Using Structural Recursion for Corecursion

Yves Bertot[1] and Ekaterina Komendantskaya[2]

[1] INRIA Sophia Antipolis, France
Yves.Bertot@inria.fr
[2] School of Computer Science, University of St Andrews, UK
ek@cs.st-andrews.ac.uk *

Abstract. We propose a (limited) solution to the problem of constructing stream values defined by recursive equations that do not respect the guardedness condition. The guardedness condition is imposed on definitions of corecursive functions in Coq, AGDA, and other higher-order proof assistants. In this paper, we concentrate in particular on those non-guarded 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 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, Guarded Corecursion, Coq

1 Introduction

Interactive theorem provers with inductive types [27, 28, 20, 16] 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 [24], but these approaches tend

* Work is partially supported by the INRIA CORDI post-doctoral program, the ANR project "A3Pat" ANR-05-BLAN-0146 and by EPSRC postdoctoral grant EP/F044046/1.

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 a larger class of terminating functions, mainly using well-founded recursion [22, 26], and this approach is now widely spread among all interactive theorem provers.

A few theorem provers [27, 28, 20] 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 must 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, which can also handle well-founded recursion; and productive corecursion, based on "guarded" corecursion [10, 15]. 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 [14] and [4, 7], which basically incorporate well-founded recursion to make sure several non-productive recursive calls are allowed as long as they ultimately become productive. In particular, [14] introduces a generalization of the concept of well-founded relation that uses an extra dimension to cover at the same time recursive or co-recursive functions; since there is an extra dimension, two notions of limits can be used and recursive values can mix terminating recursive and productive co-recursive aspects in a seamless fashion.

One essential characteristic of well-founded induction and the complete ordered families of equivalences in [14] is that the well-founded relation or families of equivalences must be given as extra data to make it possible to start the definition process. In the alternative approach described in this paper, we want to avoid this extra burden imposed on the user, and we attempt to develop a methodology that remains syntactic in nature.

We will concentrate on a class of recursive definitions where *mapping* functions interfere in the recursive equation, thus preventing the recursive equation to be recognised as *guarded by constructors*. 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 [25, 12, 13] fail to satisfy the guardedness condition, precisely because functions like `map` interfere in the recursive definition. A very elegant method of lazy differentiation [19] also gives rise to a function of multiplication for infinite streams of derivatives in the same class of definitions.

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`. In Section 3, we briefly overview the class of the functions we target.

Our proposed approach is to map every stream value to 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 5.

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 6.

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 [5]. One can also handle inductive and coinductive types within HOL (proof assistant Isabelle) [23], and within Martin-Löf type theory (proof assistant AGDA) [27].

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.
```

There is one important property we wish every function defined in Coq to possess: it is termination. To guarantee this, Coq uses a syntactic restriction on definitions of functions, called *structural recursion*. A *structurally recursive* definition is such that every recursive call is performed on a structurally smaller argument. The 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, but they can be defined using various forms of well-founded induction or induction with respect to a *predicate* [5, 8].

It is perhaps worth mentioning that there exists an approach to termination called "type-based termination" [1, 3, 17]. The essence of different methods proposed under this name is rejection of the structural recursion as being a too restrictive and narrow method for guaranteeing termination. Instead, this job is delegated to sized higher-order types. The type-based termination promises to be a powerful tool, but it is not easy to implement it. As for today, the major proof assistants still rely on structural recursion. Some non-guarded functions we formalise in this paper, can also be handled by methods of type-based termination. However, yet it gives little from the point of view of practical programming and automated proving. Therefore the value of this paper, as well as (e.g.) [5, 7, 8] is in the technical elegance and practical implementation in the existing proof assistants.

3 Guardedness

We now consider 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 -> Stream A -> Stream A.
```

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

While a structurally recursive function is supposed to rely on an inductive type for 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.*

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

```
CoFixpoint map (A B :Type)(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, and 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): Stream 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 [14, 4, 21, 7]. In particular, the mentioned papers offer a solution to the problem of formalising productive corecursive functions of this kind.

Till the rest of the paper, we shall restrict our attention to the second class of functions. To the extent of our knowledge, this paper is the first attempt to systematically formulate the functions of this class in the language of a higher-order proof assistant with guarded corecursion.

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 definition describes the stream of Fibonacci numbers:

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

Again, this recursive equation fails to satisfy **.

Example 7. The next example shows the function `dTimes` that multiplies the sequences of derivatives in the elegant method of lazy differentiation of [19, 9].

```
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 express Examples 5 - 7 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 a method of observation. For example, to prove that the two streams are *bisimilar*, we must observe that their first elements are the same, and continue the process with the rest.

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' : Stream 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 [18]. Lemmas and theorems analogous to the *coinductive proof principle* of [18] are proved in Coq and can be found in [5].

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 Soundness of recursive transformations for streams

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

As a first step, we observe how to construct a stream from a function over natural numbers:

Definition 5. *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 0 :: 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 argument `A` to be inferred from the context.

The function `stroff` has a natural inverse, the function `nth` which returns the element of a stream at a given rank:

Definition 6. *The function* `nth`[3] *is defined as follows:*

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

In the rest of this paper, we will omit the first argument (the type argument) of `nth`, following Coq's approach to implicit arguments. We will use notation $[\![s]\!]$ when talking about `(fun n => nth n s)`.

It is easy to prove that $[\![\cdot]\!]$ and $\langle \cdot \rangle$ are inverse of each other. Composing these two functions is the essence of the method we develop here. The lemmas below are essential for guaranteeing the soundness of our method.

[3] In Coq's library, this function is defined under the name `Str_nth`.

Lemma 1. *For any function f over natural numbers,* $\forall n,$ `nth` $n\ \langle f\rangle$ `=` $f\ n$.

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

Proof. Both proofs are done in Coq and available in [6].

We now want to describe a transformation for (non-guarded) recursive equations for streams. A recursive equation for a stream would normally have the form

$$a = e \tag{1}$$

where both a and e are streams, and a can also occur in the expression e; see Examples 5 - 7. We use this initial non-guarded equation to formulate a guarded equation for a of the form:

$$a = \langle e' \rangle \tag{2}$$

where e' is a function extensionally equivalent to $[\![e]\!]$. As we show later in this section, we often need to evaluate `nth` only partially or only at a certain depth, this is why the job cannot be fully delegated to `nth`.

The definition of e' will have the form

$$e'\ n = E \tag{3}$$

where e' can again occur in the expression E.

Example 8 (zeroes). For simple examples, we can go through steps (1)-(3) intuitively. Consider the corecursive guarded definition of a stream `zeroes` that contains an infinite repetition of `0`.

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

We can model the body of this corecursive definition as follows:

```
Fixpoint nzeroes (n:nat) : nat :=
  match n with 0 => 0 | S p => nzeroes p end.
```

This is a legitimate structurally recursive definition for a function that maps any natural number to zero. Note that the obtained function is extensionally equal to $[\![$`zeroes`$]\!]$.

```
Lemma nth_zeroes: forall n, nth n zeroes = nzeroes n.
```

Thus, a stream that is bisimilar to `zeroes` can be obtained by the following commands:

```
Definition zeroes' := stroff _ nzeroes.
```

By Lemma `nth_zeroes` and Lemma 2, `zeroes` and `zeroes'` are bismilar, see [6] for a proof.

The main issue is to describe a systematic transformation from the expression e in the equation 1 to the expression E in the equation (3). This "recursive" part of the work will be the main focus of the next section.

5 Recursive Analysis of Corecursive Functions

We continue to systematise the steps (1)-(3) of the transformation for a recursive equation $a = e$.

The expression e can be seen as the application of a function F to a. In this sense, the recursive definition of a expresses that a is fixpoint of F. The type of F is $\texttt{Stream}\ A \rightarrow \texttt{Stream}\ A$ for some type A. We will derive a new function F' of type $(\texttt{nat} \rightarrow A) \rightarrow (\texttt{nat} \rightarrow A)$; the recursive function a' that we want to define is a fixed point of F'. We obtain F' from F in two stages:

Step 1. We compose F on the left with $\langle\cdot\rangle$ and on the right with $[\![\cdot]\!]$. This naturally yields a new function of the required type. In practice, we do not use an explicit composition function, but perform the syntactic replacement of the formal parameter with the $\langle\cdot\rangle$ expression everywhere.

Example 9. For instance, when considering the `zeroes` example, the initial function

```
Definition zeroes_F (zeroes:Stream nat) := 0::zeroes
```

is recursively transformed into the function:

```
Definition zeroes_F' (nzeroes : nat -> nat) :=
  nth n (0::stroff nzeroes).
```

The corecursive value we consider may be a function taking arguments in types $t_1, \ldots, t_n$, that is, the function F may actually be defined as a function of type $(t_1 \rightarrow \cdots \rightarrow t_n \rightarrow \texttt{Stream}A) \rightarrow (t_1 \rightarrow \cdots \rightarrow t_n \rightarrow \texttt{Stream}A)$. The reformulated function F' that is obtained after composition with $\langle\cdot\rangle$ and $[\![\cdot]\!]$ has the corresponding type where $\texttt{Stream}\ A$ is replaced with $\texttt{nat} \rightarrow A$. Thus, it is the first argument that incurs a type modification. When one of the types t_i is itself a stream type, we can choose to leave it as a stream type, or we can choose to replace it also with a function type. When replacing t_i with a function type, we have to add compositions with $[\![\cdot]\!]$ and $\langle\cdot\rangle$ at all positions where the first argument f of F is used, to express that the argument of f at the rank i must be converted from a stream type to a function type and at all positions where the argument of the rank $i + 1$ of F is used, to express that this argument must be converted from a function type to a stream type.

We choose to perform this transformation of a stream argument into a function argument only when the function being defined is later used for another recursive definition. In this paper, this happens only for the functions `map` and `zipWith`.

Example 10. Consider the function `map` from Example 2. The function F for this case has the following form:

```
Definition map_F
 (map : forall (A B:Type)(f: A -> B), Stream A -> Stream B) :=
 fun A B f s => match s with a::s' => f a::map A B f s' end.
```

The fourth argument to `map` and the fifth argument to `map_F` have type `Stream A` and we choose to replace this type with a function type. We obtain the following new function:

```
Definition map_F'
  (map : forall (A B:Type)(f:A -> B), (nat -> A) -> nat -> B :=
fun A B f s => ⟦match ⟨s⟩ with a::s' => f a::⟨map A B f ⟦s'⟧⟩ end⟧.
```

Step 2. We go on transforming the body of F' according to rewriting rules that express the interaction between $\langle\cdot\rangle$, $[\![\cdot]\!]$, and the usual functions and constructs that deal with streams.

The Table 1 gives a summary of the rewriting rules for the transformation.

1. $\texttt{nth}\ n\ \langle f\rangle = f\ n$,
2. $\texttt{nth}\ n\ (a\texttt{::}s') = \texttt{match}\ n\ \texttt{with}\ 0\ \texttt{=>}\ a\ \texttt{|}\ \texttt{S}\ p\ \texttt{=>}\ \texttt{nth}\ p\ s'\ \texttt{end}$,
3. $\texttt{hd}\ \langle f\rangle = f\ 0$,
4. $\texttt{tl}\ \langle f\rangle = \langle\texttt{fun}\ n\ \texttt{=>}\ f\ (\texttt{S}\ n)\rangle$,
5. $\texttt{match}\ \langle f\rangle\ \texttt{with}\ a\texttt{::}s\ \texttt{=>}\ e\ a\ s\ \texttt{end} = e\ (f\ 0)\ \langle\texttt{fun}\ n\ \texttt{=>}\ f\ (\texttt{S}\ n)\rangle$,
6. β-reduction.

Table 1. Transformation rules for function representations of streams.

All these rules can be proved as theorems in the theorem prover [6]: this guarantees soundness of our approach. However, this kind of rewriting cannot be done directly inside the theorem prover, since rewriting can only be done while proving statements, while we are in the process of defining a function. Moreover, the rewriting operations must be done thoroughly, even inside lambda-abstraction, even though an operation for that may not be supported by the theorem prover (for instance, in the calculus of constructions as it is implemented in Coq, rewriting does not occur inside abstractions).

The rewriting rules make the second argument of `nth` decrease. When the recursive stream definition is guarded, this process ends with a structural function definition.

Example 11. Let us continue with the definition for `map`.

```
map_F' map A B f s n =
  nth n match ⟨s⟩ with a:: s' => f a::⟨map A B f ⟦s'⟧⟩end
= nth n (f(s 0)::⟨map A B f ⟦s'⟧⟩) end
= match n with
    0 => f(s 0) | S p => nth p ⟨map A B f ⟦⟨fun n => s (S n)⟩⟧⟩
  end
= match n with 0=>f(s 0)|S p=>map A B f ⟦⟨fun n => s (S n)⟩⟧ p end
= match n with 0=>f(s 0)|S p=>map A B f (fun n => s (S n)) p end
```

When considered as the body for a recursive definition of a function `map'`, the last right-hand side is a good structural recursive definition with respect to the initial parameter `n`. We can use this for a structural definition:

```
Fixpoint map' (A B : Type) (f : A -> B) (s: nat -> A)
     (n : nat) {struct n} :=
 match n with 0 => f a | S p => map' A B f (fun n => s (S n)) p end.
```

This function models the `map` function on streams, as a function on functions. It enjoys a particular property, which plays a central role in this paper:

Lemma 3 (Form-shifting lemmas).

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

$\forall\ f\ \ s\ \ n$, $[\![\texttt{map}]\!]$ $f\ \ s\ \ n$ = f $(s\ \ n)$.

Proof. See [6].

Thanks to the second statement of the lemma, s can be moved from an argument position to an active function position, as will later be needed for verifying structural recursion of other values relying on `map`.

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

Example 12 (Zip). The function `zipWith` can also be transformed, with the choice that both stream arguments are transformed into functions over natural numbers.

```
Definition zipWith_F
  (zipWith : forall (A B C : Type), (A -> B -> C) ->
                              Stream A -> Stream B -> Stream C)
  (A B C : Type)(f : A -> B -> C)(a : Stream A)(b : Stream B) :=
  match a, b with
    x :: a', y :: b' => f x y :: zipWith A B C f a' b'
  end.
```

Viewing arguments `a` and `b` as functions and applying the rules from Table 1 to this definition yields the following recursive equation:

```
zipWith_F' zipwith' A B C f a b n =
  match n with
    0 => f (a 0) (b 0)
  | S p => zipwith' (fun n => a (S n)) (fun n => b (S n)) p
  end
```

Here again, this leads to a legitimate structural recursive definition on the fourth argument of type `nat`. We also have form-shifting lemmas:

Lemma 4 (Form-shifting lemmas).

$$\forall\ f\ s_1\ s_2\ n,\ \texttt{nth}\ n\ (\texttt{zipWith}\ f\ s_1\ s_2) = \texttt{f}\ (\texttt{nth}\ n\ s_1)\ (\texttt{nth}\ n\ s_2)$$

$$\forall\ f\ s_1\ s_2\ n,\ [\![\texttt{zipWith}]\!]\ f\ s_1\ s_2\ n = f\ (s_1\ n)\ (s_2\ n).$$

Proof. See [6].

The second statement also moves s_1 and s_2 from argument position to function position.

Unfortunately, we do not know the way to automatically discover the Form-shifting lemmas; although the statements of these lemmas follow the same generic pattern and once stated, the proofs for them do not tend to be difficult. Instead, as we illustrate in the Conclusion, we sometimes can give a convincing argument showing that a form-shifting lemma for a particular function cannot be found; and this provides an evidence that our method is not applicable to this function. That is, existence or non/existence of the form-shifting lemmas can serve as a criterium for determining whether the function can be covered by the method.

6 Satisfying Non-Guarded Recursive Equations

Form-shifting lemmas play a role when studying recursive equations that do not satisfy the guardedness condition **, that is, when the corecursive call is made under functions like `map` or `zipWith`. To handle these functions, we simply need to add one new rule, as in Table 2, which will handle occurrences of each function that has a form-shifting lemma.

7. Let `f` be a function and `C` be a context in which arguments of `F` appear. If a form shifting lemma has the following shape:

   $$\forall a_1\ \cdots\ a_k\ s_1\ \cdots\ s_l\ n, [\![\texttt{f}]\!] a_1\ \cdots\ a_k\ s_1\ \cdots\ s_l\ n = C[a_1, \ldots, a_k, s_1\ n, \ldots, s_l\ n],$$

   then this equation should be used as an extra rewriting rule.

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

The extended set of transformation rules from Tables 1 and 2 can now be used to produce functional definitions of streams that were initially defined by non-guarded corecursive equations. The technique is as follows:

(a) Translate the equation's right-hand-side as prescribed by the rules in Tables 1 and 2,
(b) Use the equation as a recursive definition for a function,
(c) Use the function $\langle\cdot\rangle$ to obtain the corresponding stream value,

(d) Prove that this stream satisfies the initial recursive equation, using bisimilarity as the equality relation.

For the last step concerning the proof, we rely on two features provided in the Coq setting:

– For each recursive definition, the Coq system can generate a specialised induction principle, as described in [2],
– A proof that two streams are bisimilar can be transformed into a proof that their functional views are extensionally equal, using the theorem `ntheq_eqst`:

```
ntheq_eqst :
  ∀A (s1 s2:Stream A), (∀n, nth n s1 = nth n s2) -> s1 == s2
```

Using these two theorems and combining them with systematic rewriting with Lemma 1 and the form-shifting lemmas, we actually obtain a tactic we called `str_eqn_tac` in [6] which proves the recursive equations automatically.

We illustrate this method using our running examples.

Example 13. Consider the corecursive non-guarded definition of `nats` from Example 5. Here is the initial equation

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

After applying all transformation rules we obtain the following equation between functions:

$[\![\texttt{nats}]\!]$ `= fun n => if n = 0 then 1 else S (`$[\![\texttt{nats}]\!]$` (n - 1)).`

This is now a legitimate structurally recursive equation defining $[\![\texttt{nats}]\!]$, from which we define `nats` as `nats = `$\langle[\![\texttt{nats}]\!]\rangle$. 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 on the left-hand-side this reduces to the following statement:

$\forall\ n,\ [\![\texttt{nats}]\!]\ n = \texttt{nth}\ n\ \texttt{(1::map S}\ \langle[\![\texttt{nats}]\!]\rangle\texttt{)}$

We can now prove this statement by induction on the structure of the function $[\![\texttt{nats}]\!]$, as explained in [2]. This gives two cases:

$\texttt{1 = nth 0 (1::map S}\ [\![\texttt{nats}]\!]\texttt{)}$

$\texttt{S (}[\![\texttt{nats}]\!]\texttt{ p) = nth (S p) (1 :: map S}\ \langle[\![\texttt{nats}]\!]\rangle\texttt{)}$

The first goal is a direct consequence of the definition of `nth`. The second goal reduces as follows:

$\texttt{S (}[\![\texttt{nats}]\!]\texttt{ p) = nth p (map S}\ \langle[\![\texttt{nats}]\!]\rangle\texttt{)}$

Rewriting with the first form-shifting lemma for `map` yields the following goal:

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

Rewriting again with Lemma 1 yields the following trivial equality.

```
S (⟦nats⟧ p) = S (⟦nats⟧ p).
```

Example 14. 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 by the Coq system 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 5, we can define a stream `fib = ⟨⟦fib⟧⟩`, and `fib` is proved to satisfy the initial recursive equation automatically.

It is satisfactory that we have a systematic method to produce a stream value for the defining recursive equation, but we should be aware that the implementation of `fib` through a structural recursive function does not respect the intended behaviour and has a much worse complexity —exponential— while the initial equation can be implemented using lazy data-structures and have linear complexity.

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

Example 15. For the function `dTimes`, we choose to leave the two stream arguments `x` and `y` as streams. We recover the structurally recursive function `⟦dTimes⟧` from Example 7:

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

It remains to define the stream $\langle[\![\texttt{dTimes}]\!]\rangle$, which is a straightforward application of Definition 5, and to prove that it satisfies the initial recursive equation from Example 7. In [6], the proof is again handled automatically.

7 Conclusions

The practical outcome of this work is to provide an approach to model corecursive values that are not directly accepted by the "guarded-by-constructors" criterion, without relying on more advanced concepts like well-founded recursion of ordered families of equivalences. 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 [7], 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 inside constructors, but also inside interfering functions.

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 [14, 7, 11] that involved introducing particular coinductive types and manipulating 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.

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 cannot cope with the example of the Hamming sequence, as proposed in [12]. 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 corecursion, but `merge` interferes in the definition of `H` in the same way that `map` interfered in our previous examples. This time, we do not have any good form-shifting lemma for this function. The hamming sequence can probably be defined using the techniques of [14] and we were also able to find another syntactic approach for this example, this new approach is a subject for another paper.

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. G. Barthe and P. Courtieu. Efficient Reasoning about Executable Specifications in Coq. In V. Carreño, C. Muñoz, and S. Tahar, editors, *Proceedings of TPHOLs'02*, volume 2410 of *LNCS*, pages 31–46. Springer-Verlag, 2002.
3. G. Barthe, M. J. Frade, E. Giménez, L. Pinto, and T. Uustalu. Type-based termination of recursive definitions. *Mathematical Structures in Computer Science*, 14:97–141, Feb. 2004.
4. Y. Bertot. Filters and co-inductive streams, an application to Eratosthenes' sieve. In *TLCA*, volume 3461 of *LNCS*, pages 102 – 115. Springer-Verlag, 2005.
5. Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development, Coq'Art: the Calculus of Constructions.* Springer-Verlag, 2004.
6. Y. Bertot and E. Komendantskaya. Experiments on using structural recursion for corecursion: Coq code, 2008. http://hal.inria.fr/inria-00322331/.
7. Y. Bertot and E. Komendantskaya. Inductive and coinductive components of corecursive functions in Coq. *Electr. Notes Theor. Comput. Sci.*, 203(5):25–47, 2008.
8. A. Bove. *General Recursion in Type Theory.* PhD thesis, Department of Computing Science, Chalmers University of Technology, 2002.
9. S. Bronson. Posting to Coq club. Codata: problem with guardedness condition?, 30 June 2008. http://pauillac.inria.fr/pipermail/coq-club/2008/003783.html.
10. 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.
11. 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.
12. E. W. Dijkstra. Hamming's exercise in SASL, June 1981. circulated privately.
13. J. Endrullis, C. Grabmayer, D. Hendriks, A. Isihara, and J. W. Klop. Productivity of stream definitions. In *Fundamentals of Computation Theory*, volume 4639 of *LNCS*, pages 274–287. Springer-Verlag, 2007.
14. P. D. Gianantonio and M. Miculan. A unifying approach to recursive and corecursive definitions. In *TYPES*, pages 148–161, 2002.
15. E. Giménez. *Un Calcul de Constructions Infinies et son Application à la Vérification des Systèmes Communicants.* PhD thesis, Laboratoire de l'Informatique du Parallélisme, Ecole Normale Supérieure de Lyon, 1996.
16. M. J. C. Gordon and T. F. Melham. *Introduction to HOL : a theorem proving environment for higher-order logic.* Cambridge University Press, 1993.
17. J. Hughes, L. Pareto, and A. Sabry. Proving the correctness of reactive systems using sized types. In *POPL*, pages 410–423, 1996.
18. B. Jacobs and J. Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 62:222 – 259, 1997.
19. J. Karczmarczuk. Functional differentiation of computer programs. *Higher-Order and Symbolic Computation*, 14(1):35–57, 2001.

20. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
21. 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.
22. B. Nordström. Terminating general recursion. *BIT*, 28:605–619, 1988.
23. L. Paulson. Mechanizing coinduction and corecursion in higher-order logic. *Logic and Computation*, 2(7):175 – 204, 1997.
24. 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.
25. B. Sijtsma. On the productivity of recursive list definitions. *ACM Transactions on Programing Languages and Systems*, 11(4):633 – 649, 1989.
26. K. Slind. Function definition in higher order logic. In *Theorem Proving in Higher Order Logics*, volume 1125 of *LNCS*. Springer-Verlag, Aug. 1996.
27. The Agda Development Team. The agda reference manual. http://appserv.cs.chalmers.se/users/ulfn/wiki/agda.php.
28. The Coq Development Team. The Coq proof assistant reference manual. http://coq.inria.fr.