` ... `\/` $case_n$." defines the annotated sentence $(\forall Y)\,(p, case_1 \vee \ldots \vee case_n)$, where Y is the set of the variables occurring in the pattern p.

Example 3. For the first example presented in Section 1 we have one enumerated sort in S^e: `enum Bit is 0 1 .` $\widetilde{\mathcal{E}}$ has the following components:

- $\widetilde{S} = S \cup \{\texttt{Bit}\}$;
- $\widetilde{E} = E$;
- $\widetilde{\Sigma} = \Sigma \cup \{\texttt{op 0 : -> Bit .}, \texttt{op 1 : -> Bit .}\}$;
- $\widetilde{\mathcal{C}} = \{(\texttt{B},\ \texttt{B} = 0 \vee \texttt{B} = 1)\}$, where `B` is a variable of sort `Bit`.

Example 4. For the second example, we have one guarded equation in E^g:

```
geq hd(sign(S)) =
    -1 if hd(S) < 0 = true []
     0 if hd(S) = 0        []
     1 if hd(S) > 0 = true [] .,
```

where `S` is a variable of sort `Stream`. $\widetilde{\mathcal{E}}$ is given by:

- $\widetilde{S} = S$; $\widetilde{\Sigma} = \Sigma$;
- $\widetilde{E} = E \cup \{$ `ceq hd(sign(S)) = -1 if hd(S) < 0 = true .` ,
  `ceq hd(sign(S)) =  0 if hd(S) = 0 .` ,
  `ceq hd(sign(S)) =  1 if hd(S) > 0 = true .` $\}$;
- $\widetilde{\mathcal{C}} = \{(\texttt{hd(sign(S))},\ \texttt{hd(S)} < 0 = \texttt{true} \vee \texttt{hd(S)} = 0 \vee \texttt{hd(S)} > 0 = \texttt{true})\}$.

Instead of using guarded equations, one could also declare the conditional equations from $\widetilde{E}$ presented above and specify the case sentence by: "`cases pattern = hd(sign(S)) if hd(S) < 0 = true \/ hd(S) = 0 \/ hd(S) > 0 = true .`". In this way the user has the freedom to choose the syntax which describes his/her system in the best way.

It is worth noting that if a specification with cases is used, then the entailment relation is used during the proving process is that given by the associated equational interpolants (see Section 4).

5.1 Extending the circular coinduction engine

In this section we describe how we enhanced the CIRC engine with automatic case analysis, and prove the correctness of our extension. We here only consider a behavioral specification with general cases, $\widetilde{\mathcal{B}} = (\widetilde{S}, (\widetilde{\Sigma}, \Delta), (\widetilde{E}, \widetilde{\mathcal{C}}))$; other specialized case statements can be desugared into general ones, as explained above.

We extend the coinduction proving engine 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` $cond \wedge \theta(case_i) \mid i \in I\}\rangle$:

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

where $\mathcal{F}$ is the set of frozen axions and $\mathcal{G} \cup \{\boxed{t} = \boxed{t'}\ \texttt{if}\ cond\}$ is the current set of goals. By Theorem 3, the extended engine is sound because [CaseAn] is a rule associated to the interpolant defined by (2).

One of the challenges we encountered was to find the best candidate for case analysis when more than one substitution could be applied. In our experiments, the strategy that gave the best results was to identify a subterm of either t or t' with the smallest height possible where at least one of the patterns p from $\widetilde{\mathcal{C}}$ could provide a substitution.

5.2 Computing Special Contexts using Cases

In this subsection we give an intuition on what special contexts are with a few examples, and present the new algorithm for detecting special contexts using case analysis, as well as some required notions, and the result expressing the correctness of the algorithm. The formal background for special contexts is presented more in detail in [8].

We have seen in Section 2 that the frozen hypotheses cannot be used in contextual reasoning. However, there are contexts under which it is "safe" to use the frozen hypotheses. In [8] it is defined such a class of contexts, called special contexts. A context $\gamma[*{:}h]$ is called *special* if, by definition, for any experiment C for γ there is some term t such that $\mathcal{B} \vdash C[\gamma[*{:}h]] = t$ and each occurrence of $*{:}h$ in t appears in a subterm which is an experiment of depth smaller than or equal to that of C.

Example 5. Let us consider the operation f over streams of bits defined as: $f(0 : s) = 1 : f(s), f(1 : s) = 0 : 0 : f(f(f(s)))$. By making a case analysis we

deduce that $f(*{:}Stream)$ is a special context (see below). Knowing this, CIRC is able to automatically prove that $f(0^\infty) = 1^\infty$ and $f(1^\infty) = 0^\infty$.

Not all contexts are special. Consider, for instance, the operation *odd* defined by $odd(a : b : s) = a : odd(s)$. Let s and t be specified by $hd(s) = hd(t)$, $tl(s) = odd(s)$ and $tl(tl(t)) = odd(t)$. If we wrongly assume that $odd(*{:}Stream)$ is special then we manage to prove that $odd(t) = s$, which is unsound.

CIRC has been able so far to automatically detect special contexts deriving from the operators defined using unconditional equations. Now the prover may detect contexts that derive from operators defined using specifications with cases (and implicitly, conditional equations). For instance, with the enhanced algorithm, CIRC detects that $sign(*{:}Stream)$, introduced in the second motivating example, and $f(*{:}Stream)$, defined in Example 5, are special contexts.

We next present in detail the algorithm computing special contexts for specifications with case sentences and its correctness. In order to fix the terms of the discussion, for the rest of this subsection we consider the following items:

- $\mathcal{B} = ((S, S^e), (\Sigma, \Delta), (E, E^g, \mathcal{C}))$, a fixed CIRC confluent and terminating behavioral specification with cases;
- $\widetilde{\mathcal{B}} = (\widetilde{S}, (\widetilde{\Sigma}, \Delta), (\widetilde{E}, \widetilde{\mathcal{C}}))$, the compiled behavioral specification with cases;
- any derivative $\delta \in \Delta$ is $(\widetilde{S}, \widetilde{\Sigma}, \widetilde{E})$-irreducible;
- $Ctx(\Sigma)$, the set of all Σ-contexts
- $\Sigma^{hidden} \subseteq \widetilde{\Sigma}$, the set of operations with hidden result and at least one hidden argument;
- $Ctx^\circ(\Sigma^{hidden})$, the set of contexts $f(x_1, \ldots, x_n)$ with $f \in \Sigma^{hidden}$, $x_i = *$ for exactly one hidden argument i and for $j \neq i, x_j$ are variables;
- for a Δ-context C, the *hidden depth* of C is defined by $|C|^\bullet = |C|$ if C is hidden, and $|C|^\bullet = |C| - 1$ if C is visible;
- a fixed set $\Gamma \subseteq Ctx^\circ(\Sigma^{hidden})$
- a *generalized constant* is an operation whose arguments are of visible sort

The problem of deciding if a given context is special for a given specification is Π^0_2-complete. This complexity is due to the facts that $\mathcal{B} \vdash C[\gamma[*{:}h]] = t$ must be tested for all Δ-experiments C and that testing $C[\gamma[*{:}h]] = t$ with t satisfying the property from the definition of the special contexts is recursive enumerable. In practice, t is the normal form of $C[\gamma[*{:}h]]$. Moreover, the set of candidates for special contexts is also infinite. So, the best we can do is to find an algorithm which tests a property similar to the one above for a finite set of candidates and a finite set of Δ-contexts. Regarding the candidates, we are looking for a maximal set of minimal depth special contexts which is closed under composition: if γ_1 and γ_2 are special and $\gamma_1[\gamma_2]$ is defined, then $\gamma_1[\gamma_2]$ is special. The minimal set of Δ-context which must be tested is Δ itself. Therefore, as described in [8], we try to find a property $Comp(C, t)$ and a set $\Gamma \subseteq Ctx^\circ(\Sigma^{hidden})$ such that the property $Special(\Gamma)$, given by:

$$Special(\Gamma) \stackrel{\text{def}}{=} (\forall \gamma \in \Gamma)(\forall \delta \in \Delta)\, Comp(\delta, \gamma)$$

implies that each Γ-context is special. If we have an algorithm for computing the

predicate $Comp(C,t)$, then the searching for a suitable Γ requires the evaluation of the predicate for a small set of pairs (C,t).

First we present an axiomatic definition for the predicate $Comp$.

Definition 2. *A Δ-*compositional structure *for Γ is a pair $(\mathcal{T}, Comp)$, where $\mathcal{T}$ is a set of terms and $Comp(C,t)$ is a predicate defined over Δ-contexts C and terms $t \in \mathcal{T}$, which together satisfy the following conditions:*

1. *$Ctx(\Gamma) \subseteq \mathcal{T}$*
2. *$Comp(*,t) = Comp(t,*) = true$;*
3. *let C_1 and C_2 be two Δ-contexts such that $C_1[C_2]$ is defined; if $Comp(C_2,t_2)$ and $(\forall t \in \mathcal{T})\, Comp(C_1,t)$, then $Comp(C_1[C_2],t_2)$;*
4. *let γ_1 and γ_2 be two Γ-contexts such that $\gamma_1[\gamma_2]$ is defined; if $Comp(C,\gamma_1)$ and $((\forall D \in Ctx(\Delta))\ |D|^\bullet \leq |C|^\bullet$ implies $Comp(D,\gamma_2))$, then $Comp(C,\gamma_1[\gamma_2])$.*

The first main result of this section shows that the property $Special(\Gamma)$ can be extended to Γ-contexts and Δ-contexts for the case of a Δ-compositional structure.

Theorem 4. *Let $(\mathcal{T}, Comp)$ a Δ-compositional structure for Γ. If $Special(\Gamma)$, then $(\forall \gamma \in Ctx(\Gamma))(\forall C \in Ctx(\Delta))\, Comp(C,\gamma)$.*

A particular structure $(\mathcal{T}, Comp)$ for specifications without cases is given in [8]. Here we extend that structure to specifications with cases. The definition for $\mathcal{T}$ remains unchanged, namely that of (k,Γ)-composite terms; we recall it in order to make the paper self-contained.

Definition 3. *Let k be an integer number ≥ -1. A (k,Γ)-*composite *is defined as follows:*

1. *any non-star variable and any constant is a $(-1,\Gamma)$-composite;*
2. *any Δ-context C is a $(|C|^\bullet,\Gamma)$-composite;*
3. *if $f : v_1 \dots v_n \to v$ is a data operator or a generalized constant and t_i is a (k_i,Γ)-composite for $i = 1,\dots,n$, then $f(t_1,\dots,t_n)$ is a (k,Γ)-composite, where $k = \max\{k_1,\dots,k_n\}$;*
4. *if $\gamma \in \Gamma$ and t is a (k,Γ)-composite, then $\gamma[t]$ is a (k,Γ)-composite;*
5. *if C is a Δ-context, t a (k,Γ)-composite with $k = -1$ or t of the form $g(t_1,\dots,t_n)$ with g generalized constant, then $C[t]$ is a (k,Γ)-composite.*

The new definition of the predicate $Comp$ is based on the notion of normal form of a term computed in the presence of cases.

We first define the operation Eqn, which transforms the provided conjunction of cases into a set of equations in the following manner:

$$Eqn(\textstyle\bigwedge_{i\in I} u_i = v_i) = \bigcup_{i\in I} Eqn(u_i = v_i)$$

$$Eqn(u_i = v_i) = \begin{cases} \{u_i = v_i\}, & \text{the set variables from } u_i \text{ is included in that of } v_i \\ \{v_i = u_i\}, & \text{otherwise} \end{cases}$$

If $\widetilde{\mathcal{C}}$ is a set of annotated case sentences, then the *$\widetilde{\mathcal{C}}$-normal form* of a term t is the set $\mathrm{nf}_{\widetilde{E},\widetilde{\mathcal{C}}}(t)$ of pairs $\langle t', case\rangle$ satisfying: $t' = \mathrm{nf}_{\widetilde{E}\cup Eqn(case)}$ and there is no

pattern p in $\widetilde{\mathcal{C}}$ which is an instance of a subterm of t'. The component *case* is a conjunction of cases in $\widetilde{\mathcal{C}}$ used in the rewriting obtaining the irreducible term t'. An algorithm computing $\mathrm{nf}_{\widetilde{E},\widetilde{\mathcal{C}}}$ is:

$\mathrm{nf}_{\widetilde{E},\widetilde{\mathcal{C}}} \leftarrow \langle \mathrm{nf}_{\widetilde{E}}(C[t]), nil\rangle$
while $(\exists\langle t', case\rangle \in \mathrm{nf}_{\widetilde{E},\widetilde{\mathcal{C}}})(\exists\theta : Y \to \mathcal{T}_\Sigma(X))(\exists(p, \bigvee_{i\in I} case_i) \in \widetilde{\mathcal{C}})$
  such that $\theta(p)$ is a subterm of t' **do**
  $\mathrm{nf}_{\widetilde{E},\widetilde{\mathcal{C}}} \leftarrow \mathrm{nf}_{\widetilde{E},\widetilde{\mathcal{C}}} - \{\langle t', case\rangle\}$
  $\cup\ \{\langle \mathrm{nf}_{\widetilde{E}\cup Eqn(case)\cup Eqn(\theta(case_i))}(t'), case \wedge \theta(case_i)\rangle \mid i \in I\}$

Let $terms(\mathrm{nf}_{\widetilde{E},\widetilde{\mathcal{C}}}(t))$ denote the set $\{t' \mid \langle t', case\rangle \in \mathrm{nf}_{\widetilde{E},\widetilde{\mathcal{C}}}(t)\}$. We can define now the predicate *Comp*:

$$Comp(C,t) \stackrel{\text{def}}{=} (\forall t' \in terms(\mathrm{nf}_{\widetilde{E},\widetilde{\mathcal{C}}}(C[t]))))\ t' \text{ is a } (k',\Gamma)\text{-composite} \wedge k' \leq k + |C|^\bullet$$

where t is a (k,Γ)-composite, and C is a Δ-context. Since any Γ-context is a $(0,\Gamma)$-composite, $C[*] = C$ and $*[t] = t$.

Theorem 5. *Let $\mathcal{B} = (S, (\Sigma, \Delta), (E, \mathcal{C}))$ be a behavioral specification with cases and Γ a subset of $Ctx^\circ(\Sigma^{hidden})$ such that $Special(\Gamma)$ holds. Then any Γ-context γ is special.*

Example 6. Consider the operation f defined in Example 5. The normal forms of $hd(f(s))$ and $tl(f(s))$ are $\{\langle 1, hd(s) = 0\rangle, \langle 0, hd(s) = 1\rangle\}$ and $\{\langle f(tl(s)), hd(s) = 0\rangle, \langle 0 : f^3(tl(s)), hd(s) = 1\rangle\}$ respectively. If $\Gamma = \{f(*{:}Stream)\}$, then it is easy to see that $Special(\Gamma)$ holds and therefore $f(*{:}Stream)$ is a special context.

Theorem 5 is the foundation for an algorithm computing a set of context $\Gamma \subseteq Ctx^\circ(\Sigma^{hidden})$, which is a basis for special contexts. The description of the algorithm is the same as the one presented in [8], except that the call $Comp(\delta, \gamma)$ requires the computation of the normal forms with cases for $\delta[\gamma]$, as presented above. Therefore the theorem above also ensures the correctness of both versions of the algorithm.

5.3 CIRC with case analysis at work

In this section we present how CIRC theories are specified and a few commands in order to automatically prove properties using case analysis. The reader may use the web interface at `http://fsl.cs.uiuc.edu/index.php/Special:CircOnline` in order to test the examples.

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:

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

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
```

Another challenging example is inspired from [9] and consists of proving that $Rev3(N)(Rev3(N)(S)) = S$, where

$$Rev3(N)(S) = Z3(T3(N)(S), T3(N-1)(S), T3(N-2)(S))$$
$$hd(Z3(S_1,S_2,S_3)) = hd(S_1) \qquad tl(Z3(S_1,S_2,S_3)) = Z3(S_2,S_3,tl(S_1))$$
$$hd(T3(N)(S)) = hd(tl^{n \bmod 3}(S)) \quad tl(T3(N)(S)) = T3(N)(tl^3(S))$$

with N ranging over natural numbers and S over streams. Even if the definition of $hd(T3(N)(S))$ is given by cases (because of n mod 3), is not recommended to use guarded equations for specifying it because it is possible to obtain patterns of the form $hd(T3(N-1)(S))$, which forces a case analysis on "$(N-1)$ mod 3"; now we have to include in the specification how to compute "$(N-1)$ mod 3" when we know "N mod 3" and how to compute "N mod 3" when we know "$(N-1)$ mod 3" and this cannot be done using only rewriting (it is a source of non-termination). Therefore we specified it with conditional equations and we added the case sentence

$$\texttt{cases pattern} = N \texttt{ if } N \bmod 3 = 0 \vee N \bmod 3 = 1 \vee N \bmod 3 = 2 \ .$$

Even so the proof is long and complex: 12 case analyses and 14 new lemmas automatically discovered.

6 Conclusions

We presented a simple and efficient solution for automating coinductive reasoning with case analysis. The starting point was the extension of the specifications and of the entailment relation with case sentences given in [5]. A novelty of the approach presented here consists of giving semantics to case sentences by means of equational interpolants, a general technique for extending coinductive provers like CIRC introduced also here. The soundness of the use of equational interpolants in the coinductive proving process is shown and hence the soundness of the reasoning with cases is obtained as a consequence.

The basic idea is to include special syntactical constructs in the specification language. These special constructs are then processed in order to exdd tract annotated case sentences. A prover can supervise the application of the case analysis by means of proof tactics.

Using the concept of "normal form with cases", we were able to write algorithms and heuristics helping the prover. In particular, we showed that extending the algorithm computing the special contexts with case analysis, the prover was able to find a larger class of special contexts. Consequently, a larger class of

properties can be proved. The simpler the predicate *Comp* is, the faster the algorithm for detecting special contexts becomes. Therefore, as future work, there is room and motivation for improving the form of the predicate.

Case analysis based on three syntactic constructs, enumerated sorts, guarded equations and annotated case sentences, has been implemented in CIRC and experiments showed that the new prover is able to handle a large class of practical examples.

A similar approach is given in [2], where the induction and a contextual simplification technique are used to prove behavioral (observational) properties. That approach also deals with case analysis and critical contexts (which are different from special contexts). Circular coinduction is more flexible because it is parametric in the basic entailment relation, and consequently can prove more coinductive properties. On the other hand, we currently do not disproof conjectures (CIRC only reports failure to find a proof under specified constraints, but not that the property is false).

References

1. Adel Bouhoula and Michaël Rusinowitch. Automatic case analysis in proof by induction. In *IJCAI*, pages 88–94, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.
2. Adel Bouhoula and Michaël Rusinowitch. Observational proofs by rewriting. *Theor. Comput. Sci.*, 275(1-2):675–698, 2002.
3. Edsger W. Dijkstra. Guarded commands, non-determinacy and formal derivation of programs. *Commun. ACM*, 18(8):453457, 1975.
4. Joseph Goguen, Kai Lin, and Grigore 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.
5. Joseph Goguen, Kai Lin, and Grigore Roşu. Conditional circular coinductive rewriting with case analysis. In *WADT*, volume 2755 of *Lecture Notes in Computer Science*, pages 216–232. Springer, 2002.
6. 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.
7. Dorel Lucanu, Eugen-Ioan Goriac, Georgiana Caltais, and Grigore Roşu. CIRC: A behavioral verification tool based on circular coinduction. In *CALCO 2009*, volume 5728 of *LNCS*, pages 433–442. Springer, 2009.
8. Dorel Lucanu and Grigore Roşu. Circular coinduction with special contexts. In Karin Breitman and Ana Cavalcanti, editors, *ICFEM*, volume 5885 of *LNCS*, pages 639–659. Springer, 2009.
9. M. Niqui and J.J.M.M. Rutten. Sampling, splitting and merging in coinductive stream calculus. In *Mathematics of Program Construction 2010 (MPC'10)*, 2010. To appear. See CWI Technical report SEN-E0904, 2009, `http://homepages.cwi.nl/~janr/papers/`.
10. Grigore Roşu and Dorel Lucanu. Circular Coinduction – A Proof Theoretical Foundation. In *CALCO 2009*, volume 5728 of *LNCS*, pages 127–144. Springer, 2009.
11. J. J. M. M. Rutten. A coinductive calculus of streams. *Mathematical Structures in Computer Science*, 15(1):93–147, 2005.