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

When working with infinite values in a total setting it is common to require that every value is *productive* (Sijtsma 1989): 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 for coinductive types. 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 $\mathit{nats}_{\geqslant}\ n$, the stream of successive natural numbers greater than or equal to n ($\_::\_$ is the cons constructor for streams):

$$\begin{aligned}&\mathit{nats}_{\geqslant} : \mathbb{N} \to \mathit{Stream}\,\mathbb{N}\\&\mathit{nats}_{\geqslant}\ n\ =\ n :: \mathit{nats}_{\geqslant}\ (\mathsf{suc}\ n)\end{aligned}$$

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*:

$$\begin{aligned}&\mathit{bad} : \mathit{Stream}\,\mathbb{N}\\&\mathit{bad}\ =\ \mathit{tail}\ (\mathsf{zero} :: \mathit{bad})\end{aligned}$$

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:

$$\begin{aligned}&\mathit{nats} : \mathit{Stream}\,\mathbb{N}\\&\mathit{nats}\ =\ \mathsf{zero} :: \mathit{map}\ \mathsf{suc}\ \mathit{nats}\end{aligned}$$

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 equivalent guarded definitions.

The main contribution of this paper is a technique for translating 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, and that these equations have unique solutions.

The technique relies on the use of data types defined using mixed induction and coinduction (see Section 2), 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, total[1] 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 2010a).

Before we continue it may be useful to state some things which are *not* addressed by the paper:

- The paper's focus is on establishing productivity, not on representing non-productive definitions, nor on making non-productive definitions total by restricting their types (Bertot 2005).
- No attempt is made to automate the technique: as it stands it provides a manual, somewhat ad hoc method for getting productive definitions accepted by a system based on guarded corecursion.

The rest of the paper is structured as follows: Section 2 discusses induction and coinduction in the context of Agda, Sections 3–8 (as well as Appendix A) introduce the language-based approach to productivity through a number of examples, Section 9 discusses related work, and Section 10 concludes.

2 Mixed Induction and Coinduction

This section gives a quick introduction to Agda, in particular to its support for mixed induction and coinduction. For more details, see Danielsson and Altenkirch (2010, Section 2).

In Agda the type of infinite streams can be defined as follows:

$$\begin{aligned}&\textbf{data}\ \mathit{Stream}\ (A : \mathit{Set}) : \mathit{Set}\ \textbf{where}\\ &\quad \_::\_ : A \to \infty\ (\mathit{Stream}\ A) \to \mathit{Stream}\ A\end{aligned}$$

This definition states that *Stream A* is a *Set* ("type") with a single (infix) constructor $\_::\_$ of type $A \to \infty\ (\mathit{Stream}\ A) \to \mathit{Stream}\ A$. The inclusion of ∞ in the type of $\_::\_$ makes *Stream A* coinductive; without it the type would be empty. You should read $\infty\ (\mathit{Stream}\ A)$ as "delayed stream of *A*s"—the function $\infty : \mathit{Set} \to \mathit{Set}$ is analogous to the suspension type constructors which are sometimes used to introduce non-strictness in strict languages (Wadler et al. 1998), and closely related to the domain-theoretic notion of lifting. However, Agda programs are required to be total.

We can construct infinite values by guarded corecursion. For instance, we can define a function which combines two streams in a pointwise manner as follows:[2]

$$\begin{aligned}&\mathit{zipWith} : \{A\ B\ C : \mathit{Set}\} \to (A \to B \to C) \to \mathit{Stream}\ A \to \mathit{Stream}\ B \to \mathit{Stream}\ C\\ &\mathit{zipWith}\ f\ (x :: xs)\ (y :: ys) = f\ x\ y :: \sharp\ \mathit{zipWith}\ f\ (\flat\ xs)\ (\flat\ ys)\end{aligned}$$

This definition uses the coinductive delay constructor $\sharp\_$ (sharp)[3] and the force function $\flat$ (flat):

[1] Agda is an experimental system with neither a formalised meta-theory nor a verified type checker, so take words such as "total" with a grain of salt.

[2] The notation $\{A\ B\ C : \mathit{Set}\} \to \ldots$ means that *zipWith* takes three *implicit* arguments *A*, *B* and *C*, all of type *Set*. These arguments do not need to be given explicitly if Agda can infer them.

[3] The prefix operator $\sharp\_$ is the most tightly binding operator in this paper; ordinary function application binds tighter, though.

$$\begin{aligned}&\sharp\_ : \{A : Set\} \to \quad A \to \infty\, A\\ &\flat\ \ : \{A : Set\} \to \infty\, A \to \quad A\end{aligned}$$

Agda views *zipWith* as guarded, because there is no non-constructor function between the left-hand side and the corecursive call, and there is at least one use of the guarding coinductive constructor $\sharp\_$. This constructor has special status: it is treated as a constructor by Agda's productivity checker, but may not be used in patterns.[4] Instead one can use the force function: $\flat\,(\sharp\, x)$ reduces to x.

As another example, consider the following definition of equality—bisimilarity—for streams (which makes use of the fact that constructors can be overloaded):

$$\begin{aligned}&\textbf{data}\ \_{\approx}\_\ \{A : Set\} : Stream\, A \to Stream\, A \to Set\ \textbf{where}\\ &\quad \_::\_ : (x : A) \to \{xs\ ys : \infty\,(Stream\, A)\} \to \infty\,(\flat\, xs \approx \flat\, ys) \to x :: xs \approx x :: ys\end{aligned}$$

This definition states that two streams are equal if their heads are identical and their tails are equal (coinductively). Note that the elements of this type are equality *proofs*. We can establish equalities by constructing proofs using guarded corecursion. For instance, we can prove symmetry as follows:

$$\begin{aligned}&sym : \{A : Set\} \to \{xs\ ys : Stream\, A\} \to xs \approx ys \to ys \approx xs\\ &sym\,(x :: xs{\approx}ys) = x :: \sharp\, sym\,(\flat\, xs{\approx}ys)\end{aligned}$$

(Note that $xs{\approx}ys$ is an ordinary variable, albeit perhaps with an unusual name.)

Let us now consider a definition which uses both induction and coinduction. The type $SP\ A\ B$ of stream processors (Hancock et al. 2009)—representations of programs taking streams of As to streams of Bs—can be defined as follows:

$$\begin{aligned}&\textbf{data}\ SP\,(A\ B : Set) : Set\ \textbf{where}\\ &\quad \mathsf{put} : B \to \infty\,(SP\ A\ B) \to SP\ A\ B\\ &\quad \mathsf{get} : (A \to SP\ A\ B) \quad \to SP\ A\ B\end{aligned}$$

Here $\mathsf{put}\ b\ sp$ is intended to output b and continue with sp, while $\mathsf{get}\, f$ is intended to read an element a and continue with $f\ a$. You can see the type as the nested fixpoint[5] $\nu X.\ \mu Y.\ B \times X + (A \to Y)$—in fact, *all* (non-mutual) data types in the paper can be seen as nested fixpoints of the form $\nu X.\ \mu Y.\ F\ X\ Y$ (and mutually defined data types can be merged by adding an index). Note that the recursive argument of put is delayed (coinductive), whereas the recursive argument of get is not. This means that we can have an infinite number of consecutive put constructors, but only a finite number of consecutive gets; definitions such as the following one are not guarded and not accepted:

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

The definition of *sink* is not problematic in and of itself (assuming that it is not evaluated too eagerly). However, by ruling out such definitions we make other definitions possible, for instance the following one, which gives the semantics of a stream processor:

$$\begin{aligned}&[\![\_]\!] : \{A\ B : Set\} \to SP\ A\ B \to Stream\, A \to Stream\, B\\ &[\![\ \mathsf{put}\ b\ sp\ ]\!]\ as \quad\ = b :: \sharp\,([\![\ \flat\, sp\ ]\!]\ as)\\ &[\![\ \mathsf{get}\, f \quad\ ]\!]\,(a :: as) = [\![\ f\ a\ ]\!]\,(\flat\, as)\end{aligned}$$

[4] This is not entirely true in the current version of Agda, but is likely to be true in the future.

[5] This is not quite correct in Agda (Altenkirch and Danielsson 2010), but for the purposes of this paper the differences are irrelevant.

This function is accepted by Agda because it is defined using a lexicographic combination of guarded corecursion and structural recursion. In this particular example the first component of the lexicographic product is the "guardedness", and the second component is the *inductive* structure of the stream processor:

- In the first clause the corecursive call is guarded. The stream processor is not structurally smaller, due to the use of the force function ($^\flat$), but this is irrelevant.
- In the second clause the corecursive call is not guarded, but there is no non-constructor function between the left-hand side and the corecursive call, so we say that "guardedness is preserved". On the other hand, the stream processor argument is strictly structurally smaller ($f\ x$ is smaller than $\mathsf{get}\ f$ for any x).

Armed with the knowledge that there can only be a finite number of consecutive get constructors we conclude that, when evaluating $[\![\ sp\ ]\!]\ as$, we must eventually reach the first clause. At this stage we can immediately inspect the head element of the output stream, because the second clause does not introduce any interfering destructors.

As a final example, consider *filter*, which is not accepted by Agda:

$$
\begin{aligned}
&\mathit{filter} : \{A : \mathit{Set}\} \to (A \to \mathit{Bool}) \to \mathit{Stream}\ A \to \mathit{Stream}\ A\\
&\mathit{filter}\ p\ (x :: xs)\ \mathbf{with}\ p\ x\\
&\mathit{filter}\ p\ (x :: xs)\ \mid \mathsf{true}\ = x ::^\sharp \mathit{filter}\ p\ (^\flat\ xs)\\
&\mathit{filter}\ p\ (x :: xs)\ \mid \mathsf{false}\ = \mathit{filter}\ p\ (^\flat\ xs)
\end{aligned}
$$

(Here the **with** construct is used to pattern match on $p\ x$.) The first corecursive call is guarded, but in the last clause the call is not guarded, and nothing is structurally smaller, so this function is not accepted.

The explanations above should suffice to understand the definitions in this paper—in fact, most definitions use less complicated recursion principles than the one used by $[\![\_]\!]$. Readers who want to know more about Agda's criterion for accepting a function as total can refer to Danielsson and Altenkirch (2010, Section 2.5).

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:

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

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:[6]

[6] Set_1 is a type of large types; ∞ has type $\mathit{Set}_i \to \mathit{Set}_i$ for any i.

$$\begin{aligned}
&\textbf{data}\ \mathit{Stream}_{\mathrm{P}}\ :\ \mathit{Set} \to \mathit{Set}_1\ \textbf{where}\\
&\quad _::_\ \ \ \ : A \to \infty\,(\mathit{Stream}_{\mathrm{P}}\ A) \to \mathit{Stream}_{\mathrm{P}}\ A\\
&\quad \mathsf{zipWith}\ :\ (A \to B \to C) \to \mathit{Stream}_{\mathrm{P}}\ A \to \mathit{Stream}_{\mathrm{P}}\ B \to \mathit{Stream}_{\mathrm{P}}\ C
\end{aligned}$$

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 zipWiths 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:

$$\begin{aligned}
&\textbf{data}\ \mathit{Stream}_{\mathrm{W}}\ :\ \mathit{Set} \to \mathit{Set}_1\ \textbf{where}\\
&\quad _::_\ :\ A \to \mathit{Stream}_{\mathrm{P}}\ A \to \mathit{Stream}_{\mathrm{W}}\ A
\end{aligned}$$

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:

$$\begin{aligned}
&\mathit{whnf}\ :\ \mathit{Stream}_{\mathrm{P}}\ A \to \mathit{Stream}_{\mathrm{W}}\ A\\
&\mathit{whnf}\ (x :: xs) \qquad\qquad = x ::^{\flat} xs\\
&\mathit{whnf}\ (\mathsf{zipWith}\ f\ xs\ ys)\ =\ \mathit{zipWith}_{\mathrm{W}}\ f\ (\mathit{whnf}\ xs)\ (\mathit{whnf}\ ys)
\end{aligned}$$

Here $\mathit{zipWith}_{\mathrm{W}}$ is defined by simple case analysis:

$$\begin{aligned}
&\mathit{zipWith}_{\mathrm{W}}\ :\ (A \to B \to C) \to \mathit{Stream}_{\mathrm{W}}\ A \to \mathit{Stream}_{\mathrm{W}}\ B \to \mathit{Stream}_{\mathrm{W}}\ C\\
&\mathit{zipWith}_{\mathrm{W}}\ f\ (x :: xs)\ (y :: ys)\ =\ f\ x\ y :: \mathsf{zipWith}\ f\ xs\ ys
\end{aligned}$$

WHNFs can then be turned into streams corecursively:

$$\begin{aligned}
&\textbf{mutual}\\
&\quad [\![_]\!]_{\mathrm{W}}\ :\ \mathit{Stream}_{\mathrm{W}}\ A \to \mathit{Stream}\ A\\
&\quad [\![\ x :: xs\ ]\!]_{\mathrm{W}}\ =\ x ::^{\sharp} [\![\ xs\ ]\!]_{\mathrm{P}}\\
&\quad [\![_]\!]_{\mathrm{P}}\ :\ \mathit{Stream}_{\mathrm{P}}\ A \to \mathit{Stream}\ A\\
&\quad [\![\ xs\ ]\!]_{\mathrm{P}}\ =\ [\![\ \mathit{whnf}\ xs\ ]\!]_{\mathrm{W}}
\end{aligned}$$

Note that this definition is guarded. (Agda accepts definitions like this one even though it is split up over two mutually defined functions; alternatively one could write $[\![\ x :: xs\ ]\!]_{\mathrm{W}} = x ::^{\sharp} [\![\ \mathit{whnf}\ xs\ ]\!]_{\mathrm{W}}$ and define $[\![\_]\!]_{\mathrm{P}}$ separately.)

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

$$\begin{aligned}
&\mathit{fib}_{\mathrm{P}}\ :\ \mathit{Stream}_{\mathrm{P}}\ \mathbb{N} && \mathit{fib}\ :\ \mathit{Stream}\ \mathbb{N}\\
&\mathit{fib}_{\mathrm{P}}\ =\ 0 ::^{\sharp} \mathsf{zipWith}\ _+_\ \mathit{fib}_{\mathrm{P}}\ (1 ::^{\sharp} \mathit{fib}_{\mathrm{P}}) && \mathit{fib}\ =\ [\![\ \mathit{fib}_{\mathrm{P}}\ ]\!]_{\mathrm{P}}
\end{aligned}$$

One can prove that this definition satisfies the original equation for *fib* by first proving corecursively that $[\![\_]\!]_{\mathrm{P}}$ is homomorphic with respect to *zipWith*/zipWith:

$$\begin{aligned}
&\mathit{zipWith\text{-}hom}\ :\ (f\ :\ A \to B \to C) \to (xs\ :\ \mathit{Stream}\ A) \to (ys\ :\ \mathit{Stream}\ B) \to\\
&\qquad\qquad [\![\ \mathsf{zipWith}\ f\ xs\ ys\ ]\!]_{\mathrm{P}}\ \approx\ \mathit{zipWith}\ f\ [\![\ xs\ ]\!]_{\mathrm{P}}\ [\![\ ys\ ]\!]_{\mathrm{P}}\\
&\mathit{fib\text{-}correct}\ \ \ :\ \mathit{fib}\ \approx\ 0 ::^{\sharp} \mathit{zipWith}\ _+_\ \mathit{fib}\ (1 ::^{\sharp} \mathit{fib})
\end{aligned}$$

For the omitted proofs, see Danielsson (2010a). One may also want to establish that the original equation for *fib* defines the stream completely, i.e. that it has a *unique* solution. For an explanation of how this can be done, see Section 5.

It can be instructive to see what would happen if we tried to use the method above to implement the ill-defined stream *bad* from the introduction. Defining the language and giving a "definition" for *bad* is straightforward:

```
data Stream_P (A : Set) : Set where
  _::_ : A → ∞ (Stream_P A) → Stream_P A
  tail : Stream_P A → Stream_P A

bad : Stream_P ℕ
bad = tail (zero :: ♯ bad)
```

However, turning stream programs into streams becomes tricky. How would $tail_W$ be defined?

```
data Stream_W (A : Set) : Set where
  _::_ : A → Stream_P A → Stream_W A

tail_W : Stream_W A → Stream_W A
tail_W (x :: xs) = ?
```

Note that, in the body of $tail_W$, *xs* is a stream program, but we need to produce a WHNF.

4 Several Types at Once

The technique introduced in Section 3 is not limited to streams. In fact, it can be used with several types at the same time. To illustrate how this can be done I will implement circular breadth-first labelling of trees á la Jones and Gibbons (1993).

The following type of potentially infinite binary trees will be used:

```
data Tree (A : Set) : Set where
  leaf : Tree A
  node : ∞ (Tree A) → A → ∞ (Tree A) → Tree A
```

Jones and Gibbons' implementation can be described as follows. First a labelling function *lab* is defined. This function takes a tree, along with a stream of streams of new labels. The labels in a prefix of the *n*-th stream are used to label the *n*-th level of the tree, and the remaining labels are returned from *lab*:[7]

```
lab : Tree A → Stream (Stream B) → Tree B × Stream (Stream B)
lab leaf          bss           = (leaf,                bss)
lab (node l _ r) ((b :: bs) :: bss) = (node (♯ l') b (♯ r'), ♭ bs :: ♯ bss'')
  where
  (l' , bss')  = lab (♭ l) (♭ bss)
  (r', bss'')  = lab (♭ r) bss'
```

This code is not accepted by Agda, because the recursive calls are not guarded (their results are destructed). The next step in Jones and Gibbons' implementation is to construct the stream of streams of labels which is used by *lab*, and use these streams to compute the relabelled tree. This is done using a circular construction:

```
label : Tree A → Stream B → Tree B
label t bs = t'
  where (t', bss) = lab t (bs :: ♯ bss)
```

This code is not accepted by Agda, because *lab* is not a constructor, and furthermore the result of *lab* is destructed.

[7] Agda does not support pattern matching in where clauses as used here, but this is easy to work around using projection functions.

To implement breadth-first labelling in the style of Jones and Gibbons the following universe of trees, streams, products and arbitrary (small) types will be used:

```
data U : Set₁ where
  tree   : U → U
  stream : U → U
  _⊗_    : U → U → U
  ⌈_⌉    : Set → U

El : U → Set
El (tree a)     = Tree (El a)
El (stream a)   = Stream (El a)
El (a ⊗ b)      = El a × El b
El ⌈ 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 the universe U we can work with several types at once:

```
mutual
  data El_P : U → Set₁ where
    ↓    : El_W a → El_P a
    fst  : El_P (a ⊗ b) → El_P a
    snd  : El_P (a ⊗ b) → El_P b
    lab  : Tree A → El_P (stream ⌈ Stream B ⌉) → El_P (tree ⌈ B ⌉ ⊗ stream ⌈ Stream B ⌉)
  data El_W : U → Set₁ where
    leaf : El_W (tree a)
    node : ∞ (El_P (tree a)) → El_W a → ∞ (El_P (tree a)) → El_W (tree a)
    _::_ : El_W a → ∞ (El_P (stream a)) → El_W (stream a)
    _,_  : El_W a → El_W b → El_W (a ⊗ b)
    ⌈_⌉  : A → El_W ⌈ A ⌉
```

Note that only those constructor arguments which are delayed are represented as programs in the definition of El_W. Note also that the two types are defined mutually: the WHNF type is included in the type of programs using the constructor ↓. This makes the program type less usable (the program ↓ (fst p :: xs) is not well-typed, for instance), but avoids some code duplication.

The type of lab may seem a bit strange: the inner and outer streams are represented differently. One reason for this design choice can be seen in the definition of lab_W:

```
lab_W : Tree A → El_W (stream ⌈ Stream B ⌉) → El_W (tree ⌈ B ⌉ ⊗ stream ⌈ Stream B ⌉)
lab_W leaf          bss                  = (leaf,                          bss)
lab_W (node l _ r) (⌈ b :: bs ⌉ :: bss) = (node (♯ fst x) ⌈ b ⌉ (♯ fst y), ⌈ ♭ bs ⌉ :: ♯ snd y)
  where
  x = lab (♭ l) (♭ bss)
  y = lab (♭ r) (snd x)
```

Consider the second clause. If lab_W had the type

$$Tree\ A \to El_W\ (\mathsf{stream}\ (\mathsf{stream}\ b)) \to El_W\ (\mathsf{tree}\ b \otimes \mathsf{stream}\ (\mathsf{stream}\ b)),$$

then the analogue of bs would be a *program*, but the head of the resulting stream of streams ($\lceil\ ^\flat\ bs\ \rceil$ in the definition above) must be a WHNF. The use of "raw" inner streams also means that the input to the *label* function does not need to be converted.

Note that lab_W is non-recursive. The remainder of *whnf* is straightforward to implement using structural recursion:

$$
\begin{aligned}
&\mathit{fst}_{\mathrm{W}} : \mathit{El}_{\mathrm{W}}\ (a \otimes b) \to \mathit{El}_{\mathrm{W}}\ a\\
&\mathit{fst}_{\mathrm{W}}\ (x, y) = x\\
&\mathit{snd}_{\mathrm{W}} : \mathit{El}_{\mathrm{W}}\ (a \otimes b) \to \mathit{El}_{\mathrm{W}}\ b\\
&\mathit{snd}_{\mathrm{W}}\ (x, y) = y
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{whnf} : \mathit{El}_{\mathrm{P}}\ a \to \mathit{El}_{\mathrm{W}}\ a\\
&\mathit{whnf}\ (\downarrow w) = w\\
&\mathit{whnf}\ (\mathsf{fst}\ p) = \mathit{fst}_{\mathrm{W}}\ (\mathit{whnf}\ p)\\
&\mathit{whnf}\ (\mathsf{snd}\ p) = \mathit{snd}_{\mathrm{W}}\ (\mathit{whnf}\ p)\\
&\mathit{whnf}\ (\mathsf{lab}\ t\ \mathit{bss}) = \mathit{lab}_{\mathrm{W}}\ t\ (\mathit{whnf}\ \mathit{bss})
\end{aligned}
$$

It is also easy to define $[\![\_]\!]_{\mathrm{W}}$ and $[\![\_]\!]_{\mathrm{P}}$. These definitions use a lexicographic combination of guarded corecursion and structural recursion (see Section 2):

mutual

$$
\begin{aligned}
&[\![\_]\!]_{\mathrm{W}} : \mathit{El}_{\mathrm{W}}\ a \to \mathit{El}\ a\\
&[\![\ \mathsf{leaf}\ ]\!]_{\mathrm{W}} = \mathsf{leaf}\\
&[\![\ \mathsf{node}\ l\ x\ r\ ]\!]_{\mathrm{W}} = \mathsf{node}\ (\sharp\ [\![\ ^{\flat}\ l\ ]\!]_{\mathrm{P}})\ [\![\ x\ ]\!]_{\mathrm{W}}\ (\sharp\ [\![\ ^{\flat}\ r\ ]\!]_{\mathrm{P}})\\
&[\![\ x :: \mathit{xs}\ ]\!]_{\mathrm{W}} = [\![\ x\ ]\!]_{\mathrm{W}} ::^{\sharp} [\![\ ^{\flat}\ \mathit{xs}\ ]\!]_{\mathrm{P}}\\
&[\![\ (x, y)\ ]\!]_{\mathrm{W}} = ([\![\ x\ ]\!]_{\mathrm{W}}, [\![\ y\ ]\!]_{\mathrm{W}})\\
&[\![\ \lceil\ x\ \rceil\ ]\!]_{\mathrm{W}} = x\\
&[\![\_]\!]_{\mathrm{P}} : \mathit{El}_{\mathrm{P}}\ a \to \mathit{El}\ a\\
&[\![\ p\ ]\!]_{\mathrm{P}} = [\![\ \mathit{whnf}\ p\ ]\!]_{\mathrm{W}}
\end{aligned}
$$

Finally we can define *label*:

$$
\begin{aligned}
&\mathit{label}' : \mathit{Tree}\ A \to \mathit{Stream}\ B \to \mathit{El}_{\mathrm{P}}\ (\mathsf{tree}\ \lceil\ B\ \rceil \otimes \mathsf{stream}\ \lceil\ \mathit{Stream}\ B\ \rceil)\\
&\mathit{label}'\ t\ \mathit{bs} = \mathsf{lab}\ t\ (\downarrow (\lceil\ \mathit{bs}\ \rceil ::^{\sharp} \mathsf{snd}\ (\mathit{label}'\ t\ \mathit{bs})))\\
&\mathit{label} : \mathit{Tree}\ A \to \mathit{Stream}\ B \to \mathit{Tree}\ B\\
&\mathit{label}\ t\ \mathit{bs} = [\![\ \mathsf{fst}\ (\mathit{label}'\ t\ \mathit{bs})\ ]\!]_{\mathrm{P}}
\end{aligned}
$$

Note that the helper function *label*′, which corresponds to the cyclic part of the original *label*, is defined using guarded corecursion.

I have proved that the definition of *label* is correct: the resulting tree has the same shape as the original one, and a breadth-first traversal of the resulting tree produces a potentially infinite list of labels which is a prefix of the stream given to *label*. To state correctness I extended the universe with support for potentially infinite lists, and added some programs to the El_{P} type. For details of the statement and proof, see Danielsson (2010a).

5 Making Proofs Guarded

The language-based approach to guardedness introduced in Section 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 Section 3 requires a number of additions which is exponential in n, whereas if $\mathit{fib} = 0 ::^{\sharp} \mathit{zipWith}\ \_+\_\ \mathit{fib}\ (1 ::^{\sharp} \mathit{fib})$ 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 $[\![\_]\!]_{\mathrm{P}}$ 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 (for instance to establish that they are correct) 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:

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

Here *map* and *iterate* are defined as follows:

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

Hinze proves the iterate fusion law by establishing that the left and right hand sides both satisfy the same guarded equation, $f\ x\ \approx\ h\ x :: ^\sharp f\ (f_1\ x)$ (where f is the "unknown variable"):

$$\begin{aligned}&\mathit{map}\ h\ (\mathit{iterate}\ f_1\ x) && \approx\langle \text{ by definition } \rangle \\ &h\ x :: ^\sharp \mathit{map}\ h\ (\mathit{iterate}\ f_1\ (f_1\ x)) &&\end{aligned}$$

$$\begin{aligned}&h\ x :: ^\sharp \mathit{iterate}\ f_2\ (h\ (f_1\ x)) && \approx\langle \text{ assumption } \rangle \\ &h\ x :: ^\sharp \mathit{iterate}\ f_2\ (f_2\ (h\ x)) && \approx\langle \text{ by definition } \rangle \\ &\mathit{iterate}\ f_2\ (h\ x) &&\end{aligned}$$

The separately proved[8] fact that the equation has a unique solution then implies that $\mathit{map}\ h\ (\mathit{iterate}\ f_1\ x)$ and $\mathit{iterate}\ f_2\ (h\ x)$ are equal.

Note 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 Section 3. Let us define a language of equality proof "programs" as follows:

$$\begin{aligned}&\textbf{data}\ \_\approx_{\mathrm{P}}\_ : \mathit{Stream}\ A \to \mathit{Stream}\ A \to \mathit{Set}\ \textbf{where} \\ &\quad \_::\_ \quad : (x : A) \to \infty\ (^\flat xs\ \approx_{\mathrm{P}}\ ^\flat ys) \to x :: xs\ \approx_{\mathrm{P}}\ x :: ys \\ &\quad \_\approx\langle\_\rangle\_ : (xs : \mathit{Stream}\ A) \to xs\ \approx_{\mathrm{P}}\ ys \to ys\ \approx_{\mathrm{P}}\ zs \to xs\ \approx_{\mathrm{P}}\ zs \\ &\quad \_\square \quad : (xs : \mathit{Stream}\ A) \to xs\ \approx_{\mathrm{P}}\ xs\end{aligned}$$

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 (2010)). 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 $\_\square$ are both weakly binding, with $\_\approx\langle\_\rangle\_$ right associative and binding weaker than $\_\square$:

[8] Hinze proves this using a method described by Rutten (2003), which in fact is closely related to the method described here, see Section 9.

$$
\begin{aligned}
&\mathit{fusion} : (h : A \to B) \to (f_1 : A \to A) \to (f_2 : B \to B) \to \\
&\qquad ((x : A) \to h\,(f_1\,x) \equiv f_2\,(h\,x)) \to \\
&\qquad (x : A) \to \mathit{map}\,h\,(\mathit{iterate}\,f_1\,x) \approx_{\mathrm{P}} \mathit{iterate}\,f_2\,(h\,x) \\
&\mathit{fusion}\,h\,f_1\,f_2\,\mathit{hyp}\,x = \\
&\quad \mathit{map}\,h\,(\mathit{iterate}\,f_1\,x) && \approx\langle\ \mathit{by\ definition}\ \rangle \\
&\quad h\,x :: ^\sharp \mathit{map}\,h\,(\mathit{iterate}\,f_1\,(f_1\,x)) && \approx\langle\ h\,x :: ^\sharp \mathit{fusion}\,h\,f_1\,f_2\,\mathit{hyp}\,(f_1\,x)\ \rangle \\
&\quad h\,x :: ^\sharp \mathit{iterate}\,f_2\,(h\,(f_1\,x)) && \approx\langle\ h\,x :: ^\sharp \mathit{iterate\text{-}cong}\,f_2\,(\mathit{hyp}\,x)\ \rangle \\
&\quad h\,x :: ^\sharp \mathit{iterate}\,f_2\,(f_2\,(h\,x)) && \approx\langle\ \mathit{by\ definition}\ \rangle \\
&\quad \mathit{iterate}\,f_2\,(h\,x) && \Box
\end{aligned}
$$

Note that the definition of *fusion* is guarded. The definition uses some simple lemmas (*iterate-cong*, *by* and *definition*), which are omitted here.

In order to finish the proof of the iterate fusion law we have to show that $\_\approx_{\mathrm{P}}\_$ is sound with respect to $\_\approx\_$. To do this one can first define a type of WHNFs:

$$
\begin{aligned}
&\textbf{data}\ \_\approx_{\mathrm{W}}\_ : \mathit{Stream}\,A \to \mathit{Stream}\,A \to \mathit{Set}\ \textbf{where} \\
&\quad \_::\_ : (x : A) \to {}^\flat\,\mathit{xs} \approx_{\mathrm{P}} {}^\flat\,\mathit{ys} \to x :: \mathit{xs} \approx_{\mathrm{W}} x :: \mathit{ys}
\end{aligned}
$$

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

$$
\begin{aligned}
&\mathit{refl}_{\mathrm{W}} : (\mathit{xs} : \mathit{Stream}\,A) \to \mathit{xs} \approx_{\mathrm{W}} \mathit{xs} \\
&\mathit{trans}_{\mathrm{W}} : \mathit{xs} \approx_{\mathrm{W}} \mathit{ys} \to \mathit{ys} \approx_{\mathrm{W}} \mathit{zs} \to \mathit{xs} \approx_{\mathrm{W}} \mathit{zs}
\end{aligned}
$$

It follows by structural recursion that programs can be turned into WHNFs:

$$
\begin{aligned}
&\mathit{whnf} : \mathit{xs} \approx_{\mathrm{P}} \mathit{ys} \to \mathit{xs} \approx_{\mathrm{W}} \mathit{ys} \\
&\mathit{whnf}\,(x :: \mathit{xs}{\approx}\mathit{ys}) && = x :: {}^\flat\,\mathit{xs}{\approx}\mathit{ys} \\
&\mathit{whnf}\,(\mathit{xs} \approx\langle\ \mathit{xs}{\approx}\mathit{ys}\ \rangle\ \mathit{ys}{\approx}\mathit{zs}) && = \mathit{trans}_{\mathrm{W}}\,(\mathit{whnf}\ \mathit{xs}{\approx}\mathit{ys})\,(\mathit{whnf}\ \mathit{ys}{\approx}\mathit{zs}) \\
&\mathit{whnf}\,(\mathit{xs}\ \Box) && = \mathit{refl}_{\mathrm{W}}\,\mathit{xs}
\end{aligned}
$$

Finally soundness can be proved using guarded corecursion:

$$
\begin{aligned}
&\textbf{mutual} \\
&\quad \mathit{sound}_{\mathrm{W}} : \mathit{xs} \approx_{\mathrm{W}} \mathit{ys} \to \mathit{xs} \approx \mathit{ys} \\
&\quad \mathit{sound}_{\mathrm{W}}\,(x :: \mathit{xs}{\approx}\mathit{ys}) = x :: ^\sharp \mathit{sound}_{\mathrm{P}}\,\mathit{xs}{\approx}\mathit{ys} \\
&\quad \mathit{sound}_{\mathrm{P}} : \mathit{xs} \approx_{\mathrm{P}} \mathit{ys} \to \mathit{xs} \approx \mathit{ys} \\
&\quad \mathit{sound}_{\mathrm{P}}\,\mathit{xs}{\approx}\mathit{ys} = \mathit{sound}_{\mathrm{W}}\,(\mathit{whnf}\ \mathit{xs}{\approx}\mathit{ys})
\end{aligned}
$$

Note that there is no need to prove that the application $\mathit{sound}_{\mathrm{P}}\,(\mathit{fusion}\,h\,f_1\,f_2\,\mathit{hyp}\,x)$ satisfies its intended defining equation, whatever that would be, or that this equation has a unique solution.

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

As a further example, let us show that the defining equation for *fib* (see Section 3) has a unique solution. We can state the problem as follows:

$$
\begin{aligned}
&\mathit{fib\text{-}rhs} : \mathit{Stream}\,\mathbb{N} \to \mathit{Stream}\,\mathbb{N} \\
&\mathit{fib\text{-}rhs}\,\mathit{ns} = 0 :: ^\sharp \mathit{zipWith}\ \_+\_\ \mathit{ns}\,(1 :: ^\sharp \mathit{ns}) \\
&\mathit{fib\text{-}unique} : (\mathit{ms}\ \mathit{ns} : \mathit{Stream}\,\mathbb{N}) \to \mathit{ms} \approx \mathit{fib\text{-}rhs}\,\mathit{ms} \to \mathit{ns} \approx \mathit{fib\text{-}rhs}\,\mathit{ns} \to \mathit{ms} \approx_{\mathrm{P}} \mathit{ns}
\end{aligned}
$$

The type $\_\approx_{\mathrm{P}}\_$ used here is different from the one used above: the proof will make use of the congruence of *zipWith*, and the coinductive hypothesis will be an argument to this congruence, so a constructor for the congruence is included among the equality proof programs:

$$
\begin{array}{l}
\textbf{data}\ \_\approx_{\mathrm{P}}\_ : \mathit{Stream}\ A \to \mathit{Stream}\ A \to \mathit{Set}\ \textbf{where} \\
\quad \ldots \\
\quad \mathsf{zipWith\text{-}cong} : (f : A \to A \to A) \to \mathit{xs}_1 \approx_{\mathrm{P}} \mathit{ys}_1 \to \mathit{xs}_2 \approx_{\mathrm{P}} \mathit{ys}_2 \to \\
\qquad\qquad\qquad \mathit{zipWith}\ f\ \mathit{xs}_1\ \mathit{xs}_2 \approx_{\mathrm{P}} \mathit{zipWith}\ f\ \mathit{ys}_1\ \mathit{ys}_2
\end{array}
$$

It is easy to extend the definition of *whnf* to support zipWith-cong, using which we can define *fib-unique* as follows:

$$
\begin{array}{l}
\mathit{fib\text{-}unique}\ \mathit{ms}\ \mathit{ns}\ \mathit{hyp}_1\ \mathit{hyp}_2 = \\
\quad \begin{array}{ll}
\mathit{ms} & \approx\langle\ \mathit{complete}_{\mathrm{P}}\ \mathit{hyp}_1\ \rangle \\
\mathit{fib\text{-}rhs}\ \mathit{ms} & \approx\langle\ 0 :: ^{\sharp}\ \mathsf{zipWith\text{-}cong}\ \_+\_\ (\mathit{fib\text{-}unique}\ \mathit{ms}\ \mathit{ns}\ \mathit{hyp}_1\ \mathit{hyp}_2) \\
 & \qquad\qquad (1 :: ^{\sharp}\ \mathit{fib\text{-}unique}\ \mathit{ms}\ \mathit{ns}\ \mathit{hyp}_1\ \mathit{hyp}_2)\ \rangle \\
\mathit{fib\text{-}rhs}\ \mathit{ns} & \approx\langle\ \mathit{complete}_{\mathrm{P}}\ (\mathit{sym}\ \mathit{hyp}_2)\ \rangle \\
\mathit{ns} & \Box
\end{array}
\end{array}
$$

Here *sym* is the proof of symmetry of $\_\approx\_$ from Section 2, and $\mathit{complete}_{\mathrm{P}}$ shows that $\_\approx_{\mathrm{P}}\_$ is complete with respect to $\_\approx\_$:

$$
\begin{array}{l}
\mathit{complete}_{\mathrm{P}} : \mathit{xs} \approx \mathit{ys} \to \mathit{xs} \approx_{\mathrm{P}} \mathit{ys} \\
\mathit{complete}_{\mathrm{P}}\ (x :: \mathit{xs}{\approx}\mathit{ys}) = x :: ^{\sharp}\ \mathit{complete}_{\mathrm{P}}\ (^{\flat}\ \mathit{xs}{\approx}\mathit{ys})
\end{array}
$$

6 Destructors

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

$$
\begin{array}{l}
\mathit{fib} : \mathit{Stream}\ \mathbb{N} \\
\mathit{fib} = 0 :: ^{\sharp}\ (1 :: ^{\sharp}\ \mathit{zipWith}\ \_+\_\ \mathit{fib}\ (\mathit{tail}\ \mathit{fib}))
\end{array}
$$

The problem is the use of the destructor *tail*. Unrestricted use of destructors can lead to non-productive "definitions", as demonstrated by *bad* (see Section 1). 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:

$$
\begin{array}{l}
\textbf{data}\ \mathit{Stream}_{\mathrm{P}} : \mathit{Bool} \to \mathit{Set} \to \mathit{Set}_1\ \textbf{where} \\
\quad \begin{array}{lll}
[\_] & : \infty\ (\mathit{Stream}_{\mathrm{P}}\ \mathsf{true}\ A) & \to \mathit{Stream}_{\mathrm{P}}\ \mathsf{false}\ A \\
\_::\_ & : A \to \mathit{Stream}_{\mathrm{P}}\ \mathsf{false}\ A & \to \mathit{Stream}_{\mathrm{P}}\ \mathsf{true}\ A \\
\mathsf{tail} & : \mathit{Stream}_{\mathrm{P}}\ \mathsf{true}\ A & \to \mathit{Stream}_{\mathrm{P}}\ \mathsf{false}\ A \\
\mathsf{forget} & : \mathit{Stream}_{\mathrm{P}}\ \mathsf{true}\ A & \to \mathit{Stream}_{\mathrm{P}}\ \mathsf{false}\ A \\
\mathsf{zipWith} & : (A \to B \to C) \to \mathit{Stream}_{\mathrm{P}}\ b\ A \to \mathit{Stream}_{\mathrm{P}}\ b\ B & \to \mathit{Stream}_{\mathrm{P}}\ b\ C
\end{array}
\end{array}
$$

The type $\mathit{Stream}_{\mathrm{P}}\ b\ A$ 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 $\mathit{fib}_{\mathrm{P}}$ (an alternative would be to give zipWith a more general type):

$$\begin{aligned}&\mathit{fib}_{\mathrm{P}} : \mathit{Stream}_{\mathrm{P}}\ \mathsf{true}\ \mathbb{N}\\&\mathit{fib}_{\mathrm{P}} = 0 :: [^{\sharp}\,(1 :: \mathsf{zipWith}\ \_+\_\ (\mathsf{forget}\ \mathit{fib}_{\mathrm{P}})\ (\mathsf{tail}\ \mathit{fib}_{\mathrm{P}}))]\end{aligned}$$

The implementation of $[\![\_]\!]_{\mathrm{P}}$ for this language is very similar to that for the language in Section 7, so it is omitted here. For details of this implementation, the proof of correctness of $\mathit{fib}_{\mathrm{P}}$, and the proof of uniqueness of solutions of the defining equation for $\mathit{fib}_{\mathrm{P}}$, see Danielsson (2010a).

7 Other Chunk Sizes

The language of the previous section can be generalised to support other chunk sizes (see Danielsson (2010a)), i.e. other *moduli of production* (Endrullis et al. 2010). Larger chunk sizes can provide interesting situations. Consider the following alternative definition of the function *map* from Section 5:

$$\begin{aligned}&\mathit{map}_2 : (A \to B) \to \mathit{Stream}\,A \to \mathit{Stream}\,B\\&\mathit{map}_2\,f\,(x :: xs)\ \mathbf{with}\ ^{\flat}\,xs\\&\mathit{map}_2\,f\,(x :: xs)\ |\ y :: ys = f\,x ::^{\sharp} (f\,y ::^{\sharp} \mathit{map}_2\,f\,(^{\flat}\,ys))\end{aligned}$$

One can show that *map* and map_2 are extensionally equal:

$$\mathit{map}{\approx}\mathit{map}_2 : (f : A \to B) \to (xs : \mathit{Stream}\,A) \to \mathit{map}\,f\,xs \approx \mathit{map}_2\,f\,xs$$

However, assuming that pattern matching is "strict", they are not interchangeable. The following definition of the stream of natural numbers is productive, albeit not guarded:

$$\begin{aligned}&\mathit{nats} : \mathit{Stream}\,\mathbb{N}\\&\mathit{nats} = 0 ::^{\sharp} \mathit{map}\ \mathsf{suc}\ \mathit{nats}\end{aligned}$$

The definition that we get by replacing *map* by map_2, on the other hand, is not productive:

$$\begin{aligned}&\mathit{nats}_2 : \mathit{Stream}\,\mathbb{N}\\&\mathit{nats}_2 = 0 ::^{\sharp} \mathit{map}_2\ \mathsf{suc}\ \mathit{nats}_2\end{aligned}$$

The first element of nats_2 is 0, but map_2 needs to access the first *two* elements of its argument stream in order to output anything.

We can perhaps get a better picture of the situation above using the following language:

$$\begin{aligned}&\mathbf{data}\ \mathit{Stream}_{\mathrm{P}}\ (m : \mathbb{N}) : \mathbb{N} \to \mathit{Set} \to \mathit{Set}_1\ \mathbf{where}\\&\quad [\_] \ : \infty\,(\mathit{Stream}_{\mathrm{P}}\ m\ m\ A) \to \mathit{Stream}_{\mathrm{P}}\ m\ 0\ A\\&\quad \_::\_ : A \to \mathit{Stream}_{\mathrm{P}}\ m\ n\ A \to \mathit{Stream}_{\mathrm{P}}\ m\ (\mathsf{suc}\ n)\ A\\&\quad \mathsf{map} : (A \to B) \to \mathit{Stream}_{\mathrm{P}}\ m\ n\ A \to \mathit{Stream}_{\mathrm{P}}\ m\ n\ B\end{aligned}$$

$\mathit{Stream}_{\mathrm{P}}\ m\ n\ A$ is a language of programs which generate streams of As in chunks of size m, where the first chunk has size n. We can define WHNFs and the *whnf* function as follows:

$$\begin{aligned}&\mathbf{data}\ \mathit{Stream}_{\mathrm{W}}\ (m : \mathbb{N}) : \mathbb{N} \to \mathit{Set} \to \mathit{Set}_1\ \mathbf{where}\\&\quad [\_] \ : \mathit{Stream}_{\mathrm{P}}\ m\ m\ A \to \mathit{Stream}_{\mathrm{W}}\ m\ 0\ A\\&\quad \_::\_ : A \to \mathit{Stream}_{\mathrm{W}}\ m\ n\ A \to \mathit{Stream}_{\mathrm{W}}\ m\ (\mathsf{suc}\ n)\ A\end{aligned}$$

$$
\begin{aligned}
&\mathit{map}_{\mathrm{W}} : (A \to B) \to \mathit{Stream}_{\mathrm{W}}\ m\ n\ A \to \mathit{Stream}_{\mathrm{W}}\ m\ n\ B\\
&\mathit{map}_{\mathrm{W}}\ f\ [xs] \quad\ = [\mathsf{map}\ f\ xs]\\
&\mathit{map}_{\mathrm{W}}\ f\ (x :: xs) = f\ x :: \mathit{map}_{\mathrm{W}}\ f\ xs\\
&\mathit{whnf} : \mathit{Stream}_{\mathrm{P}}\ (\mathsf{suc}\ m)\ n\ A \to \mathit{Stream}_{\mathrm{W}}\ (\mathsf{suc}\ m)\ n\ A\\
&\mathit{whnf}\ [xs] \quad\ = [^{\flat}\ xs]\\
&\mathit{whnf}\ (x :: xs) = x :: \mathit{whnf}\ xs\\
&\mathit{whnf}\ (\mathsf{map}\ f\ xs) = \mathit{map}_{\mathrm{W}}\ f\ (\mathit{whnf}\ xs)
\end{aligned}
$$

Stream programs where all chunks are non-empty can then be turned into streams using guarded corecursion:

$$
\begin{aligned}
&\textbf{mutual}\\
&\quad [\![\_]\!]_{\mathrm{W}} : \mathit{Stream}_{\mathrm{W}}\ (\mathsf{suc}\ m)\ (\mathsf{suc}\ n)\ A \to \mathit{Stream}\ A\\
&\quad [\![\ x :: [xs]\ ]\!]_{\mathrm{W}} \quad\ = x :: {}^{\sharp}\ [\![\ xs\ ]\!]_{\mathrm{P}}\\
&\quad [\![\ x :: (y :: xs)\ ]\!]_{\mathrm{W}} = x :: {}^{\sharp}\ [\![\ y :: xs\ ]\!]_{\mathrm{W}}\\
&\quad [\![\_]\!]_{\mathrm{P}} : \mathit{Stream}_{\mathrm{P}}\ (\mathsf{suc}\ m)\ (\mathsf{suc}\ n)\ A \to \mathit{Stream}\ A\\
&\quad [\![\ xs\ ]\!]_{\mathrm{P}} = [\![\ \mathit{whnf}\ xs\ ]\!]_{\mathrm{W}}
\end{aligned}
$$

Using the language above we cannot define nats_2. The following code is ill-typed:

$$
\begin{aligned}
&\mathit{nats}_2 : \mathit{Stream}_{\mathrm{P}}\ 2\ 1\ \mathbb{N}\\
&\mathit{nats}_2 = 0 :: [^{\sharp}\ \mathit{map}\ \mathsf{suc}\ \mathit{nats}_2]
\end{aligned}
$$

On the other hand, the following definitions are accepted:

$$
\begin{aligned}
&\mathit{nats} : \mathit{Stream}_{\mathrm{P}}\ 1\ 1\ \mathbb{N}\\
&\mathit{nats} = 0 :: [^{\sharp}\ \mathit{map}\ \mathsf{suc}\ \mathit{nats}]
\end{aligned}
\qquad
\begin{aligned}
&\mathit{nats}'_2 : \mathit{Stream}_{\mathrm{P}}\ 2\ 2\ \mathbb{N}\\
&\mathit{nats}'_2 = 0 :: 1 :: [^{\sharp}\ \mathit{map}\ \mathsf{suc}\ \mathit{nats}'_2]
\end{aligned}
$$

Definitions which require the use of non-uniform chunk sizes cannot be handled using the language above. As an example of such a definition, consider the following presentation of the Thue-Morse sequence, due to Endrullis et al. (2010):

$$
\begin{aligned}
&\mathit{thue\text{-}morse} : \mathit{Stream}\ \mathit{Bool}\\
&\mathit{thue\text{-}morse} = \mathsf{false} :: {}^{\sharp}\ (\mathit{map}\ \mathit{not}\ (\mathit{evens}\ \mathit{thue\text{-}morse})\ \curlyvee\ \mathit{tail}\ \mathit{thue\text{-}morse})
\end{aligned}
$$

Here *evens xs* consists of every other element of *xs*, starting with the first, and $\_\curlyvee\_$ interleaves two streams: $(x :: xs)\ \curlyvee\ ys = x :: {}^{\sharp}\ (ys\ \curlyvee\ {}^{\flat}\ xs)$.[9]

8 Nested Applications

Before wrapping up, let us briefly consider nested applications of the function being defined, as in $\varphi\ (x :: xs) = x :: {}^{\sharp}\ \varphi\ (\varphi\ xs)$. Definitions with nested applications are common when programs are written using continuation-passing style. To handle such applications one can include a constructor for the function in the type of programs:

[9] A reviewer pointed me to Endrullis et al. (2010). After the paper was accepted I came up with a language which can be used to define *thue-morse*, see Danielsson (2010a).

```
data Stream_P (A : Set) : Set where
  _::_ : A → ∞ (Stream_P A) → Stream_P A
  φ_P  : Stream_P A → Stream_P A
data Stream_W (A : Set) : Set where
  _::_ : A → Stream_P A → Stream_W A

φ_W : Stream_W A → Stream_W A
φ_W (x :: xs) = x :: φ_P (φ_P xs)
whnf : Stream_P A → Stream_W A
whnf (x :: xs) = x ::♭ xs
whnf (φ_P xs)  = φ_W (whnf xs)
```

(The definition of $[\![\_]\!]_P$ is omitted above.) By turning streams into programs one can then define φ:

```
⌈_⌉ : Stream A → Stream_P A
⌈ x :: xs ⌉ = x ::♯ ⌈ ♭ xs ⌉

φ : Stream A → Stream A
φ xs = [[ φ_P ⌈ 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 Section 5, and to include a constructor for the congruence of φ_P in this language:

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

For further details, see Danielsson (2010a), who also establishes that φ's defining equation has a unique solution.

9 Related Work

This section is mainly concerned with discussing methods for establishing productivity *in systems based on guarded corecursion*. Other related work is discussed towards the end.

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 $c : E \to \mathbb{R} \times E$ 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 $E \to Stream\ \mathbb{R}$ (corresponding to $[\![\_]\!]_P$), which can be used to turn stream expressions into actual streams. Rutten then uses coinduction (expressed using bisimulations) to prove that the defined operations satisfy their intended defining equations, and that these equations have unique solutions.

There are some differences between Rutten's proof and the technique described here, other than the different settings (finality vs. guarded corecursion, bisimulations vs. guarded coinduction). One is that Rutten defines the variant of *fib* from Section 6 via two mutually recursive streams ($fib = 0 :: ♯\ fib'$ and $fib' = 1 :: ♯\ zipWith\ \_+\_\ fib\ fib'$); he does not discuss anything resembling the counting approaches of Sections 6 and 7. 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 (2010) 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 structural recursion and guarded corecursion. For completeness a short variant of this development is included in Appendix A.

Rutten's proof is closely related to a technique due to Bartels (2003). Bartels formulates the technique in a general categorical setting, and restricts the form of *whnf*, and in return proofs showing that the

definitions uniquely satisfy certain defining equations come for free. Furthermore Bartels manages to define *fib* without including it as a term in the language.

Niqui (2009, 2010) implements one of Bartels' corecursion schemes, λ-coiteration, in Coq. He states that this scheme 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.

Matthews (1999) and Di Gianantonio and Miculan (2003) describe general frameworks for defining values using a mixture of recursion and corecursion, based on functions which satisfy notions of contractivity. The methods seem to be quite general, and have been implemented (in Isabelle and Coq, respectively; note that guarded corecursion is not a primitive feature of Isabelle).

The implementations mentioned above (Matthews 1999; Di Gianantonio and Miculan 2003; Niqui 2009, 2010) provide you with unique solutions to equations, whereas when using the method described in this paper you need to prove correctness and uniqueness manually if you are interested in these properties. On the other hand, as pointed out in Section 5, there is rarely any need to pay this price when defining a proof. I suspect that circumstances determine which method is cheapest to use.

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. As mentioned in the introduction this work is orthogonal to the work presented here.

Conor McBride (personal communication) has developed a technique for establishing productivity, 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.

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

```
data _ᵛ (A : Set) : Set where
  return : A       → A ᵛ
  step   : ∞ (A ᵛ) → A ᵛ
```

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

```
data _ᵛ (A : Set) : Set₁ where
  return : A → A ᵛ
  _>>=_  : ∞ (B ᵛ) → (B → ∞ (A ᵛ)) → A ᵛ
```

One can note that this is very close 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 $\mathbb{N} \to A$, 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 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 circular breadth-first labelling function which is defined in Section 4. The technique is presented using the partial language Haskell, but Robert Atkey (personal communication) has later implemented it in Agda. 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.

Instead of working around the limitations of guarded corecursion one can include language features which make it easier to explain why programs are productive. One such feature is *sized types* (Hughes et al. 1996; Barthe et al. 2004; Abel 2009), and the λ-calculi of Buchholz (2005) provide other examples. Another approach is to use cleverer algorithms for establishing productivity. Endrullis et al. (2010, 2008) present algorithms based on eventually periodic moduli of production, which handle the definition of *thue-morse* from Section 7 automatically (except that, as presented, they only support first-order term-rewriting systems). The algorithms are tailored for streams; it seems to be hard to adapt them to, say, coinductive trees. Another algorithm is presented by Telford and Turner (1997). This algorithm does not handle *thue-morse* (Endrullis et al. 2010), but has the advantage of working for a large class of coinductive data types.

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.

10 Conclusions

I hope to have shown, through a number of examples, that the language-based approach to establishing productivity is useful. I am currently turning to it whenever I have a problem with guardedness; see Danielsson and Altenkirch (2010) and Danielsson (2010b) for some examples not included in this paper.

However, there are some problems with the method. As discussed in Section 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.

Acknowledgements. I would like to thank Andreas Abel, Thorsten Altenkirch, Conor McBride, Nicolas Oury and Anton Setzer for many discussions about coinduction, and Graham Hutton as well as several anonymous reviewers for useful feedback. I would also like to thank EPSRC for financial support (grant code: EP/E04350X/1).

A An Inductive Approximation of Stream Equality

Danielsson and Altenkirch (2010) prove that one subtyping relation is sound with respect to another using the technique described in this paper. This appendix outlines the proof, but in a simplified (and slightly different) setting: equality between streams.

Recall the definitions of *Stream* and stream equality, $\_\approx\_$, from Section 2. One can define a sound approximation of stream equality inductively as follows (using an idea due to Brandt and Henglein (1998)):

$$
\begin{array}{l}
\textbf{data}\ \_\vdash\_\approx\_\ (H : \mathit{List}\ (\mathit{Stream}\ A \times \mathit{Stream}\ A)) : \mathit{Stream}\ A \to \mathit{Stream}\ A \to \mathit{Set}\ \textbf{where} \\
\quad \_::\_ : (x : A) \to (x :: xs, x :: ys) :: H \vdash {}^\flat xs \approx {}^\flat ys \to H \vdash x :: xs \approx x :: ys \\
\quad \mathsf{hyp} : (xs, ys) \in H \to H \vdash xs \approx ys \\
\quad \mathsf{trans} : H \vdash xs \approx ys \to H \vdash ys \approx zs \to H \vdash xs \approx zs
\end{array}
$$

The intention is that, if one can prove $H \vdash xs \approx ys$, and all the assumptions in the list H are valid, then xs and ys should be equal. The first constructor of $\_\vdash\_\approx\_$ states that, in order to prove that $x :: xs$ and $x :: ys$ are equal, it suffices to show that ${}^\flat xs$ and ${}^\flat ys$ are equal, given the extra assumption that $x :: xs$ and $x :: ys$ are equal. The second constructor makes it possible to use the hypotheses in the list H ($\_\in\_$ encodes list membership), and the third constructor encodes transitivity. As an example, we can prove that the list $\mathit{repeat}\ x \approx x ::{}^\sharp \mathit{repeat}\ x$ is equal to itself as follows:

$$
\begin{array}{l}
\mathit{repeat\text{-}refl} : (x : A) \to [\,] \vdash \mathit{repeat}\ x \approx \mathit{repeat}\ x \\
\mathit{repeat\text{-}refl}\ x = x :: \mathsf{hyp}\ \mathsf{here}
\end{array}
$$

(The constructor here proves that the head of a list is a member of the list. In this case it is used at the type $(\mathit{repeat}\ x, \mathit{repeat}\ x) \in (\mathit{repeat}\ x, \mathit{repeat}\ x) :: [\,]$.)

Soundness of $\_\vdash\_\approx\_$ will now be established. The goal is to prove

$$\mathit{All}\ (\mathit{Valid}\ \_\approx\_)\ H \to H \vdash xs \approx ys \to xs \approx ys,$$

where $\mathit{All}\ P\ xs$ means that P holds for all elements in the list xs, and *Valid* is *uncurry* for stream predicates:

$$
\begin{array}{l}
\textbf{data}\ \mathit{All}\ (P : A \to \mathit{Set}) : \mathit{List}\ A \to \mathit{Set}\ \textbf{where} \\
\quad [\,] : \mathit{All}\ P\ [\,] \\
\quad \_::\_ : P\ x \to \mathit{All}\ P\ xs \to \mathit{All}\ P\ (x :: xs) \\
\mathit{Valid} : (\mathit{Stream}\ A \to \mathit{Stream}\ A \to \mathit{Set}) \to \mathit{Stream}\ A \times \mathit{Stream}\ A \to \mathit{Set} \\
\mathit{Valid}\ \_R\_\ (xs, ys) = xs\ R\ ys
\end{array}
$$

We begin by defining the program and WHNF types mutually as follows:

$$
\begin{array}{l}
\textbf{mutual} \\
\quad \textbf{data}\ \_\approx_{\mathrm{P}}\_ : \mathit{Stream}\ A \to \mathit{Stream}\ A \to \mathit{Set}\ \textbf{where} \\
\qquad \mathsf{sound} : \mathit{All}\ (\mathit{Valid}\ \_\approx_{\mathrm{W}}\_)\ H \to H \vdash xs \approx ys \to xs \approx_{\mathrm{P}} ys \\
\qquad \mathsf{trans} : xs \approx_{\mathrm{P}} ys \to ys \approx_{\mathrm{P}} zs \to xs \approx_{\mathrm{P}} zs \\
\quad \textbf{data}\ \_\approx_{\mathrm{W}}\_ : \mathit{Stream}\ A \to \mathit{Stream}\ A \to \mathit{Set}\ \textbf{where} \\
\qquad \_::\_ : (x : A) \to \infty\ ({}^\flat xs \approx_{\mathrm{P}} {}^\flat ys) \to x :: xs \approx_{\mathrm{W}} x :: ys
\end{array}
$$

Note that the first argument to the program sound refers to WHNFs. The function $\mathit{trans}_{\mathrm{W}} : xs \approx_{\mathrm{W}} ys \to ys \approx_{\mathrm{W}} zs \to xs \approx_{\mathrm{W}} zs$ can be defined using simple case analysis. The function $\mathit{sound}_{\mathrm{W}}$ is defined as follows, using structural recursion:

$$
\begin{array}{l}
\mathit{sound}_{\mathrm{W}} : \mathit{All}\ (\mathit{Valid}\ \_\approx_{\mathrm{W}}\_)\ H \to H \vdash xs \approx ys \to xs \approx_{\mathrm{W}} ys \\
\mathit{sound}_{\mathrm{W}}\ \mathit{valid}\ (\mathsf{hyp}\ h) = \mathit{lookup}\ \mathit{valid}\ h \\
\mathit{sound}_{\mathrm{W}}\ \mathit{valid}\ (\mathsf{trans}\ xs{\approx}ys\ ys{\approx}zs) = \mathit{trans}_{\mathrm{W}}\ (\mathit{sound}_{\mathrm{W}}\ \mathit{valid}\ xs{\approx}ys)\ (\mathit{sound}_{\mathrm{W}}\ \mathit{valid}\ ys{\approx}zs) \\
\mathit{sound}_{\mathrm{W}}\ \mathit{valid}\ (x :: xs{\approx}ys) = \mathit{proof} \\
\quad \textbf{where}\ \mathit{proof} = x ::{}^\sharp \mathsf{sound}\ (\mathit{proof} :: \mathit{valid})\ xs{\approx}ys
\end{array}
$$

In the first clause $\mathit{lookup} : \mathit{All}\ P\ xs \to x \in xs \to P\ x$ is used to fetch a proof from the "list" of valid

assumptions. In the third clause a *circular* proof is constructed using guarded corecursion; note that the list of valid assumptions is extended with the proof currently being defined. Given $trans_{\mathrm{W}}$ and $sound_{\mathrm{W}}$ it is easy to define *whnf* using structural recursion:

$$
\begin{aligned}
&\mathit{whnf} : \mathit{xs} \approx_{\mathrm{P}} \mathit{ys} \to \mathit{xs} \approx_{\mathrm{W}} \mathit{ys} \\
&\mathit{whnf}\ (\mathsf{sound}\ \mathit{valid}\ \mathit{xs}{\approx}\mathit{ys}) = \mathit{sound}_{\mathrm{W}}\ \mathit{valid}\ \mathit{xs}{\approx}\mathit{ys} \\
&\mathit{whnf}\ (\mathsf{trans}\ \mathit{xs}{\approx}\mathit{ys}\ \mathit{ys}{\approx}\mathit{zs}) = \mathit{trans}_{\mathrm{W}}\ (\mathit{whnf}\ \mathit{xs}{\approx}\mathit{ys})\ (\mathit{whnf}\ \mathit{ys}{\approx}\mathit{zs})
\end{aligned}
$$

The remaining pieces of the soundness proof are omitted (see Danielsson (2010a)).

References

Andreas Abel. Mixed inductive/coinductive types and strong normalization. In *Programming Languages and Systems, 5th Asian Symposium, APLAS 2007*, volume 4807 of *LNCS*, pages 286–301, 2009.

The Agda Team. The Agda Wiki. Available at `http://wiki.portal.chalmers.se/agda/`, 2010.

Thorsten Altenkirch and Nils Anders Danielsson. Termination checking in the presence of nested inductive and coinductive types. Extended abstract for talk given at the Workshop on Partiality and Recursion in Interactive Theorem Provers, 2010.

Falk Bartels. Generalised coinduction. *Mathematical Structures in Computer Science*, 13(2):321–348, 2003.

G. Barthe, M. J. Frade, E. Giménez, L. Pinto, and T. Uustalu. Type-based termination of recursive definitions. *Mathematical Structures in Computer Science*, 14(1):97–141, 2004.

Yves Bertot. Filters on coinductive streams, an application to Eratosthenes' sieve. In *Typed Lambda Calculi and Applications, 7th International Conference, TLCA 2005*, volume 3461 of *LNCS*, pages 102–115, 2005.

Yves Bertot and Ekaterina Komendantskaya. Using structural recursion for corecursion. In *Types for Proofs and Programs, International Conference, TYPES 2008*, volume 5497 of *LNCS*, pages 220–236, 2009.

Michael Brandt and Fritz Henglein. Coinductive axiomatization of recursive type equality and subtyping. *Fundamenta Informaticae*, 33(4):309–338, 1998.

Wilfried Buchholz. A term calculus for (co-)recursive definitions on streamlike data structures. *Annals of Pure and Applied Logic*, 136(1–2):75–90, 2005.

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 for Proofs and Programs, International Workshop TYPES'93*, volume 806 of *LNCS*, pages 62–78, 1994.

Nils Anders Danielsson. Code accompanying the paper. Available from `http://www.cs.nott.ac.uk/~nad/`, 2010a.

Nils Anders Danielsson. Total parser combinators. Submitted, 2010b.

Nils Anders Danielsson and Thorsten Altenkirch. Subtyping, declaratively; an exercise in mixed induction and coinduction. To appear in the proceedings of the Tenth International Conference on Mathematics of Program Construction (MPC'10), 2010.

Pietro Di Gianantonio and Marino Miculan. A unifying approach to recursive and co-recursive definitions. In *Types for Proofs and Programs, International Workshop, TYPES 2002*, volume 2646 of *LNCS*, pages 148–161, 2003.

Edsger W. Dijkstra. Hamming's exercise in SASL. EWD792 (privately circulated note), 1981.

Jörg Endrullis, Clemens Grabmayer, and Dimitri Hendriks. Data-oblivious stream productivity. In *Logic for Programming, Artificial Intelligence, and Reasoning, 15th International Conference, LPAR 2008*, volume 5330 of *LNCS*, pages 79–96, 2008.

Jörg Endrullis, Clemens Grabmayer, Dimitri Hendriks, Ariya Isihara, and Jan Willem Klop. Productivity of stream definitions. *Theoretical Computer Science*, 411(4–5):765–782, 2010.

Peter Hancock and Anton Setzer. Interactive programs in dependent type theory. In *Computer Science Logic, 14th International Workshop, 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, Proceedings of the 2008 SIGPLAN International Conference on Functional Programming*, pages 189–200, 2008.

John Hughes, Lars Pareto, and Amr Sabry. Proving the correctness of reactive systems using sized types. In *POPL '96, Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 410–423, 1996.

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.

John Matthews. Recursive function definition over coinductive types. In *Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs' 99*, volume 1690 of *LNCS*, pages 73–90, 1999.

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.

Peter Morris, Thorsten Altenkirch, and Conor McBride. Exploring the regular tree types. In *Types for Proofs and Programs, International Workshop, 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.

Milad Niqui. Coalgebraic reasoning in Coq: Bisimulation and the λ-coiteration scheme. In *Types for Proofs and Programs, International Conference, TYPES 2008*, volume 5497 of *LNCS*, pages 272–288, 2009.

Milad Niqui. Coiterative morphisms: Interactive equational reasoning for bisimulation, using coalgebras. Technical Report SEN-1003, Centrum Wiskunde & Informatica, 2010.

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.

Ben A. Sijtsma. On the productivity of recursive list definitions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 11(4):633–649, 1989.

Alastair Telford and David Turner. Ensuring streams flow. In *Algebraic Methodology and Software Technology, 6th International Conference, AMAST'97*, volume 1349 of *LNCS*, pages 509–523, 1997.

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.