Proving Equality between Streams

Frank Staals
University of Technology Eindhoven

Abstract Infinite lists, or streams are often used in mathematical problems. We investigate three models to represent these streams in Coq, a proof assistant. For each of these models we investigate what it means for streams to be equal, and how we can prove these stream equalities. In the two most fruitful models we represent streams using a coinductive type, which means coinduction is crucial in our proofs. As manual proofs become rather large, even for small examples, we present fully automated tactics to prove these stream equalities.

1 Introduction

A wide range of problems in mathematics and computer science can be expressed very concisely using so called streams. Streams are infinite lists, which are often defined by means of equations. Suppose for example that we are interested in finding the sum of the first n natural numbers which are a multiple of 3 or 5 [1]. The following Haskell[Hud+92] program defines the stream of natural numbers *nats* and a function that checks whether a number is a multiple of 3 or 5. By filtering our stream *nats* we obtain a stream containing all multiples of 3 or 5, of which we sum the first n.

```
nats = 0 : map (+1) nats
isMultiple x = x `mod` 3 == 0 || x `mod` 5 == 0
multiples = filter isMultiple nats
mySum n = sum $ take n multiples
```

This small example shows that we can program our problem very naturally using streams. Similarly streams are well suited to model all kinds of problems involving prime numbers and Fibbonaci sequences. They are also useful in a completely different setting. For example when modeling infinite paths through a labeled transition system or a Kripke structure.

Apart from these uses, streams are the simplest infinite data structure, which makes them interesting to study. We hope to extend the knowledge and techniques we gain by studying streams to more complex infinite data structures such as infinite trees.

We are mainly interested in proving equality between streams, in particular we want to automate these equality proofs. But before we can think about

[1] This is a variation of the first problem from Project Euler: http://projecteuler.net/.

(automated) equality proofs we should first describe how we model a stream. We distinguish three main models for these streams; in the first model we will represent the streams as a function mapping natural numbers to the elements stored in the stream (Section 2). The second model considers streams as a special kind of list. As streams have an infinite length we cannot model them using an inductive structure. Hence we investigate a way to model streams as a coinductive type (Section 3). The third model we consider is to specify what streams are rather than to define them. These specifications can vary widely in the level of abstraction. We choose to stay relatively close to the model using coinductive types. These stream specifications are presented in Section 4.

Related Work. We are not the first to investigate how to (automate) proving stream equalities. Goguen, Roşu, and Lin[GLR03; RG00] present an algorithm for Conditional Circular Coinductive Rewriting with Case Analysis (C4RW), which is used in the tool CIRC[LR07]. CIRC is an automated behavioral prover built as an extension of the rewriting system Maude[Cla+96]. Given a behavioral specification, represented as a set of equations, and a goal, it will try to prove the goal without further user input. The tool builds a term rewriting system from the input equations, and combines this with a circular coinduction principle (see also Section 3). This term rewriting system is then solved by Maude.

There are several descriptions [Gim96; Gim98; GC06; Chl10, Chapter 5] of how to formalize streams and very basic stream properties in the proof assistant Coq[Dow+; The04]. However none of them provide a generic and automated technique for their proofs on streams.

Finally Yves Berot[Ber05] describes a formalization in Coq[Dow+; The04] of how to use streams to implement and verify Eratosthenes' sieve.

The Coq Proof Assistant. Specialized coinductive provers are quite strong in proving stream equalities. Their main disadvantage is that it is hard to verify their correctness. They use complex algorithms with various optimizations, which makes it easy to make a mistake in the implementation. A more robust approach is taken in the Coq proof assistant[Dow+; The04]. In Coq all programs, properties and proofs are formalized in the same language, the Calculus of Inductive Constructions[BC04]. As a result, checking the validity of the proof reduces to type checking the term representing the proof.

We will formalize all our stream definitions and proofs in Coq. These definitions are similar in style to programs in a functional programming language. For example we can define the stream *ones* which has a 1 on all positions in Haskell as `ones = 1 : ones`. In Coq we write:[2]

```
CoFixpoint ones := 1 :: ones.
```

[2]By default the stream definitions in Coq are a bit more verbose. The package of Coq files that accompanies this document defines the notation that is used here.

We can prove properties in Coq by means of a sequence of *tactics*. These tactics allow the user to build and manipulate the term that represents the proof. The Coq standard library contains a large number of tactics which cover a wide range of operations. These vary from the very basics of introducing a new variable into the context (`intro`) to fully automatically solving non-quantified Presburger Arithmetic[Pug91] (`omega`). In Section 5 we show an example proof in which we will briefly explain some of these tactics. Coq also allows for users to write custom tactics by composing other tactics. We will use this to develop an automated prover for stream equalities in Section 6.

All Coq code and tactics presented in this paper, together with a number of additional proofs and definitions is available from `http://fstaals.net/tue/coq/richstream`.

Stream Basics. Before we dive into modeling streams and proving equalities we present some basic terms and functions which will be used throughout the rest of this document. Note that streams contain elements of a certain type, unless stated otherwise we will assume these elements are of some type A, and we only assume that there is a well defined equality of elements of this type.

Since Coq definitions are like program definitions in a functional programming language we will also use a functional style when writing function applications. So if we apply a function f on a variable x we will write $f\ x$ rather than $f(x)$.

We define the functions *hd* and *tl* to get the *head* respectively the *tail* of a stream. The head of a stream is the stream's first element, and the tail is the stream without it's first element. So for a stream $s = x :: xs$, we define $hd\ s = x$ and $tl\ s = xs$.

2 Modeling Streams as a Function

The first way in which we model our streams is by means of functions. A stream s is an infinite list containing elements of type A. So we consider a stream to be a function mapping the natural numbers to A. For some natural number n, the value $s\ n$ represents the n^{th} element in the stream s. We formalize this by means of the *FStream* type in Coq.

`Definition` *FStream* := $nat \to A$.

We then wonder what it means for two streams (*FStream*s) to be equal. A very natural definition for this equality, which we will call *EqFSt*, is the following. We say two *FStream*s are equal if (and only if), for any natural number n they contain the same element at position n. We can express this directly in Coq:

`Definition` *EqFSt* $(f\ g : FStream) := \forall\ n$:nat, $f\ n = g\ n$.

We can now prove the equality between two of these *FStream*s by induction to

n. As (this type of) induction is well studied this may seem like nice approach. Unfortunately it turns out that the proofs are rather clumsy and become difficult even for seemingly easy streams.

One of the main reasons for this is that it is hard to identify useful lemmas and partition the proof into smaller pieces. This means we end up with one large monolithic proof. So it seems the *FStream* model is unsuited for automating these proofs. Hence we investigate a different streams model in the next section.

3 Modeling Streams as a Coinductive Type

In our second model we focus on the fact that streams are infinite lists. We start from the definition of *finite lists*. We can inductively define a (finite) list using two constructors. The *nil* constructor creates and empty list, and using the *cons* constructor we can prepend an element to an existing list. We can formalize this in Coq by means of an *inductive type List*:

```
Inductive List A : Type :=
    | nil : List A
    | cons : A → List A → List A.
```

Similarly we can inductively define the equality *EqLSt* over these lists. For the base cases we have that two empty lists are equal, and an empty list is not equal to a non-empty list. Two non-empty lists are equal if both their first elements are equal and the remaining lists are again 'list-equal' (using *EqLst*). As these remaining lists are strictly smaller (shorter) than input lists l and m our equality *EqLst* is well defined.

If we have two lists l and m and we want to know if they are equal, then we can simply ask Coq to evaluate the term *EqLSt* l m for us. As both l and m are *finite lists* this procedure will terminate and Coq can decide whether or not l and m are equal. This allows us to use $l = m$ instead of *EqLSt* l t.

Defining a Stream Type. We would like to have similar definitions for streams, but unfortunately we cannot define streams inductively. This is perhaps the most obvious from the definition of the equality between lists. To be able to compute whether or not two lists are equal it is critical that the lists have finite length. To be more formal: in each inductive definition (or function) one of the arguments should be strictly decreasing. For streams we do not have such a decreasing argument as the tail of a stream again has an infinite length.

So instead of defining streams as an inductive type, we will use a *coinductive type*. This allows us to define the streams type as we would like:

```
CoInductive Stream A: Type :=
    Cons : A → Stream A → Stream A.
```

We can then construct streams using the *Cons* constructor which prepends an element to a stream. There is an additional constraint on our stream definitions though, we should make sure all stream definitions are *productive*. We say a stream definition is productive if for any natural number n, we can inspect the n^{th} element of the stream by unfolding it's definition finitely many times. Definitions that are not productive do not uniquely define a stream. For example consider the definition *bad1*, which is not productive. Even if we unfold this definition infinitely often it will never produce an element, and hence it does not define anything.

```
CoFixpoint bad1 : Stream A := bad1.
```

To enforce this concept of productivity Coq requires all coinductive definitions to be *guarded*. Each *corecursive call*, a reference to s when defining the stream s, should occur directly as one of the arguments of a constructor. As the above definition *bad1* contains a corecursive call which is not an argument of a *Cons*-constructor this definition is unguarded, and Coq will not accept it. Then consider the following examples in which x is an element of A, and f is a function from *Stream A* to *Stream A*.

```
CoFixpoint good : Stream A := Cons x good.
CoFixpoint bad2 : Stream A := Cons x (f bad2).
```

In the definition of *good* we see that the corecursive call *good* occurs as the top-level argument of our *Cons* constructor. This means the definition is properly guarded. Each time we unfold the definition of *good* we produce one new element. Hence the definition is productive, and therefore defines a valid stream; in this case the stream having an x at every position.

The definition *bad2* is again unguarded as the corecursive call is not the top-level argument of the *Cons*-constructor. Whether or not the definition *bad2* is productive depends on the function *f*. To be precise, this depends on the term *f bad2*. Unfortunately Coq has no way of verifying whether *f bad2* is 'safe', and therefore considers the definition unguarded. This can be a serious drawback as there are many productive stream definitions of this form. We will investigate a way to deal with this in Section 4.

A type representing Stream equality. We can now define a notion of equality on streams. This equality is very similar to the equality on lists; two streams are equal if both their heads, and their tails are equal. We represent this equality by another coinductive type:

```
CoInductive EqSt (s1 s2: Stream A) : Prop :=
    eqst :
        hd s1 = hd s2 → EqSt (tl s1) (tl s2) →
        EqSt s1 s2.
```

This now means that if we want to prove that two streams s and t are equal we should build a term of the type *EqSt s t*. As *EqSt s t* is coinductive type,

this also means that our terms of this type (our proofs) have to be guarded.

There is an isomorphism between the *FStream*s from the previous section and the *Stream*s we use here. So if *fs2rs* denotes the function that converts a *FStream* to a *Stream*, and *rs2fs* the function that converts a *Stream* into an *FStream*. Then the following theorems hold:

```
Theorem EqFSt_to_EqSt : ∀ (f g: FStream A),
                  EqFSt f g → EqSt (fs2rs f) (fs2rs g).

Theorem EqSt_to_EqFSt : ∀ (s t: Stream A),
                  EqSt s t → EqFSt (rs2fs s) (rs2fs t).
```

The proofs for these theorems can be found in the accompanying Coq files. Furthermore we note *EqSt* is a bisimulation relation.

Before we focus on how to build these *EqSt* terms we first introduce some useful notation. For our *Cons*-constructor we will use the infix notation '::', and similarly we will use '==' for our stream equality *EqSt*. So instead of *Cons* x xs and *EqSt* s t we will write $x :: xs$ and $s == t$.

Proving Stream Equalities. For an inductive type I we can define induction principles which we can use to prove properties of terms of I. Similarly we can define a *coinduction princple* for a coinductive type. We will illustrate this by the coinduction principle as it is used in CIRC[RG00; GLR03]. This so called *circular coinduction principle* is a bit different from the coinduction principle that is used in Coq. It explicitly models the productivity concept using a *freeze function*, instead of guardedness.

The circular coinduction principle states that to prove that two streams s and t are equal we should prove two properties. The first property directly corresponds with the definition of *EqSt*: the heads of both streams should be equal. The second property expresses that if we assume s and t are *behaviorally equivalent*, the tails of s and t are also behaviorally equivalent. Two streams are behaviorally equivalent if there is no way to distinguish between them. So no matter what (congruent) function we apply on both streams, the result is equal. This coinduction principle can be formulated as follows:

```
Theorem coInductionFreeze: ∀ (s t: Stream A),
  hd s = hd t →
  (∀ (B: Type) (fr: Stream A → B),
          fr s = fr t → fr (tl s) = fr (tl t)) →
  s == t.
```

Using a coinduction principle we can now create *EqSt* terms, allowing us to prove equalities between streams. As we will see in Section 5, equality proofs are still quite laborious if done by hand. Luckily this model allows for automating a large part of these proofs (Section 6).

4 Streams by Specifications

The model for our streams as presented in the previous section seems to be well suited for our equality proofs. Unfortunately the requirement that all definitions have to be guarded imposes a serious restriction on our stream definitions. For example, the *nats* stream presented in Section 1 is productive, but not guarded. So we cannot use this definition in Coq.

To avoid this problem we consider a third streams model. Instead of defining the stream by telling Coq how to construct it we simply specify properties our stream should have. We then have a choice on the abstraction level we use for these specifications. To be able to use the nice properties, principles and tactics from the streams as coinductive type model, we choose the same type definitions. So streams are again of the (coinductive) type *Stream*, and an equality proof of streams s and t is of type $EqSt\ s\ t$.

Our goal is now to specify the *behavior* of the stream. To do this we pick a set of functions $\mathcal{F}$ that together allow us to inspect the entire stream (all elements in the stream). This means that we can distinguish streams using one of the functions in $\mathcal{F}$ if and only if they are not equivalent with respect to the *EqSt* equality. We call such a set of functions $\mathcal{F}$ a *cobasis*.

For each function f in the cobasis we then specify the result of applying f to the stream. This yields a set of equations which specify the stream's behavior.

There are many such cobases for our streams. The most natural cobasis is the one consisting of the functions *hd* and *tl*. We model the *nats* stream from Section 1 using this cobasis as follows:

`Definition` $nats : Stream\ nat.\ Admitted.$
`Hypothesis` $nats\_hd : hd\ (nats) = 0.$
`Hypothesis` $nats\_tl\ : tl\ \ (nats) = map\ S\ nats.$

We can then prove stream equalities by rewriting with these equations. Tools like CIRC[LR07] use the same approach. They use these specifications as input and transform them to a term rewriting system. By combining this with rules for circular coinduction this is a powerful procedure for automatically proving stream equalities.

5 Manually Proving Stream Equalities

In this section we show two equality proofs in Coq. The first proof uses the streams as coinductive type model presented in Section 3, and the second proof uses the streams by specification model from Section 4.

The first proof uses the functions *zip* and *even*. The *zip* function takes two streams and interleaves them. The result of *zip* applied to the streams s and t is a single stream in which all elements on even positions originate from s

and the elements on odd positions from t. In Haskell we could define this as `zip (x : xs) t = x : zip t xs` [3]. The pattern matching construct in Coq is a bit more verbose, but the definition is basically the same:

```
CoFixpoint zip(s: Stream A)(t: Stream A) : Stream A :=
  match s with
    | x :: xs ⇒ x :: zip t xs
  end.
```

The function *even* takes a stream and retains only the elements on an even position:

```
CoFixpoint even(s: Stream A) := hd s :: (even (tl (tl s))).
```

We can then prove that for any two input streams if we interleave them using *zip*, the *even* function will give us back the first input stream. We prove this in Coq as follows:

```
Lemma even_zip : ∀ s t: Stream A, even (zip s t) == s.
Proof.
  coinduction CH ; destruct s as [x xs]; destruct t as [y ys].
  simpl.
  reflexivity.
  simpl.
  change (even (zip xs ys) == xs).
  apply CH.
Qed.
```

The first line of the proof uses the `coinduction` tactic, which creates two subgoals. These subgoals correspond to properties we have to prove when using circular coinduction (Section 3), but instead of using an explicit freeze function Coq checks that the proof term we are building is guarded. The `destruct s as [x xs]` tactic explicitly binds the head of s to a new variable x, and the tail to xs. By chaining this tactic after `coinduction` using the ';' operator we apply the tactic on all subgoals that are generated by `coiduction`. This leaves us the following two goals:

```
   hd (even (zip (x :: xs) (y :: ys))) = hd (x :: xs)

subgoal 2 is:
 tl (even (zip (x :: xs) (y :: ys))) == tl (x :: xs)
```

On the first subgoal we use the `simpl` tactic, which applies a series of reduction steps simplifying the goal to `x = x`. The `reflexivity` tactic then solves this subgoal. We then have to prove the second subgoal:

[3] Note that this zip is not the same as Haskell's built in zip function, which combines two lists into a list of tuples.

```
CH : forall s t : Stream A, even (zip s t) == s
...
============================
 tl (even (zip (x :: xs) (y :: ys))) == tl (x :: xs)
```

If we again use the `simpl` tactic this gives us the following scary looking goal:

```
CH : forall s t : Stream A, even (zip s t) == s
...
============================
 (cofix even (s : Stream A) : Stream A :=
    (head s :: even (tl (tl s))))
   ((cofix zip (s t : Stream A) : Stream A :=
       match s with
       | (x0 :: xs0) => (x0 :: zip t xs0)
       end) xs ys) == xs
```

Unfortunately the `simpl` tactic unfolds the goal a bit too far, showing us quite some detail on how Coq is building a coinductive term. If we look carefully we see that the left-hand side of the goal contains the definition of two coinductive functions; the functions *even* and *zip*. The `change` tactic allows us to rewrite the goal into something more readable:

```
CH : forall s t : Stream A, even (zip s t) == s
...
============================
 even (zip xs ys) == xs
```

We then solve this goal by using `apply` *CH*, which tells Coq the goal follows from a term in the context. Namely the term with label *CH*. This proves the second subgoal and completes the proof.

For the second example we use the streams by specifications model. Let *id* be the identity function on streams, then consider the following somewhat strange specification for the stream that has a 1 on all positions:

`Definition` *onesid* : *Stream nat. Admitted.*
`Hypothesis` *onesid_hd* : $hd\ (onesid) = 1$.
`Hypothesis` *onesid_tl* : $tl\ \ (onesid) = id\ onesid$.

Note that this stream is not guarded, and hence we cannot give a Coq definition. We then prove that this stream is equal to the 'regular' specification for the stream with a 1 on all positions. We call the stream given by the 'regular' specification *ones*. The complete proof is then as follows:

```
Lemma ones_eq_onesid : ones == onesid.
  Proof.
    coinduction CH.
    rewrite ones_hd ; rewrite onesid_hd.
    reflexivity.
    rewrite ones_tl ; rewrite onesid_tl.
    coinduction CH0.
    rewrite ones_hd ; rewrite id_hd ; rewrite onesid_hd.
    reflexivity.
    rewrite ones_tl ; rewrite id_tl ; rewrite onesid_tl.
    apply CH0.
Qed.
```

We highlight two intermediate states in this proof. The first one is after applying the `coinduction` tactic on the first line.

```
  id_hd : forall s : Stream nat, hd (id s) = hd s
  id_tl : forall s : Stream nat, tl (id s) = tl s
  ones_hd : hd ones = 1
  ones_tl : tl ones = ones
  onesid_hd : hd onesid = 1
  onesid_tl : tl onesid = id onesid
  ============================
   hd ones = hd onesid

subgoal 2 is:
 tl ones == tl onesid
```

We now have the specifications for all streams in the context. Which means we can rewrite the goal by using the `rewrite` tactic. The second state we highlight is the state after applying the rewrite steps on line 4. We are left with an other stream equality:

```
  id_hd : forall s : Stream nat, hd (id s) = hd s
  id_tl : forall s : Stream nat, tl (id s) = tl s
  ones_hd : hd ones = 1
  ones_tl : tl ones = ones
  onesid_hd : hd onesid = 1
  onesid_tl : tl onesid = id onesid
  CH : ones == onesid
  ============================
   ones == id onesid
```

Since we do not have useful equalities in the context we treat this as a new stream equality we have to prove. Hence we again use the `coinduction` tactic. This time we can solve the new goals using some additional rewrite steps and the coinduction hypothesis.

6 Automating Equality Proofs

Even though our streams model allows us to prove various stream equalities in Coq, it is not always immediately clear how to approach the problem. In other cases these proofs requires quite a lot of work, even for very simple streams. So we would like a tactic that proves these equalities automatically. In this section we present 2 such tactics. One that can prove stream equalities in the streams as a coinductive type model, the other that can handle stream specifications. The first tactic `solveEqSt` can fully automatically prove stream equalities in the model from Section 3.

The `solveEqst` tactic combines Coq's excellent simplification and resolution-like tactics such as `simpl` and `auto` with a slightly modified coinduction tactic. The global approach is as follows. We start by applying coinduction, this yields two subgoals, one for the head and one for the tail. The subgoal for the head we can usually completely solve using Coq's simplification tactics (and otherwise the tactic fails). On the second subgoal we apply the simplification tactics and try to apply hypotheses available in the context. If this does not solve the goal we end up with another stream equality, which we again try to solve using our `solveEqSt` tactic. We implement all of this in Ltac, Coq's tactics language. This yields the following tactic:

```
Ltac solveEqSt := let CH := fresh "CH" in
  try solveEqSt_applyHyps ; coinduction' CH ; [solveHead | solveTail ]
with solveHead := solveRewrite ; try reflexivity ;
                        fail "Solving heads failed"
with solveTail := solveRewrite ; try solveEqSt_applyHyps ; solveEqSt.
```

In this definition we see a couple of other custom tactics. The `solveRewrite` tactic uses a combination of `simpl` and `intuition`. The `solveEqSt_applyHyps` tactic tries to apply the hypothesis available in the context.

An important part of this machinery is still hidden in the `coinduction'` tactic. It creates the coinductive term representing the proof, introduces dummy variables for quantified variables, and takes care of generating the two subgoals:

```
Ltac coinduction' proof :=
  cofix proof; sdintros; constructor;
    [ clear proof | try (apply proof; clear proof) ].
```

This definition is almost identical to the 'regular' `coinduction` tactic. The important difference is in how dummy variables are added to the context. The regular tactic uses `intros` for this, whereas our tactic is using yet an other custom tactic: `sdintros`. For Coq's simplification mechanism to work it needs to match the stream onto pattern matching constructs. As Coq does not split the input stream into a head and tail, we should do this ourselves. The `sdintros` tactic takes care of this.

Running SolveEqSt. If we now use our `solveEqSt` tactic to try to prove a stream equality there are four possible outcomes:

1. The first option is that the streams are not equal. In this case the tactic will fail after a finite amount of time with the message that it could not prove that the heads of some streams are equal.
2. The second option is that the input streams are equal and that the tactic will succeed in proving this after a finite amount of time.
3. The third option the streams are again equal, but after a finite amount of time the tactic will fail, telling the user that it could not prove that the heads of some streams are equal. In this case this means the machinery in `solveHead` was simply not powerful enough to prove that the heads are really equal.
4. The fourth and final option is that the streams are equal, but the tactic will take an infinite amount of time to try to prove it.

The *even_zip* lemma from the previous section falls in the second category. The `solveEqSt` tactic is able to prove this almost instantaneously. A list of other stream equations that `solveEqSt` can automatically solve can be found in the accompanying Coq files.

There is one remaining issue with the `solveEqSt` tactic; in case the second subgoal immediately follows from the coinduction hypothesis the `coinduction` tactic will already prove this subgoal. As a result the tactic cannot apply the `solveTail` tactic (as there is no subgoal to prove) and fails.

Dealing with Stream Specifications. To handle the cases in which the input streams are not given by a definition but by a specification we have a second tactic called `solveEqSt_Eqs`. This tactic follows the same approach as `solveEqSt`, but instead of using Coq's simplification tactics it treats the equations from the specifications as a term rewriting system. This means the definition of `solveEqSt_Eqs` is the same as the definition of `solveEqSt` in which the `solveRewrite` tactic has been replaced by the following tactic:

`Ltac` *rewrite_eqst* := `repeat (progress` *find_rewrite*).

This `rewrite_eqst` tactic repeatedly applies the `find_rewrite` tactic until the goal does not change any further. The core of `find_rewrite` is formed by `find_rewrite_match` which tries to find an occurrence of one of the equations in the context, and then rewrites the goal with this equation. We can express this in Ltac using a pattern match construction:

```
Ltac find_rewrite_match :=
  match goal with
    | [ H : hd (?s) = _ ⊢ context[hd (?s) ] ] ⇒ rewrite H
    | [ H : tl (?s) = _ ⊢ context[tl (?s) ] ] ⇒ rewrite H
    | [ H : ∀ s: Stream _, hd (?f _) = _ ⊢ context[hd (?f _) ] ] ⇒ rewrite H
    | [ H : ∀ s: Stream _, tl (?f _) = _ ⊢ context[tl (?f _) ] ] ⇒ rewrite H
    | [ H : ∀ s t: Stream _, hd (?f _ _) = _ ⊢ context[hd (?f _ _) ] ]
                        ⇒ rewrite H
    | [ H : ∀ s t: Stream _, tl (?f _ _) = _ ⊢ context[tl (?f _ _) ] ]
                        ⇒ rewrite H
    | _ ⇒ idtac
  end.
```

Since our `solveEqSt_Eqs` uses the same approach as `solveEqSt` it has also has the same behavior. So when running the `solveEqSt_Eqs` tactic to prove a stream equality we also end up in one of the four cases described above.

The *ones_eq_onesid* lemma, as well as the *even_zip* lemma in which the streams are given by specifications instead of definitions, are proven almost instantaneously by the `solveEqSt_Eqs` tactic. A list of other examples is included in one of the accompanying Coq files.

7 Conclusion

We have seen three ways of modeling infinite lists, or so called streams, and we have seen how to implement these models in Coq. An easy model to regard streams as functions mapping natural numbers to elements of the stream. Unfortunately equality proofs quickly become extremely large and difficult.

Both other models represent streams as a coinductive type, which allows us to use the (circular) coinduction principle in our proofs. One of these models uses actual stream definitions. Coq requires that these definitions are guarded, which is very restrictive requirement. In the other model we only specify the behavior of the stream, avoiding this problem.

Even though both these models are better suited for equality proofs than the first model, the proofs are still rather cumbersome. Therefore we have developed two tactics which automate these proofs. The first tactic, `solveEqSt`, solves most equalities over streams we can define in Coq. The second tactic, `solveEqSt_Eqs`, can automatically prove equalities for streams given by a specification.

In particular for the `solveEqSt_Eqs` tactic there is still a lot of room for improvement. It might be worthwhile to investigate if it is possible to implement more powerful algorithms, for example the algorithms that are used in tools like CIRC. Another important angle worth investigating is how to get rid of the strict guardedness condition imposed on coinductive types by Coq.

References

[BC04] Y. Bertot and P. Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer-Verlag New York Inc, 2004. ISBN: 3540208542.

[Ber05] Y. Bertot. "Filters on coinductive streams, an application to eratosthenes sieve". In: *Typed Lambda Calculi and Applications* (2005), pp. 102–115.

[Chl10] A. Chlipala. *Certified programming with dependent types*. 2010. URL: http://adam.chlipala.net/cpdt.

[Cla+96] M. Clavel et al. "Principles of maude". In: *Electronic Notes in Theoretical Computer Science* 4 (1996), pp. 65–89. ISSN: 1571-0661.

[Dow+] G. Dowek et al. "The Coq proof assistant users guide". In: *Rapport technique* ().

[GC06] E. Giménez and P. Castéran. "A tutorial on [co-] inductive types in Coq". In: *Report technique* (2006).

[Gim96] E. Giménez. "An application of co-inductive types in Coq: verification of the alternating bit protocol". In: *Types for Proofs and Programs* (1996), pp. 135–152.

[Gim98] E. Giménez. "A Tutorial on Recursive Types in Coq". In: *Raport technique 0221* (1998).

[GLR03] J.A. Goguen, K. Lin, and G. Rosu. "Conditional circular coinductive rewriting with case analysis". In: *Recent Trends in Algebraic Development Techniques* (2003), pp. 216–232.

[Hud+92] P. Hudak et al. "Report on the programming language Haskell: a non-strict, purely functional language version 1.2". In: *ACM Sigplan Notices* 27.5 (1992), pp. 1–164. ISSN: 0362-1340.

[LR07] D. Lucanu and G. Roşu. "CIRC: A circular coinductive prover". In: *Algebra and Coalgebra in Computer Science* (2007), pp. 372–378.

[Pug91] W. Pugh. "The Omega test: a fast and practical integer programming algorithm for dependence analysis". In: *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*. ACM. 1991, p. 13. ISBN: 0897914597.

[RG00] G. Rosu and J. Goguen. "Circular Coinduction". In: *In International Joint Conference on Automated Reasoning*. Citeseer. 2000.

[The04] The Coq Development Team. *The Coq proof assistant Reference Manual*. 2004-2010. URL: http://coq.inria.fr/refman/.