CWI

Coiterative Morphisms: Interactive Equational Reasoning for Bisimulation, using Coalgebras

M. Niqui

SEN-1003

Centrum Wiskunde & Informatica

P.O. Box 94079, 1090 GB Amsterdam (NL)
Science Park 123, 1098 XG Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

ISSN 1386-369X

Coiterative Morphisms: Interactive Equational Reasoning for Bisimulation, using Coalgebras

Milad Niqui [⋆]

Department of Software Engineering
Centrum Wiskunde & Informatica, The Netherlands
M.Niqui@cwi.nl

Abstract. We study several techniques for interactive equational reasoning with the bisimulation equivalence. Our work is based on a modular library, formalised in *Coq*, that axiomatises weakly final coalgebras and bisimulation. As a theory we derive some coalgebraic schemes and an associated coinduction principle. This will help in interactive proofs by coinduction, modular derivation of congruence and co-fixed point equations and enables an extensional treatment of bisimulation. Finally we present a version of the λ-coinduction proof principle in our framework.

Keywords: coinduction; bisimulation; theorem proving
MSC Classification: 18C50 68Q85
ACM Classification: F.4.1 I.2.3

1 Introduction

One of the problem domains that theorem provers deal with is tackling the properties of infinite object. In this domain, coinduction in an important proof method. Coinduction is the dual of induction, both for computation and reasoning and while stemming from Knaster–Tarski theorem in a set theoretical framework, it can be studied from a category theoretical [23] or type theoretical point of view [10]. Many theorem provers are capable of dealing with coinductive proofs. As the Knaster–Tarski theorem needs only the basics of set theory and lattice theory, any higher order theorem prover can implement coinduction [1, 20]. Moreover, some of the theorem proving tools have built-in support for coinduction. Among the latter are the ones based on constructive type theory such as *Coq* and *Agda* where *coinductive types* serve this purpose.

In theorem proving with coinduction the main challenge is to identify mechanisable aspects and in places where full automation is impossible, to provide assistance for interactive proofs. This is general issue that concerns all systems dealing with infinite objects.

In addition to this, the systems that are based on constructive type theory face another, more fundamental, issue. Type theories where termination is crucial for finite objects enforce similar restrictions for ensuring *productivity* of functions on infinite objects. These restrictions are syntactic tests (e.g. guardedness checks in *Coq* and *Agda*) and will inevitably exclude some legitimate

[⋆] Supported by a VENI grant from the Netherlands Organisation for Scientific Research (NWO).

productive definitions. Thus not all *Haskell* programs, even those describing total functions, are accepted in coinductive type theories. An even more complex problem is the following: the restrictions for ensuring productivity rises not only in function definitions but also in coinductive proofs. The reason is that in these type theories every coinductive proof corresponds to constructing an element of a coinductive type. Thus several seemingly natural cyclic proofs would be refused by the guardedness checker.

So for the users of systems based on type theory there are two problems with coinduction based on coinductive types: (1) many total definitions are not accepted; and (2) even if they are accepted, proving their properties is problematic as there are not special facilities for coinduction.[1]

We present a framework that intends to address both of these two problems. In a high level, our approach is to develop a theory of coalgebras. This will be a library containing abstract results and parametrised by the choice of functors. The built-in coinductive types will then form one possible implementation of this library but the user have the choice of choosing other implementations. In a lower level, inside this library to tackle the aforementioned problems we directly formalise various categorical definition and proof schemes from the theory of coalgebras. Each *scheme* not only allows the formalisation of a class of specifications (or *Haskell*-like programs) satisfying a specific syntactic form, but also it comes with a suitable coinduction proof principle.

In an earlier work [18], we presented a previous version of our library in *Coq* where we aimed in a partial solution to problem (1) by implementing the λ-coiteration [2]. The present work is a sequel to that where we extend our framework with facilities to address problem (2).

Our aims is to show that by formalising a theory of coalgebras, *independent* of coinductive types, one can transfer proofs by coinduction to theorem provers. This has the pleasant side effect that our framework can be translated into systems that lack coinductive types and can be used for dealing with coinduction there. Working in *Coq*, a constraint for us is the *intensionality* of type theory (Section 2). They will become non-issue for possible translations of our work in systems with an extensional equality (overwhelming majority of systems). So the core of our work, implementing coalgebraic schemes in a logical framework is applicable to and can be potentially interesting for other theorem proving systems. A complete *Coq* formalisation of the material in this paper can be found in [19].

Related Work. Anton Setzer proposes a built-in implementation of weakly final coalgebras (as opposed to the *Haskell*-like coinductive types) in intensional type theories, which is also implemented in *Agda* [24, 25]. We follow the same general philosophy. Our emphasise is on having a light-weight modular library

[1] There is a third, rather serious problem, with equality and subject reduction for coinductive types in intensional type theories [8, 15]. But this is orthogonal to our work.

that is implementable within any logical framework. Furthermore we focus on λ-coiteration scheme as the coalgebraic definition scheme.

Extensional functors (Section 4) are used in [14, 13]. Hancock and Setzer develop the weakly final coalgebra and bisimulation for a very powerful functor capable of representing interactive IO programs in intensional type theory [10]. Their work is formalised in *Agda* [16] using induction-recursive universe which is beyond most type theoretic proof systems.

CoCasl [17] and CCSL [22] are tools that can generate proof obligations for theorem provers from coalgebraic specifications. The work on equational theorem proving with bisimilarity is an active topic of research [6, 9, 11, 12]. In these work the principal tool in coinductive proving is via bisimulation building, either explicitly or implicitly via a *circular coinduction* [9]. In [6] various heuristics for bisimulation finding are studied. In [11] several tactics for interactive and automatic bisimulation building is implemented in Isabelle/HOL and are used to derive bisimilarities for translated specifications from CoCasl. In [12] the CIRC tool is introduced which is based on hidden algebra and uses a partial decision procedure for proving bisimilarities via implicit construction of bisimulations.

These are relevant to our work and in future work we plan to work on integration of the above tools within our framework. In contrast, here our focus is not on heuristics for bisimulation finding. Rather we focus on interactive equational reasoning, as well as some automation for deriving *co-fixed point* equations from coalgebraic schemes. The other difference is that the above systems deal with ordinary bisimulation while we consider other proof principles e.g. λ-coinduction arising from more complex coalgebraic schemes. A final difference is that our work is done completely inside a theorem prover: both our corecursive functions as well as our coinductive proof principles are not simply assumed equational specifications. They correspond to concrete proof objects and (especially in *Coq*) they cab be extracted to executable code.

2 Mechanising Category Theory and Intensionality

First we should clarify the issue with intensionality as it affects our treatment of coinduction since we develop our work in the intensional setting of *Coq*. We do not aim to argue for why to use intensional over extensional systems or vice versa. We only express the practical implications.

In intensional type theory the two objects being provably equal does not entail that they are convertible. This restriction is necessary for the decidability of type checking and although it is not a theoretical obstacle for programming, it can be practically inconvenient. In particular, formalising category theory is susceptible to this inconvenience, as proving the uniqueness of arrows in universal properties of limits adheres to extensional properties of functions. On the other hand in an extensional setting the axiom of *functional extensionality* holds:

$$\forall XY \forall f g : X \to Y,\ (\forall z, f(z) = g(z)) \to f = g\ . \qquad (\textsc{Ext})$$

This allows one to properly capture the uniqueness of arrows. It is mainly the absence of this axiom that affects our treatment. First of all we should work with two equalities, one (denoting by =) is the intensional equality[2], and the other would be a defined extensional equality which in our case will be the bisimulation equivalence (denoted by $\cong$). Subsequently there will also be two kind of *uniqueness* of arrows for each type of equality. The intensional uniqueness arises only in basic cases where terms are indeed convertible while extensional uniqueness is always built on top of non-trivial proofs.

The main workaround for working extensionally in intensional type theory is is to use setoids and work modulo our extensionally defined equality. We partly follow this. This means that some of the diagrams, namely those whose commutativity depend on the extensional uniqueness, will be expressed using this extensional equality (the predicate $l_{Rel(F)}$ in Section 4).

Note that Ext can be lifted from functions to functors and its absence means that we can only work with functors that satisfy a similar condition (See Section 4. This is perhaps the most dramatic effect that working with intensional equality has on our work as it excludes some important functors. In fact in [18] we showed that exponentiation, a functor often used in modelling processes, does not satisfy functorial extensionality without assuming additional axioms.

Nevertheless we see certain positive aspects in our approach.[3] Most important advantage is that in our intensional setting our bisimilarity will coincide with the standard coinductive definition of bisimilarity [8] that is defined using coinductive types. This means the *Haskell*-like specification that we derive via coalgebraic schemes will mimic those that would be accepted in *Coq* if their productivity was detected by the guardedness checker of *Coq*. Especially since standard polynomial functors fall within our framework and hence the important cases of streams and infinite trees are covered by our treatment. Finally our work can serve as the starting point for an extensional treatment of coalgebras because we have shown that a relatively complex coalgebraic scheme is in principle implementable. In fact in future work we plan to transfer our work into a fully extensional setting using setoid functors and setoid coalgebras. It is our conviction that the high level aspects of the work (e.g. uniqueness and existence in the coalgebraic scheme) are straightforward to translate *and* they constitute the non-trivial and interesting parts of our development.

3 Coalgebras and Coinduction Proof Strategies

While coinduction can be used for proving a variety of properties, its most common application is in proving the behavioural equivalence of infinite objects by proving *bisimilarity*.

The theory of coalgebras [23] provides a framework for dealing with bisimilarity and coinduction. Recall that given a functor F an F-coalgebra consists

[2] This includes two different type of equalities which are identical in the empty context. But we do not delve into this issue here.

[3] We disregard the benefits that only affect the formalisation effort itself, e.g. that proving naturality of transformations is simply conversion of intensional equality and hence trivial.

of (X, α_X), i.e., a set[4] X, called the *state set* together with a *transition map* $\alpha_X : X \longrightarrow F(X)$. Given two coalgebras (X, α_x) and (Y, α_Y), a binary relation $R \subseteq X \times Y$ is a bisimulation between X and Y if there is a map $\gamma\colon R \longrightarrow F(R)$ making both squares in the left hand side diagram below commute (by π_i we denote the i-th projection of a tuple):

$$\begin{tikzcd}
X \arrow[d, "\alpha_X"'] & R \arrow[l, "\pi_1"'] \arrow[r, "\pi_2"] \arrow[d, dotted, "\gamma"] & Y \arrow[d, "\alpha_Y"] \\
F(X) & F(R) \arrow[l, "F\pi_1"] \arrow[r, "F\pi_2"'] & F(Y)
\end{tikzcd}
\qquad\qquad
\begin{tikzcd}
X \arrow[r, dotted, "f"] \arrow[d, "\alpha_X"'] & \mathit{\Omega} \arrow[d, "\alpha_{\mathit{\Omega}}"] \\
F(X) \arrow[r, dotted, "Ff"'] & F(\mathit{\Omega})
\end{tikzcd}
\tag{3.1}$$

The following fundamental result, taken from [23], establishes an important property of bisimulation.

Theorem 1. *i) For any two coalgebras bisimulations form a lattice, with a maximal element.*
ii) Maximal bisimulation on a single coalgebra is an equivalence relation.

Property (i) forms the basis of the coinduction proof principle: given two coalgebras proving that two elements are in the maximal bisimulation is tantamount to finding any bisimulation relation between the two coalgebras. However, in practice we are interested in bisimulation between *final coalgebras* because they model infinite objects. Recall that a coalgebra $(\mathit{\Omega}, \alpha_{\mathit{\Omega}})$ is final if for any other coalgebra (X, α_X) there exists a unique f making the right hand side diagram in (3.1) commute.

The equivalence relation that is the maximal bisimulation on $(\mathit{\Omega}, \alpha_{\mathit{\Omega}})$ is called the *bisimilarity* relation. In combination with part (i) of the above theorem one can obtain the *coinduction proof principle*: two elements of a final coalgebra are bisimilar if they are related by a bisimulation.

The advantage of this principle lies in the fact that in the standard setting (i.e., with extensional equality), one can show that bisimilarity is the same as equality [23]. So in order to prove that two infinite objects, e.g. processes or streams, are equal, one can try to search for a bisimulation relation between them. Of course there is no complete method for constructing bisimulations [21], so an effective search is impossible even if the equality holds. This means that any theorem prover dealing with bisimilarities should be equipped with alternative tools. The most obvious alternative technique would be to exploit the equational reasoning since bisimilarity is an equivalence. While in standard (extensional) setting this step is vacuously straightforward, in the intensional type theory it means that a *congruence* property should be proven for each function: given a function $g\colon \mathit{\Omega} \longrightarrow \mathit{\Omega}$ we should prove that $x \cong y \implies g(x) \cong g(y)$. In Theorem 2.ii we prove this once for each g that is defined using the coiteration scheme (i.e., using the finality in the above diagram). Such generic congruence proofs are one of the advantages of our library.

[4] We work in category **Sets**, but the notions are applicable in more general settings.

Finally, in the coinductive type theory bisimilarity itself is a coinductive type [8, 10]. To prove $x \cong y$ one has to define a function on coinductive types (e.g. in *Coq* it can be done using cofix construct). An interactive construction of this function corresponds closely to CIRC's circular coinduction. Such interactive 'coinductive' proofs can be very intuitive in the way the user gradually discovers a circularity pattern. However since all functions should be productive, such proofs too are subject to productivity tests. This is because they are function themselves. This means many intuitive *and* valid circular proofs are rejected.

So we mentioned at least three techniques for interactively proving a bisimilarity: (1) explicit building of bisimulation, (2) using the equational reasoning and congruence and (3) using interactive circular coinduction. In the remainder of the paper we present our library that enables the user in combining all these three proof methods.

4 Modular Library of Coalgebras

In this section we present the formalised library of a theory of bisimulation and weakly final coalgebras. Our formalisation uses some facets of the *Coq* system: module system, record types, and Set-valued and Prop-valued dependent sums. In other systems, most of these facilities exist but possibly in a different form (or based on different foundations). The only peculiarity is the distinction between Set and Prop but fortunately this distinction is not *essential*: it is only there to satisfy *Coq*'s fine-grained constraint on case analysis. Prop can be replaced by Set everywhere and the formalisation is still valid (and much simpler).

The library consists of a number of module types and theories. Each module type has a number of parameters and axioms. On top of each module type there will be a corresponding theory consisting of lemmas that hold for all instances of that module type. An instantiation of a module type consists of concrete values for parameters and proofs for each axioms. Once the parameters and axioms are provided the theory of the module type is available for that instantiation.

At the basis of our library lies the type of extensional Set functors. This module type consists of two operations $F\colon \mathsf{Set} \longrightarrow \mathsf{Set}$ (on objects) and $l_F\colon (X{\to}Y){\to}F(X){\to}F(Y)$ (on arrows)[5] satisfying the standard functorial properties plus the following extensionality axiom.

$$\forall XY(fg{:}X{\to}Y)x, (\forall z, f(z) = g(z)){\to}Ff(x) = Fg(x) \ .$$

Such functors are called *extensional* functors [13]. It can be easily seen that the constant functor and the identity functor and the sum, product, composition and iteration of extensional functors are extensional [13, 18].

Given an extensional Set-functor F, an F-coalgebra S is simply a record containing the set of states $S.st$ and the transition map $S.tr\colon S.st \longrightarrow F(S.st)$.

For defining bisimulations as a relation we use dependent types for subsets. We use different notation for a Prop-valued relation $R\colon X{\to}Y{\to}\mathsf{Prop}$ and the

[5] From now on, we write Ff for $l_F(f)$.

set of pairs in $\{(x, y) \in X \times Y | Rxy\}$. By $\{\overline{\exists} x \colon X, \phi(x)\}$ we denote the set of elements of X satisfying $\phi : X \rightarrow \mathsf{Prop}$. Given a relation R we write $\{\overline{\exists}(R)\}$ as a shorthand for $\{\overline{\exists} u \colon X \times Y,\ R\ \ \pi_1(u)\ \ \pi_2(u)\}$. Note that an element of $\{\overline{\exists}(R)\}$ is a 3-tuple consisting a pair from $X \times Y$ and a proof that they satisfy R.

For coalgebras S_0, S_1 a binary relation $R \colon S_0 \longrightarrow S_1 \rightarrow \mathsf{Prop}$ is *bisimulation* if there exists $\gamma \colon \{\overline{\exists}(R)\} \rightarrow F\{\overline{\exists}(R)\}$ making the diagram in (3.1) with R replaced by $\{\overline{\exists}(R)\}$ commute. We can have a similar definition for Set-valued relations leading to the definition of *Set-valued bisimulation.* Remarkably, we define a bisimulation to be *maximal* if it contains every Set-valued bisimulation. This is necessary in the proof of the transitivity of the maximal bisimulation. Our module type of bisimulations takes as parameter the maximal bisimulation between any two given coalgebra. Strictly speaking this is redundant since by Theorem 1. i the maximal bisimulation can be built abstractly for any two coalgebras. However, we prefer to parametrise the maximum bisimulations because for each concrete Set-functor such a maximum bisimulation can be defined using coinductive types and hence it will enable the user to use the interactive circular coinduction.

Any instantiation of the theory of bisimulation should prove few results about the provided parameters: first of all that the provided maximal bisimulation is indeed bisimulation and maximal. Second, we require a proof that the extensional Set-functor F have preserves weak pullbacks.[6] Again this is necessary for proving the transitivity of maximal bisimulation.

Within this theory, we can generically prove some properties of bisimulation. Notably the counterpart of Theorem 1.ii for extensional Set-functors is proven. More properties can be found in [18]. Here we present some additional properties that were added recently for equational reasoning. Recall that for F-coalgebras S_0 and S_1, $f \colon S_0.st \longrightarrow S_1.st$ is a *coalgebra homomorphism* if $\forall x,\ \ Ff(S_0.tr(x)) = S_1.tr(f(x))$.

Proposition 1. *i) Graph of a homomorphism is a bisimulation.*

ii) If $f \colon S_0.st \longrightarrow S_1.st$ is a homomorphism and R a bisimulation on S_1 then $f^{-1}(R)$ is a bisimulation on S_0.

iii) If f, g in $S_0.st \xleftarrow{f} X.st \xrightarrow{g} S_1.st$ are coalgebra homomorphisms and $Rs_0s_1 := \{\overline{\exists} x \colon X.st,\ \ f(x) = s_0 \wedge g(x) = s_1\}$, then R is a Set-valued bisimulation.

The type of weakly final coalgebras takes as parameter a weak pullback preserving extensional Set-functors F for which a bisimulation theory exists (so the maximal bisimulation is needed too). A *weakly final coalgebra* for F is an F-coalgebra $\varOmega$ such that for each F-coalgebra S we have

$$\{\overline{\exists} \mathrm{unfld}_S \colon S.st \rightarrow \varOmega.st, \forall s,\ \ \varOmega.tr(\mathrm{unfld}_S(s)) = F\mathrm{unfld}_S(S.tr(s))\}.$$

From now on we use the notation $\cong$ for maximal bisimulation on the weakly final coalgebra. *Weak* finality is due to the fact that this condition does not speak

[6] In fact the actual condition that we require is slightly weaker [18].

about the uniqueness of unfld_S. For concrete cases the following weak form of uniqueness is provable and hence can be assumed as axiom for this type.

$$\begin{aligned}\forall S \forall fg\colon S.st{\to}\mathit{\Omega}.st, (\forall s_0, \mathit{\Omega}.tr(f(s_0)) = Ff(S.tr(s_0))){\to} \\ (\forall s_0, \mathit{\Omega}.tr(g(s_0)) = Fg(S.tr(s_0))){\to}\forall s, f(s) \cong g(s) \ . \end{aligned} \tag{4.1}$$

Our final axiom expresses the commutativity of diagram (3.1) modulo bisimilarity. First we define the lifting of a relation to the image of F:

$$l_{Rel(F)}(S_1, S_2, R, z_x, z_y) := \exists xy, Rxy \wedge z_x{=}S_1.tr(x) \wedge z_y{=}S_2.tr(y) \ .$$

Using this our final axiom reads:

$$\forall X \forall fg\colon X{\to}\mathit{\Omega}.st \forall y, \ \big(\forall x, f(x) \cong g(x)\big) \ \to \ l_{Rel(F)}\big(\mathit{\Omega}, \mathit{\Omega}, \cong, Ff(y), Fg(y)\big) \ .$$

This concludes the definition of the module type for weakly final coalgebras. From here on the rest of the library includes theory development. Basically the remainder can be considered the theory of weakly final coalgebra but we divide in three different sub theories: general theory of weakly final coalgebras (given below), the coalgebraic schemes (Section 5) and a version of λ-coinduction (Section 7).

In the general theory of weakly final coalgebras we can prove some useful tools for interactive coinductive reasoning namely the coinduction proof principle, the congruence of unfld_S and a left inverse (up to bisimilarity) for $\mathit{\Omega}.tr$.

Theorem 2. *Let $\mathit{\Omega}$ be a weakly final F-coalgebra for F an extensional weak pullback preserving* Set*-functor. Then*

i) If R is a bisimulation relation on $\mathit{\Omega}.st$ then $\forall x, y\colon \mathit{\Omega}.st, \ Rxy \implies x \cong y$.
ii) For any F-coalgebra S we have $\forall x, y\colon S.st, \ x \sim_S y \Leftrightarrow \mathrm{unfld}_S(x) \cong \mathrm{unfld}_S(y)$ where $\sim_S$ denotes the maximal bisimulation on S.
iii) For the coalgebra $S_c := (F(\mathit{\Omega}.st), F\mathit{\Omega}.tr)$ we have $\forall x, \mathrm{unfld}_{S_c}(\mathit{\Omega}.tr(x)) \cong x$.

The second part, whose proof needs Theorem 1.i–ii, is a fundamental property; it implies that if one considers $(\mathit{\Omega}, \cong)$ as a setoid, then $f := \mathrm{unfld}_F$ is a setoid morphism. This means that if our goal involves bisimilarity between applications of f we can *rewrite* the bisimilarities between arguments of f into the goal. In other words bisimilarity becomes rewritable. This improves the equational reasoning, especially in a context where several such arrows are composed. In *Coq* we use the *setoid rewriting* library to declare the maximal bisimulation as a parametric setoid relation and unfld_F as a parametric morphisms [5, § 24].

5 Coalgebraic Schemes

A coalgebraic definition scheme consist of syntactic constraints that will lead to a valid definition (i.e., existence and uniqueness) of a function whose codomain is a final coalgebra. There is no restriction on the domain: schemes can also be used for defining constants as functions from the unit set. The most basic scheme is the *coiteration scheme* given by the universal property in right hand side diagram (3.1). Given a set X this scheme requires a transition map $\alpha_X : X \longrightarrow F(X)$ to define a function from X to $\mathit{\Omega.st}$.

The λ-coiteration scheme by Bartels [2] is a very expressive scheme in that it subsumes many other schemes including coiteration. In [18] we showed how to implement the λ-coiteration scheme in our library. This enabled us to 'define' functions whose productivity is difficult to capture by the basic guardedness tests. Consider this specification for the stream of Fibonacci numbers:

$$\texttt{fibs} := 0 :: \; \oplus_3 (1, \texttt{fibs}, \texttt{fibs}) \ , \tag{5.1}$$

where $\oplus_3$ is a ternary operation defined as:

$$\oplus_3(x_0,\ x :: xs,\ y :: ys) := x_0 + y :: \; \oplus_3 (x, xs, ys) \ .$$

There is a unique stream satisfying the specification in (5.1), namely the Fibonacci stream. Ideally (5.1) should be accepted as a definition. However the guardedness test of *Coq* refuses this definition. Fortunately, as we show below, the shape of this definition fits the λ-coiteration scheme. In fact our scheme provides the necessary existence theorem. Of course the absence of the intensional uniqueness in our framework means that the (nullary) function we obtain from our scheme is bisimilar to the stream of Fibonacci but for the purpose of computation and observation of behaviour this bisimilarity is all one needs.

Here we present the adaptation of this scheme to our setting (cf. [18, Theorem 2]). From here on we assume B to be a weak pullback preserving extensional Set-functor, T to be an extensional Set-functor and $\mathit{\Omega}$ to be the weakly final B-coalgebra. Let $\mathit{\Lambda} \colon TB \Longrightarrow BT$ be a natural transformation. Note that with the map $\mathit{\Lambda}_{\mathit{\Omega.st}} \circ T(\mathit{\Omega.tr}) \colon T(\mathit{\Omega.st}) \longrightarrow BT(\mathit{\Omega.st})$ one can form a B-coalgebra $S_0 := (T(\mathit{\Omega.st}), \mathit{\Lambda}_{\mathit{\Omega.st}} \circ T(\mathit{\Omega.tr}))$. Let $\beta := \mathrm{unfld}_{S_0}$ (so $\beta \colon T(\mathit{\Omega.st}) \longrightarrow \mathit{\Omega.st}$).

Theorem 3. *With $\mathit{\Lambda}, \beta$ as above assume a map $g \colon X \longrightarrow BT(X)$ is given (for an arbitrary set X). Then there exists an arrow f making the diagram below commute up to bisimilarity.*

$$\begin{tikzcd}
X \arrow[rr, dotted, "f"] \arrow[d, "g"'] & & \mathit{\Omega.st} \arrow[d, "\mathit{\Omega.tr}"] \\
BT(X) \arrow[r, dotted, "BT(f)"'] & BT(\mathit{\Omega.st}) \arrow[r, "B(\beta)"'] & B(\mathit{\Omega.st})
\end{tikzcd}$$

That is to say

$$l_{Rel(B)}\Big(\mathit{\Omega}, \mathit{\Omega}, \cong, \mathit{\Omega.tr}\big(f(x)\big), B\beta \circ BTf \circ g(x)\Big) \ .$$

The map f given by the above theorem is called the λ-*coiterative arrow induced by g*. In fact we can show the extensional uniqueness of f (see Section 7) but for the material in this section the existence is enough.

Presuming that we want to define functions into $\Omega.st$, this scheme needs as parameters T, Λ and g. Other (simpler) schemes can be obtained from this scheme by taking specific functors for T and concrete natural transformations for Λ. For example, λ-coiteration schemes subsumes coiteration and primitive corecursion.

5.1 Coiteration scheme from λ-coiteration

Taking the identity functor $T(X) := X$, and the identity natural transformation $\Lambda_X(x) := x$ we have the ordinary coiteration scheme. In this case $\beta\colon \Omega.st \longrightarrow \Omega.st$ will be the identity as a coalgebra homomorphism, i.e., the function `id` satisfying following specification:

$$\mathtt{id}(x :: xs) := x :: \mathtt{id}(xs) \enspace .$$

In this case providing $g\colon X \longrightarrow B(X)$ is the same as providing a coalgebraic transition structure on X, hence the above diagram will be equivalent to (3.1). However the diagrams would not become intensionally equal. There will be an occurrence of `id` in the λ-coiterative version.

We demonstrate this by an example. For the coalgebra of streams of natural numbers $(\mathbb{N}^\omega, \langle \mathtt{hd}, \mathtt{tl}\rangle)$, taking $g(\sigma) := \langle \mathtt{hd}(\sigma), \mathtt{tl}(\sigma)\rangle$ and applying the above theorem results in the λ-coiterative definition of the function $\mathtt{x2}_1$ that doubles each element of a stream. Applying the coiteration scheme directly (via diagram (3.1)) and using the coalgebra $S := (\mathbb{N}^\omega, g)$ we get a function $\mathtt{x2}_1 := \mathrm{unfld}_S$. The two functions satisfy the following.

$$\begin{aligned} \mathtt{x2}_1(x :: xs) &= x :: \mathtt{x2}_1(xs) \enspace , \\ \mathtt{x2}_2(x :: xs) &\cong x :: \mathtt{id}(\mathtt{x2}_2(xs)) \enspace . \end{aligned}$$

The first one follows from the weak finality hence it is an intensional equality. The second is obtained from Theorem 3. We can prove that both $\mathtt{x2}_i$ satisfy the following bisimilarity.

$$\mathtt{x2}_i(x :: xs) \cong x :: \mathtt{x2}_i(xs) \enspace .$$

5.2 Primitive Corecursion scheme from λ-coiteration

Primitive corecursion scheme [7] is useful in some situations where ordinary coiteration does not work. An example, taken from [2], is the function `insert` that satisfies this specification:

$$\mathtt{insert}\quad n \quad (x :: xs) := \begin{cases} x :: \mathtt{insert}\quad n \quad xs & \text{if } x \leq n, \\ n :: x :: xs & \text{otherwise} \enspace . \end{cases} \tag{5.2}$$

This specification does not fall under ordinary coiteration. The primitive corecursion scheme takes as parameter a map $g\colon X \longrightarrow B(X + \Omega.st)$ and ensures the existence of the *primitive corecursive* map f making the diagram below commute.

$$\begin{tikzcd} X \arrow[r, dotted, "f"] \arrow[d, "g"'] & \Omega.st \arrow[d, "\Omega.tr"] \\ B(X+\Omega.st) \arrow[r, dotted, "{B[f,id]}"'] & B(\Omega.st) \end{tikzcd} \tag{5.3}$$

(Here $[f, id]\colon X + \Omega.st \longrightarrow \Omega.st$ is the cotupling map given by the property of coproduct). Let $\imath_l, \imath_r$ be the maps $X \xrightarrow{\imath_l} X{+}\Omega.st \xleftarrow{\imath_r} \Omega.st$. In [2] it is pointed out that by taking $T(X) := X + \Omega.st$ and $\Lambda_X := [B\imath_l, B\imath_2 \circ \Omega.tr]$ this scheme is a special case of the λ-coiteration scheme. In our framework we can prove that

$$\forall x\colon X + \Omega.st,\quad \beta(x) \cong [id, id](x) \ .$$

From this, Theorem 2.iii and Theorem 3 we obtain the following.

Theorem 4. *Given a map $g\colon X \longrightarrow B(X + \Omega.st)$ there exists a map f making the diagram (5.3) commute up to bisimilarity. I.e.,*

$$l_{Rel(B)}\Big(\Omega, \Omega, \cong,\ \Omega.tr\big(f(x)\big),\ B[f, id] \circ g(x)\Big) \ .$$

This enables us to define the `insert` function as a primitive corecursive map by taking

$$g\langle n, \sigma\rangle := \begin{cases} \langle \mathtt{hd}(\sigma), \imath_l\langle n, \mathtt{tl}(\sigma)\rangle\rangle & \text{if } \mathtt{hd}(\sigma) \leq n, \\ \langle n, \imath_r(\sigma)\rangle & \text{otherwise} \ . \end{cases}$$

Subsequently, Theorem 4 entails that `insert` defined using primitive corecursion with the above g satisfies (5.2) up to bisimilarity.

5.3 Beyond coiteration and primitive corecursion

Primitive corecursion, while enhancing upon the ordinary coiteration, still has a limited format. For example the specification (5.1) does not fall under either of these schemes, but fits the λ-coiteration scheme. We define `fibs` as the λ-coiterative arrow from $\mathbf{1} = \{*\}$ into $\Omega.st$. For this we need to apply Theorem 3 with the following parameters.

$$\begin{aligned} &T(X) := \mathbb{N} \times X \times X \ , \\ &\Lambda_X(x) := \langle \pi_1(x) + \pi_4(x), \langle \pi_2(x), \pi_3(x), \pi_5(x)\rangle\rangle \ , \\ &g(x) := \langle 0, \langle 1, *, *\rangle\rangle \ . \end{aligned}$$

Note that here we do not need to define $\oplus_3$ explicitly. This is because $\oplus_3 = \beta$, and β is calculated automatically from T and Λ. More examples of definitions using λ-coiteration-scheme in our framework can be found in [18].

6 Equational Reasoning

In this section we explain some practical theorem proving aspects of our library in proving bisimilarities. First of all, note that we distinguish between two kind of goals. One is the bisimilarity that captures the specification of an infinite object. This can be seen as the equation 'defining' an infinite object but stated in terms of bisimilarity. We call these the *co-fixed point equations.* I.e., if one was to declare the co-fixed point equation e.g. in *Haskell* one would get a well-defined infinite object. As an example

$$\texttt{fibs} \;\cong\; 0{::}\; \oplus_3 (1, \texttt{fibs}, \texttt{fibs}) \tag{6.1}$$

is the co-fixed point equation corresponding to the specification in (5.1). The characteristic property of a co-fixed point equation is that its left hand side contains a single function f (possibly applied to curried arguments) and it always has a unique solution in f:

$$f \quad x_0 \quad \cdots x_k \cong \phi(f, x_0, \cdots, x_k) \ . \tag{6.2}$$

Any other type of bisimilarity is usually about the algebraic properties of infinite objects, therefore we call these *algebraic bisimilarities.* These need not have a unique solution. Below we will show some examples of the the two type of goals. Although in most cases the distinction between them is clear, we emphasise that this distinction is a practical matter and it depends on the perspective of the user.

6.1 Co-fixed point equations

In our framework the co-fixed point equations are proven with very little interaction from the user. The prerequisite is that the equation for a function f falls under the syntactic shape of the λ-coiteration scheme. The procedure then would be to first use 3 (or 4) to obtain f as a λ-coiterative arrow. Second, we derive the co-fixed point equation from the commutativity of the diagram. But, the second step cannot be done generically for all functors. Rather, for each concrete instance of functor B we should prove a theorem deriving a co-fixed point equation from the l_{RelB} predicate that states commutativity up to bisimilarity.

For any B we can prove that l_{RelB} implies the following which is a maximal bisimulation on $B(\Omega.st)$.

$$\Omega.tr(f(x)) \quad \sim_{B(\Omega.st)} \quad B(\beta \circ Tf) \circ g(x) \ . \tag{6.3}$$

The co-fixed point equation in fact applies the left inverse of $\Omega.tr$, the so called coconstructor (cf. Theorem 2.iii) to both sides of (6.3). Since the coconstructor is a setoid morphism by Theorem 1.ii, this will entail

$$f(x) \cong \mathrm{unfld}_{S_c}\big(B(\beta \circ Tf) \circ g(x)\big) \ .$$

But in order to have a more simplified version in the right hand side, we need the information about the functor B. For example, if B is the stream functor to get the right hand side in terms of $::$ (which is the usual functional programming style) we have to have an implementation of the theory of weakly final coalgebra of streams e.g. using coinductive types. As a side advantage, we will also be able to use the implementation-specific pattern matching for the arguments in the left hand side (e.g. having $x :: xs$ as an argument). Note that we only have to know the shape of functor B, the remaining parameters namely T, Λ and g can remain abstract. For the stream coalgebra we have the following in our framework.

$$\forall x,\ \ f(x) \ \ \cong \ \ \pi_1\big(B(\beta \circ Tf) \circ g(x)\big) \ :: \ \pi_2\big(B(\beta \circ Tf) \circ g(x)\big) \ .$$

Although such lemma has to be derived for each B, still afterwards one such derivation can be used for many different specifications. For example for all the stream examples in this paper and in [18] we need one single lemma. As the last step, each concrete co-fixed point equation will be derived by replacing the values of Λ, T and g in the above bisimilarity. This latter step is the only place where some interaction from user is needed. In *Coq* even this step can be done semi automatically, using proof scripts that are almost identical (see [19] where (6.1), (5.2) and more examples are derived).

6.2 Algebraic bisimilarities

We have defined `fib` as a λ-coiterative arrow and we have derived its co-fixed point equation. Now we want to prove that this stream satisfies *another* specification for the Fibonacci stream, namely the one used in [4]. I.e., we want to prove:

$$\mathtt{fibs} \cong 0 :: 1 :: \mathtt{tl}(\mathtt{fibs}) \oplus \mathtt{fibs} \ . \tag{6.4}$$

Here $\oplus$ is the binary pointwise addition on streams and can be defined using ordinary coiteration. Its co-fixed point equation reads as

$$xs \oplus ys \ \ \cong \ \ \mathtt{hd}(xs) + \mathtt{hd}(ys) \ \ :: \ \ \big(\, \mathtt{tl}(xs) \ \oplus \ \mathtt{tl}(ys)\, \big) \ .$$

Our goal (6.4) can be transformed, using the equational theory, to two sub goals:

$$\begin{aligned} &\oplus_3\,(x, xs, ys) \ \cong (x :: xs) \oplus ys \ ; \\ &xs \oplus ys \ \cong \ ys \oplus xs \ . \end{aligned} \tag{6.5}$$

Both of these can be proven in two different ways. One method is by using 2.i and explicitly providing a bisimulation. Of course 'guessing' the bisimulation is the duty of the user; however in this case there are procedures that can help in finding a suitable bisimulation by examining the shape of the goal and using it as a starting point [6]. A direction for future work would be to integrate those within the existing interactive theorem provers. The second technique would be to use the implementation of $\cong$. Recall that bisimilarity was a parameter in our modular theory and for each implementation had to come up with an instance. In *Coq* for example, where we can instantiate weakly final coalgebras by coinductive types, we can use the fact that $\cong$ is a coinductive type itself. Hence a proof of the above becomes constructing an inhabitant this type. As mentioned earlier this process is in fact an interactive version of the circular coinduction. In this particular case, since both specifications ($\oplus$ and $\oplus_3$) are obtained using ordinary coiteration *Coq* will not have any problem with detecting productivity of our proof (likewise, CIRC will readily find an automatic coinductive proof). But in general this method is less robust than the explicit bisimulation method. This is is partly due to productivity checker and partly because of its reliability on a particular implementation of the theory.

Our tools for dealing with this kind of goal are the three techniques that we mentioned in Section 3. Mainly we use the equational theory of bisimilarity, the facilities for dealing with setoids (rewriting congruences and morphisms) and the generic lemmas that our library provides. These generic lemmas provide some automated steps but most of the reasoning relies on the interaction with user. When the user proves more and more lemmas they may be used to fabricate an algebraic structure and use more advanced automation tools. For example, based on (6.5) and a couple of more lemmas on associativity and neutrality the user can prove that the streams of integers form an abelian group with $\oplus$. Subsequently decision strategies for dealing with word problems on groups can be used to prove more complex goals, independent of coinduction and bisimulations.

7 λ-Coinduction

The coinduction proof principle reduces the bisimilarity goals to finding bisimulations, which is still not an easy problem [21]. In case of functions defined using λ-coiteration scheme one can use an alternative proof principle that reduces the bisimilarity goals to finding another type of relations called λ-bisimulations. The formal definition of λ-bisimulation and λ-bialgebra can be found in Appendix A. Using those we could develop a theory of λ-bisimulation following [2]. Most notably we formalised the following result which cumulatively entail the λ-coinduction principle. A crucial step in the proof relies on Proposition 1.iii.

Theorem 5. *Let R be a λ-bisimulation between λ-bialgebras (X, β_X, α_X) and (Y, β_Y, α_Y). Then*

i) There is a Set-valued bisimulation between coalgebras (X, α_X) and (Y, α_Y).
ii) If Rxy then $x \sim_{XY} y$; where $\sim_{XY}$ is the maximal bisimulation between (X, α_X) and (Y, α_Y).
iii) $(\Omega.st, \beta, \Omega.tr)$, with β as in Section 5, is a λ-bialgebra.
iv) Let R_Ω be a λ-bisimulation on $(\Omega.st, \beta, \Omega.tr)$. Then $R_\Omega xy$ implies $x \cong y$.

The advantage of λ-coinduction is that in some cases where the suitable bisimulations have a complicated shape, the λ-bisimulation relation has a simple intuitive shape. We demonstrate this by an example taken from [2]. Let shuffle product, denoted by $\otimes$, be the function on streams of natural numbers satisfying the following co-fixed point equations.

$$(x :: xs) \otimes (y :: ys) \quad \cong \quad x \cdot y \ :: \ (xs \otimes (y :: ys)) \oplus ((x :: xs) \otimes ys) \ .$$

This operation can be defined using λ-coiteration [2, 18]. Assume we want to prove that the shuffle product is commutative. It is very difficult to prove this by finding a bisimulation.[7] On the other hand the relation $R := \exists x \exists y,\ \sigma = x \otimes y \wedge \tau = y \otimes x$ is *extensionally* a λ-bisimulation.

Since our definition of λ-bisimulation and bisimulation is based on intensional commutativity, in order to prove that R is a bisimulation we needed to assume an axiom stating that on streams bisimilarity implies equality: $x \cong y \implies x = y$. It is easy to prove that this axiom is equivalent to the axiom that states the intensional uniqueness of the arrow into the final coalgebra (and hence turns weak finality into finality). In a fully extensional setting (even within *Coq*, with setoids) this axiom is validated automatically.

We conclude by stating that λ-coinduction enables us to prove the uniqueness of the λ-coiterative arrow up to the extensional equality (similar to (4.1)). This was missing in [18] where we only showed the existence.

8 Conclusions and Further Work

We presented a modular library that axiomatises weakly final coalgebras and bisimulation. As a theory we derived some coalgebraic schemes and an associated coinduction principle. These provide tools for reasoning about bisimilarities. Our framework has three layers: (1) generic results for all extensional functors, (2) results for each concrete functor, and (3) the specific functions to be formalised by the user. The first is fully provided by our library. The second is dependent on implementation. In [18] we have given an implementation for streams and natural numbers using coinductive types but the user is free to use other implementation. For example, instead of coinductive type of streams one could use $N \to N$ to represent streams of natural numbers as a function type.

Our work is done in *Coq*, but in principle it can transferred to other logical frameworks. Doing so may actually relax some of the constraints that the intensional type theory has posed on us, while the essential coalgebraic parts (the schemes and λ-coinduction) wily remain intact.

In future we plan to construct a setoid version of our framework (still within the intensional type theory) and work on the integration of bisimulation search techniques in it.

References

[1] F. Andersen and K. D. Petersen. Recursive boolean functions in HOL. In M. Archer, J. J. Joyce, K. N. Levitt, and P. J. Windley, editors, *Proc. of the*

[7] In fact attempts to prove this in *Coq* by using coinductive types fails due to guardedness. It seems that CIRC too, at least when tried with the standard stack limit and number of steps, is unable to find this bisimulation automatically.

1991 Int. Workshop on the HOL theorem proving system and its Applications, pages 367–377. IEEE, 1992.

[2] F. Bartels. *On Generalised Coinduction and Probabilistic Specification Formats: Distributive Laws in Coalgebraic Modelling.* PhD thesis, Vrije Universiteit Amsterdam, 2004.

[3] S. Berardi, F. Damiani, and U. de'Liguoro, editors. *Proceedings of TYPES 2008 Workshop*, volume 5497 of *LNCS*. Springer, 2009.

[4] Y. Bertot and E. Komendantskaya. Using structural recursion for corecursion. In Berardi et al. [3], pages 220–236.

[5] The Coq Development Team. *Reference Manual, Version 8.2.* INRIA, June 2008. `http://coq.inria.fr/V8.2/doc/html/refman/`, [cited 18 Jan. 2010].

[6] L. Dennis. *Proof Planning Coinduction.* PhD thesis, University of Edinburgh, 1998.

[7] H. Geuvers. Inductive and coinductive types with iteration and recursion. In B. Nordström, K. Petersson, and G. Plotkin, editors, *Informal Proc. of Types'92*, pages 193–217. Chalmers Univ. of Technology, 1992.

[8] E. Giménez. *Un Calcul de Constructions Infinies et son Application a la Verification des Systemes Communicants.* PhD thesis 96-11, ENS Lyon, Dec. 1996.

[9] J. Goguen, K. Lin, and G. Roşu. Circular coinductive rewriting. In *Proc. of ASE'00*, pages 123–132. IEEE, 2000.

[10] P. Hancock and A. Setzer. Interactive programs and weakly final coalgebras in dependent type theory. In L. Crosilla and P. Schuster, editors, *From Sets and Types to Topology and Analysis. Towards Practicable Foundations for Constructive Mathematics*, volume 48 of *Oxford Logic Guides*, pages 115–134. Oxford University Press, 2005.

[11] D. Hausmann, T. Mossakowski, and L. Schröder. Iterative circular coinduction for CoCasl in Isabelle/HOL. In M. Cerioli, editor, *Proc. of FASE 2005*, volume 3442 of *LNCS*, pages 341–356. Springer, 2005.

[12] D. Lucanu and G. Roşu. CIRC : A circular coinductive prover. In T. Mossakowski, U. Montanari, and M. Haveraaen, editors, *Proc. of CALCO 2007*, volume 4624 of *LNCS*, pages 372–378. Springer, 2007.

[13] R. Matthes. An induction principle for nested datatypes in intensional type theory. *J. Funct. Programming*, 19(3–4):439–468, 2009.

[14] C. McBride. *Dependently Typed Programs and their Proofs.* PhD thesis, University of Edinburgh, 1999.

[15] C. McBride. Let's see how things unfold: Reconciling the infinite with the intensional (extended abstract). In A. Kurz, M. Lenisa, and A. Tarlecki, editors, *Proc. of CALCO 2009*, volume 5728 of *LNCS*, pages 113–126. Springer, 2009.

[16] M. Michelbrink. Interfaces as functors, programs as coalgebras - a final coalgebra theorem in intensional type theory. *Theoret. Comput. Sci.*, 360(1–3):415–439, 2006.

[17] T. Mossakowski, L. Schröder, M. Roggenbach, and H. Reichel. Algebraic-coalgebraic specification in CoCasl. *J. Log. Algebr. Program.*, 67(1–2):146–197, 2006.

[18] M. Niqui. Coalgebraic reasoning in Coq: Bisimulation and the λ-coiteration scheme. In Berardi et al. [3], pages 272–288.

[19] M. Niqui. `http://www.cwi.nl/~milad/coalgebras/` [cited 18 Jan. 2010], Jan. 2010. Files for Coq v. 8.2.

[20] L. C. Paulson. Mechanizing coinduction and corecursion in higher-order logic. *J. Logic Comput.*, 7(2):175–204, Apr. 1997.

[21] G. Roşu. Equality of streams is a Π_2^0-complete problem. In J. H. Reppy and J. L. Lawall, editors, *Proc. of ICFP'06*, pages 184–191. ACM Press, 2006.

[22] J. Rothe, H. Tews, and B. Jacobs. The coalgebraic class specification language ccsl. *J. Universal Comput. Sci.*, 7(2):175–193, 2001.

[23] J. J. M. M. Rutten. Universal coalgebra: a theory of systems. *Theoret. Comput. Sci.*, 249(1):3–80, Oct. 2000.

[24] A. Setzer. Coalgebras in dependent type theory. `http://unit.aist.go.jp/cvs/symposium/AIM9/setzer.pdf`, [cited 18 Jan. 2010], 2008. Talk at AIM 9 : Agda Intensive Meeting 9. Sendai, Japan, 27 Nov. - 4 Dec. 2008.

[25] A. Setzer. Coalgebras and codata in agda. `http://www.cs.swan.ac.uk/~csetzer/slides/wessexSeminarMarch2009.pdf`, [cited 18 Jan. 2010], 2009. Talk at 3rd Wessex Theory Seminar. Bath, UK, March 2009.

A $\boldsymbol{\lambda}$-Bialgebra and $\boldsymbol{\lambda}$-Bisimulation

These definitions are based on [2] but are adapted to our framework. Let B be an extensional weak pullback preserving Set-functor, T be a Set-functor and $\Lambda\colon TB \Longrightarrow BT$ a natural transformation. We call (X, β_X, α_X) a λ-*bialgebra* if the following diagram commutes.

$$\begin{tikzcd}
T(X) \arrow[rr, "\beta_X"] \arrow[dr, "T\alpha_X"'] & & X \arrow[rr, "\alpha_X"] & & B(X) \arrow[dl, "B\beta_X"] \\
& TB(X) \arrow[rr, "\Lambda_X"] & & BT(X) &
\end{tikzcd}$$

Let (X, β_X, α_X) and (Y, β_Y, α_Y) be two λ-bialgebras. Then a binary relation $R\colon X \longrightarrow Y \to \mathsf{Prop}$ is a λ-*bisimulation* if there exists a unique arrow γ making the diagram below commute.

$$\begin{tikzcd}
T(X) \arrow[r, "\beta X"] & X \arrow[d, "\alpha_X"'] & \{\overline{\exists}(R)\} \arrow[l, "\pi_1"'] \arrow[r, "\pi_2"] \arrow[d, dotted, "\gamma"] & Y \arrow[d, "\alpha_Y"] & T(Y) \arrow[l, "\beta Y"'] \\
& B(X) & BT(\{\overline{\exists}(R)\}) \arrow[l, "B(\beta_X \circ T\pi_1)"] \arrow[r, "B(\beta_X \circ T\pi_2)"'] & B(Y) &
\end{tikzcd}$$

Centrum Wiskunde & Informatica (CWI) is the national research institute for mathematics and computer science in the Netherlands. The institute's strategy is to concentrate research on four broad, societally relevant themes: earth and life sciences, the data explosion, societal logistics and software as service.

Centrum Wiskunde & Informatica (CWI) is het nationale onderzoeksinstituut op het gebied van wiskunde en informatica. De strategie van het instituut concentreert zich op vier maatschappelijk relevante onderzoeksthema's: aard- en levenswetenschappen, de data-explosie, maatschappelijke logistiek en software als service.

Bezoekadres:
Science Park 123
Amsterdam

Postadres:
Postbus 94079, 1090 GB Amsterdam
Telefoon 020 592 93 33
Fax 020 592 41 99
info@cwi.nl
www.cwi.nl

CWI
Centrum Wiskunde & Informatica