Formalizing the Hamming Stream in Coq

Eelis van der Weegen

Radboud University Nijmegen, Faculty of Science

Bachelor's Thesis
Supervisor: Milad Niqui

January 2007

Contents

1 Introduction **2**

2 The Hamming Stream **4**

2.1 Specification 4

2.2 Implementations 5

3 Streams as Functions **7**

3.1 Productivity 7

3.2 Merge-based Implementation 9

3.3 Queue-based Implementation 10

4 Coinductive Streams **12**

4.1 Corecursion 13

4.2 Convertibility 14

4.3 Hamming Specification 15

4.4 Merge-based Implementation 16

4.5 Queue-based Implementation 17

4.6 Correctness 17

5 Evolving Initial Segments: Merge Redux **21**

5.1 Toward a New Merge-based Implementation 21

5.2 Correctness 24

5.3 Reflection 26

6 Program Extraction **28**

7 Conclusion **30**

Bibliography **30**

Chapter 1

Introduction

Infinite sequences, also called "streams", are studied extensively throughout mathematics and computer science. In this article we look at several approaches to working with streams using Coq [11]. Coq is a versatile proof assistant with which formal specifications, programs, and proofs can be developed and mechanically checked. It is founded on the Predicative Calculus of (Co)Inductive Constructions [1], a constructive and dependent type theory interpreted along the lines of the Curry-Howard correspondence, with support for algebraic data types.

Two aspects in particular are of interest, namely:

1. How can streams be represented in Coq? What type do we assign them?
2. What obstacles (if any) arise when we try to specify, define recursively, and reason about streams? And how can we work around those obstacles?

The Hamming stream, described in Chapter 2, is used as a case study. It is a stream with a straightforward specification for which elegant, concise recursive implementations exist—at least in other systems. To what extent those implementations can be mimicked in Coq will be examined in the rest of this article, which is mostly organized following the pursuit of certified Hamming stream implementations in Coq.

In Chapter 3 we look at a stream representation where streams are viewed as functions. Next, in Chapter 4, we look at a stream representation based on coinductive data types. Finally, in Chapter 5 we look at a stream representation based on growing lists of finite length. In all three chapters, we attempt to use the respective stream representations for the Hamming stream.

In Chapter 6, when our Hamming formalization efforts have come to an end, we briefly look at the practicality of program extraction applied to our results.

Basic familiarity with the Coq proof assistant—its calculus, specification language, and standard libraries—is assumed. The article is accompanied by a complete Coq formalization of all definitions and proofs. Although all our definitions and proofs primarily relate to the calculus on which Coq is founded rather than the Coq system itself, in our discussion we identify the two and simply say "Coq" when either is meant.

We only consider strictly infinite streams with a beginning and no end. We do not consider finite or potentially-finite streams, or streams that are infinite in two directions (such as $\mathbb{Z}$ when viewed as a stream).

Notation We use ML-style parentheses everywhere (e.g. $(f\ \ (g\ \ x)\ \ y)$ instead of $f(g(x), y)$).

Lists and List Operations

We will often use Coq's `list` data type, which is assumed to be familiar. We will use two different list indexing functions for it. The first, `nth`, is part of the Coq standard library and has type $\forall\ (i : \mathbb{N})\ (l :$ `list` $T)\ (d : T),\ T$. The dummy argument d is returned when $i \geq$ `length` l. The second indexing function, `snth` (for "safe n'th"), is not part of the Coq standard library and has type $\forall\ (i : \mathbb{N})\ (l :$ `list` $T), i <$ `length` $l \rightarrow T$. Instead of a dummy value it takes a proof that excludes the case where `nth` would have returned its dummy value.

In our formalization we occasionally need to work with lists that are known not to be empty. For these, we define the type `ne_list` of non-empty lists:

```
Inductive ne_list (T: Set): Set :=
  | one: T -> ne_list T
  | cons: T -> ne_list T -> ne_list T.
```

In addition to obvious operations on non-empty lists like `head` and `tail`, one operation that will pop up in our definitions is `ne_list.from_plain`, of type $\forall\ T,\ T \rightarrow$ `list` $T \rightarrow$ `ne_list` T. It constructs a non-empty list from a (potentially empty) plain list and a separate head element.

Also, `ne_list` T has been made coercible to `list` T, meaning that there is a conversion function of type $\forall\ T,$ `ne_list` $T \rightarrow$ `list` T that is applied implicitly whenever a term of type `ne_list` T occurs in a context where a term of type `list` T was expected.

Chapter 2

The Hamming Stream

2.1 Specification

The Hamming stream consists of those natural numbers whose prime divisors are all ≤ 5, listed in increasing order without duplicates. Its first 20 elements are:

$$1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, 16, 18, 20, 24, 25, 27, 30, 32, 36$$

It was popularized by Edsger Dijkstra [3], who also gave a proof [5] of correctness of the merge-based implementation discussed in the next section [1].

Generalizing the constant 5 leads to what are called streams of k-smooth numbers. A number is k-smooth if its prime divisors are all $\leq k$. In this article and its accompanying Coq formalization, we adopt a slightly broader notion of smoothness, namely that a number is smooth relative to a list of numbers l if it can be written as a product of numbers in l:

```
Parameter multipliers: ne_list nat.

Hypothesis multipliers_nontrivial:
  forall m, In m multipliers -> m > 1.

Inductive smooth: nat -> Prop :=
  | smooth_one: smooth 1
  | smooth_more x y:
      smooth x -> In y multipliers -> smooth (x * y).
```

We use this broader notion of smoothness because it is both easier to work with (since it turns out that this way the formalization does not need any theory about prime numbers), and strengthens our results (since normal k-smoothness is reduced to a mere special case where the list of multipliers contains the first successive primes up to and including k).

Our entire formalization is parameterized by the list of multipliers. Plugging in the list of numbers 2, 3, and 5 yields definitions and correctness proofs of the normal Hamming stream. For illustrative purposes, definitions shown in this article often use the numbers 2, 3, and 5 directly. In our discussion we will use the term "Hamming stream" to refer to any and all of the variants above, generalized or not.

[1] Unfortunately, to the extent that Dijkstra's proof is formal, the logical framework it uses (if any) seems entirely incompatible with the one used by Coq, so it is of little use to us.

The precise formulation of a Coq specification of the Hamming stream depends on the chosen stream representation, but we can characterize the three main properties as follows.

1. If a number occurs in the stream, it is smooth.
2. If a number is smooth, it occurs in the stream.
3. The elements occur in increasing order without duplicates.

We will refer to the first and second properties as “soundness” and “completeness”, respectively. For convenience, we will say that a list or stream “increases” if its elements occur in increasing order without duplicates. With this, the third property can simply be stated as: the stream increases.

In Section 4.3 a precise Coq formulation is given for a coinductive stream representation.

2.2 Implementations

As mentioned, in some other systems the Hamming stream can be implemented very concisely. Our goal will be to approximate two such implementation—one existing and one new—in Coq. The first implementation, here expressed in Haskell, is as follows.

```
merge (x:xs) (y:ys)
  | x < y = x : merge xs (y:ys)
  | x > y = y : merge (x:xs) ys
  | x == y = x : merge xs ys

hamming = 1 :
  merge (map (* 2) hamming) (
    merge (map (* 3) hamming) (
      map (* 5) hamming ))
```

We call this the *merge-based* implementation. It is an often-cited example demonstrating both the expressive power and potential for efficiency of fully lazily evaluated functional programming languages.

Much can be said about to what extent the implementation is intuitive or elegant. However, ultimately this has proven to be very much in the eye of the beholder, arguably depending above all on their affinity for and experience with the functional programming paradigm in general.

The second implementation, again in Haskell, is somewhat similar to the previous one.

```
enqueue x [] = [x]
enqueue x (y:ys)
  | x < y = x : y : ys
  | x > y = y : enqueue x ys
  | x == y = y : ys

ham_from (h:t) = h : ham_from (
  enqueue (h * 2) (
    enqueue (h * 3) (
      enqueue (h * 5) t )))

hamming = ham_from [1]
```

We call this the *queue-based* implementation[2]. It differs from the merge-based implementation in that it explicitly processes one element at a time while the merge-based implementation manipulates entire

[2] The term “queue” is used very informally here. It merely reflects this author’s impression of the algorithm’s behavior.

streams at a time. It is in this sense somewhat more imperative in nature. We will see later that this makes all the difference—for the better—in our formalization efforts. This is no accident, as it was designed explicitly with Coq formalization in mind.

Note that in both implementations, Haskell's lazy evaluation allows its ordinary list data type to be used to represent infinite lists. This approach and its applicability in Coq is discussed in detail in Chapter 4.

Having familiarized ourselves with the Hamming stream, we now look at our first stream representation candidate.

Chapter 3

Streams as Functions

A very straightforward way of representing streams is as functions from $\mathbb{N}$ to some type T. For example, the popular Fibonacci sequence can be specified as follows.

$$\begin{aligned} \text{fib } 0 &= 0 \\ \text{fib } 1 &= 1 \\ \text{fib } (n+2) &= \text{fib } n + \text{fib } (n+1) \end{aligned}$$

This representation and form of definition can be used in Coq:

```
Definition f_stream (T: Set) := nat -> T.

Definition fib: f_stream nat :=
  fix f (n: nat) {struct n}: nat :=
    match n with
    | 0 => 0
    | 1 => 1
    | S (S n'' as n') => f n' + f n''
    end.
```

(The '`f_`' prefix is used to distinguish this functional approach from alternative approaches in later chapters.)

3.1 Productivity

We used the `fix` construct to define `fib` recursively. In doing so, Coq required that we identify a structurally decreasing argument (pointed to by `{struct n}`), to ensure that the recursion eventually terminates—a property Coq requires for all recursive function definitions. This Coq definition then is not just a direct translation of the set of equations that made up the original specification, but also provides a proof that, using this definition, the value of any element of the stream can be computed in finite time.

Definition 1. A stream definition that provides the means to compute any desired element value in finite time is called *productive* [10, 4].

Note that non-recursive stream definitions are trivially productive. Informally, productivity is what separates stream implementations from mere specifications. In the case of fib, turning the specification into an implementation by showing its productivity was easy; all we had to do was point at the parameter n, which

happened to nicely decrease at the recursive call. In general though, showing the productivity of a specification can be very hard, or indeed impossible if the specification interpreted as an implementation would simply not be productive. As a (silly) example of the latter, consider the following alternative specification of the Fibonacci stream.

$$\begin{aligned} \text{fib' } 0 &= 0 \\ \text{fib' } 1 &= 1 \\ \text{fib' } n &= \text{fib' } (n+2) - \text{fib' } (n+1) \end{aligned}$$

As a specification, this set of equations is equivalent to fib, and denotes the same stream. However, it cannot be used as an implementation, because when used to compute, say, fib' 3, the computation would never complete. In Coq fib' simply cannot be written down as a recursive function definition, because there is no decreasing argument. We *can* formalize this specification, but only in the form of a passive predicate over streams which does not provide the means to compute anything:

```
Definition is_fib (s: f_stream nat): Prop :=
  s 0 = 0 /\
  s 1 = 1 /\
  forall n, s n = s (n + 2) - s (n + 1).
```

For the Fibonacci stream there is little point in using this alternative specification in predicate form, as the original specification was both more intuitive and productive. For the Hamming stream, however, giving a specification in predicate form allows us to directly state the three properties from Section 2.1 without any concern for productivity, so we will do this in Section 4.3.

In constructive logics such as the one used by Coq, providing an implementation of something is equated to proving its very existence. From this perspective our goal of coming up with an implementation of the Hamming stream can actually be stated in much more grandiose terms: our goal is to prove the existence of the Hamming stream (or rather of a family of streams that includes the Hamming stream).

Since Coq demands provable termination for *all* computation, it follows that it will only permit provably productive stream implementations, regardless of stream representation. In this functional representation, this productivity requirement manifests itself in the form of recursion being structural rather than unbounded. In Chapters 4 and 5 we will see how the productivity requirement manifests itself in other stream representations.

Proceeding with this function-based representation of streams, we can define a few convenient functions that operate on streams:

```
Definition f_head T (s: f_stream T): T := s 0.

Definition f_tail T (s: f_stream T): f_stream T := fun n => s (S n).

Definition f_cons (T: Set) (x: T) (s: f_stream T): f_stream T :=
  fun n => match n with 0 => x | S n' => s n' end.

Definition f_map (T U: Set) (g: T -> U) (s: f_stream T): f_stream U :=
  fun n => g (s n).
```

These are all accepted by Coq. With these fundamentals in place, we now attempt to implement the Hamming stream, starting with the implementation based on merge.

3.2 Merge-based Implementation

First, we must define the merge function, which happens to be recursive. Translating the Haskell version to Coq as directly as possible, we arrive at:

```
Fixpoint f_merge (a b: f_stream nat) {struct ?}: f_stream nat :=
  match lt_eq_lt_dec (f_head a) (f_head b) with
  | inleft (left _) => f_cons (f_head a) (f_merge (f_tail a) b)
  | inleft (right _) => f_cons (f_head a) (f_merge (f_tail a) (f_tail b))
  | inright _ => f_cons (f_head b) (f_merge a (f_tail b))
  end.
```

(The type of `lt_eq_lt_dec` is $\forall\ (n\ m : \mathbb{N}),\ \{n < m\} + \{n = m\} + \{m < n\})$.

Unfortunately, there is nothing to write at the question mark, where the decreasing argument would be specified (as it was in the definition of `fib`). A decreasing argument must be of an inductive type, which `f_stream nat` is not; `f_tail` s is not structurally smaller than s, at least not in an inductive sense. Fortunately, we can open up the `f_stream nat` abstraction to reveal the position parameter, on which we can perform straightforward recursion. This works because regardless of the outcome of the heads comparison, the result term is a `f_cons` application that trivially decomposes into head and tail components corresponding to the cases where the position argument is zero and non-zero, respectively.

```
Definition f_merge: f_stream nat -> f_stream nat -> f_stream nat :=
  fix g (a b: f_stream nat) (n: nat) {struct n}: nat :=
  match n with
  | 0 =>
      match lt_eq_lt_dec (f_head a) (f_head b) with
      | inleft _ => f_head a
      | inright _ => f_head b
      end
  | S n' =>
      match lt_eq_lt_dec (f_head a) (f_head b) with
      | inleft (left _) => g (f_tail a) b n'
      | inleft (right _) => g (f_tail a) (f_tail b) n'
      | inright _ => g a (f_tail b) n'
      end
  end.
```

While not as elegant, this works.

Next up is the Hamming stream itself. An initial attempt fails analogously to the initial `f_merge` attempt, albeit in an even more dramatic way:

```
Fixpoint f_hamming {struct ?}: f_stream nat :=
  f_cons 1 (f_merge (f_map (mult 2) f_hamming)
    (f_merge (f_map (mult 3) f_hamming)
      (f_map (mult 5) f_hamming))).
```

There is simply no argument to recurse on. Unlike before, however, opening up `f_stream` cannot save us:

```
Fixpoint f_hamming (n: nat) {struct n}: nat :=
  match n with
  | 0 => 1
  | S n' =>
    (f_merge (f_map (mult 2) f_hamming)
      (f_merge (f_map (mult 3) f_hamming) (f_map (mult 5) f_hamming))) n'
  end.
```

Coq rejects this definition because it only allows recursive calls that are passed a structurally smaller argument directly, while in the above definition any actual recursive calls are delegated to other functions. The recursion is not simple enough for Coq to recognize the productivity of the definition.

This time no obvious fix comes to mind. There appears to be no clear-cut way to convince Coq of the productivity of this definition, so for now we turn our attention to the implementation based on queues. We will revisit the merge-based implementation in Chapters 4 and 5.

3.3 Queue-based Implementation

We begin with the `enqueue` function, for which the Haskell code from Chapter 2 can be translated almost directly. The only real change is that we change some data types to non-empty lists and point at the structurally decreasing argument.

```
Fixpoint enqueue (n: nat) (l: list nat) {struct l}: ne_list nat :=
  match l with
  | nil => one n
  | cons h t =>
    match lt_eq_lt_dec n h  with
    | inleft (left _) => ne_list.from_plain n l
    | inleft (right _) => ne_list.from_plain h t
    | inright _ => cons h (enqueue n t)
    end
  end.
```

Next, we attempt the `ham_from` function.

```
Fixpoint ham_from (l: ne_list nat) {struct ?}: f_stream nat :=
  f_cons (head l) (ham_from (
    enqueue (head l * 2) (
      enqueue (head l * 3) (
        enqueue (head l * 5) (tail l))))).
```

This attempt fails because there is no suitable argument to recurse on (*l* most certainly does not get structurally smaller). Fortunately, the trick of opening up `f_stream nat` can help us out again, thanks to the

fact that the result term is a `f_cons` application:

```
Fixpoint ham_from (l: ne_list nat) (n: nat) {struct n}: nat :=
  match n with
  | 0 => head l
  | S n' => ham_from (
      enqueue (head l * 2) (
        enqueue (head l * 3) (
          enqueue (head l * 5) (tail l)))) n'
  end.
```

Finally, `hamming` is just an application of `ham_from`:

```
Definition hamming: f_stream nat := ham_from (one 1).
```

We will not prove the correctness of this implementation here. In Section 4.5 we will find that the queue-based definition can also be expressed using a coinductive stream representation, and we will prove correctness of a coinductive queue-based Hamming stream implementation there.

This queue-based implementation was basically accepted in its original form, while Coq would not accept the merge-based implementation. The reason for this is that the productivity of the former was obvious enough for Coq to be able to recognize it in the form of structural recursion, while the productivity of the latter was too complex for Coq to recognize—at least in this functional stream representation. To gain more insight into whether this is just an artifact of this particular stream representation or a problem inherent in the merge-based implementation, we now leave the functional stream representation for what it is and look at another stream representation.

Chapter 4

Coinductive Streams

In the previous chapter we viewed streams as functions from $\mathbb{N}$ to some element type. A completely different way of looking at streams is to view them as instances of some kind of algebraic data type. Looking at the canonical inductive definition of list,

```
Inductive list (T: Set): Set :=
  nil | cons: T -> list T -> list T.
```

it might occur to us that a hypothetical term consisting of an infinite sequence of applications of the `cons` constructor,

$$\texttt{cons}\ x_0\ (\texttt{cons}\ x_1\ (\texttt{cons}\ x_2\ (\texttt{cons}\ x_3\ \ldots))),$$

would bear the hallmarks of a stream.

In some systems, such as Haskell, this is a valid representation of streams (Haskell examples using this representation were given in Chapter 2). Coq however is not one such system, as terms of inductive types can only be finite.

Definition 2. We call a term *finite* if it always reduces to a term in canonical form (of finite size) in finitely many steps.

In Coq, the guaranteed finiteness of terms of inductive types is inextricably tied to guaranteed termination of recursive computation. Through structural recursion the two both guarantee and depend on one another: constructing an infinite term of inductive type would require an infinite recursive computation, but because that recursion would be structural it would require an infinite term of inductive type, thus completing the cyclic dependency that simultaneously precludes both infinite terms and non-terminating computations. Consequently, we cannot represent streams using inductive types.

In practical terms, while we could simply write:

```
Inductive i_stream (A: Set): Set :=
  i_cons: A -> i_stream A -> i_stream A.
```

there simply would be no way to ever construct a term of this type.

Proposition 1. No term of type `i_stream` T exists (for any T).

Proof. To say that such a term does not exist is to say that its existence implies inconsistency, so we prove `i_stream` $T \to \bot$. The lack of a terminal constructor for `i_stream` has caused Coq to generate a somewhat curious induction principle:

$$\forall\, P,\ \texttt{i_stream}\ T \to \texttt{Prop}, \Big(\forall x\ s, P\ s \to P\,(\texttt{i_cons}\ \ x\ \ s)\Big) \to \forall s, P\ \ s$$

We can apply this directly, taking $(\lambda s.\bot)$ for P, leaving a trivial recursive step of $\bot \to \bot$. □

Coq provides a variant of inductive types called *coinductive* types [2, 7] [1, Ch.13], the goal of which is to facilitate (potentially) infinite terms without introducing the potential for non-terminating computation. Coinductive definitions look very similar to inductive ones. The type of potentially infinite lists can be written coinductively as:

```
CoInductive c_list (T: Set): Set :=
  cl_nil | cl_cons: T -> c_list T -> c_list T.
```

Since our streams are always infinite, we simply drop the nil constructor:

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

This definition is part of the Coq standard library (in the Coq.Lists.Streams module).

Coinductive types differ from inductive types in two important ways, both of which reflect their potential for infinity. First, the decreasing argument in a structural recursion may not be of a coinductive type, because those are not well-founded. Similarly, no induction or recursion principles are generated for coinductive types.

Second, terms of coinductive types can be constructed corecursively.

4.1 Corecursion

Corecursion allows one to construct infinite terms of coinductive types (ordinary recursion can be used to construct finite terms of coinductive types). Coq supports corecursion in the form of `CoFixpoint` constructs:

```
CoFixpoint enumerate (T: Set) (g: nat -> T) (n: nat): Stream T :=
  Cons (g n) (enumerate g (S n)).
```

Naive addition of corecursion to a strongly normalizing system would introduce infinite reduction paths for terms involving corecursion (since corecursion is seemingly unbounded), resulting in the loss of the strong normalization property. The solution is to employ a lazy reduction strategy for (sub)terms of coinductive types. That is, to only reduce them if they are the subject of a pattern matching construct that is being evaluated, and then to reduce only as much as is needed to match one of the patterns.

Lazy reduction of terms of coinductive types is not sufficient to regain strong normalization, however. Consider,

```
CoFixpoint silly: Stream nat := silly.
```

Were Coq to accept this, no amount of reduction of `silly` in

```
match silly with Cons h t => h end
```

would ever result in a term matching the pattern. What is needed is a guarantee that continued reduction of corecursive terms steadily reveals ever more constructor applications, so that when they are reduced as the subject of match constructs, eventually one of the patterns will match. This property coincides with the notion of productivity as defined in Section 3.1. Thus, a mechanism is needed to ensure that only productive corecursive definitions are permitted, so that definitions like that of `silly` are rejected. The mechanism used by Coq is a syntactic requirement, or *guard condition* [2], on corecursive definitions.

Definition 3. A corecursive definition is *guarded* if, regardless of which branches are selected in pattern match constructs (if any), corecursive occurrences in the result term are nested, and nested only, in applications of constructors of the coinductive type. In the subject terms of pattern match constructs there may be corecursive occurrences, but only if the patterns and result term obviously do not depend on values of the stream being defined.[1]

It is easy to see that guarded stream definitions are indeed productive. It is even easier to see that the merge-based Hamming implementation is not guarded, while the queue-based implementation is. More on that in a moment.

4.2 Convertibility

Interestingly, this representation of streams is trivially to- and from-convertible with the functional representation from the previous chapter:

```
Definition c_to_f T (s: Stream T): f_stream T := fun i => Str_nth i s.
Definition f_to_c T (s: f_stream T): Stream T := enumerate s 0.
```

To show that these are really eachother's inverse, ideally we would like to prove

$$\left(\forall f, \texttt{c_to_f (f_to_c } f) = f\right) \wedge \left(\forall c, \texttt{f_to_c (c_to_f } c) = c\right).$$

Unfortunately, Coq's usual Leibniz equality is too strong on either side, for reasons we will not go into here. We will instead prove extensional equality on both sides. For `Stream`, extensional equality is expressed as a coinductive predicate:

```
CoInductive EqSt T (s1 s2: Stream T): Prop :=
  eqst: hd s1 = hd s2 -> EqSt (tl s1) (tl s2) -> EqSt s1 s2.
```

Proposition 2. `c_to_f` and `f_to_c` are eachother's inverse. That is,

$$\left(\forall f\ n,\ \texttt{c_to_f (f_to_c } f)\ n = f\ \ n\right) \wedge \left(\forall c,\ \texttt{EqSt (f_to_c (c_to_f } c))\ c\right).$$

Proof. We begin with the left side of the conjunction. Unfolding `c_to_f` and `f_to_c`, we get

$$\forall f\ n,\ \texttt{Str_nth}\ n\ (\texttt{enumerate}\ f\ \ 0) = f\ \ n.$$

Rewriting the right side of this equation as $\texttt{Str_nth}\ 0\ \ (\texttt{enumerate}\ f\ \ n)$ we get a specific instance of the more general statement that

$$\forall f\ x\ u\ m, \texttt{Str_nth}\ (u + x)\ \ (\texttt{enumerate}\ f\ m) = \texttt{Str_nth}\ u\ (\texttt{enumerate}\ f\ \ (m + x))$$

with $u = m = 0$ and $x = n$. This more general statement is easily proved by induction on x.

Next, the right side of the conjunction. Unfolding `f_to_c` on the left side of the equation and replacing c with $\texttt{Str_nth_tl}\ 0\ \ c$ on the right side, we get

$$\forall c, \texttt{EqSt (enumerate (c_to_f } c)\ 0)\ (\texttt{Str_nth_tl}\ 0\ \ c).$$

We now generalize 0 and apply coinduction. The heads are immediately equal (after reduction). For the tails we apply the coinduction hypothesis after minor rewriting. □

[1] The use of the word "obvious" here doesn't sound very syntactically verifiable. Unfortunately, we cannot accurately characterize the syntactical nature of this obviousness without discussing Coq internals. This is because Coq does not actually check the guardedness condition until it has internally performed several syntactical rewritings on the definition, which may shuffle around or optimize away pattern match constructs.

4.3 Hamming Specification

We now take a moment to revisit the Hamming specification given in Chapter 2 and state it in terms of the coinductive stream type `Stream nat`. Recall that our specification was comprised of the following three properties:

1. If a number occurs in the stream, it is smooth. ("soundness")
2. If a number is smooth, it occurs in the stream. ("completeness")
3. The elements occur in increasing order without duplicates.

We define the following record type to represent coinductive streams having these properties.

```
Record hamming_stream: Set :=
  { s: Stream nat
  ; s_sound: everywhere smooth s
  ; s_complete: forall n, smooth n -> in_stream (eq n) s
  ; s_increases: increases s
  }.
```

Here, `everywhere`, `in_stream` and `increases` are defined as follows.

```
CoInductive ForAll (A: Set) (P: Stream A ->Prop) (x: Stream A): Prop :=
    HereAndFurther: P x -> ForAll P (tl x) ->ForAll P x

Definition everywhere (A: Set) (p: A -> Prop):
  Stream A -> Prop := ForAll (fun s => p (hd s)).

Inductive Exists (A: Set) (P: Stream A -> Prop) (x: Stream A): Prop :=
    Here: P x -> Exists P x | Further: Exists P (tl x) -> Exists P x


Definition in_stream (A: Set) (p: A -> Prop): Stream A -> Prop :=
  Exists (fun s => p (hd s)).

CoInductive increases: Stream nat -> Prop :=
  | increases_ctor x y r: x < y -> increases (Cons y r) ->
      increases (Cons x (Cons y r)).
```

Of these, `ForAll` and `Exists` are part of the Coq standard library.

We intend that our Hamming specification identify a unique stream, and we can prove that it does.

Proposition 3. For any two objects of type `hamming_stream`, their stream subobjects are extensionally equal. That is,

$$\forall\,(a\;b : \texttt{hamming_stream}),\ \texttt{EqSt (s } a\texttt{) (s } b\texttt{)}.$$

Proof. The proof is by coinduction, meaning that we prove the heads equal and invoke the coinduction hypothesis for the tails. We do not apply coinduction on the proposition as stated though, because it would make a useless coinduction hypothesis. It would be useless because we would not be able to apply it to the stream tails, because those tails are not themselves `hamming_stream`'s (after all, they lack the first Hamming number, 1).

We therefore first generalize the statement into a form better suited for coinduction. Since the completeness requirement in `hamming_stream` is what prevents us from using coinduction, it should not come as a surprise that it is this property that we generalize:

$$
\begin{gathered}
\forall\,(a\;b : \texttt{Stream nat}),\\
\texttt{everywhere smooth}\;a \to \texttt{increases}\;a \to\\
\texttt{everywhere smooth}\;b \to \texttt{increases}\;b \to\\
(\forall\,n, \texttt{min (hd}\;a\texttt{) (hd}\;b\texttt{)} \le n \to \texttt{smooth}\;n \to \texttt{in_stream (eq}\;n\texttt{)}\;a) \to\\
(\forall\,n, \texttt{min (hd}\;b\texttt{) (hd}\;a\texttt{)} \le n \to \texttt{smooth}\;n \to \texttt{in_stream (eq}\;n\texttt{)}\;b) \to\\
\texttt{EqSt}\;a\;b.
\end{gathered}
$$

As a coinduction hypothesis this new statement can be applied to the tails of a and b without any problems. What remains is to prove that the heads of a and b are equal. We prove this by showing that their inequality would lead to contradiction. If they are unequal, then one is bigger than the other. (These last two steps are constructively permissible because equality and order on natural numbers are decideable.) Call the stream with the bigger head p. Since p increases, this means that the value of the smaller head *does not* occur in p. But since that value is both smooth (by the soundness property of its stream) and greater than or equal to the lower bound of the generalized completeness property, the latter states that it *does* occur in p. Hence contradiction, so the heads must be equal. □

4.4 Merge-based Implementation

Using corecursion, we can define `merge` exactly the way we would like to:

```
CoFixpoint merge (a b: Stream nat): Stream nat :=
  match a, b with Cons ha ta, Cons hb tb =>
    match lt_eq_lt_dec ha hb with
    | inleft (left _) => Cons ha (merge ta b)
    | inleft (right _) => Cons ha (merge ta tb)
    | inright _ => Cons hb (merge a tb)
    end
  end.
```

Unfortunately, as we observed earlier the merge-based definition of the Hamming stream itself does not satisfy Coq's guardedness condition described in Section 4.1. The `hamming` occurrences in the following definition are not guarded because they are enclosed in `merge` and `map` applications.

```
CoFixpoint hamming: Stream nat :=
  Cons 1 (merge (map (mult 2) hamming)
    (merge (map (mult 3) hamming)
      (map (mult 5) hamming))).
```

The situation is very similar to the one we faced when we tried to build a merge-based Hamming implementation using the functional stream representation: in both cases Coq's syntactic requirements designed to enforce termination/productivity are not sophisticated enough to recognize this property for this form of Hamming implementation.

Seeing no solution, we again temporarily give up on the merge-based implementation and focus on the implementation based on queues. We will revisit the merge-based implementation one final time in Chapter 5.

4.5 Queue-based Implementation

We re-use the enqueue definition from Section 3.3 and proceed immediately with ham_from:

```
CoFixpoint ham_from (l: ne_list nat): Stream nat :=
  Cons (head l) (ham_from (
    enqueue (head l * 2) (
      enqueue (head l * 3) (
        enqueue (head l * 5) (tail l))))).
```

This definition satisfies the guardedness condition, and is accepted. Again, we get the Hamming stream by applying ham_from:

```
Definition hamming: Stream nat := ham_from (one 1).
```

Apart from a little overhead caused by the issue of potential list emptiness, this is a very succinct definition indeed, quite closely resembling the Haskell version.

Before we proceed to prove the correctness of this implementation, we note that again the queue-based implementation was basically accepted in its original form, while Coq again would not accept the merge-based implementation. We conclude that the latter is simply inherently less obviously productive, making it less suitable for Coq formalization. Nevertheless, we will return to the merge-based implementation in Chapter 5, where we present an alternative indirect definition scheme that externalizes the productivity property as an (optional) proof obligation, allowing us to give a merge-based implementation after all.

But first, we prove the correctness of this queue-based implementation.

4.6 Correctness

Before we start, we write ham_from as:

```
CoFixpoint ham_from (l: ne_list nat): Stream nat :=
  Cons (head l) (ham_from (process l)).
```

where process is defined as:

```
Definition process (l: ne_list nat): ne_list nat :=
  enqueue (head l * 2) (
    enqueue (head l * 3) (
      enqueue (head l * 5) (
        tail l))).
```

Recall from Section 2.1 that the actual Coq formalization is generalized over the list of multipliers. For process, this means that the actual definition used in our proofs is implemented as a fold over the list of multipliers:

```
Definition process (l: ne_list nat): ne_list nat :=
  enqueue (head l * head multipliers)
    (fold_right (fun h => enqueue (head l * h)) (tail l) (tail multipliers)).
```

For a list of multipliers consisting of the numbers 2, 3, and 5, this definition is equivalent to the first one. One enqueue operation was taken outside the fold to ensure that the resulting list is non-empty.

The three properties that make up correctness—soundness, completeness, and increase—are all "layered" in the sense that proofs of these properties for composed operations generally depend on lemmas that state properties similar in spirit for the operation's primary suboperation. For example, we will see that the soundness proof for `hamming` depends on a soundness lemma for `process`, the proof of which in turn depends on a soundness lemma for `enqueue`.

4.6.1 Soundness

Following the layered approach described above, we start with a soundness lemma about `enqueue`.

Lemma 1. `enqueue` is sound with regard to its purpose. That is,

$$\forall\, l\; e\; x,\;\; x \in \texttt{enqueue}\;\; e\;\; l \rightarrow \big(x = e \vee x \in l\big).$$

Proof. The proof is by induction on l. □

Next, we use this lemma to establish soundness of `process`.

Lemma 2. `process` is sound with regard to its purpose. That is,

$$\forall\, l\; x,\;\; x \in \texttt{process}\;\; l \rightarrow \big(x \in \texttt{tail}\;\; l \vee \exists\, m \in \texttt{multipliers}, x = (\texttt{head}\;\; l) * m\big).$$

Proof. Since `process` is just repeated application of `enqueue`, the lemma follows from repeated application of Lemma 1. □

With this lemma, we prove soundness of `hamming`.

Theorem 1. `hamming` is sound. That is,

$$\texttt{everywhere smooth hamming}.$$

Proof. Unfolding `hamming` and generalizing (`one` 1) to any (non-empty) list whose elements are smooth, we get

$$\forall\, l,\; (\forall\, n, n \in l \rightarrow \texttt{smooth}\;\; n) \rightarrow \texttt{everywhere smooth}\;\; (\texttt{ham_from}\;\; l).$$

We proceed by coinduction. Unfolding `ham_from` once, the goal becomes

$$\texttt{everywhere smooth}\;\; (\texttt{Cons}\;\; (\texttt{head}\;\; l)\;\; (\texttt{ham_from}\;\; (\texttt{process}\;\; l))).$$

The head is smooth because it is in l, of which all elements are known to be smooth. The tail is smooth by coinduction hypothesis, provided that all elements in `process` l are smooth. By Lemma 2 we know that such elements either occur in the tail of l, or are the result of multiplying the head of l (which is a smooth number) by a number from the list of multipliers. In both cases their smoothness follows immediately. □

4.6.2 Increase

We start with an increase lemma about `enqueue`.

Lemma 3. `enqueue` preserves increase. That is,

$$\forall\, l,\; \texttt{increases}\;\; l \rightarrow \forall\, e,\; \texttt{increases}\;\; (\texttt{enqueue}\;\; e\;\; l).$$

Proof. The proof is by induction on l. □

Next, we need two lemmas about `process`' increase.

Lemma 4. `process` preserves increase. That is,

$$\forall\, l,\ \texttt{increases}\ l \rightarrow \texttt{increases}\ (\texttt{process}\ l).$$

Proof. Again, since `process` is just repeated application of `enqueue`, the lemma follows from repeated application of Lemma 3. □

Lemma 5.

$$\forall\, l,\ \texttt{increases}\ l \rightarrow 0 < \texttt{head}\ l \rightarrow \texttt{head}\ l < \texttt{head}\ (\texttt{process}\ l).$$

Proof. By Lemma 2, `head` (`process` l) is known to be either in the tail of l, or to be the result of multiplying the head of l by a number from the list of multipliers. In the first case, it is bigger than `head` l because l increases. In the second case, it is bigger than `head` l because multiplying any number with another number greater than 1 yields a bigger number, and by `multipliers_nontrivial` we know the multipliers to be greater than 1. □

With these, we prove increase of `hamming`.

Theorem 2. The elements in `hamming` occur in increasing order. That is,

$$\texttt{increases hamming}.$$

Proof. Unfolding `hamming` and generalizing (`one` 1) to any (non-empty) increasing list with a positive head, we get

$$\forall\, l,\ \texttt{increases}\ l \rightarrow 0 < \texttt{head}\ l \rightarrow \texttt{increases}\ (\texttt{ham_from}\ l).$$

Unfolding `ham_from` once, the goal becomes

$$\texttt{increases}\ (\texttt{Cons}\ (\texttt{head}\, l)\ (\texttt{ham_from}\, (\texttt{process}\ l))).$$

We proceed by coinduction, yielding the coinduction hypothesis

$$\forall\, l,\ \texttt{increases}\ l \rightarrow 0 < \texttt{head}\ l \rightarrow \texttt{increases}\ (\texttt{Cons}\ (\texttt{head}\ l)\ (\texttt{ham_from}\ (\texttt{process}\ l))).$$

Unfolding `ham_from` in our goal a second time, it now looks like:

$$\begin{array}{c}\texttt{increases}\ (\texttt{Cons}\ (\texttt{head}\ l)\ (\texttt{Cons}\ (\texttt{head}\ (\texttt{process}\, l))\\ (\texttt{ham_from}\ (\texttt{process}\ (\texttt{process}\ l))))).\end{array}$$

This stream does indeed increase, because it increases from its first to its second element per Lemma 5, and increases beyond the first element per coinduction hypothesis, provided that `process` l is increasing and has a positive head, which by Lemmas 4 and 5 is the case. □

4.6.3 Completeness

Completeness is the most involved of the three properties. We start with a completeness lemma about `enqueue`.

Lemma 6. `enqueue` is complete with regard to its purpose. That is,

i. $\forall\, x\, l,\ x \in l \rightarrow \forall y, x \in \texttt{enqueue}\ \ y\ \ l$, and

ii. $\forall\, e\, l,\ e \in \texttt{enqueue}\ \ e\ \ l.$

Proof. Both proofs are by induction on l. □

Next, we use this lemma to establish completeness of `process`.

Lemma 7. `process` is itself complete with regard to its purpose. That is,

i. $\forall\, l\ p,\ p \in \texttt{multipliers} \rightarrow (\texttt{head}\ \ l * p) \in \texttt{process}\ l$, and

ii. $\forall\, t\ l,\ t \in \texttt{tail}\ \ l \rightarrow t \in \texttt{process}\ \ l.$

Proof. Since `process` is just repeated application of `enqueue`, the two statements follow from repeated application of Lemma 6. □

We need two more lemmas before we can tackle completeness of `hamming`.

Lemma 8. Any tail of `ham_from` l with l an increasing list with positive head is itself of the form `ham_from` l' with l' an increasing list with positive head. That is,

$$\forall\, (n : \texttt{ne_list nat}),\ \texttt{increases}\ n \land 0 < \texttt{head}\ \ n \rightarrow$$
$$\texttt{ForAll}\ (\texttt{fun}\ \ s \Rightarrow \exists\, l,\ s = \texttt{ham_from}\ \ l \land \texttt{increases}\ \ l \land 0 < \texttt{head}\ \ l)\ \ (\texttt{ham_from}\ \ n).$$

Proof. This follows immediately from the definition of `ham_from` combined with Lemmas 4 and 5. □

Lemma 9. `ham_from`'s list parameter behaves like a "queue" in that all elements in it eventually show up in the generated stream. That is,

$$\forall\, l,\ \texttt{increases}\ \ l \rightarrow 0 < \texttt{head}\ \ l \rightarrow \forall\, x \in l,\ \texttt{in_stream}\ \ (\texttt{eq}\ \ x)\ \ (\texttt{ham_from}\ \ l).$$

Proof. The proof is by induction on the difference between x and the head of l. If the difference is zero, then x is the head of l and shows up as the first element in `ham_from` l. If the difference is nonzero, then x occurs in the tail of l. We unfold `ham_from` l once, showing

$$\texttt{Cons}\ \ (\texttt{head}\ \ l)\ \ (\texttt{ham_from}\ \ (\texttt{process}\ \ l)).$$

We now invoke the induction hypothesis to prove that x occurs in the tail. We are allowed to do this because x occurs in `process` l by Lemma 7ii, and the difference between x and the head has decreased by Lemma 5. □

Theorem 3. `hamming` is complete. That is,

$$\forall\, n, \texttt{smooth}\ \ n \rightarrow \texttt{in_stream}\ \ (\texttt{eq}\ \ n)\ \ \texttt{hamming}.$$

Proof. The proof is by induction on n being smooth (recall that `smooth` is an inductively defined predicate). In the base case where n is smooth because it is 1, the goal holds because 1 occurs as the head of `hamming`. In the recursive case where n is smooth because it is of the form $x * y$ with x a smooth number and y in the list of multipliers, the induction hypothesis states that x occurs in `hamming`. From this it now has to be proved that $x * y$ occurs in it, too. Unfolding `hamming` and generalizing (`one` 1) to any increasing list with positive head, we get

$$\forall\, l,\ 0 < \texttt{head}\ \ l \rightarrow \texttt{increases}\ \ l \rightarrow$$
$$\texttt{in_stream}\ \ x\ \ (\texttt{ham_from}\ \ l) \rightarrow \texttt{in_stream}\ \ (x\ *\ y)\ \ (\texttt{ham_from}\ \ l).$$

If x occurs in `ham_from` l, then it must be so that $x = (\texttt{head}\ \ s)$ where $s = (\texttt{tail}^m\ \ (\texttt{ham_from}\ \ l))$ for some m. We prove that $x * y$ occurs in s. From Lemma 8 we know that s must be of the form `ham_from` l' with l' an increasing list with positive head. Unfolding s once, it becomes

$$\texttt{Cons}\ \ (\texttt{head}\ \ l')\ \ (\texttt{ham_from}\ \ (\texttt{process}\ \ l')).$$

Next, by Lemma 7i we know that $x * y \in \texttt{process}\ \ l'$, which by Lemma 9 means that it occurs in the stream. □

With soundness, completeness, and increase proven, `hamming` has been proven correct.

Chapter 5

Evolving Initial Segments: Merge Redux

5.1 Toward a New Merge-based Implementation

As we concluded earlier, our problems trying to use the merge-based implementation were caused by it being inherently less obviously productive. A logical reaction is to try and force its productive behavior out into the open by explicitly examining the recursion step by step rather than all at once. To make this easier, it helps to recognize that the merge-based stream definition is incrementally productive.

Definition 4. A recursive stream definition is *incrementally productive* if the computation of the element value at a position i only depends on the element values at positions below i.

Non-incrementally productive recursive stream definitions exist, but are uncommon. The following is an example of one such stream definition in Haskell.

```
s  =  (s !! 1) : 1 : s
```

The “!!” is Haskell’s `nth`-like operator. The computation of the first element of s depends on the value of the second element, but s is nevertheless productive. This can be easily seen when we first expand the first occurance of s, and then evaluate the outer indexing operation:

```
s  =  (s !! 1) : 1 : s
   =  ((s !! 1) : 1 : s) !! 1 : 1 : s
   =  1 : 1 : s
```

For incrementally productive stream definitions, the step-by-step recursion approach mentioned above can be realized quite straightforwardly by viewing the body of the definition as a unary function taking a list of the first n elements and producing a list of the first m elements, where m is always $\geq n$ and at least occasionally $> n$.

Writing this down for the merge-based Hamming stream definition, this looks like:

```
Definition ham_body (l: list nat): list nat :=
  cons 1 (merge (map (mult 2) l)
   (merge (map (mult 3) l) (map (mult 5) l))).
```

The `cons` used here is the ordinary `cons` for finite lists. Also, in this definition we need a `merge` that operates on finite lists. Writing one is not hard, but we do not show ours here because the definition is

rather cluttered. The clutter is caused by the fact that straightforward recursion on one of its two arguments alone cannot work, since neither steadily decreases at each level in the recursion (this can be seen from the corecursive `merge` definition in Section 4.4). Instead, the recursion is bounded by the sum of the lengths of the two list parameters, which requires a bit of rather verbose Coq juggling.

This `ham_body` is a huge step in the direction of a Coq merge-based Hamming stream implementation, because it manages to capture the essence of the definition while at the same time completely avoiding issues of recursion or stream representation. If we manage to come up with some mechanism that takes a function of this form and turns it into a stream, we are set.

Before we get ahead of ourselves, though, let us take a look at what repeated application of `ham_body` to the empty list actually produces. We can automate the repeated application using `repeat_apply`.

```
Fixpoint repeat_apply T (f: T -> T) (x: T) (n: nat) {struct n}: T :=
  match n with
  | 0 => f x
  | S n' => f (repeat_apply f x n')
  end.
```

The lists thus obtained are as follows.

n	`repeat_apply ham_body nil` n
0	1
1	1, 2, 3, 5
2	1, 2, 3, 4, 5, 6, 9, 10, 15, 25
3	1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, 18, 20, 25, 27, 30, 45, 50, 75, 125

While at a first glance we might be tempted to think that these are simply increasingly long initial segments of the Hamming stream, the mismatch at the fourth position between the second and third iterations (among others) reveal that what we have here is somewhat more subtle. To accurately describe the contents of these lists, we introduce the notion of bounded smoothness.

Definition 5. A number is *bounded smooth* with bound b if it can be written as a product of b or less numbers from the list of multipliers:

```
Inductive bounded_smooth: forall (bound: nat), nat -> Prop :=
  | bounded_smooth_1: bounded_smooth 0 1
  | bounded_smooth_S b n: bounded_smooth b n ->
      bounded_smooth (S b) n
  | bounded_smooth_mult b n: bounded_smooth b n ->
      forall m, In m multipliers -> bounded_smooth (S b) (n * m).
```

Lemma 10. Bounded smoothness and unbounded smoothness imply eachother. That is,

i. $\forall\, b\, n$, `bounded_smooth` b $n \rightarrow$ `smooth` n , and

ii. $\forall\, n$, `smooth` $n \rightarrow \exists\, b$, `bounded_smooth` b n.

Proof. In both cases the proof is by induction on the premise. □

Lemma 11. The i'th iteration contains precisely those numbers that are bounded smooth with bound i. That is,

$$\forall\, i\ x,\ \texttt{bounded_smooth}\ i\ x \leftrightarrow x \in \texttt{repeat_apply ham_body nil}\ i.$$

Proof. The proofs in either direction are by induction on i. □

Lemma 12. Every iteration increases. That is,

$$\forall\, i,\ \texttt{increases}\ (\texttt{repeat_apply ham_body nil}\ i).$$

Proof. The proof is by induction on i and relies on lemmas about soundness and increase of `merge`, `map`, along with some other minor lemmas. Discussing it in any detail here would only cause us to lose track of the big picture, however, so we will not do so. □

With this better understanding of the contents of these lists, we can now reason about to what extent initial segments of them correspond to initial segment of equal length of the Hamming stream.

If an initial segment of an iteration differs from a finite initial segment of equal length of the Hamming stream, it must be because a future iteration adds a number that falls inside the range of the segment (as we saw at the mismatch at the fourth position between the second and third iterations). It is not hard to see that the lowest new number introduced in the i'th iteration must be at least 2^i. This means that from the i'th iteration onwards, at least the initial segment containing the numbers below 2^i is stable and corresponds to an initial segment of equal length of the Hamming stream. (Note that this is different from saying that the first 2^i elements are stable from the i'th iteration onwards.)

Definition 6. An initial segment of the list at iteration i is *stable* if it remains the initial segment in all future iterations.

There is a relation between the index of an iteration and the length of its initial stable segment. This relation could be considered `ham_body`'s productive behavior. The exact length of these initial stable segments in a given iteration is hard to accurately describe. Fortunately we may not need complete accuracy; if we can prove that the first i elements of iteration i are stable, then the diagonal obtained by taking every i'th iteration's i'th element would correspond to the Hamming stream.

Of course, for this to make sense, that diagonal needs to exist. There are two ways to deal with this issue: either we can prove that it exists and then take it, or we can choose to fill any holes in it with some dummy value.[1] For sufficiently productive stream definition, such holes will not occur anyway. The two options translate to Coq as follows.

```
Definition take_diagonal_or_dummy (T: Set)
  (g: nat -> list T) (dummy: T): f_stream T :=
    fun n => nth n (g n) dummy.

Definition take_diagonal (T: Set) (g: nat -> list T)
  (p: forall n, n < length (g n)): f_stream T :=
    fun n => snth (g n) n (p n).
```

For now we choose to use the second, but will return to the matter later. To use the second, we first need to prove growth.

[1] It is tempting to think that this use of a dummy value can be eliminated by using something like Coq's `Option` type (similar to Haskell's `Maybe` type). However, with this approach the stream one gets when taking the diagonal would be of type `Stream (Option nat)`, which is not what we want.

Lemma 13. grow: $\forall\, i,\ i <$ length (repeat_apply ham_body nil i).

Proof. The proof is by induction on i. The base case reduces to $0 <$ length (cons 1 nil). The recursive step amounts to showing that the list resulting from applying ham_body to a list l has a length greater than l. This follows naturally from a lemma about merge (that we will not prove here) which states that

$$\forall\, a\, b,\ \texttt{max (length } a\texttt{) (length } b\texttt{)} \leq \texttt{length (merge } a\ b\texttt{)}. \qquad \square$$

With this, we can use take_diagonal:

```
Definition f_hamming: f_stream nat :=
  take_diagonal (repeat_apply ham_body nil) grow.
```

Note that Lemma 13 does not prove productivity, because it doesn't take into consideration that only initial segments actually correspond to the Hamming stream. A property much closer to what we would call productivity is the one mentioned earlier, namely that the first i elements of iteration i are stable.

Lemma 14. Repeated application of ham_body produces a sequence of lists that is stable below its diagonal. That is,

$$\forall\, i,\ \texttt{equal_upto (repeat_apply ham_body nil i) (repeat_apply ham_body nil (S } i\texttt{)) (S } i\texttt{)},$$

where equal_upto is defined as follows.

```
Definition equal_upto T (a a': list T) n: Prop :=
  forall i (ip: i < n) (v: i < length a) (v': i < length a'),
    snth a i v = snth a' i v'.
```

Proof. The proof is by induction on i. It relies on similar stability properties of merge and map, and is hard to summarize in a few key steps. The interested reader can find the proof along with all sublemmas in all their glory in the accompanying Coq formalization. □

This lemma will be important in our correctness proof.

5.2 Correctness

We want to keep using the specification of the Hamming stream given in Section 4.3, so we use the coinductive stream representation:

```
Definition c_hamming: Stream nat := enumerate f_hamming 0.
```

Overall, correctness proofs of this merge-based Hamming stream implementation are quite a bit trickier than those of the queue-based Hamming stream implementation. In terms of Coq code, the combined proofs are about twice as long. The reason for this is simply that the trick of taking the diagonal adds a thick layer of indirection that must be worked through to get to the essence of things. Whereas the proofs for the queue-based implementation dealt directly with the issues at hand, the proofs for the merge-based implementation are dominated by intricate arguments about stability and diagonals. This was particularly so in Lemma 14 which we more or less skipped for this very reason. The rest of the proof outlines will also be much more sketchy than those in the previous chapter.

5.2.1 Soundness

Theorem 4. `c_hamming` is sound. That is,

$$\texttt{everywhere smooth c_hamming}.$$

Proof. Unfolding `hamming` and generalizing 0, we get:

$$\forall\, n,\ \texttt{everywhere smooth (enumerate f_hamming}\ n).$$

We prove this by coinduction. The coinduction hypothesis applies directly to the tail, leaving only the head. The head is just `f_hamming` n, which is bounded smooth (with some bound b) by Lemma 11, and therefore smooth by Lemma 10i. □

5.2.2 Completeness

Theorem 5. `c_hamming` is complete. That is,

$$\forall\, n,\ \texttt{smooth}\ n \to \texttt{in_stream (eq}\ n)\ \texttt{c_hamming}.$$

Proof. From Lemma 10ii we know that n is bounded smooth with some bound b. From Lemma 11 we know that this means it occurs in some iteration. We distinguish between three cases:

- If n occurs *on* the diagonal, then it immediately follows that it occurs in `c_hamming`.
- If n occurs *below* the diagonal, then by Lemma 14 it must have occurred *on* the diagonal in some previous iteration (and therefore occurs in `c_hamming`).
- If n occurs *above* the diagonal, then we can perform induction on the difference between n and the value on the diagonal. If the difference is zero, then it has to be *on* the diagonal, contradicting the premise. If the difference is nonzero, then the difference will decrease in the next iteration where n is still present but the value on the diagonal has increased, allowing us to invoke the induction hypothesis. □

5.2.3 Increase

Lemma 15. If a list increases, elements occurring after other elements are greater than those elements. That is,

$$\forall\, l\ a\ b\ (p : a < \texttt{length}\ l)\ (q : b < \texttt{length}\ l),\ \texttt{increases}\ l \to a < b \to \texttt{snth}\ l\ a\ p < \texttt{snth}\ l\ b\ q.$$

Proof. The proof is by induction on l. □

Theorem 6. `c_hamming` increases. That is,

$$\texttt{increases c_hamming}$$

Proof. Unfolding `c_hamming`, then `f_hamming`, and then generalizing 0, we get

$$\forall\, n,\ \texttt{increases (enumerate (take_diagonal (repeat_apply ham_body nil) grow)}\ n).$$

We prove this by coinduction. Unfolding the `enumerate` application twice, we get

```
increases
  (Cons (take_diagonal (repeat_apply ham_body nil) grow n)
    (Cons (take_diagonal (repeat_apply ham_body nil) grow (S n))
      (enumerate (take_diagonal (repeat_apply ham_body nil) grow) (S (S n)))))
```

We invoke the coinduction hypothesis to prove increase beyond the first element, leaving increase from the first to the second element for us to prove. That is,

take_diagonal (repeat_apply ham_body nil) grow $n <$
take_diagonal (repeat_apply ham_body nil) grow (S n).

Unfolding take_diagonal, we get

snth (repeat_apply ham_body nil n) (grow n) $<$
snth (repeat_apply ham_body nil (S n)) (grow (S n)).

Using Lemma 14 we can rewrite this into

snth (repeat_apply ham_body nil (S n)) $P <$
snth (repeat_apply ham_body nil (S n)) (grow (S n)).

with P a proof of

$n <$ length (repeat_apply ham_body nil (S n))

obtained from Lemma 13. Applying Lemma 15, the goal becomes

increases (repeat_apply ham_body nil (S n)),

which follows from Lemma 12. □

With soundness, completeness, and increase proven, c_hamming has been proven correct.

5.3 Reflection

It is important to note that our gentle redressing of the merge-based implementation did not magically make Coq "see" its productivity the way it was able to see the productivity of the queue-based implementations. Instead, we placed an upper bound of n recursive calls in the computation of the n'th element, and then separately proved that this was sufficient for this merge-based implementation.

As we noted earlier, this last proof was actually optional at the time. In our implementation we could have chosen to ignore the productivity issue altogether and have potential holes in the diagonal filled with dummy values. However, in subsequent correctness proofs the issue would most certainly have come back to haunt us, most likely in a more vicious form. Still, the option is nice to have, as not all Coq stream implementations need necessarily be certified.

This then brings up the question of to what extent this approach is generic. As far as this author can see, the approach should work for any stream definition that meets the following requirements.

1. The stream definition must be incrementally productive.
2. Any n'th element must take at most n recursive calls to compute. (Perhaps *linearly productive* would make a good term for this.)
3. Any operations the stream definition would perform on infinite lists must be expressible for finite lists (like map and merge were in our Hamming case).

In retrospect, the first requirement can probably be eliminated by using f_stream's directly, starting with an f_stream that yields dummy values for all positions, and slowly "filling" it by repeatedly applying the body function. However, the presence of dummy values would likely make correctness proofs much harder.

We briefly show how the approach can be directly applied to two other stream definitions that would not be accepted by Coq in direct recursive or corecursive form due to their productivity not being obvious enough. Since at this point we only wish to show definitions and do not want to have to do full correctness proofs, we refrain from proving linear productivity and instead let potential holes be filled with a dummy value.

First up is the Fibonacci stream, which can be written in Haskell as follows. (A simpler definition of this happens to be available and was given in Chapter 3, but that is beside the point.)

```
fib = 0 : 1 : zip_with (+) fib (tail fib)
```

In Coq, using our approach this translates to:

```
Definition fib_body l := 0 :: 1 :: zip_with plus l (tail l).

Definition fib: Stream nat := take_diagonal_or_dummy (repeat_apply fib_body nil) 0.
```

(The definitions of Haskell's and Coq's `zip_with` are not very interesting.)

The second example is the Thue-Morse sequence, which can be written in Haskell as follows.

```
morse = False : True : zip (tail morse) (map not (tail morse))
```

In Coq, using our approach this translates to:

```
Definition morse_body l := false :: true :: zip (tail l) (map negb (tail l)).

Definition morse: Stream nat :=
  take_diagonal_or_dummy (repeat_apply morse_body nil) false.
```

(The definitions of Haskell's and Coq's `zip` are not very interesting.)

Looking at the definition of the `ham_body` function, and at the stability requirement needed to prove that the result is indeed the Hamming stream, there appear to be a strong parallels to how the Knaster-Tarski Fixpoint Theorem yields fixpoints of monotonic functions over complete lattices. It would be interesting to put the finite-initial-segments trick discussed in this chapter into a more theoretical context and explore this relation, similar to how Lawrence Paulson applies the Knaster-Tarski Fixpoint Theorem to justify coinductive definitions in [9]. However, doing so is beyond the scope of this article.

Finally, Dimitri Hendriks used a similar trick based on taking the diagonal of a stream of streams in his formalization [8] of the Thue-Morse sequence.

Chapter 6

Program Extraction

Efficiency and suitability for program extraction were not part of our initial stated goals, nor was an analysis of space/time complexity of the different implementations. That said, once we have a working Hamming stream definition in Coq, its fully automated program extraction makes it very hard to resist taking what it produces for a quick spin.

Both queue-based and merge-based implementations were extracted to Haskell and OCaml. The extracted OCaml version of the merge-based implementation was useless, as it crashed from a stack overflow after computing the first 8 stream elements. The extracted OCaml version of the queue-based implementation, as well as the extracted Haskell versions of both the merge-based and queue-based implementations, all worked properly.

Figure 6.1 shows the time efficiency of the three working extracted programs with the list of multipliers set to produce the ordinary Hamming stream. The merge-based implementation is about as slow in Haskell as it is in OCaml. The Haskell version of the queue-based implementation progresses a little further before exploding.

All three versions are extremely inefficient compared to the direct, non-extracted merge-based Haskell implementation from Chapter 2, which can compute the millionth element in less than seven seconds on the same machine. While it would be interesting to analyze the reason our extracted implementations are so inefficient, and to see if mild tweaking can make a difference, this falls outside the scope of this article.

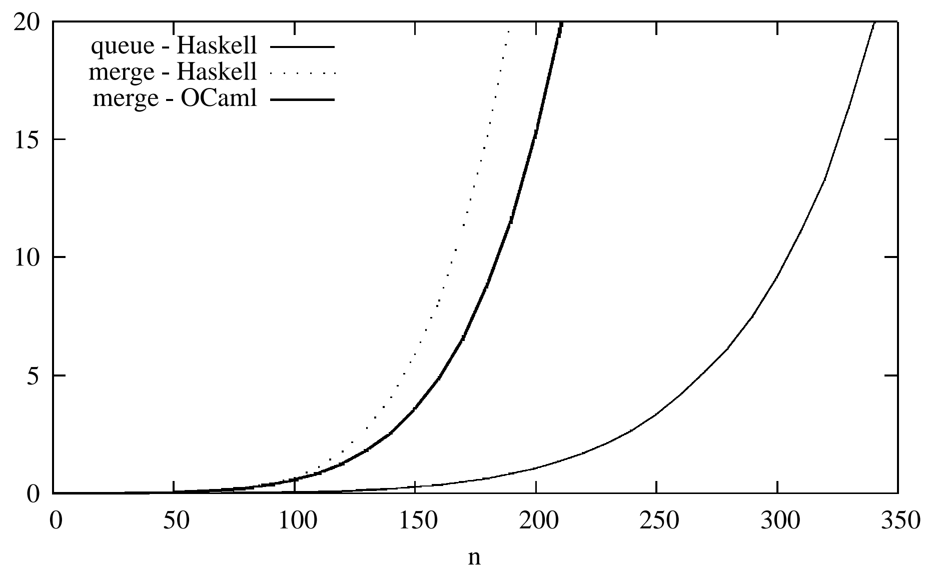

Figure 6.1: Time required to compute the value of the n-th element, in seconds. (Hardware: AMD Athlon 64 3200+ CPU, 1 GiB RAM.)

Chapter 7

Conclusion

More than anything else, our efforts to formalize the Hamming stream provide a demonstration that the extent to which a recursive stream definition's productivity is obvious can make or break its suitability for formalization in Coq. Stream definitions that are obviously productive, such as our queue-based implementation, can be used and certified in Coq straightforwardly using either the functional or the coinductive stream representation.

The approach using evolving initial segments can in some cases be used for stream definitions that are not obviously productive, but the resulting implementations are relatively hard to certify. It really is a hack, and a highly intrusive one at that. A more proper solution would address the problem at its root, in the mechanism used by the system to ensure productivity of corecursive definitions. Fortunately, a lot of research has been and is being done to find more sophisticated mechanisms (see for example [6]). Hopefully the fruits of this research can make tricks like this one obsolete in some future version of Coq.

Bibliography

[1] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer-Verlag, 2004.

[2] Thierry Coquand. Infinite objects in type theory. In H. Barendregt and T. Nipkow, editors, *Selected Papers 1st Int. Workshop on Types for Proofs and Programs, TYPES'93, Nijmegen, The Netherlands, 24–28 May 1993*, volume 806, pages 62–78. Springer-Verlag, Berlin, 1994.

[3] Edsger W. Dijkstra. An exercise attributed to R.W.Hamming. In *A Discipline of Programming*, chapter 17. Prentice-Hall, 1976.

[4] Edsger W. Dijkstra. On the productivity of recursive definitions. Personal note EWD 749, September 1980.

[5] Edsger W. Dijkstra. Hamming's exercise in SASL. Personal note EWD 792, June 1981.

[6] P. Di Gianantonio and M. Miculan. A unifying approach to recursive and corecursive definitions. In H. Geuvers and F. Wiedijk, editors, *Proceedings of TYPES'02, LNCS 2646*. Springer-Verlag, 2003.

[7] Eduardo Giménez. Codifying guarded definitions with recursive schemes. In *TYPES '94: Selected papers from the International Workshop on Types for Proofs and Programs*, pages 39–59, London, UK, 1995. Springer-Verlag.

[8] Dimitri Hendriks. The Thue-Morse sequence in Coq (or: how to bypass guardedness). September 2005.

[9] Lawrence C. Paulson. A fixedpoint approach to implementing (co)inductive definitions. In *CADE-12: Proceedings of the 12th International Conference on Automated Deduction*, pages 148–161, London, UK, 1994. Springer-Verlag.

[10] Ben A. Sijtsma. On the productivity of recursive list definitions. *ACM Trans. Program. Lang. Syst. (TOPLAS)*, 11(4):633–649, October 1989.

[11] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.1 gamma*, November 2006. http://coq.inria.fr.