HAL open science

Using Structural Recursion for Corecursion

Yves Bertot, Ekaterina Komendantskaya

To cite this version:

Yves Bertot, Ekaterina Komendantskaya. Using Structural Recursion for Corecursion. 2008. inria-00322331v3

HAL Id: inria-00322331

https://inria.hal.science/inria-00322331v3

Preprint submitted on 22 Sep 2008 (v3), last revised 23 Mar 2009 (v4)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Using Structural Recursion for Corecursion

Yves Bertot[1] and Ekaterina Komendantskaya[2]

[1] INRIA Sophia Antipolis, France
Yves.Bertot@inria.fr
[2] INRIA Sophia Antipolis, France
ekaterina.komendantskaya@inria.fr *

Abstract. We address the problem of constructing stream values defined by recursive equations that do not respect directly the "guardedness" condition, and we concentrate in particular on equations where recursive calls appear under functions. We use a correspondence between streams and functions over natural numbers to show that some classes of non-guarded definitions can still be modelled through the encoding as structural recursive functions. In practice, this work extends the class of stream values that can be defined in a constructive type theory-based theorem prover with inductive and coinductive types, structural recursion and guarded corecursion.
Key words: Constructive Type Theory, Structural Recursion, Coinductive types, Corecursion, Coq

1 Introduction

Interactive theorem provers with inductive types [25, 26, 18] provide a restricted programming language together with a formal meta-theory for reasoning about the language. This language is very close to functional programming languages, so that the verification of a program in a conventional functional programming language can often be viewed as a simple matter of adapting the program's formulation to a theorem prover's syntax, thus obtaining a faithful prover-level model. Then one can reason about this model in the theorem prover. This approach has inspired studies of a large collection of algorithms, starting from simple examples like sorting algorithms to more complex algorithms, like the ones used in the computation of Gröbner bases, the verification of the four-colour theorem, or compilers.

However, the prover's programming language is restricted, especially concerning recursion. For instance, *structural* restriction ensures that all programs terminate, so that values are never undefined; we give details in Section 2. Approaches to cope with potentially non-terminating programs are available, especially by encoding domain theory as in HOLCF [22], but these approaches tend to make the description of programs more cumbersome, because the exceptional case where a computation may not terminate needs to be covered at every stage. An alternative is to manage wider class of terminating functions, mainly using

* The authors thank

well-founded recursion [20, 24], and this approach is now widely spread among all interactive theorem provers.

A few theorem provers [25, 26, 18] also support coinduction. Coinductive datatypes provide a way to look at infinite data objects. In particular, streams of data can be viewed as infinite lists. Coinductive datatypes also provide room for a new class of recursive objects, known as corecursive objects.

Termination is not required anymore for these functions, but termination still plays a role, since every finite value should still be computable in finite time, even if the computation involves an interaction with a corecursive value. This constraint boils down to a concept of *productivity*. Roughly speaking, infinite sequences of recursive calls where no data is being produced should be avoided. For recursive programs, productivity is undecidable for the same reason that termination is. For this reason, a more restrictive criterion is used to describe corecursive functions that are legitimate in theorem provers.

A theorem prover like Coq provides two kinds of recursion: terminating recursion, initially based on structural recursion for inductive types, and later extended to well-founded recursion, which is a simple extension of structural recursion; and productive corecursion, based on "guarded" corecursion [9, 14]. Efforts have been made to extend the basic guarded corecursion in the same spirit that well-founded recursion extends the basic structural recursion. We can mention [13] and [2, 5], which basically incorporate well-founded recursion to make sure several non-productive recursive calls are allowed as long as they ultimately become productive.

The problem we address in this paper has a different nature: the stream appears to be both produced and consumed in the recursive definition, and we need to make sure that no data is consumed before it can be produced. More precisely, the guardedness condition imposes that (a) each corecursive call is made under at least one constructor; and (b) if the recursive call is under a constructor, it does not appear as an argument of any function. The work cited in [13, 2, 5] addresses recursive definitions that violate the former condition. However, there are many interesting recursive equations that violate the latter condition. The infinite sequences of Fibonacci numbers (considered e.g., in [1]) and of natural numbers (see Example 5) are famous representatives of the class. Many of the corecursive values studied, for example, in [23, 11, 12] fail to satisfy the second guardedness condition. A very elegant method of lazy differentiation [16] also gives rise to a function of multiplication for infinite lists of derivatives that fails to satisfy the second guardedness condition. In Section 3 we take a closer look at the functions of this kind and the two guardedness conditions.

In this paper, we provide solutions to non-guarded cases when recursive calls occur as arguments to functions like `map`, which produces a stream of values by applying a given function to all elements of an input stream. The equations we are interested in are the ones where the argument of the `map` function is precisely the value being defined. A simple example is the following recursive equation (studied later as Example 5):

```
nats = 1::map S nats
```

A quick analysis shows that we can use this equation to infer the value of each element in the stream: the first value is given directly, the second element is obtained from the first one through the behaviour of the `map` function, and so on. This recursive equation is a legitimate specification of a stream, and it can actually be used as a definition in a conventional lazy functional programming language like Haskell.

Thus the question studied in this article is: given a recursive equation like the one concerning `nats`, can we build a corecursive value that satisfies this equation, using only structural recursion and guarded corecursion? We will describe a partial solution to this problem. We will also show that this solution can incorporate other interfering functions than `map`.

Our proposed approach is to map every stream value with a function over natural numbers in a reversible way: a stream $s_0{::}s_1{::}\cdots$ is mapped to the function $[\![s]\!] : i \mapsto s_i$, and the reverse map is an easily defined guarded corecursive function. It appears that all legitimate guarded corecursive values are mapped to structurally recursive functions and that the question of productivity is transformed into a question of termination. We discuss it in Section 4.

Moreover, uses of the `map` function and similar operations are transformed into program fragments that still respect the constraints of structural recursion. Thus, there are stream values whose recursive definitions as streams are mapped to structural recursive definitions, even though the initial equations did not respect guardedness constraints. For these stream values, we propose to define the corresponding recursive function using structural recursion, and then to produce the stream value using the reverse map from functions over natural numbers to streams. We present this method in Section 5.

The attractive features of this approach is that it is systematic and simple. It appears to be simpler than, e.g., related work done in [5, 10] that involved introduction of particular coinductive types and/or manipulation with ad-hoc predicates. Although the current state of our experiments relies on manual operations, we believe the approach can be automated in the near future, yielding a command in the same spirit as the `Function` command of Coq recent versions.

2 Structurally Recursive Functions

We start with defining the notions of inductive and coinductive types, and recursive/corecursive functions. We will use the syntax of Coq throughout. For a more detailed introduction to Coq, see [3]. The related work was done in HOL and mechanised using Isabelle [21], and in Martin-Löf type theory and formalised in AGDA [25].

Inductive data types are defined by introducing a few basic constructors that generate elements of the new type.

Definition 1. *The definition of the inductive type of natural numbers is built using two constructors `O` and `S`:*

```
Inductive nat : Set := O : nat | S : nat -> nat.
```

This definition also implies that the type supports both pattern-matching and recursion: on the one hand, all values in the type are either of the form `O` or of the form `(S x)`; on the other hand, all values are finite and a function is well defined when its value on `O` is given and the value for `S x` can be computed from the value for `x`.

After the inductive type is defined, one can define its inhabitants and functions on it. Most functions defined on the inductive type must be defined recursively, that is, by describing values for different patterns of the constructors and by allowing calls to the same function on variables taken from the patterns.

Example 1. The recursive function below computes the n-th Fibonacci number.

```
Fixpoint fib (n:nat) : nat :=
  match n with
  | O => 1
  | S O => 1
  | S (S p as q) => fib p + fib q
  end.
```

A *structurally recursive* definition is such that every recursive call is performed on a structurally smaller argument. In this way we can be sure that the recursion terminates.

This function `fib` is *structurally recursive*: all recursive calls are made on variables (here `p` and `q`) that were obtained through pattern-matching from the initial argument.

There are many useful functions and algorithms that are not structurally recursive, but general recursive. They are not accepted by Coq or similar proof assistants directly. See e.g. [3, 7] for examples and surveys of methods that allow to implement such functions in constructive type theory.

3 Guardedness

It was observed in [15], that induction gives rise to initial algebras, while coinduction gives rise to final coalgebras; and the basic duality between algebras and coalgebras can be expressed as *construction* versus *observation*. In practice, observation is provided by pattern-matching, construction is provided by the constructors. However, the initiality property of initial algebras is a direct justification for structural recursion and the finality property of final coalgebras is a direct justification for guarded corecursion.

The following is the definition of a coinductive type of infinite streams, built using one constructor `Cons`.

Definition 2. *The type of streams is given by*

```
CoInductive Stream (A:Set) : Set := Cons: A -> str A ->  str A.
```

In the rest of this paper, we will write `a::tl` for `Cons _ a tl`, leaving the first argument of `Cons` to be inferred from the context.

While a structurally recursive function is supposed to rely on an inductive type of its domain and is restricted in the way recursive calls are using this input, a corecursive function is supposed to rely on a co-inductive type for its co-domain and is restricted in the way recursive calls are used for producing the output.

Definition 3 (Guardedness). *A position in an expression is* pre-guarded *if it occurs as the root of the expression, or if it is a direct sub-term of a pattern-matching construct or a conditional statement, which is itself in a pre-guarded position.*

A position is guarded if it occurs as a direct sub-term of a constructor for the co-inductive type that is being defined and if this constructor occurs in a pre-guarded position or a guarded position. A corecursive function is guarded *if all its corecursive calls occur in guarded positions.*

A corecursive definition is guarded *if all recursive calls occur in guarded positions with respect to the definition's right-hand side.*

Example 2. The coinductive function `map` applies some given function `f` to a given infinite stream.

```
CoFixpoint map (A B :Set)(f: A -> B)(s: Stream A): Stream B :=
match s with x::s' => f x::map A B f s' end.
```

In this definition's right-hand side, the match construct and the expression `f x::...` are in pre-guarded positions, the expression `map A B f s'` is in guarded position, the definition is guarded.

Example 3. The coinductive function `nums` takes as argument a natural number n and produces a stream of natural numbers starting from n.

```
CoFixpoint nums (n: nat): str nat := n::nums (S n).
```

In this definition's right-hand side, the expression `n::nums (S n)` is in a pre-guarded position, the expression `nums (S n)` is in a guarded position.

Example 4. The following function `zipWith` is guarded:

```
CoFixpoint zipWith (A B C: Set)(f: A -> B -> C)
     (s: Stream A)(t: Stream B) : Stream C :=
match (s, t) with  (x :: s',  y :: t')  =>
(f x y):: (zipWith A B C f s' t')
end.
```

Informally speaking, the guardedness condition insures that

* each corecursive call is made under at least one constructor;

** if the recursive call is under a constructor, it does not appear as an argument of any function.

Violation of any of these two conditions makes a function non-guarded. According to the two guardedness conditions above, we will be talking about the two classes of non-guarded functions - (*) and (**).

A more subtle analysis of the corecursive functions that fail to satisfy the guardedness condition *, can be found in [2, 19, 5]. In particular, the mentioned papers offer a solution to the problem of formalising productive corecursive functions of this kind. We could briefly mention here one famous function in this family - a *filter* function that, given a stream `s` and a property `P`, outputs the stream of those values from `s` that satisfy `P`.

Till the rest of the paper, we shall restrict our attention to the other class of functions.

Example 5. Consider the following equation:

```
nats = 1::map S nats
```

This definition is not guarded, the expression `map S nats` occurs in a guarded position, but `nats` is not; see the guardedness condition **. Despite of this, the value `nats` is well-defined.

Example 6. The following function computes the stream of Fibonacci numbers:

```
fib =  0 ::  1 :: (zipWith nat nat plus (tl fib) fib).
```

Again, this recursive equation fails to satisfy **.

The next example shows the function `dTimes` that multiplies the sequences of derivatives in the elegant method of lazy differentiation; [16, 8].

Example 7.
```
dTimes x y =  match x, y with
  | x0 :: x', y0 :: y' =>
  (x0 * y0) :: (zipWith Z Z plus (dTimes x'  y)  (dTimes x y'))
   end.
```

Again, this function fails to satisfy **.

In the next section, we will develop a method that makes it possible to describe these three functions as guarded corecursive values.

Values in co-inductive types usually cannot be observed as a whole, because of their infiniteness. To prove some properties of infinite streams, we use the method of observation. For example, to prove that the two lists are *bisimilar*, we must observe that their first elements are the same, and continue the process with the next.

Definition 4. *Bisimilarity is expressed in the definition of the following coinductive type:*

```
CoInductive EqSt:  Stream A -> Stream A -> Prop :=
| eqst : forall (a : A) (s s' : str A), EqSt s s' ->
                     EqSt (a::s)(a::s').
```

In the rest of this paper, we will write a==b for `EqSt` a b. The definition of a==b corresponds to the conventional notion of bisimilarity as given, e.g. in [15]. Lemmas and theorems analogous to the *coinductive proof principle* of [15] are proved in Coq and can be found in [3].

Bisimilarity expresses that two streams are observationally equal. Very often, we will only be able to prove this form of equality, but for most purposes this will be sufficient.

4 Recursive Analysis of Corecursive Functions

In this section, we show that streams can be replaced with functions. Because there are a wide variety of techniques to define functions, this will make it possible to increase the class of functions that we can reason about. Our approach will be to start from a recursive equation known to describe a stream, to transform systematically this recursive equation into a recursive equation for a function which will happen to be structurally recursive, and then to transform this function back into a stream using a guarded corecursive scheme.

The first step is to transform an arbitrary stream into a function over natural numbers. This is simply done by relying on the `nth` function[3].

Definition 5. *The function* `nth` *is defined as follows:*

```
Fixpoint nth (A:Type) (n:nat) (l: Stream A) {struct n}: A :=
match l with  a :: l' =>
  match n with | O =>  a | S p => nth A p l' end
end.
```

It is a regular structural recursive function with structural argument n. In the following we will note $[\![s]\!]$ to represent the functional view of the stream s. In practice, $[\![s]\!]$ will be extensionally equal to `fun n => nth n` s.

Definition 6. *Given a function f over natural numbers, it can be transformed into a stream using the following function:*

```
Cofixpoint stroff (A:Type)(f:nat->Type) : Stream A :=
  f O :: stroff A (fun x => f (1+x)).
```

This definition is guarded by constructors. In the rest of this paper, we will write $\langle s \rangle$ for `stroff _` s leaving the first argument of `stroff` to be inferred from the context.

Thus, the direct interpretation of a stream as a function is given by the function `nth`. This function can be inverted using the function $\langle \cdot \rangle$. It is easy to prove that $[\![\cdot]\!]$ and $\langle \cdot \rangle$ are inverse of each other. In particular, we have the following two lemmas.

Lemma 1. *For any function* `f` *over natural numbers, we have the following statement:*

[3] In Coq pre-defined library of streams, this function is called `Str_nth`.

$\forall\ f\ n$, `nth` $n\ \langle f \rangle$ `=` $f\ n$.

Lemma 2. *For any stream* `s`, `s` $= \langle [\![s]\!] \rangle$.

Proof. Proofs for both lemmas are done in Coq and available in [4].

When observing the interaction of $[\![\cdot]\!]$ with other constructs, we establish that we can transform stream expressions in the following recursive manner:

1. $[\![$`e`$]\!]$ `= e` if `e` contains no objects of type `Stream A` or if it is a locally bound variable;
2. The head `hd s` of the stream `s` is interpreted as follows: $[\![$`hd s`$]\!]$ `=` $[\![$`s`$]\!]$ `0`;
3. For the tail `tl s` of the stream `s`, we put $[\![$`tl s`$]\!]$ `= fun n =>` $[\![$`s`$]\!]$ `(1+n)`.
4. $[\![$`a b`$]\!]$ `=` $[\![$`a`$]\!]$ $[\![$`b`$]\!]$;
5. $[\![$`fun x => e`$]\!]$ `= fun x =>` $[\![$`e`$]\!]$;
6. $[\![$`match e with` p_1 `=>` e_1 `| ... end`$]\!]$ `= match` $[\![$`e`$]\!]$ `with` p_1 `=>` $[\![e_1]\!]$ `| ... end`, if e does not have type `Stream A`;
7. Pattern-matching constructs on streams should be reformulated using `hd` and `tl`.
8. Finally, the stream `s = a :: s'` receives the following interpretation:

   $[\![$`a::s'`$]\!]$ `= fun n => match n with 0 => a | S p =>` $[\![$`s'`$]\!]$ `p end.`

Table 1. Rules for recursive transformation of streams.

Having these rules, we can infer new recursive functions from given streams. We show how this method works on several examples.

Example 8 (zeroes). Consider the recursive guarded function `zeroes`:

```
CoFixpoint zeroes := 0 :: zeroes.
```

According to item 8 in Table 1, we can model the body of this corecursive definition as follows:

`fun n => match n with 0 => 0 | S p =>` $[\![$`zeroes`$]\!]$ `p end.`

Now, we can prove that $[\![$`zeroes`$]\!]$ is equal to this expression. This can be expressed by a legitimate structurally recursive definition for a function that maps any natural number to zero. Thus, it is the simple way to obtain a model for a stream that contains only zeros.

Example 9. Consider the following reformulation of `map` from Example 2:

```
CoFixpoint map (f: A -> B)(s: str A): str B :=
 f (hd s)::map f (tl s).
```

Note that, following the item 7 in Table 1, we have switched to the head/tail representation of the matching construct for `s`.

We give its step-by-step translation into a recursive function using instructions from Table 1. By item 8 we have:

```
⟦map f s⟧ = fun n => match n with
| 0 => f ⟦hd s⟧
| S p => ⟦map (f (tl s))⟧ p
end.
```

By item 6, the pattern matching construct above need not to be transformed any further, neither does the "`fun n`" peace of code, according to item 5. By items 2, 4, we transform the above function into:

```
⟦map⟧ ⟦f⟧ ⟦s⟧ = fun n => match n with
| 0 => f  (⟦s⟧ 0)
| S p => ⟦map⟧ ⟦f⟧ ⟦tl s⟧ p
end.
```

We apply items 3, 4 to the third line and item 1 to the variables f and e.

```
⟦map⟧ f s = fun n => match n with
| 0 => f  (s 0)
| S p => ⟦map⟧ f (fun x => s (1+x)) p
end.
```

The resulting function is a legitimate definition for structural recursion on the third argument. Remarkably, `map` and ⟦`map`⟧ yield the following two lemmas. These lemmas will play an important role when we formalise corecursive definitions of streams where `map` interferes with the guardedness constraints.

Lemma 3 (Form-shifting lemmas).

$\forall\ f\ s\ n$, `nth` n `(map` f s`) =` f `(nth` n s`)`

$\forall\ f\ s$, ⟦`map`⟧ f s n `=` f `(`s n`).`

Proof. See [4].

Thanks to these lemmas, `s` can be moved from an argument position to an active function position, as needed for verifying structural recursion.

Finally, we show the same formalisation for the function `zipWith`:

Example 10 (Zip).

Reformulation of the function `zipWith` from example 4 with `tl` and `hd` gives

```
CoFixpoint zipWith (A B C: Set)(f: A -> B -> C)
  (s1: Stream A)(s2: Stream B): str C :=
 f (hd s1)(hd s2)::zipWith A B C f (tl s1) (tl s2).
```

Applying the rules from Table 1 to this definition yields the following recursive equation:

```
⟦zipWith⟧ f s1 s2 = fun n =>
match n with
| 0 => f (s1 0) (s2 0)
| S p => ⟦zipWith⟧ f (fun x => s1 (1+x))(fun x => s2 (1+x)) p
end.
```

Here again, the result is a legitimate recursive definition for structural recursion on the fourth argument.

For `zipWith` we also have two form-shifting lemmas:

Lemma 4 (Form-shifting lemmas).

$\forall\ f\ s_1\ s_2\ n$, `nth` n `(zipWith` $f\ s_1\ s_2$`) = f (nth` $n\ s_1$`) (nth` $n\ s_2$`)`

$\forall$ `n, ⟦zipWith⟧ f` $\mathtt{s}_1\ \mathtt{s}_2$ `n = f (`$\mathtt{s}_1$ `n) (`$\mathtt{s}_2$ `n).`

Proof. See [4].

These lemmas also move `s1` and `s2` from argument position to function position.

5 Satisfying Non-Guarded Recursive Equations

Form-shifting lemmas play a role when studying recursive equations to define streams that do not satisfy the guardedness condition **. To handle these functions, we simply need to add one new rule to the Table 1, which will handle occurrences of each function that has a form-shifting lemma.

9. If `F` is a function, `C` is the context in which arguments of `F` appear, and the Form-shifting lemma for `F` and `C` has the form $\forall a_1\ \cdots\ a_k\ s_1\ \cdots\ s_l\ n, [\![\mathtt{F}]\!] a_1\ \cdots\ a_k\ s_1\ \cdots\ s_l n = C[a_1, \ldots, a_k, s_1\ n, \ldots, s_l\ n]$, then ⟦ `F` $\mathtt{a}_1\ \mathtt{a}_2\ \mathtt{s}_1\ \mathtt{s}_2$ `n⟧ = C[⟦`$\mathtt{a}_1$`⟧,...,` ⟦$\mathtt{a}_k$⟧, ⟦$\mathtt{s}_1$⟧ `n,...,⟦`$\mathtt{s}_l$⟧ `n]`

Table 2. Rule for recursive transformation of non-guarded streams

In this manner, many recursive non-guarded definitions of streams based on `map` or `zipWith` will be transformed into recursive definitions of functions that satisfy structural recursion criteria. Thus, given a non-guarded corecursive definition, we define a systematic method of recovering a structural recursive function that can be in its turn converted into a stream using $\langle\cdot\rangle$. Table 3 describes this procedure, and [4] contains a Coq tactic `str_eqn_tac` that formalises the item (d) of Table 3. So, the tactic can be used automatically for all similar functions.

We illustrate this method using our running examples.

(a) Perform translation of the left-hand side of the equation following the construction of the Tables 1 and 2 and Form-shifting lemmas.
(b) Obtain a structurally recursive model of the non-guarded function.
(c) Use the function $\langle\cdot\rangle$ to obtain the corresponding stream value.
(d) Prove that this function satisfies initial recursive equation, using bisimilarity as the equality relation. This proof is done in three steps:
    1. First use a Coq `Streams` package theorem (`ntheq_eqst`) that shows that two streams are bisimilar if and only the nth elements of the two streams are equal for any n.
    2. Then perform a proof by induction using the specific induction theorem attached to the recursive function used to model the stream.
    3. Then systematically rewrite with the form-shifting and lemmas that express the interaction between `nth`, $\langle\cdot\rangle$, `hd` and `tl`.

Table 3. Formalisation of non-guarded streams

Example 11. Consider the corecursive non-guarded function `nats` from Example 5. The recursive model of `nats` is obtained by applying rules from Table 1:

```
⟦nats⟧ = fun n => if n = 0 then 1 else ⟦map S  ⟦nats⟧⟧ (n - 1).
```

Including also the Form-shifting lemma for `map` and the rule of Table 2, we get the following definition:

```
⟦nats⟧ = fun n => if n = 0 then 1 else S ⟦nats⟧ (n - 1).
```

This is now a legitimate structurally recursive equation, with which we obtain, by Definition 6, the function $\langle$⟦`nats`⟧$\rangle$ and then prove that `nats ==`$\langle$⟦`nats`⟧$\rangle$. The next step is to show that `nats` satisfies the equation of Example 5.

```
nats == 1::map S nats
```

Using the theorem `ntheq_eqst` and Lemma 1 this reduces to the following statement:

∀ n, ⟦`nats`⟧ n `= nth` n `(1::map S` $\langle$⟦`nats`⟧$\rangle$`)`

We can now prove this statement by induction using the induction principle that can be derived from the recursive definition of ⟦`nats`⟧. This gives two cases, which reduce to the following statements:

```
1 = nth 0 (1::map S ⟦nats⟧)
```

```
S (⟦nats⟧ p) = nth (S p) (1 :: map S ⟨⟦nats⟧⟩)
```

The first goal is provable as a direct consequence of the definition of `nth`. For the second goal, the left hand-side reduces as follows, again according to the definition of `nth`.

```
S (⟦nats⟧ p) = nth p (map S ⟨⟦nats⟧⟩)
```

Rewriting with the Form-shift lemma for `map` yields the following goal:

```
S (⟦nats⟧ p) = S (nth p ⟨⟦nats⟧⟩)
```

Rewriting with Lemma 1 yields the following trivial equality.

```
S (⟦nats⟧ p) = S (⟦nats⟧ p)
```

The proof is fully handled by the tactic `str_eqn_tac`, see [4].

Example 12. The sequence of Fibonacci numbers can be defined by the following equation:

```
fib = 1::1::zipWith plus fib (tl fib)
```

When processing the left-hand side of this equation using the rules from Tables 1, 2 and the Form-shifting lemma for `zipWith`, we obtain the following code:

```
⟦fib⟧ =
 fun n =>
 match n with
 | 0 => 1
 | S p =>
   match p with 0 => 1 | S q => (⟦fib⟧ q) (⟦fib⟧(1+q)) end
 end
```

This is still not accepted as a legitimate structurally recursive equation, because `(1+q)` is not a variable term, however it is semantically equivalent to `p`, and the following text is accepted:

```
⟦fib⟧ =
 fun n =>
 match n with
 | 0 => 1
 | S p =>
   match p with 0 => 1 | S q => ⟦fib⟧ q + ⟦fib⟧ p end
 end
```

Again, by Definition 6, we can define a stream `fib == ⟨⟦fib⟧⟩`, and `fib` is proved to satisfy the initial recursive equation automatically, using only the tactic `str_eqn_tac`; [4].

This model has positive and negative aspects. On the positive side, it is satisfactory that we can prove that the stream `fib` satisfies the original recursive equation. On the negative side, this model does not have the same computational complexity as the intended stream: computations of `fib` are repeated over and over so that a rough estimate gives an exponential complexity to obtain the nth element of the stream. Actually, the more efficient stream defined as `fib' 1 1` with the following more efficient and guarded definition can also be shown to satisfy the recursive equation:

```
CoFixpoint fib' (n m:nat) := n::fib' m (n+m).
```

However, we currently have not designed a systematic approach to derive the definition of `fib'` from the original recursive equation.

Finally, we illustrate the work of this method on the function `dTimes` from Example 7:

Example 13. We recover the structurally recursive function ⟦`dTimes`⟧ from Example 7, using rules from Tables 1 and 2:

```
⟦dTimes⟧ (x y:Stream nat) (n:nat){struct n} =
  match x, y, n with
  | x0 :: x', y0 :: y', 0 => x0 * y0
  | x0 :: x', y0 :: y', S p => (⟦dTimes⟧ x' y p) + (⟦dTimes⟧ x y' p)
  end.
```

Note that, unlike the `nats` and `fib` examples, here the two arguments to ⟦`dTimes`⟧ are of coinductive type. So, the method is quite flexible with respect to the input data types of the non-guarded corecursive function we consider.

It remains to define the stream ⟨⟦`dTimes`⟧⟩, which is a straightforward application of Definition 6, and to prove that it satisfy the initial recursive equation from Example 7. The proof is handled automatically by the tactic `str_eqn_tac`; [4].

6 Conclusions

There are several interesting conclusions that one can make from the method below. First of all, the method is very simple, and is based solely on the duality between recursive and corecursive functions, without constructing additional auxiliary coinductive types or ad-hoc predicates as was done (e.g.) in [5, 10].

The practical outcome of this work is to provide an approach to model more corecursive values than the ones that are directly accepted by the "guarded-by-constructors" criterion. With this approach we can address formal verification for a wider class of functional programming languages. The work presented here is complementary to the work presented in [5], since the method in that paper only considers definitions where recursive calls occur outside of any constructor, while the method in this paper considers definitions where recursive calls are made inside constructors and inside interfering functions.

Although the current state of our experiments relies on manual operations, we believe the approach can be automated in the near future, yielding a command in the same spirit as the `Function` command of Coq recent versions.

The Coq system also provides a mechanism known as extraction which produces values in conventional functional programming languages. When it comes to producing code for the solution of one of our recursive equations on streams, we have the choice of using the recursive equation as a definition, or the extracted code corresponding to the structurally recursive model. We suggest that

the initial recursive equation, which was used as our specification, should be used as the definition, because the structural recursive value may not respect the intended computational complexity. This was visible in the model we produced for the Fibonacci sequence, which does not take advantage of the value re-use as described in the recursive equation. We still need to investigate whether using the specification instead of the code will be sound with respect to the extracted code.

The method presented here is still very limited: it can only cope with recursive definitions of streams where each value is computed using values at a fixed distance in recursively defined streams: in the `nats` stream, the value at rank n is computed using the value at rank $n-1$, in the `fibs` stream, the value at rank n is computed using the values at rank $n-1$ and $n-2$, in the `dTimes` streams, the value at rank n are computed using values at rank $n-1$, even if these values are taken from other streams. The encoding fails for programs where the value at rank n is computed using values at rank $n-p$, where p is arbitrary large. An example of such a programs is the example of the Hamming sequence, as proposed in [11]. A recursive definition of this stream is:

```
H = 1::merge (map (Zmult 2) H) (map (Zmult 3) H)
```

In this definition, `merge` is the function that takes two streams and produces a new stream by always taking the least element of the two streams: when the input streams are ordered, the output stream is an ordered enumeration of all values in both streams. Such a `merge` function is easily defined by guarded co-recursion, but it interferes in the definition of `H` in the same way that `map` interfered in our previous examples. We have a solution for defining `H`, actually inspired from the "recursion-by-iteration" technique exposed in [3], but we cannot expose this solution here, for lack of space.

It would be desirable to analyse further how the practical methods we suggested here relate to the existing categoroical view on duality between coalgebraic and algebraic aspects of computations following [6, 17].

References

1. A. Abel. *Type-Based Termination. A Polymorphic Lambda-Calculus with Sized Higher-Order Types.* PhD thesis, Fakültat für Mathematik, Informatik und Statistik der Ludwig-Maximilians-Universität München, 2006.
2. Y. Bertot. Filters and co-inductive streams, an application to eratosthenes' sieve. In *TLCA*, volume 3461 of *LNCS*, pages 102 – 115. Springer Verlag, 2005.
3. Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development, Coq'Art: the Calculus of Constructions.* Springer-Verlag, 2004.
4. Y. Bertot and E. Komendantskaya. Experiments on using structural recursion for corecursion: Coq code, 2008. http://hal.inria.fr/inria-00322331/en/.
5. Y. Bertot and E. Komendantskaya. Inductive and coinductive components of corecursive functions in coq. *Electr. Notes Theor. Comput. Sci.*, 203(5):25–47, 2008.
6. M. Bidoit, R. Hennicker, and A. Kurz. On the duality between observability and reachability. In *FoSSaCS*, pages 72–87, 2001.

7. A. Bove. *General Recursion in Type Theory*. PhD thesis, Department of Computing Science, Chalmers University of Technology, 2002.
8. S. Bronson. Posting to coq club. codata: problem with guardedness condition?, 30 June 2008. http://pauillac.inria.fr/pipermail/coq-club/2008/003783.html.
9. T. Coquand. Infinite objects in type theory. In *Types for Proofs and Programs, Int. Workshop TYPES'93*, volume 806 of *LNCS*, pages 62–78. Springer-Verlag, 1994.
10. N. A. Danielsson. Posting to coq club. codata: problem with guardedness condition?; An ad-hoc approach to productive definitions. 1,4 Aug 2008. http://pauillac.inria.fr/pipermail/coq-club/2008/003859.html and http://sneezy.cs.nott.ac.uk/fplunch/weblog/?p=109.
11. E. W. Dijkstra. Hamming's exercise in SASL, June 1981. circulated privately.
12. J. Endrullis, C. Grabmayer, D. Hendriks, A. Isihara, and J. W. Klop. Productivity of stream definitions. In *FCT*, pages 274–287, 2007.
13. P. D. Gianantonio and M. Miculan. A unifying approach to recursive and corecursive definitions. In *TYPES*, pages 148–161, 2002.
14. E. Gimenez. *Un Calcul de Constructions Infinies et son Application a la Verification des Systemes Communcants*. PhD thesis, Laboratoire de l'Informatique du Parallelism, Ecole Normale Superiere de Lyon, 1996.
15. B. Jacobs and J. Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 62:222 – 259, 1997.
16. J. Karczmarczuk. Functional differentiation of computer programs. *Higher-Order and Symbolic Computation*, 14(1):35–57, 2001.
17. G. Malcolm. Behavioural equivalence, bisimulation, and minimal realisation. In *COMPASS/ADT*, pages 359–378, 1995.
18. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.
19. M. Niqui. Coinductive field of exact real numbers and general corecursion. In *Proc. of CMCS'06*, volume 164 of *ENTCS*, pages 121–139. ELSEVIER, 2006.
20. B. Nordström. Terminating general recursion. *BIT*, 28:605–619, 1988.
21. L. Paulson. Mechanizing coinduction and corecursion in higher-order logic. *Logic and Computation*, 2(7):175 – 204, 1997.
22. F. Regensburger. Holcf: Higher order logic of computable functions. In *In Theorem Proving in Higher Order Logics, volume 971 of LNCS*, pages 293–307. Springer-Verlag, 1995.
23. B. Sijtsma. On the productivity of recursive list definitions. *ACM Transactions on Programing Languages and Systems*, 11(4):633 – 649, 1989.
24. K. Slind. Function definition in higher order logic. In *Theorem Proving in Higher Order Logics*, volume 1125 of *Lecture Notes in Computer Science*. Sprinter Verlag, Aug. 1996.
25. The Agda Development Team. The agda reference manual. http://appserv.cs.chalmers.se/users/ulfn/wiki/agda.php.
26. The Coq Development Team. The coq proof assistant reference manual. http://coq.inria.fr.