Streams from a rewriting perspective

Hans Zantema

Technische Universiteit Eindhoven and Radboud Universiteit Nijmegen
P.O. Box 513, 5600 MB, Eindhoven, The Netherlands
email: h.zantema@tue.nl

Streams Seminar, February 22, 2010
This presentation

Streams from a rewriting perspective

Introduction
Techniques
Examples, conclusions

Visualization of streams
Term rewriting
Infinite terms

How to define and compute streams?
How to prove that stream specifications are well-defined or productive?

Term rewriting is our central mechanism.
Focus on uniquely defined streams by requiring orthogonality.
Focus on computer supported techniques: enter stream specification in appropriate tool, after pressing a button a proof of the desired property is generated fully automatically by the tool.

We omit several technical details.
This presentation

- How to define and compute streams?

Hans Zantema Streams from a rewriting perspective
This presentation

- How to define and compute streams?
- How to prove that stream specifications are *well-defined* or *productive*?
This presentation

- How to define and compute streams?
- How to prove that stream specifications are *well-defined or productive*?
- *Term rewriting* is our central mechanism
This presentation

- How to define and compute streams?
- How to prove that stream specifications are \textit{well-defined} or \textit{productive}?
- \textit{Term rewriting} is our central mechanism
- Focus on uniquely defined streams by requiring \textit{orthogonality}
This presentation

- How to define and compute streams?
- How to prove that stream specifications are well-defined or productive?
- Term rewriting is our central mechanism
- Focus on uniquely defined streams by requiring orthogonality
- Focus on computer supported techniques: enter stream specification in appropriate tool, after pressing a button a proof of the desired property is generated fully automatically by the tool
This presentation

- How to define and compute streams?
- How to prove that stream specifications are *well-defined* or *productive*?

*Term rewriting* is our central mechanism

Focus on uniquely defined streams by requiring *orthogonality*

Focus on computer supported techniques: enter stream specification in appropriate tool, after pressing a button a proof of the desired property is generated fully automatically by the tool

- We omit several technical details
Streams as a result of computation

Many computations go on forever, yielding a stream. Examples:

- stream ones defined by ones = 1 : ones
- stream of Hamming numbers
- stream of all prime numbers defined by sieve of Eratosthenes

If the produced elements are in a set $D$, then the result is a stream over $D$, that is, a function from $\mathbb{N}$ to $D$.

Streams are often defined by a set of equations, that are used to compute the elements of the stream. Example:

One equation ones = 1 : ones

ones → 1 : ones → 1 : 1 : ones → 1 : 1 : 1 : ones → \cdots
Streams as a result of computation

- Many computations go on forever, yielding a stream
Streams as a result of computation

- Many computations go on forever, yielding a *stream*

- Examples:
Streams as a result of computation

- Many computations go on forever, yielding a *stream*

- Examples:
  - stream ones defined by ones = 1 : ones
Streams as a result of computation

- Many computations go on forever, yielding a *stream*

- Examples:
  - stream ones defined by ones = 1 : ones
  - stream of Hamming numbers
Streams as a result of computation

- Many computations go on forever, yielding a *stream*

- Examples:
  - stream ones defined by $\text{ones} = 1 : \text{ones}$
  - stream of Hamming numbers
  - stream of all prime numbers defined by sieve of Erathosthenes
Streams as a result of computation

- Many computations go on forever, yielding a *stream*

- Examples:
  - stream ones defined by ones = 1 : ones
  - stream of Hamming numbers
  - stream of all prime numbers defined by sieve of Eratosthenes

- If the produced elements are in a set $D$, then the result is a *stream* over $D$, that is, a function from $\mathbb{N}$ to $D$
Many computations go on forever, yielding a *stream*.

Examples:
- stream ones defined by $\text{ones} = 1 : \text{ones}$
- stream of Hamming numbers
- stream of all prime numbers defined by sieve of Erathosthenes

If the produced elements are in a set $D$, then the result is a *stream* over $D$, that is, a function from $\mathbb{N}$ to $D$.

Streams are often defined by a set of equations, that are used to compute the elements of the stream.
Many computations go on forever, yielding a *stream*

Examples:
- stream ones defined by $\text{ones} = 1 : \text{ones}$
- stream of Hamming numbers
- stream of all prime numbers defined by sieve of Erathosthenes

If the produced elements are in a set $D$, then the result is a *stream* over $D$, that is, a function from $\mathbb{N}$ to $D$

Streams are often defined by a set of equations, that are used to compute the elements of the stream

*Example:*
One equation $\text{ones} = 1 : \text{ones}$

$\text{ones} \rightarrow 1 : \text{ones} \rightarrow 1 : 1 : \text{ones} \rightarrow 1 : 1 : 1 : \text{ones} \rightarrow \cdots$
Extremely simple sets of equations often give rise to amazing streams
Extremely simple sets of equations often give rise to amazing streams

The Fibonacci stream

\[ \text{Fib} = 0 : 1 : 0 : 0 : 1 : 0 : 0 : 1 : 0 : 0 : 1 : 0 : 1 : 0 : 0 : 1 : \cdots \]

is uniquely defined by

\[
\begin{align*}
\text{Fib} & = f(\text{Fib}) \\
f(0 : \sigma) & = 0 : 1 : f(\sigma) \\
f(1 : \sigma) & = 0 : f(\sigma)
\end{align*}
\]
Extremely simple sets of equations often give rise to amazing streams

The Fibonacci stream

\[ \text{Fib} = 0 : 1 : 0 : 0 : 1 : 0 : 1 : 0 : 0 : 1 : 0 : 0 : 1 : 0 : 1 : 0 : 0 : 1 : \ldots \]

is uniquely defined by

\[
\begin{align*}
\text{Fib} & = f(\text{Fib}) \\
f(0 : \sigma) & = 0 : 1 : f(\sigma) \\
f(1 : \sigma) & = 0 : f(\sigma)
\end{align*}
\]

or equivalently by

\[
\begin{align*}
\text{Fib} & = 0 : c \\
c & = 1 : f(c) \\
f(0 : \sigma) & = 0 : 1 : f(\sigma) \\
f(1 : \sigma) & = 0 : f(\sigma)
\end{align*}
\]
Visualization of streams

Boolean streams can be visualized by a turtle figure. Goes back to 70s or 80s, LOGO programming. Drawing algorithm: fix angles $\alpha, \beta$, fix initial direction. Traverse first $N$ elements of the stream. For every 0, the direction goes $\alpha$ to the right. For every 1, the direction goes $\beta$ to the left. After every symbol read a unit step is done.
Boolean streams can be visualized by a *turtle* figure.
Goes back to 70s or 80s, LOGO programming.
Boolean streams can be visualized by a *turtle* figure.

Goes back to 70s or 80s, LOGO programming.

**Drawing algorithm:**

- fix angles $\alpha, \beta$, fix initial direction
- traverse first $N$ elements of the stream
- for every 0 the direction goes $\alpha$ to the right
- for every 1 the direction goes $\beta$ to the left
- after every symbol read a unit step is done
Example turtle figure

Taking the first 11 elements 10011010001 of a stream, starting horizontally and choose $\alpha = \beta = 80^\circ$ yields
Fibonacci stream

\[ N = 18.000 \]

\[ \alpha = 20^\circ \]

\[ \beta = 160^\circ \]
Fibonacci stream

$\alpha = 30^\circ$

$\beta = 150^\circ$

200,000 steps
In computing (an initial part of) a stream the equations are applied from left to right
Streams by rewriting

In computing (an initial part of) a stream the equations are applied from left to right.

Computation by equations that are only applied from left to right is usually called term rewriting.
Streams by rewriting

In computing (an initial part of) a stream the equations are applied from left to right.

Computation by equations that are only applied from left to right is usually called \textit{term rewriting}.

First we sketch the basic concept of term rewriting, and then we apply it to computing streams, and to prove the following two basic notions of stream specifications:
Streams by rewriting

In computing (an initial part of) a stream the equations are applied from left to right.

Computation by equations that are only applied from left to right is usually called term rewriting.

First we sketch the basic concept of term rewriting, and then we apply it to computing streams, and to prove the following two basic notions of stream specifications:

- **well-definedness**: there exists exactly one stream satisfying the given equations.
Streams by rewriting

In computing (an initial part of) a stream the equations are applied from left to right.

Computation by equations that are only applied from left to right is usually called *term rewriting*.

First we sketch the basic concept of term rewriting, and then we apply it to computing streams, and to prove the following two basic notions of stream specifications:

- **well-definedness**: there exists exactly one stream satisfying the given equations
- **productivity**: when using the equations as rewrite rules, all elements of the stream can be computed
A term rewriting system (TRS) is a set of rules of the shape $\ell \rightarrow r$, where $\ell, r$ are terms over a given signature $\Sigma$ and a set $V$ of variables.
A term rewriting system (TRS) is a set of rules of the shape \( \ell \rightarrow r \), where \( \ell, r \) are terms over a given signature \( \Sigma \) and a set \( V \) of variables.

**Example**

\[
\begin{align*}
0 + x & \rightarrow x \\
\text{s}(x) + y & \rightarrow \text{s}(x + y) \\
0 \ast x & \rightarrow 0 \\
\text{s}(x) \ast y & \rightarrow y + (x \ast y)
\end{align*}
\]
Term rewriting

A term rewriting system (TRS) is a set of rules of the shape $\ell \rightarrow r$, where $\ell, r$ are terms over a given signature $\Sigma$ and a set $V$ of variables.

**Example**

\[
\begin{align*}
0 + x & \rightarrow x \\
s(x) + y & \rightarrow s(x + y) \\
0 * x & \rightarrow 0 \\
s(x) * y & \rightarrow y + (x * y)
\end{align*}
\]

Here $x, y \in V$ and $\Sigma = \{0, s, +, *\}$.
A term rewriting system (TRS) is a set of rules of the shape \( \ell \rightarrow r \), where \( \ell, r \) are terms over a given signature \( \Sigma \) and a set \( V \) of variables.

**Example**

\[
0 + x \rightarrow x \\
s(x) + y \rightarrow s(x + y) \\
0 \ast x \rightarrow 0 \\
s(x) \ast y \rightarrow y + (x \ast y)
\]

Here \( x, y \in V \) and \( \Sigma = \{0, s, +, \ast\} \).

Every symbol in \( \Sigma \) has a fixed \textit{arity}: the number of arguments it requires.
Term rewriting

A *substitution* is a map from variables to terms.
A substitution is a map from variables to terms.

If $t$ is a term and $\sigma$ is a substitution, then $t\sigma$ is the term obtained from $t$ by replacing every variable $x$ by $\sigma(x)$.
Term rewriting

A substitution is a map from variables to terms

If \( t \) is a term and \( \sigma \) is a substitution, then \( t\sigma \) is the term obtained from \( t \) by replacing every variable \( x \) by \( \sigma(x) \)

If \( R \) is a TRS, then a rewrite step \( t \rightarrow_R t' \) is the replacement of \( t \) by \( t' \), where \( t' \) is obtained from \( t \) by replacing a subterm of \( t \) of the shape \( \ell\sigma \) by \( r\sigma \), for some rule \( \ell \rightarrow r \) of \( R \)
A substitution is a map from variables to terms.

If $t$ is a term and $\sigma$ is a substitution, then $t\sigma$ is the term obtained from $t$ by replacing every variable $x$ by $\sigma(x)$.

If $R$ is a TRS, then a rewrite step $t \rightarrow_R t'$ is the replacement of $t$ by $t'$, where $t'$ is obtained from $t$ by replacing a subterm of $t$ of the shape $\ell\sigma$ by $r\sigma$, for some rule $\ell \rightarrow r$ of $R$.

In this way we can compute $2 + 2 = 4$:

$$s(s(0)) + s(s(0)) \rightarrow_R s(s(0) + s(s(0)))$$

$$\rightarrow_R s(s(0 + s(s(0))))$$

$$\rightarrow_R s(s(s(s(0))))$$
Why does this work well?

- Ground term: term without variables
- Two types of symbols: constructors (0, s) and defined symbols (+, ∗)
- The intended result is a constructor ground term on every ground term containing a defined symbol.
- A rule is applicable: the TRS is exhaustive.
- The TRS is terminating, so starting in a ground term, rewriting will always end in a constructor ground term.
- The TRS is confluent, so this resulting constructor ground term is independent of possible choice in the computation.

Hans Zantema

Streams from a rewriting perspective
Why does this work well?

- *ground term* = term without variables
Why does this work well?

- *ground term* = term without variables

- two types of symbols: *constructors* 0, s and *defined symbols* +, *

Hans Zantema

Streams from a rewriting perspective
Why does this work well?

- *ground term* = term without variables

- two types of symbols: *constructors* $0, s$ and *defined symbols* $+, \ast$

- the intended result is a constructor ground term
Why does this work well?

- *ground term* = term without variables

- two types of symbols: *constructors* 0, s and *defined symbols* +, *

- the intended result is a constructor ground term

- on every ground term containing a defined symbol a rule is applicable: the TRS is *exhaustive*
Why does this work well?

- *ground term* = term without variables
- two types of symbols: *constructors* 0, s and *defined symbols* +, *
- the intended result is a constructor ground term
- on every ground term containing a defined symbol a rule is applicable: the TRS is *exhaustive*
- the TRS is *terminating*, so starting in a ground term, rewriting will always end in a constructor ground term
Why does this work well?

- **ground term** = term without variables

- two types of symbols: **constructors** 0, s and **defined symbols** +, *

- the intended result is a constructor ground term

- on every ground term containing a defined symbol a rule is applicable: the TRS is **exhaustive**

- the TRS is **terminating**, so starting in a ground term, rewriting will always end in a constructor ground term

- the TRS is **confluent**, so this resulting constructor ground term is independent of possible choice in the computation
How about streams?

The setting is two-sorted: there is a sort $d$ for data elements and a sort $s$ for streams. There is only one constructor of sort $s$: the symbol ':', having type $d \times s \rightarrow s$. Data elements are constructor normal forms of sort $d$, for instance $N$ composed from $s$ and 0. There are no finite constructor ground terms of sort $s$. The intended stream is an infinite constructor ground term of sort $s$, and can be seen as the limit of an infinite reduction.
The setting is *two-sorted*: there is a sort $d$ for data elements and a sort $s$ for streams.
How about streams?

- The setting is *two-sorted*: there is a sort \( d \) for data elements and a sort \( s \) for streams.

- There is only one constructor of sort \( s \): the symbol \( ':' \), having type \( d \times s \rightarrow s \).
How about streams?

- The setting is *two-sorted*: there is a sort \( d \) for data elements and a sort \( s \) for streams.

- There is only one constructor of sort \( s \): the symbol ‘:’, having type \( d \times s \to s \).

- Data elements are constructor normal forms of sort \( d \), for instance \( N \) composed from \( s \) and 0.
How about streams?

- The setting is *two-sorted*: there is a sort $d$ for data elements and a sort $s$ for streams.

- There is only one constructor of sort $s$: the symbol ':', having type $d \times s \rightarrow s$.

- Data elements are constructor normal forms of sort $d$, for instance $\mathbf{N}$ composed from $s$ and 0.

- There are no finite constructor ground terms of sort $s$. 
The setting is **two-sorted**: there is a sort \( d \) for data elements and a sort \( s \) for streams.

There is only one constructor of sort \( s \): the symbol \('\)\,'\), having type \( d \times s \rightarrow s \).

Data elements are constructor normal forms of sort \( d \), for instance \( \mathbf{N} \) composed from \( s \) and 0.

There are no finite constructor ground terms of sort \( s \).

The intended stream is an infinite constructor ground term of sort \( s \), and can be seen as the limit of an infinite reduction.
For example, the specification \( \text{ones} = 1 : \text{ones} \) coincides with the TRS \( \text{ones} \rightarrow 1 : \text{ones} \), yielding an infinite reduction

\[
\text{ones} \rightarrow 1 : \text{ones} \rightarrow 1 : 1 : \text{ones} \rightarrow \cdots
\]

resulting in the infinite constructor ground term

\[
1 : 1 : 1 : 1 : \cdots
\]

in the limit.
For example, the specification ones = 1 : ones coincides with the TRS ones → 1 : ones, yielding an infinite reduction

\[
\text{ones} \rightarrow 1 : \text{ones} \rightarrow 1 : 1 : \text{ones} \rightarrow \cdots
\]

resulting in the infinite constructor ground term

\[
1 : 1 : 1 : 1 : \cdots
\]

in the limit

Here ones is the defined symbol; computation continues as long the term contains a defined symbol
For example, the specification \( \text{ones} = 1 : \text{ones} \) coincides with the TRS \( \text{ones} \rightarrow 1 : \text{ones} \), yielding an infinite reduction

\[
\text{ones} \rightarrow 1 : \text{ones} \rightarrow 1 : 1 : \text{ones} \rightarrow \cdots
\]

resulting in the infinite constructor ground term

\[
1 : 1 : 1 : 1 : \cdots
\]

in the limit

Here \( \text{ones} \) is the defined symbol; computation continues as long the term contains a defined symbol

This computation goes on forever: it is intended to produce an infinite stream
For example, the specification \( \text{ones} = 1 : \text{ones} \) coincides with the TRS \( \text{ones} \rightarrow 1 : \text{ones} \), yielding an infinite reduction

\[
\text{ones} \rightarrow 1 : \text{ones} \rightarrow 1 : 1 : \text{ones} \rightarrow \cdots
\]

resulting in the infinite constructor ground term

\[
1 : 1 : 1 : 1 : \cdots
\]

in the limit

Here \( \text{ones} \) is the defined symbol; computation continues as long the term contains a defined symbol

This computation goes on forever: it is intended to produce an infinite stream

So the TRS is never terminating
Infinite terms?

We focus on streams, that can be seen as *infinite terms* over the constructor ': ' and $D$.
Infinite terms?

We focus on streams, that can be seen as *infinite terms* over the constructor ’:’ and $D$

Various ways to define general infinite terms:
Infinite terms?

We focus on streams, that can be seen as *infinite terms* over the constructor ’:’ and $D$

Various ways to define general infinite terms:
- As a map from positions $\in \mathbb{N}^*$ to symbols
We focus on streams, that can be seen as *infinite terms* over the constructor ‘:’ and \( D \).

Various ways to define general infinite terms:

- As a map from positions \( \in \mathbb{N}^* \) to symbols
- As the limit of finite terms
We focus on streams, that can be seen as *infinite terms* over the constructor ‘:’ and $D$

Various ways to define general infinite terms:

- As a map from positions $\in N^*$ to symbols
- As the limit of finite terms
- As some final coalgebra
Infinite terms?

We focus on streams, that can be seen as infinite terms over the constructor ’:’ and $D$

Various ways to define general infinite terms:

- As a map from positions $\in \mathbb{N}^*$ to symbols
- As the limit of finite terms
- As some final coalgebra

However, we already have a definition of streams: maps from $\mathbb{N}$ to $D$ (closely related to first option) which we prefer to keep
Infinite terms?

We focus on streams, that can be seen as *infinite terms* over the constructor ’:’ and $D$

Various ways to define general infinite terms:

- As a map from positions $\in \mathbb{N}^*$ to symbols
- As the limit of finite terms
- As some final coalgebra

However, we already have a definition of streams: maps from $\mathbb{N}$ to $D$ (closely related to first option) which we prefer to keep

The resulting stream is an infinite term being the limit of an infinite sequence of rewrite steps, each on *finite terms*
We focus on streams, that can be seen as *infinite terms* over the constructor ’:’ and $D$

Various ways to define general infinite terms:

- As a map from positions $\in \mathbb{N}^*$ to symbols
- As the limit of finite terms
- As some final coalgebra

However, we already have a definition of streams: maps from $\mathbb{N}$ to $D$ (closely related to first option) which we prefer to keep.

The resulting stream is an infinite term being the limit of an infinite sequence of rewrite steps, each on *finite terms*.

So the process of computing a stream by rewriting is *not* about infinite terms.
Example

Thue-Morse stream Morse:

\[
\begin{align*}
D &= \{0, 1\} \\
\text{Morse} &\rightarrow 0 : c \\
&\quad \rightarrow 1 : f(c) \\
&\quad f(0 : xs) &\rightarrow 0 : 1 : f(xs) \\
&\quad f(1 : xs) &\rightarrow 1 : 0 : f(xs) \\
Morse &\rightarrow 0 : c &\rightarrow 0 : 1 : f(c) &\rightarrow 0 : 1 : f(f(c)) \\
&\quad \rightarrow 0 : 1 : 0 : f(f(f(c))) &\rightarrow 0 : 1 : 1 : 0 : f(f(f(f(c)))) &\rightarrow \cdots
\end{align*}
\]

Hans Zantema

Streams from a rewriting perspective
Example

Thue-Morse stream Morse:

\[ D = \{0, 1\} \]

\[
\begin{align*}
\text{Morse} & \rightarrow 0 : c \\
\text{c} & \rightarrow 1 : f(c) \\
f(0 : xs) & \rightarrow 0 : 1 : f(xs) \\
f(1 : xs) & \rightarrow 1 : 0 : f(xs)
\end{align*}
\]
Thue-Morse stream Morse: $D = \{0, 1\}$

\[
\begin{align*}
\text{Morse} & \rightarrow 0 : c \\
\text{c} & \rightarrow 1 : f(c) \\
f(0 : xs) & \rightarrow 0 : 1 : f(xs) \\
f(1 : xs) & \rightarrow 1 : 0 : f(xs)
\end{align*}
\]

\[
\begin{align*}
\text{Morse} \rightarrow 0 : c & \rightarrow 0 : 1 : f(c) \rightarrow 0 : 1 : f(1 : f(c)) \rightarrow \\
0 : 1 : 1 : 0 & : f(f(c)) \rightarrow 0 : 1 : 1 : 0 : f(f(1 : f(c))) \rightarrow \\
0 : 1 : 1 & : 0 : f(1 : 0 : f(f(c))) \rightarrow \\
0 : 1 : 1 : 0 & : 1 : 0 : f(0 : f(f(c))) \rightarrow \\
0 : 1 : 1 : 0 : 1 : 0 : 0 : 1 & : f(f(f(c))) \rightarrow \cdots
\end{align*}
\]
Example: Stream Ham of Hamming numbers

\[ D = \mathbb{N} \]
Example: Stream Ham of Hamming numbers

\[ D = \mathbb{N} \]

\[
\begin{align*}
\text{Ham} & \quad \rightarrow \quad s(0) : \text{merge}(m_2(\text{Ham}), m_3(\text{Ham})) \\
m_2(x : xs) & \quad \rightarrow \quad d(x) : m_2(xs) \\
d(0) & \quad \rightarrow \quad 0 \\
d(s(x)) & \quad \rightarrow \quad s(s(d(x))) \\
m_3(x : xs) & \quad \rightarrow \quad t(x) : m_3(xs) \\
t(0) & \quad \rightarrow \quad 0 \\
t(s(x)) & \quad \rightarrow \quad s(s(s(t(x))))
\end{align*}
\]
Example: Stream Ham of Hamming numbers

\[
\begin{align*}
\text{Ham} & \rightarrow s(0) : \text{merge}(m_2(\text{Ham}), m_3(\text{Ham})) \\
m_2(x : xs) & \rightarrow d(x) : m_2(xs) \\
d(0) & \rightarrow 0 \\
d(s(x)) & \rightarrow s(s(d(x))) \\
m_3(x : xs) & \rightarrow t(x) : m_3(xs) \\
t(0) & \rightarrow 0 \\
t(s(x)) & \rightarrow s(s(t(x)))) \\
\text{merge}(x : xs, y : ys) & \rightarrow F(c(x, y), x : xs, y : ys) \\
F(0, x : xs, y : ys) & \rightarrow x : \text{merge}(xs, ys) \\
F(s(0), x : xs, ys) & \rightarrow x : \text{merge}(xs, ys) \\
F(s(s(x)), xs, y : ys) & \rightarrow y : \text{merge}(xs, ys)
\end{align*}
\]
Example: Stream Ham of Hamming numbers

\[
\begin{align*}
\text{Ham} & \rightarrow s(0) : \text{merge}(m_2(\text{Ham}), m_3(\text{Ham})) \\
m_2(x : xs) & \rightarrow d(x) : m_2(xs) \\
d(0) & \rightarrow 0 \\
d(s(x)) & \rightarrow s(s(d(x))) \\
m_3(x : xs) & \rightarrow t(x) : m_3(xs) \\
t(0) & \rightarrow 0 \\
t(s(x)) & \rightarrow s(s(s(t(x)))) \\
\text{merge}(x : xs, y : ys) & \rightarrow F(c(x, y), x : xs, y : ys) \\
F(0, x : xs, y : ys) & \rightarrow x : \text{merge}(xs, ys) \\
F(s(0), x : xs, ys) & \rightarrow x : \text{merge}(xs, ys) \\
F(s(s(x)), xs, y : ys) & \rightarrow y : \text{merge}(xs, ys) \\
c(0, 0) & \rightarrow 0 \\
c(s(x), s(y)) & \rightarrow c(x, y) \\
c(0, s(x)) & \rightarrow s(0) \\
c(s(x), 0) & \rightarrow s(s(0)) \\
\end{align*}
\]

\[c(x, y) \rightarrow^* 0 \text{ if } x = y\] \[c(x, y) \rightarrow^* s(0) \text{ if } x < y\] \[c(x, y) \rightarrow^* s(s(0)) \text{ if } x > y\]
Essentially the same example as in Jan Rutten’s presentation on coalgebra

Differences:
Essentially the same example as in Jan Rutten’s presentation on coalgebra

Differences:

- **Syntax:**
  
  $\text{Ham} \rightarrow s(0) : \text{merge}(m_2(\text{Ham}), m_3(\text{Ham}))$

  $m_2(x : xs) \rightarrow d(x) : m_2(xs)$

  instead of

  $\text{Ham}(0) = 1, \quad \text{Ham}' = \text{merge}(m_2(\text{Ham}), m_3(\text{Ham}))$

  $m_2(\sigma)(0) = d(\sigma(0)), \quad m_2(\sigma)' = m_2(\sigma')$
Essentially the same example as in Jan Rutten’s presentation on coalgebra

Differences:

- **Syntax:**
  
  \[
  \begin{align*}
  \text{Ham} & \rightarrow s(0) : \text{merge}(m_2(\text{Ham}), m_3(\text{Ham})) \\
  m_2(x : xs) & \rightarrow d(x) : m_2(xs)
  \end{align*}
  \]

  instead of

  \[
  \begin{align*}
  \text{Ham}(0) = 1, & \quad \text{Ham'} = \text{merge}(m_2(\text{Ham}), m_3(\text{Ham})) \\
  m_2(\sigma)(0) = d(\sigma(0)), & \quad m_2(\sigma') = m_2(\sigma')
  \end{align*}
  \]

- There **N** was assumed to exist, with +, *, <, >, = with their usual meaning, while we encode this by term rewriting as part of our specification
Essentially the same example as in Jan Rutten’s presentation on coalgebra

Differences:

- **Syntax:**
  
  \[
  \begin{align*}
  \text{Ham} & \rightarrow s(0) : \text{merge}(m_2(\text{Ham}), m_3(\text{Ham})) \\
  m_2(x : xs) & \rightarrow d(x) : m_2(xs)
  \end{align*}
  \]

  instead of

  \[
  \begin{align*}
  \text{Ham}(0) = 1, \quad \text{Ham}' &= \text{merge}(m_2(\text{Ham}), m_3(\text{Ham})) \\
  m_2(\sigma)(0) &= d(\sigma(0)), \quad m_2(\sigma)' = m_2(\sigma')
  \end{align*}
  \]

- There \( \mathbb{N} \) was assumed to exist, with \(+, \ast, <, >, =\) with their usual meaning, while we encode this by term rewriting as part of our specification.

- There non-trivial human effort was required to prove that this specification is well-defined \(\Rightarrow\) has a unique solution.
Differences, continued:

- By term rewriting we have several techniques to prove well-definedness fully automatically:
The only human effort is typing the example in the appropriate tool; the rest is done by a computer.
Differences, continued:

- By term rewriting we have several techniques to prove well-definedness fully automatically:
  The only human effort is typing the example in the appropriate tool; the rest is done by a computer.

- The coalgebra proof can be generalized to a general format, and checking whether a specification meets this format is easily checked by a program too.
Differences, continued:

- By term rewriting we have several techniques to prove well-definedness fully automatically:
  The only human effort is typing the example in the appropriate tool; the rest is done by a computer

- The coalgebra proof can be generalized to a general format, and checking whether a specification meets this format is easily checked by a program too
  However, this general format is much more restricted than the approaches in rewriting
Differences, continued:

- By term rewriting we have several techniques to prove well-definedness fully automatically:
  The only human effort is typing the example in the appropriate tool; the rest is done by a computer.

- The coalgebra proof can be generalized to a general format, and checking whether a specification meets this format is easily checked by a program too.
  However, this general format is much more restricted than the approaches in rewriting.

- Several rewriting techniques prove the stronger notion of **productivity** = the solution is unique and can be computed by rewriting.
Differences, continued:

- By term rewriting we have several techniques to prove well-definedness fully automatically:
  The only human effort is typing the example in the appropriate tool; the rest is done by a computer.

- The coalgebra proof can be generalized to a general format, and checking whether a specification meets this format is easily checked by a program too. However, this general format is much more restricted than the approaches in rewriting.

- Several rewriting techniques prove the stronger notion of *productivity* = the solution is unique and can be computed by rewriting.

Before discussing details of productivity first we present a rewriting technique for proving well-definedness.
The Observational Variant

Instead of producing the stream, we only want to observe it, by introducing head and tail
The Observational Variant

Instead of producing the stream, we only want to observe it, by introducing head and tail

**Idea:**
If we can observe the $n$-th element $\text{head}(\text{tail}^{n-1}(\sigma))$ of a stream $\sigma$ for every $n$, and this result is unique, then the whole stream is well-defined
The Observational Variant

Instead of producing the stream, we only want to observe it, by introducing head and tail

*Idea:*
If we can observe the $n$-th element $\text{head}(\text{tail}^{n-1}(\sigma))$ of a stream $\sigma$ for every $n$, and this result is unique, then the whole stream is well-defined

Polish the TRS $R$ such that this holds, resulting in the observational variant $\text{Obs}(R)$
The Observational Variant

Instead of producing the stream, we only want to observe it, by introducing head and tail

_Idea:_
If we can observe the $n$-th element $\text{head}(\text{tail}^{n-1}(\sigma))$ of a stream $\sigma$ for every $n$, and this result is unique, then the whole stream is well-defined

Polish the TRS $R$ such that this holds, resulting in the observational variant $\text{Obs}(R)$

_Theorem (Z09)_

If $R$ is a stream specification for which $\text{Obs}(R)$ is terminating, then $R$ is well-defined
Example

Ham \rightarrow s(0) : \text{merge}(m_2(\text{Ham}), m_3(\text{Ham}))

m_2(x : xs) \rightarrow d(x) : m_2(xs)
Example

Ham $\rightarrow$ $s(0) : \text{merge}(m_2(\text{Ham}), m_3(\text{Ham}))$

$m_2(x : xs) \rightarrow d(x) : m_2(xs)$

transforms to

head(\text{Ham}) $\rightarrow$ $s(0)$

\text{tail}(\text{Ham}) $\rightarrow$ $\text{merge}(m_2(\text{Ham}), m_3(\text{Ham}))$

head($m_2(xs)$) $\rightarrow$ $d(\text{head}(xs))$

\text{tail}($m_2(xs)$) $\rightarrow$ $m_2(\text{tail}(xs))$
**Example**

\[
\begin{align*}
\text{Ham} & \rightarrow \ s(0) : \text{merge}(m_2(\text{Ham}), m_3(\text{Ham})) \\
\text{m}_2(x : xs) & \rightarrow \ d(x) : m_2(xs)
\end{align*}
\]

transforms to

\[
\begin{align*}
\text{head(Ham)} & \rightarrow \ s(0) \\
\text{tail(Ham)} & \rightarrow \ \text{merge}(m_2(\text{Ham}), m_3(\text{Ham})) \\
\text{head(m}_2(xs)) & \rightarrow \ d(\text{head(xs)}) \\
\text{tail(m}_2(xs)) & \rightarrow \ m_2(\text{tail(xs)})
\end{align*}
\]

Closely related to coalgebra format
Example

\[
\begin{align*}
\text{Ham} & \rightarrow s(0) : \text{merge}(m_2(\text{Ham}), m_3(\text{Ham})) \\
\text{m}_2(x : \text{xs}) & \rightarrow d(x) : \text{m}_2(\text{xs})
\end{align*}
\]

transforms to

\[
\begin{align*}
\text{head}(\text{Ham}) & \rightarrow s(0) \\
\text{tail}(\text{Ham}) & \rightarrow \text{merge}(m_2(\text{Ham}), m_3(\text{Ham})) \\
\text{head}(m_2(\text{xs})) & \rightarrow d(\text{head}(\text{xs})) \\
\text{tail}(m_2(\text{xs})) & \rightarrow m_2(\text{tail}(\text{xs}))
\end{align*}
\]

Closely related to coalgebra format

Standard termination tools like AProVE succeed in proving termination of \text{Obs}(R) fully automatically for the Hamming example.
(VAR x y xs ys)
(RULES
hd(ham) -> s(0)
tl(ham) -> mer(m2(ham),m3(ham))
hd(m2(xs)) -> d(hd(xs))
tl(m2(xs)) -> m2(tl(xs))
d(0) -> 0
d(s(x)) -> s(s(d(x)))
hd(m3(xs)) -> t(hd(xs))
tl(m3(xs)) -> m3(tl(xs))
t(0) -> 0
t(s(x)) -> s(s(t(x)))
hd(mer(xs,ys)) -> hd(F(c(hd(xs),hd(ys)),xs,ys))
tl(mer(xs,ys)) -> tl(F(c(hd(xs),hd(ys)),xs,ys))
hd(F(0,xs,ys)) -> hd(xs)
tl(F(0,xs,ys)) -> mer(tl(xs),tl(ys))
hd(F(s(0),xs,ys)) -> hd(xs)
tl(F(s(0),xs,ys)) -> mer(tl(xs),ys)
hd(F(s(s(x)),xs,ys)) -> hd(ys)
tl(F(s(s(x)),xs,ys)) -> mer(xs,tl(ys))
c(0,0) -> 0
c(s(x),s(y)) -> c(x,y)
c(0,s(x)) -> s(0)
c(s(x),0) -> s(s(0))
)
Productivity

Intuitively:

productivity = the solution is unique and can be computed by rewriting.

We restrict to orthogonal systems (non-overlapping and left-linear), for which unicity of the result is obtained for free as soon as result can be computed.

Definition

A stream specification $R$ is productive for a ground term $t$ if for every $n \in \mathbb{N}$ there exists a reduction $t \rightarrow^* R d_1 : d_2 : \cdots : d_n$.
Intuitively: \textit{productivity} = the solution is unique and can be computed by rewriting.
Productivity

*Intuitively:* 

*productivity* = the solution is unique and can be computed by rewriting

We restrict to *orthogonal* systems (non-overlapping and left-linear), for which unicity of the result is obtained for free as soon as result can be computed
Intuitively: productivity = the solution is unique and can be computed by rewriting

We restrict to orthogonal systems (non-overlapping and left-linear), for which unicity of the result is obtained for free as soon as result can be computed.

**Definition**

A stream specification $R$ is productive for a ground term $t$ if for every $n \in \mathbb{N}$ there exists a reduction

$$t \xrightarrow{\star}_R d_1 : d_2 : \cdots : d_n : t'$$

$n$ elements produced
Note that this definition does not refer to infinitary rewriting.
Productivity

Note that this definition does not refer to infinitary rewriting

*Equivalent:*
A stream specification $R$ is productive for a ground term $t$ if and only if $t$ admits an infinite constructor normal form, that is, an infinite term composed from "::" and data elements
Productivity

Note that this definition does not refer to infinitary rewriting

*Equivalent:*
A stream specification $R$ is productive for a ground term $t$ if and only if $t$ admits an infinite constructor normal form, that is, an infinite term composed from """:" and data elements

Productivity implies well-definedness, but not conversely
Note that this definition does not refer to infinitary rewriting

**Equivalent:**
A stream specification $R$ is productive for a ground term $t$ if and only if $t$ admits an infinite constructor normal form, that is, an infinite term composed from "::" and data elements

Productivity implies well-definedness, but not conversely

**Example:**

$$
\begin{align*}
c & \rightarrow f(c) \\
f(x : xs) & \rightarrow 0 : c
\end{align*}
$$

is well-defined, but not productive
[EGHIK07] and [EGH08] identify decidable criteria

- in [EGHIK07]: weakly guarded
- in [EGH08]: flat / friendly nesting

for which productivity can be decided
[EGHIK07] and [EGH08] identify decidable criteria

- in [EGHIK07]: weakly guarded
- in [EGH08]: flat / friendly nesting

for which productivity can be decided

**Idea:**

for all function symbols investigate how many elements they consume / produce
[EGHIK07] and [EGH08] identify decidable criteria

- in [EGHIK07]: weakly guarded
- in [EGH08]: flat / friendly nesting

for which productivity can be decided

**Idea:**
for all function symbols investigate how many elements they consume / produce

\[ f(x : xs) \rightarrow 0 : 1 : f(xs) \]

produces two elements 0, 1, after consuming one element \( x \)
[EGHIK07] and [EGH08] identify decidable criteria
  - in [EGHIK07]: weakly guarded
  - in [EGH08]: flat / friendly nesting
for which productivity can be decided

**Idea:**
for all function symbols investigate how many elements they consume / produce

\[ f(x : xs) \rightarrow 0 : 1 : f(xs) \]

produces two elements 0, 1, after consuming one element \( x \)

A *pebble flow net* mimicks the production / consumption of elements starting by the given constant
Pebble flow

\[ M \rightarrow 0 : \text{zip}(\text{inv}(\text{even}(M)), \text{tail}(M)) \]
The system is productive for the given constant if this produces elements forever, that is, in this process there are never more elements consumed than have been produced.
Pebble flow

The system is productive for the given constant if this produces elements forever, that is, in this process there are never more elements consumed than have been produced.

This can be checked by a decision procedure, which is in itself a rewriting procedure.
The system is productive for the given constant if this produces elements forever, that is, in this process there are never more elements consumed than have been produced.

This can be checked by a decision procedure, which is in itself a rewriting procedure.

Has been implemented.
Web interface of pebble flow tool

Online Version

Enter your stream specification, and press "GO" to check for productivity.

```
Signature(
    M : stream(bit),
    0,1 : bit,
    tail : stream(x) -> stream(x),
    zip : stream(x) -> stream(x) -> stream(x),
    inv : stream(bit) -> stream(bit)
)

M = 0:zip(inv(M),tail(M))

zip(a:s,t) = a:zip(t,s)
tail(a:s) = s
inv(0:s) = 1:inv(s)
inv(1:s) = 0:inv(s)
```
- Gives proofs fully automatically, for productivity for a single constant.
• Gives proofs fully automatically, for productivity for a single constant

• Successful for wide range of examples, including Hamming numbers, Thue Morse stream, Fibonacci stream, ⋯
- Gives proofs fully automatically, for productivity for a single constant

- Successful for wide range of examples, including Hamming numbers, Thue Morse stream, Fibonacci stream, ···

- Restricts to *data oblivious* productivity $\approx$ independent of data values, so fails to prove productivity of

\[
\begin{align*}
M & \rightarrow f(c) \\
c & \rightarrow 1 : c \\
f(0 : xs) & \rightarrow f(xs) \\
f(1 : xs) & \rightarrow 1 : f(xs)
\end{align*}
\]
Other techniques to prove productivity

Fruitful lemma:

A stream specification $R$ is productive for all (finite) ground terms if and only if every ground term $t$ rewrites to a term of the shape $d$:

$$t'$$

So proving productivity for all ground terms may be easier than for a single constant [ZR09]: A stream specification $R$ is productive for all (finite) ground terms if and only if $R \cup \{x : xs \rightarrow \text{overflow}\}$ is balanced, outermost terminating.
Fruitful lemma:

**Lemma**

A stream specification $R$ is productive for all (finite) ground terms if and only if every ground term $t$ rewrites to a term of the shape $d : t'$.
Other techniques to prove productivity

Fruitful lemma:

**Lemma**

A stream specification $R$ is productive for all (finite) ground terms if and only if every ground term $t$ rewrites to a term of the shape $d : t'$

So proving productivity for all ground terms may be easier than for a single constant
Other techniques to prove productivity

Fruitful lemma:

Lemma

A stream specification $R$ is productive for all (finite) ground terms if and only if every ground term $t$ rewrites to a term of the shape $d : t'$

So proving productivity for all ground terms may be easier than for a single constant

[ZR09]: A stream specification $R$ is productive for all (finite) ground terms if and only if $R \cup \{x : xs \rightarrow \text{overflow}\}$ is balanced outermost terminating
Example

\[
\begin{align*}
M & \rightarrow f(c) \\
c & \rightarrow 1 : c \\
f(0 : xs) & \rightarrow f(xs) \\
f(1 : xs) & \rightarrow 1 : f(xs) \\
x : xs & \rightarrow \text{overflow}
\end{align*}
\]

is outermost terminating, as is proved automatically by several tools.
Example

\[
\begin{align*}
M & \rightarrow f(c) \\
c & \rightarrow 1 : c \\
f(0 : xs) & \rightarrow f(xs) \\
f(1 : xs) & \rightarrow 1 : f(xs) \\
x : xs & \rightarrow \text{overflow}
\end{align*}
\]

is outermost terminating, as is proved automatically by several tools.

So this approach proves productivity for this example, where data oblivious productivity does not hold.
Example

\[
\begin{align*}
M & \rightarrow f(c) \\
c & \rightarrow 1 : c \\
f(0 : xs) & \rightarrow f(xs) \\
f(1 : xs) & \rightarrow 1 : f(xs) \\
x : xs & \rightarrow \text{overflow}
\end{align*}
\]

is outermost terminating, as is proved automatically by several tools.

So this approach proves productivity for this example, where data oblivious productivity does not hold.

Unfortunately, for many productive examples plain outermost termination does not hold, while there are no techniques for proving balanced outermost termination automatically.
Lemma (ZR10)

Let \( R' \) be obtained from \( R \) by replacing \( \ell \rightarrow r \) by \( \ell \rightarrow r' \), where \( r \rightarrow^*_R r' \), and \( R' \) is productive

Then \( R \) is productive
Lemma (ZR10)

Let $R'$ be obtained from $R$ by replacing $\ell \rightarrow r$ by $\ell \rightarrow r'$, where $r \rightarrow^*_{R} r'$, and $R'$ is productive

Then $R$ is productive

Lemma (ZR10)

Let all rhs of $R$ be of the shape $d : t$

Then $R$ is productive
Lemma (ZR10)

Let $R'$ be obtained from $R$ by replacing $\ell \rightarrow r$ by $\ell \rightarrow r'$, where $r \rightarrow^*_R r'$, and $R'$ is productive.

Then $R$ is productive.

Lemma (ZR10)

Let all rhs of $R$ be of the shape $d : t$.

Then $R$ is productive.

Lemma (ZR10)

Let $R$ be terminating with respect to the context-sensitive rewrite relation is which rewriting in the right argument of ":" is disallowed.

Then $R$ is productive.
Carefully combining these lemmas and calling a standard tool for proving termination of context-sensitive rewriting yields a powerful technique for proving productivity automatically.
Carefully combining these lemmas and calling a standard tool for proving termination of context-sensitive rewriting yields a powerful technique for proving productivity automatically.

Has been implemented by R; a web interface is available.
Carefully combining these lemmas and calling a standard tool for proving termination of context-sensitive rewriting yields a powerful technique for proving productivity automatically.

Has been implemented by R; a web interface is available.

Successful for wide range of examples, including
- Hamming numbers, Thue Morse stream, Fibonacci stream, ...
- examples that do not satisfy data oblivious productivity
- more general infinite data structures, like binary trees
Streams are specified by a set of equations = rewrite rules

Desired properties: well-definedness and productivity

Techniques for proving well-definedness:
- Coalgebra
- Prove termination of observational variant
- Apply transformations that preserve well-definedness

Techniques for proving productivity:
- Pebble flow
- Prove outermost termination or context-sensitive termination
- Apply transformations that preserve productivity

Several techniques make use of termination tools
Summary of results

- Streams are specified by a set of equations = rewrite rules
Summary of results

- Streams are specified by a set of equations = rewrite rules
- Desired properties: well-definedness and productivity
Summary of results

- Streams are specified by a set of equations = rewrite rules
- Desired properties: well-definedness and productivity
- Techniques for proving well-definedness:
  - Coalgebra
  - Prove termination of observational variant
  - Apply transformations that preserve well-definedness
Streams are specified by a set of equations = rewrite rules

Desired properties: well-definedness and productivity

Techniques for proving well-definedness:
- Coalgebra
- Prove termination of observational variant
- Apply transformations that preserve well-definedness

Techniques for proving productivity:
- Pebble flow
- Prove outermost termination or context-sensitive termination
- Apply transformations that preserve productivity
Summary of results

- Streams are specified by a set of equations = rewrite rules
- Desired properties: well-definedness and productivity
- Techniques for proving well-definedness:
  - Coalgebra
  - Prove termination of observational variant
  - Apply transformations that preserve well-definedness
- Techniques for proving productivity:
  - Pebble flow
  - Prove outermost termination or context-sensitive termination
  - Apply transformations that preserve productivity
- Several techniques make use of termination tools
Apart from proving well-definedness or productivity there are more interesting issues.
Other issues

Apart from proving well-definedness or productivity there are more interesting issues.

In particular, for two stream specification one may wonder whether they specify the same stream.
Apart from proving well-definedness or productivity there are more interesting issues.

In particular, for two stream specification one may wonder whether they specify the same stream.

Typically, this may be done by equational reasoning and *coinduction* $\approx$ make a *bisimulation relation*.
Apart from proving well-definedness or productivity there are more interesting issues.

In particular, for two stream specification one may wonder whether they specify the same stream.

Typically, this may be done by equational reasoning and coinduction $\approx$ make a bisimulation relation.

General coinduction is hard to implement, but the tool CIRC (Lucanu & Roşu) implements circular coinduction by which several equalities can be proved automatically.
Apart from proving well-definedness or productivity there are more interesting issues.

In particular, for two stream specification one may wonder whether they specify the same stream.

Typically, this may be done by equational reasoning and coinduction $\approx$ make a bisimulation relation.

General coinduction is hard to implement, but the tool CIRC (Lucanu & Roşu) implements circular coinduction by which several equalities can be proved automatically.

Other interesting issues include levels of complexity.
Other issues

Apart from proving well-definedness or productivity there are more interesting issues.

In particular, for two stream specification one may wonder whether they specify the same stream.

Typically, this may be done by equational reasoning and coinduction \( \approx \) make a bisimulation relation.

General coinduction is hard to implement, but the tool CIRC (Lucanu & Roşu) implements circular coinduction by which several equalities can be proved automatically.

Other interesting issues include levels of complexity.

We conclude this presentation by some more examples.
Paper folding

$P$ is the boolean stream defined by

$$P = \text{zip}(\text{alt}, P)$$

$$\text{alt} = 0 : 1 : \text{alt}$$

$$\text{zip}(x : xs, ys) = x : \text{zip}(ys, xs)$$

in which $x$ is a boolean variable and $xs, ys$ are stream variables.
Paper folding

$P$ is the boolean stream defined by

$$P = \text{zip}(\text{alt}, P)$$

$$\text{alt} = 0 : 1 : \text{alt}$$

$$\text{zip}(x : xs, ys) = x : \text{zip}(ys, xs)$$

in which $x$ is a boolean variable and $xs, ys$ are stream variables.

Productivity and well-definedness is easily proved by any of the techniques we discussed.
Paper folding

$P$ is the boolean stream defined by

$$P = \text{zip}(\text{alt}, P)$$

$$\text{alt} = 0 : 1 : \text{alt}$$

$$\text{zip}(x : xs, ys) = x : \text{zip}(ys, xs)$$

in which $x$ is a boolean variable and $xs, ys$ are stream variables.

Productivity and well-definedness is easily proved by any of the techniques we discussed.

The turtle figure with $\alpha = \beta = 90^\circ$ is called the *dragon curve* or *dragon fractal*.
The dragon fractal
The Kolakoski stream

Let $f, g$ be defined by

$$f(0 : xs) = 0 : 0 : g(xs)$$
$$f(1 : xs) = 0 : g(xs)$$
$$g(0 : xs) = 1 : 1 : f(xs)$$
$$g(1 : xs) = 1 : f(xs)$$

The Kolakoski stream $K$ is the unique fixpoint of $g$, and satisfies

$$K = 1 : 0 : c$$
$$c = 0 : g(c)$$

for which productivity and well-definedness is easily proved by any of the techniques we discussed.
The Kolakoski stream

Let \( f, g \) be defined by

\[
\begin{align*}
  f(0 : xs) &= 0 : 0 : g(xs) \\
  f(1 : xs) &= 0 : g(xs) \\
  g(0 : xs) &= 1 : 1 : f(xs) \\
  g(1 : xs) &= 1 : f(xs)
\end{align*}
\]

The Kolakoski stream \( K \) is the unique fixpoint of \( g \), and satisfies

\[
K = 1 : 0 : c
\]

\[
c = 0 : g(c)
\]

for which productivity and well-definedness is easily proved by any of the techniques we discussed.

\( K \) behaves like a completely random boolean stream, and shows up many open problems.
The Kolakoski stream
One more example...

\[ Q = 1 : f(Q) \]
\[ f(0 : xs) = 1 : 0 : f(xs) \]
\[ f(1 : xs) = 0 : 0 : 1 : f(xs) \]
One more example...

\[ Q = 1 : f(Q) \]
\[ f(0 : xs) = 1 : 0 : f(xs) \]
\[ f(1 : xs) = 0 : 0 : 1 : f(xs) \]

Again a very simple stream specification for which productivity and well-definedness are easily proved.
One more example...

\[ Q = 1 : f(Q) \]
\[ f(0 : xs) = 1 : 0 : f(xs) \]
\[ f(1 : xs) = 0 : 0 : 1 : f(xs) \]

Again a very simple stream specification for which productivity and well-definedness are easily proved

Taking the first 200,000 elements and choose \( \alpha = 118^\circ \) yields the following turtle figure
Hans Zantema
Streams from a rewriting perspective
For all of these pictures the initial part of the stream was constructed by rewriting: start by the corresponding constant, and apply rewrite rules until the required number of elements has been produced.
For all of these pictures the initial part of the stream was constructed by rewriting: start by the corresponding constant, and apply rewrite rules until the required number of elements has been produced.

Next the turtle figure was drawn by a few lines of code
For all of these pictures the initial part of the stream was constructed by rewriting: start by the corresponding constant, and apply rewrite rules until the required number of elements has been produced.

Next the turtle figure was drawn by a few lines of code

Thank you for your attention