Co-inductive Proofs for Streams in PVS*

Hanne Gottliebsen[1]

Department of Computer Science, Queen Mary, University of London
hago@dcs.qmul.ac.uk

Abstract. We present an implementation in the theorem prover PVS of co-inductive stream calculus. Stream calculus can be used to model signal flow graphs, and thus provides a nice mathematical foundation for reasoning about properties of signal flow graphs, which are again used to model a variety of systems such as digital signal processing. We show how proofs by co-induction are used to prove equality of streams, and present a strategy to do this automatically.

1 Introduction

How do we reason in an abstract way about signals and circuits? Classical mathematical techniques require analysis relying on numerical methods. An alternative approach is to take a more abstract view, for example using signal flow graphs [4]. This allows for a much more precise model of the circuits, and thus provides a better basis for analysis.

The work we present here is part of the Logical Structures for Control project, which is concerned with reasoning about dynamical systems. The projects has two main strands. One strand is building a Hoare logic for dynamical systems in general [3, 1], which can then be applied to various models of dynamical systems, such as block diagrams or signal flow graphs. The other strand is constructing a practical implementation for reasoning about models of dynamical systems. It is within this second strand that the work presented here falls.

It turns out that for several practical research areas, signal flow graphs provide an elegant, modular and scalable notation for modelling. Signal flow graphs were originally introduced by Mason [4] in 1953 for modelling linear networks. Since then, they have been used for a variety of systems, for example modelling circuit transposition for circuits [11], test generation for mixed-signal devices [8] and for digital filters [10]. Thus they are generally used as a modelling tool.

Stream calculus was introduced by Escardó and Pavlović [7]. Their main idea is to interpret the stream elements as factors in approximation for functions, thus establishing the connection between stream calculus and classical calculus. The basic operations of stream calculus are conceptually very simple, for instance differentiation corresponds to taking the tail of a stream, and so stream calculus provides a way to work on problems such as differential equations.

* This work is supported in part by The Nuffield Foundation.

Rutten [9] showed how signal flow graphs can be modelled very nicely using stream calculus. This gives us a precise mathematical notion of what a signal flow graph represents, and allows us to perform mathematical analysis of signal flow graphs in a precise way. Based on this idea of stream calculus as a model for signal flow graphs, we did an implementation in the theorem prover PVS of stream calculus, as we wanted to investigate formal reasoning about signal flow graphs.

PVS (Prototype Verification System) [6] is an interactive theorem prover. The specification language is powerful and allows the use of predicate subtyping as well as higher-order classical logic. Although PVS is an interactive theorem prover, a good level of automation is provided and PVS also facilitates the addition of user-defined automation. Of particular importance to our implementation is the existing support in PVS for co-inductive datatypes. Our implementation of stream calculus in PVS allows us to reason in the most rigorous sense about properties streams and functions over streams.

The implementation is based on co-inductive streams, that is streams are defined as infinite lists rather than as functions from the natural numbers to their element types. We have found that many definitions and also proofs are more intuitive using this technique, although of course conceptually (if not in implementation) the two basic ideas would model the same streams. Using stream calculus allows a very simple way to solve differential equations, and we use this in some of our examples. Many proofs in the implementation relies on the co-induction principle, which in turns requires a bisimulation between two streams to be given. Just like working out an induction hypothesis for a regular proof by induction can be difficult, so can choosing the "right" bisimulation. However, we show how one can systematically "guess" a bisimulation from the proof goal. This is important, as it makes the proofs a lot simpler. We have implemented this as a strategy in PVS and present here for the first time this strategy and some applications of it.

The current implementation is the first step in providing a platform for formal reasoning about signal flow graphs using stream calculus. It is evident that such a platform would be useful for many different topics.

1.1 Structure

In Sect. 2 we explain in some detail the basic concepts of stream calculus. Section 3 outlines the implementation of stream calculus in PVS, with the use of a co-inductive datatype. Section 4 explains how we may automate proofs by co-induction, thus making the current implementation much easier to use. Finally, Sect. 5 contains some conclusions and directions for further work.

2 Stream Calculus

The notion of stream calculus was introduced by Escardó and Pavlović [7] as a means to do symbolic computation (in for example computer algebra and theorem proving) using co-induction.

Streams in general can be defined over any kind of element type, however since we intend to model signal flow graphs, we will restrict our element type to the set $\mathbb{R}$ of real numbers. Following the definitions of Rutten [9], we view a stream as a function from $\mathbb{N}$ to $\mathbb{R}$, and let the set of streams over the reals be denoted by $\mathbb{R}^\omega$:

$$\mathbb{R}^\omega = \{\sigma | \sigma : \mathbb{N} \to \mathbb{R}\}$$

Following the tradition of Escardó and Pavlović [7] we use the following terminology: we call $\sigma(0)$ the *initial value* of the stream σ, and the *derivative* σ' of the stream σ is given by

$$\sigma'(n) = \sigma(n+1)$$

These are more commonly known as *head* and *tail* in computer science, however having this notion of a derivative allows the development of a *calculus of streams* [7] which is fairly close to that of classical functional analysis. We may use :: to denote appending elements to streams, for example

$$\sigma = a_0 :: a_1 :: \rho$$

where σ and ρ are streams and a_0 and a_1 are stream elements.

We can now define addition and multiplication of streams as follows. The *sum*, $\sigma + \tau$ of streams σ and τ is element-wise, that is

$$\forall n \in \mathbb{N} : (\sigma + \tau)(n) = \sigma(n) + \tau(n)$$

The *convolution product*, $\sigma \times \tau$ of streams σ and τ is given by

$$\forall n \in \mathbb{N} : (\sigma \times \tau)(n) = \sum_{k=0}^{n} \sigma(k) \cdot \tau(n-k)$$

We can embed the real numbers into the streams by defining the following stream. Let $r \in \mathbb{R}$. Then $[r]$ is defined as follows:

$$[r] = (r, 0, 0, 0, \dots)$$

This essentially allows us to add and multiply real numbers and streams:

$$[r] + \sigma = (r + \sigma(0), \sigma(1), \sigma(2), \dots)$$
$$[r] \times \sigma = (r \cdot \sigma(0), r \cdot \sigma(1), r \cdot \sigma(2), \dots)$$

Often we will simply use r to denote the stream $[r]$, it will be clear from the context if r is a real number or the stream related to the number.

Finally, we can define a constant stream of particular interest, X:

$$X = (0, 1, 0, 0, \dots)$$

The effect of multiplying a stream by X is a delay of 1, that is:

$$X \times \sigma = (0, \sigma(0), \sigma(1), \sigma(2), \dots)$$

We see that multiplication by X is essentially an *antiderivative*, in the sense that if we multiply a stream by X and then differentiate, we get the original stream back:

$$(X \times \sigma)' = \sigma$$

However, the reverse is only true if the initial value of σ is 0. This corresponds to the constant of integration in analysis being 0.

With the above definitions of differentiation, addition and multiplication, we can obtain the following facts about differentiation of sums and products by applying the basic operations:

$$(\sigma + \tau)' = \sigma' + \tau'$$
$$(\sigma \times \tau)' = ([\sigma(0)] \times \tau') + (\sigma' \times \tau)$$

We see that the sum behaves exactly as in classical calculus, however multiplication does not.

3 Stream Calculus in PVS

PVS has an extensive set of libraries, and provides easy ways for users to define new datatypes and related operations. In particular, defining a co-inductive datatype is simple with PVS automatically providing the basic operations and properties of the datatype based on the definition. With PVS being strongly typed and supporting predicate subtyping, we can use types to fully define certain functions, something which we use extensively in this implementation.

We have a basic implementation of stream calculus in PVS. The implementation covers all the operations described in the previous section.

3.1 Basic Notion of Streams

We are using the co-inductive datatype constructor in PVS to implement the streams:

```
stream : CODATATYPE
  BEGIN
   str(car: real, cdr:stream):str?
  END stream
```

This gives us a datatype `stream` which works essentially like infinite lists, with `car` and `cdr` denoting initial value and derivative, respectively. With the co-inductive datatype we get (for free, from the implementation of `CODATATYPE` in PVS) various theorems, most importantly for us the definition of co-induction:

```
coinduction: AXIOM
  FORALL (B: (bisimulation?), x: stream, y: stream):
    B(x, y) => x = y;
```

In PVS, there is a built-in polymorphic type for sequences, modelled as a function from the natural number to the element type – thus much closer to the definition of streams given in Sect. 2. This of course could also be used to model streams. Indeed, when we first started this work, that is the approach we took [2]. Normally, one would argue in favour of using the co-inductive datatype mainly if one wants to model finite streams alongside the infinite streams. At the moment we are not doing this, although we are considering it. However, we chose to use the co-inductive datatype regardless, since many of the existing proofs (in papers) of properties of stream calculus uses co-induction. In fact, we have basic implementations of stream calculus in both notations, and we found that many proofs are simpler using the co-inductive datatype. We also wanted to investigate closer how the co-inductive datatype works in PVS, and hope that this implementation can serve as an example for PVS users in general. Another reason for choosing to use the co-inductive datatype, is that, with a little work, it allows us to give definitions very close in style to those with self-reference corresponding to

$$A = a_0 :: a_1 :: A$$

This is not in general simple in PVS, where items must be declared before they can be used.

Declaring co-inductive streams in PVS, as well as the main result about doing proofs using co-induction, we also get various functions used in constructing co-inductive streams, for example `inj_str` and `coreduce`, most of which are somewhat cumbersome to work with in part due to being automatically generated. In order to simplify the notation compared to the automatically generated notation, we introduce our own notation, inspired by Miner [5]:

```
f, g : VAR [real -> real]
a : VAR real

corec(f,g)(a) : stream =
  coreduce(lambda (b: real): inj_str(f(b), g(b)))(a)
```

The meaning of this is that given two functions over the reals f and g and a real number a, $corec(f, g)(a)$ returns the following stream:

$$corec(f, g)(a) = f(a) :: f(g(a)) :: f(g(g(a))) :: \ldots$$

That is, element n in the stream is defined by successively applying g and then finally applying f once. In fact, we defined `corec` and its associated operations and properties in a polymorphic way, as this was useful for implementing certain stream operators, such as addition, Sect. 3.2.

Now, if we want to be able to define a constant stream such as $a :: a :: a :: \ldots$, it can be done as follows:

```
const(a) : stream = corec[real](id,id)(a)
```

where a is a real number and id is the identity function on the real numbers. We can then use co-induction to prove the following lemma, confirming that *const* does indeed give us the constant stream.

```
const_fact : LEMMA const(a) = str(a,const(a))
```

There are many functions which can be defined quite nicely using corecursion, for example *map* and *iter* (short for iterate):

```
map(f,s) : stream = corec[stream](lambda t1 : f(car(t1)),
                                  lambda t1 : cdr(t1))(s)

iter(f,a) : stream = corec[real](id, f)(a)
```

where $f : \mathbb{R} \to real$, $a : \mathbb{R}$ and s is a stream. Then $map(f, s)$ returns the stream of f applied element-wise to s, and $iter(f, a)$ returns the stream where f is applied to a n times for element n. Thus

$$map(f, s) = f(s_0) :: f(s_1) :: f(s_2) :: \ldots$$
$$iter(f, a) = a :: f(a) :: f(f(a)) :: \ldots$$

3.2 Calculating with Streams

We now define the basic operations on streams:

```
initial(sigma) : real = car(sigma)

derivative(sigma) : stream = cdr(sigma)

% Adding two streams
+(s1,s2) : stream = corec(lambda t1,t2 : car(t1) + car(t2),
                          lambda t1,t2 : (cdr(t1),cdr(t2)))(s1,s2)

% Scalar multiplication
m(a,s) : stream = map(lambda x : a * x, s)

% Register/Delay
R(s) : stream = str(0,s)

% Some fact to show that the definitions work as expected
add_fact : LEMMA
  +(s1,s2) = str(car(s1) + car(s2), +(cdr(s1),cdr(s2)))

m_fact : LEMMA m(a,s) = str(a*car(s), m(a,cdr(s)))

R_inv_deriv : LEMMA derivative(R(s)) = s
```

One operator which is made slightly more complicated because we are using the co-inductive datatype rather than streams as functions over the natural numbers, is the convolution product. This really highlights another difference between the two notations. In order to calculate the convolution product, we

need *history*, in the sense that for each element of the resulting stream, we need to know all preceding elements of each of the two input streams. There is no way to avoid this. In our implementation we get around this by essentially going back to considering element number n, something which is a bit unnatural in the co-inductive definition, but nonetheless works.

3.3 Differential Equations

One of the interesting questions about a given signal flow graph is whether it implements a solution to a certain differential equation or not. Thus, we might consider the following first-order stream differential equation, and ask what s should be:

$$s(0) = a \ \wedge \ s' = s \tag{1}$$

Just by looking at the stream and remembering the definition of the derivative of a stream, we get

$$\begin{aligned} s &= a :: s \\ &= a :: a :: s \\ &= a :: a :: a :: s \end{aligned}$$

So we see that s is the constant stream over a. Then if our signal flow graph outputs this stream, it gives a solution to the differential equation.

In general, a first-order stream differential equation is of this form:

$$t(0) = a \wedge t = s \tag{2}$$

where a is some real number and s and t are streams of real numbers. This leads us to the following definition in PVS of the solution to a first-order differential equation as in (2):

```
fode(s,a) : stream = { t | car(t) = a and cdr(t) = s }
```

This means that that the function $fode$ takes arguments s and a and returns a single stream with initial value a and derivative s. That is, $fode(s, a)$ is the solution to (2). So we immediately have the following result:

```
fode_fact : LEMMA
  fode(s,a) = str(a,s)
```

We can now prove the result of our example (1) above:

```
example_1_3 : LEMMA
  s = fode(s,a) IMPLIES s = const(a)
```

We have implemented second-order differential equations in essentially the same manner:

```
sode(s,a,b) : stream =
  { t | car(t) = a and car(cdr(t)) = b and cdr(cdr(t)) = s }

sode_fact : LEMMA
  sode(s,a,b) = str(a,str(b,s))
```

As an example, consider the following second-order stream differential equation:

$$\begin{aligned} s(0) &= a \\ s(1) &= b \\ s'' &= s \end{aligned}$$

Again, by looking at the stream, we can find s:

$$\begin{aligned} s &= a :: b :: s \\ &= a :: b :: a :: b :: s \end{aligned}$$

We see that this corresponds to the constant streams over a and b respectively *zipped*, in PVS:

```
example_1_4 : LEMMA
  s1 = sode(s1,a,b) IMPLIES s1 = zip(const(a),const(b))
```

where $zip(s,t) = s_0 :: t_0 :: s_1 :: t_1 :: \ldots$.

To summarise, we can define streams and functions over streams. We can also do basic interactive proofs in PVS, but in the next section we will discuss new automation for proofs by co-induction, making most of the proofs of general lemmas about streams and functions over streams much easier.

4 Automation of Co-inductive Proofs

In this section we will discuss co-induction in general, and see how we can automate proofs by co-induction in PVS. The strategy presented here is set up specifically to work with the implementation of stream calculus, but we see no reason why it cannot be generalised to co-inductive datatypes in general.

Following Rutten [9] we first introduce formally the notion of bisimulation on streams and the co-induction principle for streams. A bisimulation is a relation, defined as follows:

Definition 1 (Bisimulation). *Let R be a binary relation on streams. Then R is a* bisimulation *if for all streams s and t the following holds:*
$sRt \Rightarrow car(s) = car(t) \wedge cdr(s)Rcdr(t)$

We see that equality is an obvious bisimulation. However, not all bisimulations are equivalent to equality, for example the relation

$$R(s,t) = (s = const(1) \wedge t = const(1))$$

is a bisimulation, but the stream (1,0,0,...) is not related to itself under this bisimulation, although it is obviously equal to itself. In PVS the above definition is generated automatically when declaring a co-inductive datatype, such as our streams.

Let us remind ourselves about the co-induction principle as seen in Sect. 3.1:

Theorem 1 (Co-induction).
Let two streams, s and t, and a bisimulation on streams, $\equiv$, be given. Then, if $s \equiv t$ then $s = t$.

Again, this theorem is generated automatically by PVS for any co-inductive datatype, obviously with the appropriate types used.

Every time we want to do a proof by co-induction, we need to "invent" a bisimulation which fits the statement we are trying to prove. This is not necessarily a simple task, and as usual with theorem proving, having better support for automation makes proving so much easier. Let us look at an example:

Example 1 (Proof by Co-induction – map_iter).
We want to prove that for the functions *map* and *iter* (Sect. 3.1), a function $f : \mathbb{R} \to \mathbb{R}$ and a real number x, we have the following:

$$map(f, iter(f, x)) = iter(f, f(x))$$

Since both *map* and *iter* return streams, this is an equality between streams.

Now consider the relation R, where

$$R(s,t) = \exists g : \mathbb{R} \to \mathbb{R}, y : real : s = map(g, iter(g, y)) \wedge t = iter(g, g(y)) \quad (3)$$

It is clear that $R(map(f, iter(f, x)), iter(f, f(x)))$ since we can choose f for g and x for y. So, assuming that R is indeed a *bisimulation*, we can use Theorem 1 to prove equality of the two stream.

Thus we need to prove that R is indeed a bisimulation, that is $sRt \Rightarrow car(s) = car(t) \wedge cdr(s) R cdr(t)$ for all streams s and t. This can be easily done by unfolding the definitions of *map* and *iter*.

In the above example, we see that given an appropriate relation, in this case R as in (3), the rest of the proof is relatively simple, if tedious. So it seems that the step, when doing proof by co-induction, which requires "invention" is determining which relation to use. However, it is possible to do this in a very mechanistic way for many cases of stream equality. Looking again at the above example, we see that the r suggested can be obtained simply by considering the structure of the equality, we want to prove. One may wonder why we need to introduce new, existentially quantified variables g and y in the definition of the relation. The reason for this is to ensure that the relation is indeed a bisimulation, for which we need the cdr's of two related streams to also be related.

In Ex. 1 the proof of the relation being a bisimulation is quite simple, using only rewriting and simplification, but in some cases this step requires real insight and relies on using other lemmas about properties of the functions and streams involved. For this reason, this step has not yet been automated.

The simple pattern matching of the actual formula being proven illustrated above is not in general enough. Often, in a proof, we have various assumptions which may also be used in the proof, and the very basic relation above does not capture those. Therefore the relation may not be a bisimulation, although a bisimulation may indeed exist. One such example is (1). So, as well as looking at the actually equality we want to prove, we also need to consider the rest of the current proof goal.

4.1 The PVS strategy

The basic structure of the PVS strategy `bisim` is as follows:

Collect Variables In order to determine which variables, and their types, to have existentially quantified in the relation, we construct a list of the original variables, their types and new variables, which will be of the same type.

Assumptions We need to decide which assumption needs to be added to the relation. This is done based on the occurrence of any of the variables found above. That is, if an assumption contains any of the variables collected, then a version of this assumption, with suitable substitutions, should be included in the relation.

Build Relation We use the original formulae, but substitute with our new variables to build the string used to instantiate the Theorem of Co-induction.

Instantiate Theorem of Co-induction We then instantiate the theorem, using the relation just built and the original streams also.

Prove the Assumption Since the theorem says $R(s,t) \Rightarrow s = t$, we need to prove that the relation holds on the two original streams. However, since the relation is built based on pattern matching with the original streams, this is easily done using the "obvious" instantiation.

Prove that Relation is a Bisimulation Finally, we need to prove that the relation is a bisimulation. This is done by proving each of the statements on the car's being equal and the cdr's being related.

We see that most of the work of this strategy goes into setting up terms to be used in the instantiation of the theorem. In fact, the actual PVS proof does only the following: skolemize, introduce the theorem, instantiation (both the theorem, as explained above, and automatic instantiation provided by PVS), replacing terms and simplifying. Other than the instantiation of the theorem, none of these requires any particular insight, and indeed this part of the automation is very simple.

Example 2 (PVS proof of map_iter).
Consider again the statement

$$map(f, iter(f, x)) = iter(f, f(x))$$

where $f : \mathbb{R} \to \mathbb{R}$ and $x : \mathbb{R}$.

In PVS, we can prove this by using our strategy `bisim`:

```
map_iter :

{1}   FORALL (f: [real -> real], x: real):
        map(f, iter(f, x)) = iter(f, f(x))
```

Remember that `bisim` leaves us to prove only the two criteria for the relation being a bisimulation. So the two subgoals produced here correspond to the two parts of the definition of a bisimulation.

The first subgoal is then

```
map_iter.1 :

{1}   car(map(g, iter(g, y))) = car(iter(g, g(y)))
```

This is then completed by expanding *map* and *iter* to their definitions in terms of *corec* and then using rewrites about *car* and *corec*.

The second subgoal is

```
map_iter.2 :

{1}   EXISTS (g1: [real -> real], y1: real):
        cdr(map(g, iter(g, y))) = map(g1, iter(g1, y1))
         AND cdr(iter(g, g(y))) = iter(g1, g1(y1))
```

Again, we expand *map* and *iter* and use rewrites about *cdr* and *corec*. After this PVS is able to automatically guess the correct instantiation of $g1 = g$ and $y1 = g(y)$ to complete the proof. If we attempt to instantiate before the expansion, PVS will guess the wrong instantiation in this case, thus the expansion should take place first.

This completes the proof.

In both subgoals, new variables generated by the strategy are used. For the sake of readability we have substituted simpler names above.

4.2 Applicability of Strategy

It is clear that the strategy is quite basic, however it works quite well on typical examples of equalities which occur in for example Rutten [9]. There are some areas where the strategy does not perform so well:

Foundational Statements For some foundational statements, which are part of the basic definitions and properties for streams, it turns out that the generated relation is not a bisimulation. In these cases we often need a relation which is stronger than that provided by the strategy.

Streams as Sequences For streams or functions viewed as sequences (functions over the natural numbers), it is often more appropriate to use elementwise equality as the bisimulation, however this is not produced by the strategy.

Complicated Bisimulations In some cases, the relation is such that the proof that it is a bisimulation relies on other lemmas. In many cases, it is possible by systematic rewriting and simplification to present the goal to PVS in a form where PVS can make the correct instantiation, however this is not always possible. If instantiation by the user is needed, we then have a problem with the automatically generated variable names, since for a rerun of the proof, these will be different. Thus a proof constructed this way will not work if rerun.

Others There are some statements which are just not on the right form for this strategy to work. One example is the following:

```
double_zip : LEMMA double(s) = zip(s,s)
```

where $double(s)$ duplicates each element in s, that is $double(s) = s_0 :: s_0 :: s_1 :: s_1 :: \ldots$.
In this case, the relation needs to look two elements ahead rather than only one, something not accounted for in the strategy.

There are various ways we can try to alleviate the above problems. For example, the problem of the naming and renaming of fresh variables might be handled differently, allowing us to fix the names *locally* within each proof, and so prevent this particular problem of not being able to rerun proofs automatically. In fact, with the current version it is possible to get around this by using the *glass box* version of the `bisim` strategy. This stores as the proof the individual proof steps and thus keeps a record of the instantiation involving the variable names, keeping them the same for any subsequent reruns of the proof. For statements where a straight forward proof by co-induction is not feasible, we can set up other, related proof strategies. It turns out that the lemma `double_zip` is can be proven using what we might call *even-odd co-induction*: Essentially, if we have a bisimulation between the even elements of two streams and a bisimulation between the odd elements, then the two streams are equal. We have proven this principle in PVS and used it to prove `double_zip` in a manner very similar to that of using `bisim`. However, `bisim` does not at this time handle such proofs. Based on our study of Rutten's examples [9] it seems that it may be beneficial to support automatic proof using element-wise equality as the bisimulation. In many cases using this would overly complicate the proof, essentially converting to a sequence notation for the streams, however in some cases it works well, particularly where the stream (or any functions) involved are defined by considering streams as sequences.

We will be using the first-order differential equation from (1) as an example, and want to prove that $s = const(a)$, when $s = fode(s, a)$. In any given proof, we may have not only the actual equality that we are trying to prove, but also other equalities for the streams, which we may use as assumptions in our proof. If we have assumptions of the form $s = s1(x_0, \ldots, x_n)$ (and similarly for t), we take as the general bisimulation the following function:

$$\lambda s2, t2 : \exists x, y, s1, t1 : s2 = s \wedge s = s1(x) \wedge t2 = t \wedge t = t1(y)$$

For our example, this becomes

$$\lambda s2, t2 : \exists a, s1 : s2 = s1 \wedge s1 = fode(s1, a) \wedge t2 = const(a)$$

as there are no extra assumption on $const(a)$. In this particular case, it looks like the term $s2 = s1$ is not necessary, however this method is for the general case where $s1$ is a compound term, in which case this extra term is needed.

After having found the "correct" relation to use, the co-induction theorem gives us the equality between the streams. However, we still need to prove that the relation is indeed a bisimulation. Of course, any bisimulation would work, but for the sake of completing the proof as painlessly as possible, it makes sense to choose one which is not too difficult to handle. The method outlined above for finding a bisimulation does not necessarily lead to the smartest, simplest or indeed the most intuitive bisimulation, but so far it has proven to work quite well.

Let us consider a more complex example using PVS. In this case, we are looking at the second-order differential stream equation from Sect. 3.3. After some initial steps, we have the following proof goal to work on:

```
{-1}  FORALL (B: (bisimulation?), x: stream, y: stream):
                      B(x, y) => x = y
[-2]  s1!1 = sode(s1!1, a!1, b!1)
  |-------
[1]   s1!1 = zip(const(a!1), const(b!1))
```

We need to prove that formulae -1 and -2 implies formula 1.

We can now instantiate formula -1 according to the rules set out above:

```
(inst -1 "lambda (s2,t2) : exists (s1,a,b) :
          s2 = s1 and s1 = sode(s1,a,b) and
                      t2 = zip(const(a),const(b))"
          "s1!1" "zip(const(a!1),const(b!1))")
```

This gives us the following goal:

```
{-1}  (EXISTS (s1, a, b):
         s1!1 = s1 AND
          s1 = sode(s1, a, b) AND
           zip(const(a!1), const(b!1)) = zip(const(a), const(b)))
       => s1!1 = zip(const(a!1), const(b!1))
[-2]  s1!1 = sode(s1!1, a!1, b!1)
  |-------
[1]   s1!1 = zip(const(a!1), const(b!1))
```

where formula -1 has as a consequent exactly the thing we are trying to prove in formula 1, and the assumption in formula -1 is trivially true if we choose the obvious instantiations for s1, a and b, something which PVS will do for us automatically.

The other part of the proof is then that the relation we entered before must be a bisimulation:

```
[-1]  s1!1 = sode(s1!1, a!1, b!1)
  |-------
{1}   bisimulation?(LAMBDA (s2, t2):
                      EXISTS (s1, a, b):
                        s2 = s1 AND
                         s1 = sode(s1, a, b) AND
                          t2 = zip(const(a), const(b)))
[2]   s1!1 = zip(const(a!1), const(b!1))
```

The proof of this is a bit longer and more tedious, however no great insight is required, so it is now quite simple. Thus using the method of guessing a bisimulation has helped us complete the proof.

5 Conclusions

One of the main discussion points in this project has been which underlying datatype to use for the streams: functions over natural numbers or a co-inductive datatype. Of course, using only infinite streams, the two datatypes are equivalent, but due to the datatypes and the support for definitions and proofs, the choice does matter. Having tried both, we have concluded that using the co-inductive datatype leads to more natural definitions (once past the initial less pretty automatically generated constructors) and proofs.

We have shown that with our implementation, we can define and solve stream differential equations amongst other things. In general, stream calculus is simply, but elegant and thus provides a very neat application for mechanised reasoning.

One of the main issues of the use of an implementation in a theorem prover is automation, since this has a big impact on efficiency and accessibility. We have addressed this through our implementation of the strategy `bisim`, which for many equalities guess the correct bisimulation to use for a proof by co-induction. It is clear that our strategy works well for a large class of equations over streams, however we have also identified some classes where it fails and given suggestions for possible solutions in these cases.

The next stage of our project is to do a case study of a signal flow graph, for instance one modelling a filter. This would show the strength of the implementation as a tool for formal analysis of a very practical application of signal flow graphs. PVS already has extensive libraries for classical functional analysis, and we intend to also connect the notion of streams as representations of Taylor series expansions to the existing analysis libraries in PVS.

6 Acknowledgements

We would like to thank César Muñoz for his assistance with the initial version of the strategy `bisim`, Sam Owre for help in understanding how the co-inductive datatypes in PVS work. Thanks also to Paul Miner for his insight on using co-induction in general and on the usefulness of the function *corec* in particular.

References

1. Boulton, R., Gottliebsen, H., Hardy, R., Kelsey, T., Martin, U.: Design verification for control engineering. In Eerke Boiten, J.D..G.S., ed.: Integrated Formal Methods, 4th International Conference, IFM 2004, Canterbury, UK, April 4-7, 2004, Proceedings. Volume 2999 of Lecture Notes in Computer Science., Springer-Verlag (2004) 21–35
2. Gottliebsen, H.: A PVS implementation of stream calculus for signal flow graphs. In: Theorem Proving in Higher Order Logics: Emerging Trends Proceedings. (2005)
3. Martin, U., Mathiesen, E.A., Oliva, P.: Hoare logic in the abstract. In Ésik, Z., ed.: Computer Science Logic, 20th International Workshop, CSL 2006, 15th Annual Conference of the EACSL, Szeged, Hungary, September 25–29, 2006, Proceedings. Volume 4207 of Lecture Notes in Computer Science., Springer-Verlag (2006) 501–515
4. Mason, S.J.: Feedback theory – some properties of signal flow graphs. In: Proceedings of the I.R.E. Volume 41., IEEE (1953) 1144–1156
5. Miner, P.S.: Hardware verification using coinductive assertions. PhD thesis, Indiana University (1998)
6. Owre, S., Rushby, J.M., , Shankar, N.: PVS: A prototype verification system. In Kapur, D., ed.: 11th International Conference on Automated Deduction (CADE). Volume 607 of Lecture Notes in Artificial Intelligence., Saratoga, NY, Springer-Verlag (June 1992) 748–752
7. Pavlovic, D., Escardo, M.H.: Calculus in coinductive form. In: Proceedings of the 13th Annual IEEE Symposium on Logic in Computer Science. (1998) 408–417
8. Ramadoss, R., Bushnell, M.L.: Test generation for mixed-signal devices using signal flow graphs. In: Proceedings of the 9th International Conference on VLSI Design, IEEE Computer Society (1996) 242–248
9. Rutten, J.: An application of coinductive stream calculus to signal flow graphs (2003)
10. Samadi, S., Nishihara, A., Iwakura, H.: Filter-generating systems. IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing **47**(3) (March 2000) 214–221
11. Schmid, H.: Circuit transposition using signal flow graphs. In: IEEE International Symposium on Circuits and Systems, ISCAS 2002. Volume 2., IEEE Computer Society (2002) 25–28