

Beating the Productivity Checker Using Embedded Languages

Nils Anders Danielsson

University of Nottingham

Abstract. Some total languages, like Agda and Coq, allow the use of guarded corecursion to construct infinite values and proofs. Guarded corecursion is a form of recursion in which arbitrary recursive calls are allowed, as long as they are guarded by a coinductive constructor. Guardedness ensures that programs are productive, i.e. that every finite prefix of an infinite value can be computed in finite time. However, many productive programs are not guarded, and it can be nontrivial to put them in guarded form.

This paper gives a method for turning a productive program into a guarded program. The method amounts to defining a problem-specific language as a data type, writing the program in the problem-specific language, and writing a guarded interpreter for this language.

1 Introduction

Coinduction and corecursion are useful techniques for defining and reasoning about things which are potentially infinite, including streams and other (potentially) infinite data types (Coquand 1994; Giménez 1996; Turner 2004), process congruences (Milner 1990), congruences for functional programs (Gordon 1999), and semantics for divergence of programs (Cousot and Cousot 1992; Hughes and Moran 1995; Leroy and Grall 2009; Nakata and Uustalu 2009).

When working with infinite values in a total setting it is common to require *productivity*: even though a value is conceptually infinite, it should always be possible to compute the next unit of information in finite time. The primitive methods for defining infinite values in the proof assistants Agda and Coq are based on *guarded corecursion* (Coquand 1994), which is a conservative approximation of productivity. The basic idea of guarded corecursion is that “corecursive calls” may only take place under guarding constructors, thus ensuring that the next unit of information—the next constructor—can always be computed. For instance, consider the following definition of $nats_{\geq n}$, the stream of natural numbers greater than or equal to n ($_{::}$ is the cons constructor for streams):

$$\begin{aligned}nats_{\geq} &: \mathbb{N} \rightarrow Stream \mathbb{N} \\nats_{\geq} n &= n :: nats_{\geq} (\text{succ } n)\end{aligned}$$

This definition is guarded, and has the property that the next natural number can always be computed in finite time. As another example, consider *bad*:

```
bad : Stream N
bad = tail (zero :: bad)
```

This definition is not guarded (due to the presence of *tail*), nor is it productive: *bad* is not well-defined. Finally consider the following definition of the stream of natural numbers:

```
nats : Stream N
nats = zero :: map suc nats
```

This definition is productive, but unfortunately it is not guarded, because *map* is not a constructor. In fact, many productive definitions are not guarded, and it can be nontrivial to find an equivalent guarded definition.

The main contribution of this paper is a technique for turning a large class of productive but unguarded definitions into guarded definitions. The basic observation of the technique is that many productive definitions would be guarded if some functions were actually constructors. For instance, if *map* were a constructor, then *nats* would be guarded. The technique then amounts to defining a problem-specific language as a data type which includes a constructor for every function like *map*, implementing the productive definitions in a guarded way using this language, and implementing a guarded interpreter for the language. Optionally one can also prove that the resulting definitions satisfy their intended defining equations.

The technique relies on the use of data types defined using mixed induction and coinduction (see Sect. 2.4), so it requires a programming language with support for such definitions. The examples in the paper have been implemented using Agda (Norell 2007; Agda Team 2010), a dependently typed, total¹ functional programming language with good support for mixed induction and coinduction. The supporting source code is at the time of writing available to download (Danielsson 2010).

The rest of the paper is structured as follows: Section 2 discusses induction and coinduction in the context of Agda, Sects. 3–7 introduce the language-based approach to productivity through a number of examples, Sect. 8 discusses related work, and Sect. 9 concludes.

2 Induction and Coinduction

This section gives a brief introduction to induction and coinduction, with an emphasis on how these concepts are realised in Agda.² For more formal accounts of induction and coinduction see, for instance, the theses of Hagino (1987) and Mendler (1988). For a more detailed discussion of mixed induction and coinduction in Agda, see Danielsson and Altenkirch (2009a).

¹ Agda is an experimental system with neither a formalised meta-theory nor a verified type checker, so take phrases such as “total” with a grain of salt.

² This section with background material is very similar to a section in a paper by Danielsson and Altenkirch (2009b).

2.1 Induction

Let us start with a simple inductive definition. In Agda the type of *finite* lists can be defined as follows:

```
data List (A : Set) : Set where
  []   : List A
  _::_ : A → List A → List A
```

This states that $List\ A$ is a type (or Set) with two constructors, $[]$ of type $List\ A$ and $_::_$ of type $A \rightarrow List\ A \rightarrow List\ A$. The constructor $_::_$ is an infix operator; the underscores mark the argument positions. The type $List\ A$ is isomorphic to the least fixpoint $\mu X. 1 + A \times X$ in the category of types and total functions.³

Agda has a termination checker which ensures that all code is terminating (or productive, see below). The termination checker allows lists to be destructured using structural recursion:

```
mapList : {A B : Set} → (A → B) → List A → List B
mapList f []           = []
mapList f (x :: xs) = f x :: mapList f xs
```

The use of braces in $\{A\ B\ : Set\} \rightarrow \dots$ means that the two type arguments A and B are *implicit*; they do not need to be given explicitly if Agda can infer them. (Note that in this context $A\ B$ is not an application, it is a sequence of variables.)

2.2 Coinduction

If we want to have infinite lists, or streams, we can use the following coinductive definition instead (note that constructors, such as $_::_$, can be overloaded in Agda):

```
data Stream (A : Set) : Set where
  _::∞ : A → ∞ (Stream A) → Stream A
```

The type $Stream\ A$ is isomorphic to the greatest fixpoint $\nu X. A \times X$. The type function $\infty : Set \rightarrow Set$ marks its argument as being coinductive. It is analogous to the suspension type constructors which are sometimes used to implement non-strictness in strict languages (Wadler et al. 1998). The type comes with a force function and a delay constructor:

```
! : {A : Set} → ∞ A → A           #_ : {A : Set} → A → ∞ A
```

The constructor $\#_$ is a tightly binding prefix operator. Ordinary function application binds tighter, though.

³ At the time of writing this is not exactly true in Agda (Danielsson and Altenkirch 2009a, Section 7.1), but the difference between $List\ A$ and the fixpoint is irrelevant for the purposes of this paper. Similar considerations apply to greatest fixpoints.

Values of coinductive types can be constructed using guarded corecursion:

$$\begin{aligned} \mathit{map} &: \{A\ B : \mathit{Set}\} \rightarrow (A \rightarrow B) \rightarrow \mathit{Stream}\ A \rightarrow \mathit{Stream}\ B \\ \mathit{map}\ f\ (x :: xs) &= f\ x :: \sharp\ \mathit{map}\ f\ (\flat\ xs) \end{aligned}$$

The definition of map is accepted by Agda's termination checker because the corecursive call is guarded by \sharp , without any non-constructor function between the left-hand side and the corecursive call.

2.3 Coinductive Relations

Let us now consider a coinductively defined *relation*: stream equality, or bisimilarity. Two streams are equal if they have identical heads and their tails are equal (coinductively):

$$\frac{\flat\ xs \approx \flat\ ys}{x :: xs \approx x :: ys} \quad (\text{coinductive})$$

This inference system can be represented using an indexed data type:

$$\begin{aligned} \mathbf{data}\ _ \approx _ &: \{A : \mathit{Set}\} : \mathit{Stream}\ A \rightarrow \mathit{Stream}\ A \rightarrow \mathit{Set}\ \mathbf{where} \\ _ :: _ &: (x : A) \{xs\ ys : \infty\ (\mathit{Stream}\ A)\} \rightarrow \infty\ (\flat\ xs \approx \flat\ ys) \rightarrow \\ & x :: xs \approx x :: ys \end{aligned}$$

Note the use of *dependent* function spaces $((x : A) \rightarrow B$ where x can occur in B) to set up dependencies of types on values. Note also that the elements of the type $xs \approx ys$ are *proofs* witnessing the equality of xs and ys , and that the termination checker ensures that these proofs are productive.

Elements of coinductively defined relations can be constructed using corecursion. As an example, let us prove the map-iterate property (Gibbons and Hutton 2005): $\mathit{map}\ f\ (\mathit{iterate}\ f\ x) \approx \mathit{iterate}\ f\ (f\ x)$. The function $\mathit{iterate}$ is defined corecursively:

$$\begin{aligned} \mathit{iterate} &: \{A : \mathit{Set}\} \rightarrow (A \rightarrow A) \rightarrow A \rightarrow \mathit{Stream}\ A \\ \mathit{iterate}\ f\ x &= x :: \sharp\ \mathit{iterate}\ f\ (f\ x) \end{aligned}$$

This function repeatedly applies a function to a seed element and collects the results in a stream; the unfolding of $\mathit{iterate}\ f\ x$ is $x :: \sharp\ (f\ x :: \sharp\ (f\ (f\ x) :: \dots))$. The map-iterate property can be proved by using guarded corecursion (the term *guarded coinduction* could also be used):

$$\begin{aligned} \mathit{map}\text{-}\mathit{iterate} &: \{A : \mathit{Set}\} (f : A \rightarrow A) (x : A) \rightarrow \\ & \mathit{map}\ f\ (\mathit{iterate}\ f\ x) \approx \mathit{iterate}\ f\ (f\ x) \\ \mathit{map}\text{-}\mathit{iterate}\ f\ x &= f\ x :: \sharp\ \mathit{map}\text{-}\mathit{iterate}\ f\ (f\ x) \end{aligned}$$

To see how this proof works, consider how it can be built up step by step (as in an interactive Agda session):

$map\text{-}iterate\ f\ x = ?$

The type of the *goal* $?$ is $map\ f\ (iterate\ f\ x) \approx iterate\ f\ (f\ x)$. Agda types should always be read up to normalisation, so this is equivalent to⁴

$f\ x :: \# map\ f\ (b\ (\# iterate\ f\ (f\ x))) \approx f\ x :: \# iterate\ f\ (f\ (f\ x))$.

(Note that normalisation does not involve reduction under $\#$, and that $b\ (\# x)$ reduces to x .) This type matches the result type of the equality constructor $_::-$, so we can refine the goal:

$map\text{-}iterate\ f\ x = f\ x :: ?$

The new goal type is $\infty (map\ f\ (iterate\ f\ (f\ x)) \approx iterate\ f\ (f\ (f\ x)))$, so the proof can be finished by an application of the coinductive hypothesis under the guarding constructor $\#$.

2.4 Mixed Induction and Coinduction

The types above are either inductive or coinductive. Let us now consider a type which uses both induction and coinduction. Hancock et al. (2009) define a language of stream processors, representing functions of type $Stream\ A \rightarrow Stream\ B$, using a nested fixpoint: $\nu Y.\mu X. B \times Y + (A \rightarrow X)$. We can represent this fixpoint in Agda as follows:

```
data SP (A B : Set) : Set where
  put : B → ∞ (SP A B) → SP A B
  get : (A → SP A B) → SP A B
```

The stream processor `put b sp` outputs b , and continues processing according to sp . The processor `get f` reads one element a from the input stream, and continues processing according to $f\ a$. In the case of `put` the recursive argument is coinductive, so it is fine to output an infinite number of elements, whereas in the case of `get` the recursive argument is inductive, which means that one can only read a finite number of elements before writing the next one. This ensures that the output stream can be generated productively.

We can implement a simple stream processor which copies the input to the output as follows:

```
copy : {A : Set} → SP A A
copy = get (λ a → put a (# copy))
```

This definition is guarded. Note that `copy` contains an infinite number of `get` constructors. This is fine, even though `get`'s argument is inductive, because there is never a stretch of infinitely many `get` constructors without an intervening

⁴ This is a simplification of the current behaviour of Agda.

delay constructor ($\#$). On the other hand, the following definition of a sink is not guarded, and is not accepted by Agda:

$$\begin{aligned} \text{sink} &: \{A B : \text{Set}\} \rightarrow SP\ A\ B \\ \text{sink} &= \text{get } (\lambda _ \rightarrow \text{sink}) \end{aligned}$$

We can also compute the semantics of a stream processor:

$$\begin{aligned} \llbracket _ \rrbracket &: \{A B : \text{Set}\} \rightarrow SP\ A\ B \rightarrow Stream\ A \rightarrow Stream\ B \\ \llbracket \text{put } b\ sp \rrbracket as &= b :: \# (\llbracket \text{ } sp \rrbracket as) \\ \llbracket \text{get } f \rrbracket (a :: as) &= \llbracket f\ a \rrbracket (\text{ } as) \end{aligned}$$

($\llbracket _ \rrbracket$ is a mixfix operator.) This definition uses a lexicographic combination of guarded corecursion and higher-order structural recursion. In the first clause the corecursive call is guarded. In the second clause it “preserves guardedness” (it takes place under zero coinductive constructors rather than one), and the first argument is structurally smaller.

Note that $\llbracket _ \rrbracket$ could not have been implemented if $SP\ A\ B$ had been defined purely coinductively (because then sink could be implemented with B equal to the empty type). By using both induction and coinduction in the definition we rule out certain stream processors which would otherwise have been accepted, and in return we can implement functions like $\llbracket _ \rrbracket$.

Before we continue note that, in order to reduce clutter, the declarations of implicit arguments have been omitted in the remainder of the paper.

3 Making Programs Guarded

As noted in the introduction guardedness is sometimes an inconvenient restriction: there are productive programs which are not syntactically guarded. This section introduces a language-based technique for putting definitions in guarded form.

Consider the following definition of the stream of Fibonacci numbers:

$$\begin{aligned} \text{fib} &: Stream\ \mathbb{N} \\ \text{fib} &= 0 :: \# \text{zipWith } _+ _ \text{fib } (1 :: \# \text{fib}) \end{aligned}$$

The definition uses the function zipWith , which combines two streams:

$$\begin{aligned} \text{zipWith} &: (A \rightarrow B \rightarrow C) \rightarrow Stream\ A \rightarrow Stream\ B \rightarrow Stream\ C \\ \text{zipWith } f\ (x :: xs)\ (y :: ys) &= f\ x\ y :: \# \text{zipWith } f\ (\text{ } xs)\ (\text{ } ys) \end{aligned}$$

While the definition of fib is productive, it is not guarded, because the function zipWith is not a constructor. If zipWith were a constructor the definition would be guarded, though, and this presents a way out: we can define a problem-specific language which includes zipWith as a constructor, and then define an interpreter for the language by using guarded corecursion.

A simple language of stream programs can be defined as follows:

data $Stream_P : Set \rightarrow Set_1$ **where**
 $_{::_} : A \rightarrow \infty (Stream_P A) \rightarrow Stream_P A$
 $zipWith : (A \rightarrow B \rightarrow C) \rightarrow Stream_P A \rightarrow Stream_P B \rightarrow Stream_P C$

(Set_1 is a type of large types; ∞ has type $Set_i \rightarrow Set_i$ for any i .) Note that the stream program argument of $_{::_}$ is coinductive, while the arguments of $zipWith$ are inductive; a stream program consisting of an infinitely deep application of $zipWith$ s would not be productive.

Stream programs will be turned into streams in two steps. First a kind of weak head normal form (WHNF) for stream programs is computed recursively, and then the resulting stream is computed corecursively. The WHNFs are defined in the following way:

data $Stream_W : Set \rightarrow Set_1$ **where**
 $_{::_} : A \rightarrow Stream_P A \rightarrow Stream_W A$

Note that the stream argument to $_{::_}$ is a (“suspended”) program, not a WHNF. The function $whnf$ which computes WHNFs can be defined by structural recursion:

$whnf : Stream_P A \rightarrow Stream_W A$
 $whnf (x :: xs) = x ::^b xs$
 $whnf (zipWith f xs ys) = zipWith_W f (whnf xs) (whnf ys)$

Here $zipWith_W$ is defined by simple case analysis:

$zipWith_W : (A \rightarrow B \rightarrow C) \rightarrow Stream_W A \rightarrow Stream_W B \rightarrow Stream_W C$
 $zipWith_W f (x :: xs) (y :: ys) = f x y :: zipWith f xs ys$

WHNFs can then be turned into streams corecursively:

$\llbracket _ \rrbracket_W : Stream_W A \rightarrow Stream A$ $\llbracket _ \rrbracket_P : Stream_P A \rightarrow Stream A$
 $\llbracket x :: xs \rrbracket_W = x ::^\# \llbracket xs \rrbracket_P$ $\llbracket xs \rrbracket_P = \llbracket whnf xs \rrbracket_W$

Note that this definition is guarded. (Agda accepts definitions like this one even though it is split up over two mutually defined functions.)

Given the language above we can now define the stream of Fibonacci numbers using guarded corecursion:

$fib_P : Stream_P \mathbb{N}$ $fib : Stream \mathbb{N}$
 $fib_P = 0 ::^\# zipWith _+_ fib_P (1 ::^\# fib_P)$ $fib = \llbracket fib_P \rrbracket_P$

One can prove that this definition satisfies the original equation for fib by first proving coinductively that $\llbracket _ \rrbracket_P$ is homomorphic with respect to $zipWith/zipWith$:

$zipWith-hom : (f : A \rightarrow B \rightarrow C) (xs : Stream A) (ys : Stream B) \rightarrow$
 $\llbracket zipWith f xs ys \rrbracket_P \approx zipWith f \llbracket xs \rrbracket_P \llbracket ys \rrbracket_P$
 $fib-correct : fib \approx 0 ::^\# zipWith _+_ fib (1 ::^\# fib)$

(The proofs are omitted.)

4 Several Types at Once

The technique introduced in Sect. 3 is not limited to streams. It should be obvious how to generalise it to other types. In fact, it can be used with several types at the same time.

Consider the following universe of streams, products and other types:

data $U : Set_1$ where	$El : U \rightarrow Set$
$stream : U \rightarrow U$	$El (stream\ a) = Stream (El\ a)$
$product : U \rightarrow U \rightarrow U$	$El (product\ a\ b) = El\ a \times El\ b$
$lift : Set \rightarrow U$	$El (lift\ A) = A$

The type U defines codes for elements of the universe, and El interprets these codes. By indexing the program and WHNF types by codes from this universe we can work with several types at once:

data $El_P : U \rightarrow Set_1$ where	
...	
$interleave : El_P (stream\ a) \rightarrow El_P (stream\ a) \rightarrow El_P (stream\ a)$	
$fst : El_P (product\ a\ b) \rightarrow El_P\ a$	
$snd : El_P (product\ a\ b) \rightarrow El_P\ b$	
data $El_W : U \rightarrow Set_1$ where	
$-\!::-\ : El_W\ a \rightarrow El_P (stream\ a) \rightarrow El_W (stream\ a)$	
$-\!,-\ : El_W\ a \rightarrow El_W\ b \rightarrow El_W (product\ a\ b)$	
$[-] : A \rightarrow El_W (lift\ A)$	

Note that only those constructor arguments which are usually delayed are represented as programs in the definition of El_W . The implementation of $whnf : El_P\ a \rightarrow El_W\ a$ and $\llbracket - \rrbracket_P : El_P\ a \rightarrow El\ a$ is left as an exercise for the reader.

Using a universe similar to the one above I have implemented—and proved correct—insanely circular breadth-first labelling of trees à la Jones and Gibbons (1993), see Danielsson (2010). To see why a universe such as the one above can be useful, consider the circular part of the algorithm (this is not Agda code):

$label : Tree\ A \rightarrow Stream\ B \rightarrow Tree\ B$
$label\ t\ ls = t' \mathbf{where} (t', lss) = label'\ t (ls :: lss)$
$label' : Tree\ A \rightarrow Stream (Stream\ B) \rightarrow Tree\ B \times Stream (Stream\ B)$
$label'\ t\ lss = \dots$

5 Making Proofs Guarded

The language-based approach to guardedness introduced in Sect. 3 has some problems when applied to programs:

- The interpretive overhead, compared to a direct implementation, can be substantial. For instance, computing the n -th element of the stream fib defined

in Sect. 3 requires a number of additions which is exponential in n , whereas if $fib = 0 :: \# \text{zipWith } _+ _ fib \ (1 :: \# fib)$ is implemented directly in a language which uses call-by-need this computation only requires $\mathcal{O}(n)$ additions. The reason for this discrepancy is that the interpreter $\llbracket _ \rrbracket_{\text{P}}$ does not preserve sharing. One could perhaps work around this problem by writing a more complicated interpreter, but this seems counterproductive: why spend effort writing a new interpreter when one is already provided by the host programming language (or the underlying hardware)?

- Proving properties about the interpreted definitions can be awkward, because this amounts to proving properties about the interpreter.

However, these problems are usually irrelevant for *proofs*: the run-time complexity of proofs is rarely important, and any proof of a property is usually as good as any other. Hence the approach is likely to be more useful for making proofs guarded, than for making programs guarded.

This section shows how the technique can be applied to proofs. Hinze (2008) advocates proving stream identities using a uniqueness property. One example in his paper is the iterate fusion law:

$$\begin{aligned} \text{fusion} & : (h : A \rightarrow B) (f_1 : A \rightarrow A) (f_2 : B \rightarrow B) \rightarrow \\ & ((x : A) \rightarrow h (f_1 x) \equiv f_2 (h x)) \rightarrow \\ & (x : A) \rightarrow \text{map } h (\text{iterate } f_1 x) \approx \text{iterate } f_2 (h x) \end{aligned}$$

Hinze proves this law by establishing that the left and right hand sides both satisfy the same guarded equation, $f x \approx h x :: \# f (f_1 x)$ (where f is the “unknown variable”):

$$\begin{aligned} \text{map } h (\text{iterate } f_1 x) & \quad \approx \langle \text{by definition} \rangle \\ h x :: \# \text{map } h (\text{iterate } f_1 (f_1 x)) & \\ \\ h x :: \# \text{iterate } f_2 (h (f_1 x)) & \quad \approx \langle \text{assumption} \rangle \\ h x :: \# \text{iterate } f_2 (f_2 (h x)) & \quad \approx \langle \text{by definition} \rangle \\ \text{iterate } f_2 (h x) & \end{aligned}$$

The separately proved fact that the equation has a unique solution then implies that $\text{map } h (\text{iterate } f_1 x)$ and $\text{iterate } f_2 (h x)$ are equal.

The attentive reader may have noticed that the proof above is almost a proof by guarded coinduction: the two equational reasoning blocks can be joined by an application of the coinductive hypothesis. However, the second block uses transitivity, thus destroying guardedness. We can work around this problem by following the approach introduced in Sect. 3. Let us define a language of equality proof “programs” as follows:

$$\begin{aligned} \text{data } _ \approx_{\text{P}} _ & : \text{Stream } A \rightarrow \text{Stream } A \rightarrow \text{Set} \text{ where} \\ _ :: _ & : (x : A) \rightarrow \infty ({}^b xs \approx_{\text{P}} {}^b ys) \rightarrow x :: xs \approx_{\text{P}} x :: ys \\ _ \approx \langle _ \rangle _ & : (xs : \text{Stream } A) \rightarrow xs \approx_{\text{P}} ys \rightarrow ys \approx_{\text{P}} zs \rightarrow xs \approx_{\text{P}} zs \\ _ \square & : (xs : \text{Stream } A) \rightarrow xs \approx_{\text{P}} xs \end{aligned}$$

The last two constructors represent transitivity and reflexivity, respectively. Note that the transitivity constructor is inductive; a coinductive transitivity constructor would make the relation trivial (see Danielsson and Altenkirch (2009b)). The somewhat odd names were chosen to make the proof of the iterate fusion law more readable, following Norell (2007). Just remember that $\approx\langle_ \rangle$ and \sqsubseteq are both weakly binding, with $\approx\langle_ \rangle$ right associative and binding weaker than \sqsubseteq :

$$\begin{aligned}
\text{fusion} & : (h : A \rightarrow B) (f_1 : A \rightarrow A) (f_2 : B \rightarrow B) \rightarrow \\
& ((x : A) \rightarrow h (f_1 x) \equiv f_2 (h x)) \rightarrow \\
& (x : A) \rightarrow \text{map } h (\text{iterate } f_1 x) \approx_{\text{P}} \text{iterate } f_2 (h x) \\
\text{fusion } h f_1 f_2 \text{ hyp } x & = \\
\text{map } h (\text{iterate } f_1 x) & \approx\langle \text{by definition} \rangle \\
h x :: \# \text{map } h (\text{iterate } f_1 (f_1 x)) & \approx\langle h x :: \# \text{fusion } h f_1 f_2 \text{ hyp } (f_1 x) \rangle \\
h x :: \# \text{iterate } f_2 (h (f_1 x)) & \approx\langle h x :: \# \text{iterate-cong } f_2 (\text{hyp } x) \rangle \\
h x :: \# \text{iterate } f_2 (f_2 (h x)) & \approx\langle \text{by definition} \rangle \\
\text{iterate } f_2 (h x) & \square
\end{aligned}$$

Note that the definition of *fusion* is guarded. The definition uses some simple lemmas (*iterate-cong*, *by* and *definition*), which are left as exercises for the reader.

In order to finish the proof of the iterate fusion law we have to show that \approx_{P} is sound with respect to \approx . (Completeness is easy to prove, but not relevant for the task at hand.) To do this one can first define a type of WHNFs:

$$\begin{aligned}
\mathbf{data} \ \approx_{\text{W}} & : \text{Stream } A \rightarrow \text{Stream } A \rightarrow \text{Set} \ \mathbf{where} \\
\text{---} & : (x : A) \rightarrow \text{! } xs \approx_{\text{P}} \text{! } ys \rightarrow x :: xs \approx_{\text{W}} x :: ys
\end{aligned}$$

It is easy to establish, by simple case analysis, that this relation is a preorder:

$$\begin{aligned}
\text{refl}_{\text{W}} & : (xs : \text{Stream } A) \rightarrow xs \approx_{\text{W}} xs \\
\text{trans}_{\text{W}} & : xs \approx_{\text{W}} ys \rightarrow ys \approx_{\text{W}} zs \rightarrow xs \approx_{\text{W}} zs
\end{aligned}$$

It follows by structural recursion that programs can be turned into WHNFs (note that $xs \approx ys$ and $ys \approx zs$ are variable names):

$$\begin{aligned}
\text{whnf} & : xs \approx_{\text{P}} ys \rightarrow xs \approx_{\text{W}} ys \\
\text{whnf } (x :: xs \approx ys) & = x :: \text{! } xs \approx ys \\
\text{whnf } (xs \approx\langle xs \approx ys \rangle ys \approx zs) & = \text{trans}_{\text{W}} (\text{whnf } xs \approx ys) (\text{whnf } ys \approx zs) \\
\text{whnf } (xs \ \square) & = \text{refl}_{\text{W}} xs
\end{aligned}$$

Finally soundness can be proved using guarded corecursion:

$$\begin{aligned}
\text{sound}_{\text{W}} : xs \approx_{\text{W}} ys \rightarrow xs \approx ys & \quad \text{sound}_{\text{P}} : xs \approx_{\text{P}} ys \rightarrow xs \approx ys \\
\text{sound}_{\text{W}} (x :: xs \approx ys) = & \quad \text{sound}_{\text{P}} xs \approx ys = \\
x :: \# \text{sound}_{\text{P}} xs \approx ys & \quad \text{sound}_{\text{W}} (\text{whnf } xs \approx ys)
\end{aligned}$$

Note that, unlike in Sect. 3, there is no need to prove that the application

$sound_P$ ($fusion\ h\ f_1\ f_2\ hyp\ x$) satisfies its “intended defining equation”, whatever that would be.

Using the language-based approach to guardedness I have formalised a number of examples from Hinze’s paper, see Danielsson (2010). Rephrasing the proofs using guarded coinduction turned out to be unproblematic.

6 Destructors

The following, alternative definition of the Fibonacci sequence is not directly supported by the framework outlined in previous sections:

$$\begin{aligned} fib &: Stream\ \mathbb{N} \\ fib &= 0 :: \# (1 :: \# zipWith\ _+_ fib\ (tail\ fib)) \end{aligned}$$

The problem is the use of the destructor *tail*. Unrestricted use of destructors can lead to non-productive definitions, as demonstrated by the definition *bad* in the introduction. However, destructors can be incorporated by extending the program type with an index which indicates when they can be used.

Consider the following type of stream programs:

$$\begin{aligned} \mathbf{data}\ Stream_P &: Bool \rightarrow Set \rightarrow Set_1 \mathbf{where} \\ [-] &: \infty (Stream_P\ true\ A) \rightarrow Stream_P\ false\ A \\ _::_ &: A \rightarrow Stream_P\ b\ A \rightarrow Stream_P\ true\ A \\ \mathbf{tail} &: Stream_P\ true\ A \rightarrow Stream_P\ false\ A \\ \mathbf{forget} &: Stream_P\ true\ A \rightarrow Stream_P\ false\ A \\ \mathbf{zipWith} &: (A \rightarrow B \rightarrow C) \rightarrow \\ &Stream_P\ b\ A \rightarrow Stream_P\ b\ B \rightarrow Stream_P\ b\ C \end{aligned}$$

The type $Stream_P\ b\ A$ stands for streams generated in chunks of size (at least) one, where the first chunk is guaranteed to be non-empty if the index b is true. The constructor $[-]$ marks the end of a chunk. Note how the indices ensure that a finished chunk is always non-empty, and that *tail* may only be used to inspect the chunk currently being constructed. The constructor *forget* is used to “forget” that a chunk is already finished; *forget* represents the identity function. This constructor is used in the implementation of fib_P (an alternative would be to give *zipWith* a more general type):

$$\begin{aligned} fib_P &: Stream_P\ true\ \mathbb{N} \\ fib_P &= 0 :: \# (1 :: zipWith\ _+_ (\mathbf{forget}\ fib_P)\ (\mathbf{tail}\ fib_P)) \end{aligned}$$

The implementation of $[-]_P$ and the proof of correctness of fib_P are left as exercises for the reader (solutions are provided by Danielsson (2010)). The following type of WHNFs may be useful:

$$\begin{aligned} \mathbf{data}\ Stream_W &: Bool \rightarrow Set \rightarrow Set_1 \mathbf{where} \\ [-] &: Stream_P\ true\ A \rightarrow Stream_W\ false\ A \\ _::_ &: A \rightarrow Stream_P\ true\ A \rightarrow Stream_W\ true\ A \end{aligned}$$

Before leaving the subject of destructors, note that the language above can be generalised to support other chunk sizes, i.e. other moduli of continuity (see Danielsson (2010)).

7 Nested Applications

Before wrapping up, let us briefly consider nested applications of the function being defined, as in $\varphi (x :: xs) = x :: \# \varphi (\varphi xs)$. To handle such applications one can include a constructor for the function in the type of programs:

```
data StreamP (A : Set) : Set where
  _::_ : A → ∞ (StreamP A) → StreamP A
  φP : StreamP A → StreamP A
```

By lifting streams to programs one can then define φ :

```
[_] : Stream A → StreamP A      φ : Stream A → Stream A
[x :: xs] = x :: # [^ xs]      φ xs = [[ φP [ xs ] ]]P
```

In order to prove that φ satisfies its intended defining equation it can be helpful to use an equality proof language, as in Sect. 5, and to include a constructor for the congruence of φ_P in this language (see Danielsson (2010)):

```
data _≈P_ : Stream A → Stream A → Set where
  ...
  φP-cong : (xs ys : StreamP A) →
    [[ xs ]]P ≈P [[ ys ]]P → [[ φP xs ]]P ≈P [[ φP ys ]]P
```

8 Related Work

Rutten (2003) proves that certain operations on streams are well-defined by using a technique which is very similar to the one described in this paper. He defines a language E of real number stream expressions inductively (this language is similar to $Stream_P \mathbb{R}$), and defines a stream coalgebra $c : E \rightarrow \mathbb{R} \times E$ by recursion over the structure of E (this corresponds to *whnf*). The type of streams is a final coalgebra, so from c one obtains a function of type $E \rightarrow Stream \mathbb{R}$ (corresponding to $[[_]]_P$), which can be used to turn stream expressions into actual streams. Rutten then uses coinduction (expressed using bisimilarity) to prove that the defined operations satisfy their intended defining equations. He also proves uniqueness, i.e. that the operations are the only operations satisfying these equations.

There are some differences between Rutten's proof and the technique described here, other than the different settings (informal vs. formal, bisimulations vs. guarded coinduction). One is that the approach described here can handle

functions like *tail* (even compositions of *tail* with itself, corresponding to second and higher derivatives in Rutten’s terminology). Another difference is that Rutten’s language E is inductive, whereas $Stream_P$ uses mixed induction and coinduction. A simple consequence of this difference is that when Rutten defines *fib* he includes it as a term in E ; with the method described here one can get much further using a fixed language. Danielsson and Altenkirch (2009b)⁵ also take advantage of this difference when proving that one subtyping relation is sound with respect to another. In this proof the program and WHNF types are defined mutually, using mixed induction and coinduction, and the *whnf* function constructs its result using a combination of guarded corecursion and structural recursion.

Rutten’s proof is closely related to a technique, due to Bartels (2003), which is formulated in a general categorical setting. Bartels restricts the form of *whnf*, and in return the proofs showing that the definitions satisfy their intended defining equations come almost for free. Furthermore Bartels manages to define *fib* without including it as a term in the language.

Conor McBride (personal communication) has developed a technique for ensuring guardedness, based on the work of Hancock and Setzer (2000). The idea is to represent the right-hand sides of function definitions using a type $RHS\ g$, where g indicates whether the context is guarding or not, and to only allow corecursive calls in a guarding context.

Di Gianantonio and Miculan (2003) describe a general framework for defining values using a mixture of recursion and corecursion, based on functions which satisfy a notion of contractivity. The method seems to be quite general, and has been implemented in Coq.

Bertot (2005) implements a filter function for streams in Coq. An unrestricted filter function is not productive, so Bertot restricts the function’s inputs using predicates of the form “always (eventually P)”. The always part is defined coinductively, and the eventually part inductively.

Capretta (2005) defines the partiality monad, which can be used to represent potentially non-terminating computations, as follows:

```

data  $_^\nu$  ( $A : Set$ ) :  $Set$  where
  return :  $A \rightarrow A^\nu$ 
  step   :  $\infty (A^\nu) \rightarrow A^\nu$ 

```

The constructor **return** returns a result, and **step** postpones a computation. It is easy to define **bind** for this monad: $_{\gg} : A^\nu \rightarrow (A \rightarrow B^\nu) \rightarrow B^\nu$. Unfortunately it can be inconvenient to use this definition of **bind** in systems based on guarded corecursion, because $_{\gg}$ is not a constructor. Megacz (2007) suggests (essentially) the following alternative definition:

```

data  $_^\nu$  ( $A : Set$ ) :  $Set_1$  where
  return :  $A \rightarrow A^\nu$ 
   $_{\gg}$  :  $\infty (B^\nu) \rightarrow (B \rightarrow \infty (A^\nu)) \rightarrow A^\nu$ 

```

⁵ The focus of this submitted paper is not on guardedness. The technique is only used to show that two proofs are productive; about two pages is spent on the technique.

One can note that this corresponds directly to the first step of the technique presented in this paper. Megacz does not translate from the second to the first type, though.

Bertot and Komendantskaya (2009) describe a method for replacing corecursion with recursion. They map values of type *Stream* A to and from the isomorphic type $\mathbb{N} \rightarrow A$, and values of this type can be defined recursively. The authors state that the method is still very limited and that, as presented, it cannot handle van de Snepscheut’s corecursive definition of the Hamming numbers (Dijkstra 1981), which can easily be handled using the method described in this paper.

Niqui (2009) implements a corecursion scheme called λ -coiteration (Bartels 2003) in Coq. He states that this scheme cannot handle van de Snepscheut’s corecursive definition of the Hamming numbers.

McBride (2009) defines an applicative functor which captures the notion of “be[ing] ready a wee bit later”. Using this structure he defines various corecursive programs, including the insanely circular breadth-first labelling which is discussed in Sect. 4. The technique is presented using the partial language Haskell, but Robert Atkey has later implemented it in Agda (personal communication). The technique has not been developed very far yet: as far as I am aware no one has tried to prove any properties about functions defined using it. It seems promising, though: I have not seen any other implementation of insanely circular labelling in a total language (except for my own).

Morris et al. (2006) use the technique of replacing functions with constructors to show *termination* rather than productivity (see Morris et al. (2007) for an explanation of the technique). They replace a partially applied recursive call (which is not necessarily structural, because it could later be applied to anything), nested inside another recursive call, with a constructor application. If this constructor application is later encountered it is handled using structural recursion.

The technique presented here also shares some traits with Reynolds’ *defunctionalisation* (1972). Defunctionalisation is used to translate programs written in higher-order languages to first-order languages, and it basically amounts to representing function spaces using application-specific data types, and implementing interpreters for these data types.

9 Conclusions

I hope to have shown, through a number of examples, that the language-based approach to establishing guardedness is useful. I am currently turning to it whenever I have a problem with guardedness; see Danielsson and Altenkirch (2009b) for some examples not included in this paper.

However, there are some problems with the method. As discussed in Sect. 5 it is not very useful if efficiency is a concern. Furthermore it can be disruptive: if one decides to use the method after already having developed a large number of functions in some project, and many of these functions have to be reified as

constructors in a program data type, then a lot of work may be necessary. In fact, this problem—in one shape or another—is likely to apply to *all* approaches to making definitions guarded.

In the long term I believe that it would be useful to adopt a more modular approach to productivity than guardedness. In the short term it does not seem hard to make Agda’s productivity checker much less restrictive: if it remembered which functions “preserve guardedness”, then many definitions which are currently rejected would be guarded. Note that the technique presented in this paper provides an indication of why such an extension of the productivity checker would be correct.

Acknowledgements I would like to thank Thorsten Altenkirch, Conor McBride, Nicolas Oury and Anton Setzer for many discussions about coinduction, and EPSRC for financial support.

References

- The Agda Team. The Agda Wiki. Available at <http://wiki.portal.chalmers.se/agda/>, 2010.
- Falk Bartels. Generalised coinduction. *Mathematical Structures in Computer Science*, 13(2):321–348, 2003.
- Yves Bertot. Filters on coinductive streams, an application to Eratosthenes’ sieve. In *TLCA 2005*, volume 3461 of *LNCS*, pages 102–115, 2005.
- Yves Bertot and Ekaterina Komendantskaya. Using structural recursion for corecursion. In *TYPES 2008*, volume 5497 of *LNCS*, pages 220–236, 2009.
- Venanzio Capretta. General recursion via coinductive types. *Logical Methods in Computer Science*, 1(2):1–28, 2005.
- Thierry Coquand. Infinite objects in type theory. In *TYPES’93*, volume 806 of *LNCS*, pages 62–78, 1994.
- Patrick Cousot and Radhia Cousot. Inductive definitions, semantics and abstract interpretations. In *POPL ’92*, pages 83–94, 1992.
- Nils Anders Danielsson. Code accompanying the paper. Currently available from <http://www.cs.nott.ac.uk/~nad/>, 2010.
- Nils Anders Danielsson and Thorsten Altenkirch. Mixing induction and coinduction. Draft, 2009a.
- Nils Anders Danielsson and Thorsten Altenkirch. Subtyping, declaratively; An exercise in mixed induction and coinduction. Submitted, available from <http://www.cs.nott.ac.uk/~nad/>, 2009b.
- Pietro Di Gianantonio and Marino Miculan. A unifying approach to recursive and co-recursive definitions. In *TYPES 2002*, volume 2646 of *LNCS*, pages 148–161, 2003.
- Edsger W. Dijkstra. Hamming’s exercise in SASL. EWD792 (privately circulated note), 1981.
- Jeremy Gibbons and Graham Hutton. Proof methods for corecursive programs. *Fundamenta Informaticae*, 66(4):353–366, 2005.
- Eduardo Giménez. *Un Calcul de Constructions Infinies et son Application à la Vérification de Systèmes Communicants*. PhD thesis, Ecole Normale Supérieure de Lyon, 1996.

- Andrew D. Gordon. Bisimilarity as a theory of functional programming. *Theoretical Computer Science*, 228(1–2):5–47, 1999.
- Tatsuya Hagino. *A Categorical Programming Language*. PhD thesis, University of Edinburgh, 1987.
- Peter Hancock and Anton Setzer. Interactive programs in dependent type theory. In *CSL 2000*, volume 1862 of *LNCS*, pages 317–331, 2000.
- Peter Hancock, Dirk Pattinson, and Neil Ghani. Representations of stream processors using nested fixed points. *Logical Methods in Computer Science*, 5(3:9), 2009.
- Ralf Hinze. Functional pearl: Streams and unique fixed points. In *ICFP’08*, pages 189–200, 2008.
- John Hughes and Andrew Moran. Making choices lazily. In *FPCA ’95, Proceedings of the seventh international conference on Functional programming languages and computer architecture*, pages 108–119, 1995.
- Geraint Jones and Jeremy Gibbons. Linear-time breadth-first tree algorithms: An exercise in the arithmetic of folds and zips. Technical Report 071, Department of Computer Science, The University of Auckland, 1993.
- Xavier Leroy and Hervé Grall. Coinductive big-step operational semantics. *Information and Computation*, 207(2):284–304, 2009.
- Conor McBride. Time flies like an applicative functor. Available at <http://www.e-pig.org/epilogue/?p=186>, 2009.
- Adam Megacz. A coinductive monad for Prop-bounded recursion. In *PLPV’07, Proceedings of the 2007 workshop on Programming languages meets program verification*, pages 11–20, 2007.
- Paul Francis Mendler. *Inductive Definition in Type Theory*. PhD thesis, Cornell University, 1988.
- Robin Milner. Operational and algebraic semantics of concurrent processes. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*. The MIT Press and Elsevier, 1990.
- Peter Morris, Thorsten Altenkirch, and Conor McBride. Exploring the regular tree types. In *TYPES 2004*, volume 3839 of *LNCS*, pages 252–267, 2006.
- Peter Morris, Thorsten Altenkirch, and Neil Ghani. Constructing strictly positive families. In *Theory of Computing 2007, Proceedings of the Thirteenth Computing: The Australasian Theory Symposium (CATS2007)*, pages 111–121, 2007.
- Keiko Nakata and Tarmo Uustalu. Trace-based coinductive operational semantics for While; Big-step and small-step, relational and functional styles. In *TPHOLs 2009*, volume 5674 of *LNCS*, pages 375–390, 2009.
- Milad Niqui. Coalgebraic reasoning in Coq: Bisimulation and the λ -coiteration scheme. In *TYPES 2008*, volume 5497 of *LNCS*, pages 272–288, 2009.
- Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology and Göteborg University, 2007.
- John C. Reynolds. Definitional interpreters for higher-order programming languages. In *ACM ’72, Proceedings of the ACM annual conference*, volume 2, pages 717–740, 1972.
- J.J.M.M. Rutten. Behavioural differential equations: a coinductive calculus of streams, automata, and power series. *Theoretical Computer Science*, 308(1–3):1–53, 2003.
- D. A. Turner. Total functional programming. *Journal of Universal Computer Science*, 10(7):751–768, 2004.
- Philip Wadler, Walid Taha, and David MacQueen. How to add laziness to a strict language, without even being odd. In *Proceedings of the 1998 ACM SIGPLAN Workshop on ML*, 1998.