Exacte Wetenschappen

Aanvraagformulier Vrije competitie 2009 EW
Free Competition 2009 EW Application Form

Aanvraagnummer/application number: 600.065.120.09N07

1. Aanvrager(s)/applicant(s)

a. *Hoofdaanvrager/contactpersoon/main applicant/contact person*

volledige naam/name:	Prof. dr. J.W. Klop
organisatie/organisation:	Vrije Universiteit Amsterdam Afdeling Informatica Sectie Theoretische Informatica De Boelelaan 1081a 1081 HV AMSTERDAM

telefoon/telephone:	fax:	e-mail:
020-598 7735	020-598 7653	jwk@cs.vu.nl

b. *Overige aanvragers/co-applicants*

volledige naam/name	organisatie/organisation
Dr. R.C. de Vrijer	Vrije Universiteit Amsterdam Afdeling Informatica Sectie Theoretische Informatica

2. Titel/Title

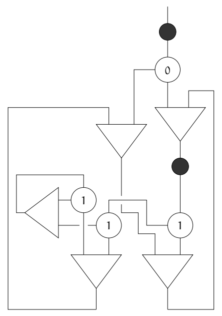
Lazy Productivity

3. Samenvatting/Abstract

Infinite objects like streams of natural numbers or other data can be specified by a variety of methods. Our framework is that of recursive equations, common in functional languages, based on infinitary term rewriting. For many specifications such as alt = 0 : 1 : alt, defining the stream of alternating bits, well-definedness is evident. For more involved specifications such as fib = 0 : 1 : (fib + tail(fib)) well-definedness is not obvious, and decision methods are needed. This is the issue of productivity, consisting of determining whether a specification yields an infinite normal form composed of constructors. In a slogan: productivity is for infinite objects what termination is for finite data specifications. There are various methods to ensure productivity. For `pure' stream specifications such as above, a powerful automated method has been developed based on a fine-grained pebbleflow network analysis.

Pebbles * are abstracted data: 0 = 1 = *. This data-oblivious approach can be extended by taking the identity of data into consideration (data-aware). In this proposal we extend the approach by admitting stream dependent data functions such as head of a stream. This involves dealing with `foresight' as in Y = 0 : head(tail(tail(Y))) : Y. Our primary concern is to treat stream dependent data functions, by explicitation of the input-output behaviour via a dependency analysis of the stream elements. Thus, we can deal also with partially defined stream terms. Second, we aim to generalize the methods to infinite data structures other than streams, such as infinite trees, lambda terms, and proof objects.

Research Proposal
Computer Science
Vrije Competitie NWO/EW
February 2009



Lazy Productivity (LaPro)

Applicants:

Prof. dr. Jan Willem Klop

Dr. Roel de Vrijer

Vrije Universiteit Amsterdam (VUA)

Principal investigator: Prof. dr. Jan Willem Klop
Department of Computer Science
Vrije Universiteit Amsterdam
Tel: +31 20 5987770
Fax: +31 20 5987653
E-mail: `jwk@cs.vu.nl`

1 Application

Project Title

Lazy Productivity

Project Acronym

LaPro

Principal Investigator

Jan Willem Klop

Renewed Application

This is a renewed application. The present proposal is a minor modification of our proposal submitted last year which was received very positively. The data of the original proposal are:

dossiernr. 600.065.100.08N002, **corr. nr. 2008/06610/EW**.

The main change with respect to the previous version is that we have taken up the encouraging suggestions by the referees. We now plan to:

i. formalize the production calculus and implement a decision method for productivity of stream specifications in the proof assistant Coq. This will offer a more flexible definitional mechanism for streams in Coq;

ii. investigate the applicability of our approach to dataflow (stream-based) synchronous programming languages, which have proven to be of use in the design of electronic circuits;

iii. study the run-time complexity of the decision procedures to be developed for the calculus of dependency streams.

2 Abstracts

Summary

Infinite objects like streams of natural numbers or other data can be specified by a variety of methods. Our framework is that of recursive equations, common in functional languages, based on infinitary term rewriting. For many specifications such as $\mathsf{alt} = 0 : 1 : \mathsf{alt}$, defining the stream of alternating bits, well-definedness is evident. For more involved specifications such as $\mathsf{fib} = 0 : 1 : (\mathsf{fib} + \mathsf{tail}(\mathsf{fib}))$ well-definedness is not obvious, and decision methods are needed. This is the issue of productivity, consisting of determining whether a

specification yields an infinite normal form composed of constructors. In a slogan: productivity is for infinite objects what termination is for finite data specifications. There are various methods to ensure productivity. For 'pure' stream specifications such as above, a powerful automated method has been developed based on a fine-grained pebbleflow network analysis. Pebbles $\bullet$ are abstracted data: $0 = 1 = \bullet$. This data-oblivious approach can be extended by taking the identity of data into consideration (data-aware). In this proposal we extend the approach by admitting stream dependent data functions such as head of a stream. This involves dealing with 'foresight' as in $\mathsf{X} = 0 : \mathsf{X}(n) : \mathsf{X}$ where $\mathsf{X}(n)$ is shorthand for $\mathsf{head}(\mathsf{tail}^n(\mathsf{X}))$. Our primary concern is to treat stream dependent data functions, by explicitation of the input-output behaviour via a dependency analysis of the stream elements. Thus, we can deal also with partially defined stream terms. Second, we aim to generalize the methods to infinite data structures other than streams, such as infinite trees, lambda terms, and proof objects.

Abstract for Laymen (in Dutch)

Oneindige objecten zoals oneindige lijsten (ook stromen genoemd) van natuurlijke getallen of andere data zijn dagelijkse kost in functionele programmeertalen. Ze worden veelal gespecificeerd door middel van stelsels recursievergelijkingen, die voor hun evaluatie steunen op de rigoreuze grondslag van oneindig termherschrijven. Voor veel voorkomende gevallen is de welgedefinieerdheid evident, zoals bijvoorbeeld voor $\mathsf{alt} = 0 : 1 : \mathsf{alt}$. Voor ingewikkelder vergelijkingen, zoals bijvoorbeeld $\mathsf{fib} = 0 : 1 : (\mathsf{fib} + \mathsf{tail}(\mathsf{fib}))$ is het soms zeer lastig de welgedefinieerdheid in te zien of te garanderen. Op dit punt rijst de noodzaak van beslissingsmethoden voor de welgedefinieerdheid, of zoals dit bekend staat, voor de eigenschap 'productiviteit' van de specificatie. Een productieve specificatie levert een oneindige normaalvorm op die bestaat uit constructoren, de bouwstenen van zulke oneindige objecten. In dit onderzoeksvoorstel breiden we de huidige beschikbare methoden uit om productiviteit, en daarmee correctheid, te garanderen van programma's die met oneindige objecten werken. We bouwen voort op een verfijnde analyse door middel van pebbleflow netwerken, waarin de gebeurtenissen in het verwerken van specificaties van oneindige objecten op atomair niveau in hun causaal verband worden bekeken. Het cruciale nieuwe element in dit voorstel is dat we een klasse toelaten van 'stroom-afhankelijke' functies die data als output hebben, zoals head van een stroom. Dit brengt een fenomeen van 'foresight', met zich mee, zoals in het voorbeeld $\mathsf{X} = 0 : \mathsf{X}(n) : \mathsf{X}$ waarbij $\mathsf{X}(n)$ staat voor $\mathsf{head}(\mathsf{tail}^n(\mathsf{X}))$. De toegenomen complexiteit wordt bestudeerd aan de hand van een 'dependency' analyse van het input-output gedrag van de stroom-elementen. Bij dit alles is het komen tot een daadwerkelijk inzetbaar 'tool' een leidraad.

Keywords

productivity — well-definedness — recursive specifications
program correctness — lazy evaluation — infinitary term rewriting

3 Classification

This proposal belongs to the discipline *computer science*, and is best placed within the NOAG-ict 2005–2010 theme *Methoden voor ontwerpen en bouwen*.

4 Composition of the Research Team

personalia	*specialty*	*affiliation*	*fte*
prof. dr. J.W. Klop	rewriting	VU	0.2
dr. V. van Oostrom	rewriting	UU	0.2
prof. dr. J.J.M.M. Rutten	coalgebra	CWI, VU	0.1
dr. R.C. de Vrijer	lambda calculus	VU	0.4

The candidate will be appointed as a postdoc for a period of three years.

5 Research School

The postdoc will participate in the research school IPA (Institute for Programming research and Algorithmics).

6 Description of the Proposed Research

6.1 Introduction

An important aspect of program correctness is termination. When dealing with programs that construct or process infinite objects, one cannot require termination. However, the question whether such a program is able to produce a canonical normal form remains of vital importance. This is the question of productivity. Productivity captures the intuitive notion of unlimited progress, of 'working' programs producing values indefinitely, programs immune to deadlock. We can already formulate the principle aim of this project:

> *This project is concerned with productivity of rewrite rules designed for lazy evaluation. We want to develop a 'production calculus' that faithfully represents delayed evaluation.*

This general formulation will be worked out into the more specific research questions Q1–Q7 in Sections 6.4 and 6.6.

In lazy functional programming languages such as Miranda [59], Clean [47] or Haskell [46], usage of infinite structures such as infinite lists (streams), is common practice. For the correctness of programs dealing with such structures one must guarantee that every finite part of the infinite structure can be evaluated; that is, the specification of the infinite structure must be productive. This holds both for terminating programs employing infinite structures (termination is then possible as only finite parts of the lazy evaluated infinite structures are queried), as well as for non-terminating programs that directly construct or process infinite objects. Of particular interest are synchronous dataflow (stream-based) programming languages like Esterel [9], Lucid [62], Lustre [29], or Lucid Synchrone [16]. Our research will enable more liberal definition schemes for these languages.

The application perspective follows straightforward from the need for program correctnes as mentioned. We are convinced that this perspective is not far away, but close at hand. The tool developed in the first part of this endeavour, for data-oblivious pure stream specifications[1] can be easily included in any functional programming environment. For tools corresponding to the extensions strived for in the present proposal, the same concern will be a guideline in the development.

To simplify the exposition we focus on the paradigmatic example of streams (infinite lists) and productivity of recursive stream specifications. However, an important goal of the project is to generalize the obtained results for stream specifications to specifications of other coinductive structures.

We study stream specifications in the framework of term rewriting. Here they are presented as systems of equations. A stream specification is called productive if not only can the specification be evaluated continually to build up a unique infinite normal form, but the resulting infinite expression is also meaningful in the sense that it is a constructor normal form which allows to read off consecutively individual elements of the stream. Since productivity of stream specifications is undecidable in general, the challenge is to find ever larger classes of stream specifications significant to programming practice for which productivity is decidable, or for which at least a powerful method for proving productivity exists.

The set of streams over a given set A is defined by: $A^{\omega} := \{\sigma \mid \sigma : \mathbb{N} \to A\}$, and elements $\sigma \in A^{\omega}$ are denoted $\sigma(0) : \sigma(1) : \sigma(2) : \ldots$, where ':' is the infix stream constructor prepending an element to a stream. In the paradigm of functional programming, stream programs are usually written as a system of recursive equations that are intended to be evaluated from left to right, as rewrite rules. Productivity is not just unique solvability of such a system, but also the potential to obtain the solution by evaluation. Consider the stream specification $\mathsf{Z} = \mathsf{z}(\mathsf{Z})$ with $\mathsf{z}(x : \sigma) = \mathsf{0} : \mathsf{z}(\sigma)$, which has as its unique solution the stream of zeros, but cannot be evaluated to obtain this solution.

[1]Available at `http://infinity.few.vu.nl/productivity/`, based on theory in [21, 20].

We give two examples, taken from [21]:

$$\mathsf{M} = 0 : \mathsf{zip}(\mathsf{inv}(\mathsf{even}(\mathsf{M})), \mathsf{tail}(\mathsf{M})) \tag{6.1}$$

$$\mathsf{E} = 0 : 1 : \mathsf{even}(\mathsf{E}) \tag{6.2}$$

zip zips two streams alternatingly into one: $\mathsf{zip}(x{:}\sigma, \tau) = x{:}\mathsf{zip}(\tau, \sigma)$, tail returns a stream without its first element: $\mathsf{tail}(x{:}\sigma) = \sigma$, inv inverts bit streams (streams over $\{0, 1\}$) element-wise: $\mathsf{inv}(x : \sigma) = \overline{x} : \mathsf{inv}(\sigma)$ with $\overline{0} = 1$ and $\overline{1} = 0$, and even returns a stream consisting of its even positions: $\mathsf{even}(x : \sigma) = x : \mathsf{odd}(\sigma)$ and $\mathsf{odd}(x{:}\sigma) = \mathsf{even}(\sigma)$. Equation (6.1) is an example of a productive specification; it defines the Thue–Morse sequence $0 : 1 : 1 : 0 : 1 : 0 : 0 : 1 : \ldots$.[2] Equation (6.2) is an example of a non-productive stream specification. Evaluating this specification yields $0{:}1{:}0{:}0{:}\mathsf{even}(\mathsf{even}(\ldots))$, that is, no more than four elements are produced.

To give a preview of the main technical focus of this proposal, note that the data rules in the Examples (6.1) and (6.2) (for inv only) are only depending on data, not streams. By contrast the rule for $\mathsf{head}(\sigma)$ delivering the first element of σ is typically stream dependent. This is an important distinction, and the succes of the method developed in [21] is because stream dependent data functions are disallowed. But in the present proposal we also consider stream dependent data functions. To amplify on this issue we now discuss, in Section 6.2, some strands of previous approaches, and then discuss, in Section 6.3, our proposed strategy in main lines, and in more detail in Section 6.4.

6.2 Related Work

The notion of productivity (sometimes also referred to as liveness) was first mentioned by Dijkstra [19]. Since then several papers [61, 54, 18, 32, 56, 14] have been devoted to criteria ensuring productivity. The common essence of these approaches is a quantitative analysis. They can be understood as describing the quantitative input/output behaviour of a stream function f by a 'modulus of production' $\nu_{\mathsf{f}} : \mathbb{N}^k \to \mathbb{N}$ with the property that the first $\nu_{\mathsf{f}}(n_1, \ldots, n_k)$ elements of $\mathsf{f}(t_1, \ldots, t_k)$ can be computed whenever the first n_i elements of t_i are defined.

In [61], Wadge uses dataflow networks to model fixed points of equations. He devises a so-called *cyclic sum test*, using production moduli of the form $\nu(n_1, \ldots, n_k) = \min(n_1 + a_1, \ldots, n_r + a_k)$ with $a_i \in \mathbb{Z}$, i.e. the output *leads* or *lags* the input by a fixed value a_i. Sijtsma [54] points out that this class of production moduli is too restrictive to capture the behaviour of commonly used stream functions like even or zip. For instance, the specification (6.1) cannot be dealt with by the cyclic sum test. Therefore Sijtsma develops an approach allowing arbitrary production moduli $\nu : \mathbb{N}^k \to \mathbb{N}$, an important notion, but less amenable for automation in full generality.

[2]Actually (6.1) can be simplified: if the occurrence of even is omitted, M still evaluates to the Thue–Morse sequence; even is included only to compare existing approaches that deal with stream productivity.

Coquand [18] defines a syntactic criterion called 'guardedness' for ensuring productivity in a type theoretical framework. This criterion was adapted for the Calculus of Constructions by Giménez [26], and implemented in the Coq proof assistant [55]. Guardedness is generally viewed as too restrictive for programming practice, because it disallows function applications to recursive calls. Another seminal approach is that of Telford and Turner [56], who extend the notion of guardedness with a method in the flavour of Wadge. They use a sophisticated counting scheme to compute the 'guardedness level' of a stream function, an element in $\mathbb{Z} \cup \{-\omega, \omega\}$. With this, a stream specification is recognized to be productive if the result of computing its guardedness level (by plain addition in the case of unary functions) from the guardedness levels of the stream functions occurring is positive. Still, their approach does not meet Sijtsma's criticism: their production moduli are essentially the same as Wadge's. Determining a guardedness level x, hence a modulus of the form $n \mapsto n + x$, for the stream function even leaves $x = -\omega$ as the only possibility. As a consequence, their algorithm does not recognize the specification (6.1) of M to be productive.

Hughes, Pareto and Sabry [32] introduce a type system using production moduli with the property that $\nu(a \cdot x + b) = c \cdot x + d$ for some $a, b, c, d \in \mathbb{N}$. For instance, the type they have to assign to the stream function tail is $\mathrm{ST}^{i+1} \to \mathrm{ST}^{i}$. Hence their system rejects the stream specification for M because the subterm $\mathsf{tail}(\mathsf{M})$ cannot be typed. Moreover, their class of moduli is not closed under composition, leading to the need of approximations and a loss of power.

Buchholz [14] presents a formal type system for proving productivity, whose basic ingredients are, closely connected to [54], unrestricted production moduli $\nu : \mathbb{N}^k \to \mathbb{N}$. In order to obtain an automatable method, Buchholz also devises a syntactic criterion to ensure productivity. This criterion easily handles all the examples of [56], but seems at a loss to deal with functions that have a negative effect like even, and hence with the specification (6.1) of M above.

Endrullis et al. [21, 20, 22] show that productivity is decidable for a rich class of recursive stream specifications (of which (6.1) and (6.2) are instances) that hitherto could not be handled automatically. Their decision algorithm exploits production moduli that are 'rational' functions $\nu : \mathbb{N}^k \to \mathbb{N}$ which, for $k = 1$, have eventually periodic difference functions $\Delta_\nu(n) := \nu(n+1) - \nu(n)$, that is $\exists n, p \in \mathbb{N}.\ \forall m \geq n.\ \Delta_\nu(m) = \Delta_\nu(m + p)$. This class of moduli is effectively closed under composition, and allows to calculate fixed points of unary functions. Rational production moduli generalise those employed by [61, 32, 18, 56], and provide the means to precisely capture the consumption/production behaviour of a large class of stream functions.

6.3 Delayed Evaluation of Stream Elements

We take a closer look at the syntactic class of 'pure' stream specifications whose productivity can be decided by the algorithm given in [21]. Pure specifications are orthogonal term rewriting systems consisting of three layers: a *stream layer* (top) where stream constants are specified using stream and data function symbols that are defined by the rules of the *function layer* (middle) and the *data-layer* (bottom). Each layer may use symbols defined in a lower layer, but not vice-versa. Only the rules in the function layer are subjected to syntactic restrictions (e.g. only 'flat' rules are allowed: rules without nested occurrences of function symbols in their right-hand sides) but no conditions other than well-sortedness are imposed on how the defining rules for the stream constant symbols in the stream layer can make use of the symbols in the other two layers.

An important restriction, from the perspective of decidability, imposed by the hierarchical setup of pure stream specifications in [21] is that symbols defined in the data-layer are *stream independent*. The reason for [21] to exclude data rules that make use of stream terms is to prevent the output of possibly undefined stream elements. Dropping this restriction significantly increases the complexity of the problem of recognizing productivity. Yet this is what we want to investigate, in order to account for 'lazy' function specifications. We elaborate on this topic below.

Consider the following two rules for a stream function which only 'reads' an incoming stream and outputs its successive elements:

$$\mathsf{read}_1(x : \sigma') = x : \mathsf{read}_1(\sigma') \tag{6.3}$$

$$\mathsf{read}_2(\sigma) = \mathsf{head}(\sigma) : \mathsf{read}_2(\mathsf{tail}(\sigma)) \tag{6.4}$$

where head and tail are defined by $\mathsf{head}(x : \sigma') = x$ and $\mathsf{tail}(x : \sigma') = \sigma'$. The rule (6.3) falls in the format of [21], but (6.4) does not. A term $\mathsf{read}_1(t)$ is a redex w.r.t. rule (6.3) only in case $t \equiv d : u$, whereas $\mathsf{read}_2(t)$ constitutes a redex w.r.t. (6.4) for *any* stream term t, and so $\mathsf{head}(t)$ can be undefined. The difference in evaluating these rules is that the 'lazy rule' (6.4) *postpones* pattern matching. The algorithm of [21] calculates the length of the longest prefix of defined stream elements a stream term evaluates to. It cannot deal with partially defined stream terms where intermediate elements are undefined.

We propose to investigate a finer semantics for modelling stream specifications, in particular for defining rules for stream functions. We start with a motivating example, taken from [54]:

$$\mathsf{S} = 0 : \mathsf{S}(2) : \mathsf{S} \tag{6.5}$$

$$\mathsf{T} = 0 : \mathsf{T}(3) : \mathsf{T} \tag{6.6}$$

where $\sigma(n)$ denotes the n-th element of σ; so $\sigma(n) = \mathsf{head}(\mathsf{tail}^n(\sigma))$. How do we check whether these definitions are productive? The difficulty in both cases is the reference to an element that becomes available only in future unfoldings of these equations: the element of S with index 1 depends on the element with index 2: $\mathsf{S}(1) = \mathsf{S}(2)$; likewise $\mathsf{T}(1) = \mathsf{T}(3)$. As said before, a function like $\lambda\sigma.\,\lambda n.\,\sigma(n)$ possibly creating 'look-ahead', falls outside the format allowed by [21]. Hence, we cannot use that method to show productivity of (6.5) and non-productivity of (6.6).

In the terminology of [54] a unary stream function f is 'ν_{f}-productive' with $\nu_{\mathsf{f}} : \mathbb{N} \to \mathbb{N}$, if, whenever the first n elements of a stream σ are defined, (at least) the first $\nu_f(n)$ elements of $f(\sigma)$ are defined. Sijtsma shows that if $\nu_f(n) > n$ for all $n \in \mathbb{N}$, then $\mathrm{lfp}(f)$ is a productive stream [54, Thm. 32], where $\mathrm{lfp}(f)$ denotes the least fixed point of f. To return to (6.5), note that $\mathsf{S} = \mathrm{lfp}(\mathsf{s})$, where:

$$\mathsf{s}(\sigma) = 0 : \sigma(2) : \sigma\,. \tag{6.7}$$

This function is ν_s-productive, with ν_s defined by: $\nu_\mathsf{s}(n) = 1$ if $n \leq 2$, and $\nu_\mathsf{s}(n) = n + 2$ if $n > 2$. Clearly, it is not the case that $\forall n \in \mathbb{N}.\ \nu_\mathsf{s}(n) > n$.

So far we have discussed what Sijtsma calls 'segment productivity' [54]. For the restricted class of pure stream specifications, for which the work [21] gives a decidability algorithm, productivity and segment productivity coincide. Sijtsma also develops a more general notion, called 'set productivity', with production moduli in $\mathcal{P}(\mathbb{N}) \to \mathcal{P}(\mathbb{N})$. These again specify the production behaviour of the stream functions they model, but can be used for proving productivity of a larger class of stream definitions including (6.5). Sijtsma formulates sufficient conditions for set productivity, but does not provide a method to compute these conditions. One of the main goals of LaPro is:

Identify a class of lazy stream specifications for which productivity is decidable / computable criteria exist.

Here by a 'lazy' stream specification we mean a stream specification that allows for stream dependent data functions. In contrast to pure specifications, in lazy ones matching the pattern of a stream can be 'delayed' by using head and tail in the right-hand sides of defining rules; e.g. (6.4) is the lazy version of (6.3).

6.4 The Production Calculus

We propose a simple semantics for modelling the production functions of stream functions that allows the delay of evaluating stream elements, necessary to deal with examples like (6.5) above, exploiting Sijtsma's meta-theory. Instead of 'I/O sequences' used by [21] to model input/output behaviour of stream functions, we want to employ a finer notion, something which we provisionally call 'dependency streams', or just d-streams for short.

Develop the calculus of d-streams. (Q1)

For the sake of simplicity we restrict the exposition to unary stream operations. The set Δ of d-streams is defined by $\Delta := \mathcal{P}(\mathbb{N})^\omega$, i.e. streams of sets of natural numbers. We want a uniform, production behaviour preserving, translation which maps stream functions to their corresponding d-streams (and make this correspondence precise).

Define a translation from stream functions to d-streams. (Q2)

For now, we use the notation δ_f ($\in \Delta$) to denote the d-stream corresponding to a defining rewrite rule for a stream function symbol f. For all $i \in \mathbb{N}$ the index set $\delta_f(i)$ at position i indicates the elements of the incoming stream it 'depends' on. For instance we have $\delta_{\mathsf{even}} = \{0\}:\{2\}:\{4\}:\dots$, and for the function s defined by (6.7) we find: $\delta_{\mathsf{s}} = \varnothing:\{2\}:\{0\}:\{1\}:\{2\}:\dots$. To get a taste of the calculus we have in mind, we give the corecursive definition of composition of d-streams:

$$(A:\delta)\circ\gamma := (\cup_{i\in A}\,\gamma(i)):(\delta\circ\gamma)\,.$$

This is an associative operation, and for the translation of stream functions it should hold that composition is preserved: $\delta_{\mathsf{f}\circ\mathsf{g}} = \delta_{\mathsf{f}}\circ\delta_{\mathsf{g}}$.

Identify a class of d-streams that is closed under, and allows for the computation of: composition, infimum, and fixed points. (Q3)

Dependency streams are interpreted as follows: The *production function* $\pi_\delta : \mathcal{P}(\mathbb{N}) \to \mathcal{P}(\mathbb{N})$ *of a d-stream* δ, is defined by:

$$\pi_\delta(A) := \{i \mid \delta(i) \subseteq A\}\,.$$

The idea is that, when given a stream function f and a stream term t with as its only defined elements those indexed by the elements of a set $A \subseteq \mathbb{N}$, then the set $\pi_{\delta_{\mathsf{f}}}(A)$ is the set of indices of the defined elements of the stream expression $\mathsf{f}(t)$. For composition it should hold that $\pi_{\delta\circ\gamma} = \pi_\delta\circ\pi_\gamma$. If we have that $\mathrm{lfp}(\pi_{\delta_{\mathsf{f}}}) = \mathbb{N}$, then $\mathsf{X} = \mathsf{f}(\mathsf{X})$ is a productive stream definition.

Relate the calculus of d-streams to the theory of set productivity [54]. (Q4)

To see that d-streams are a refinement of 'I/O sequences' (used by [21, 20] to represent periodially increasing production moduli), note that the latter correspond to monotone d-streams. A d-stream δ is monotone if $\delta(i) \subseteq \delta(j)$ for all $i \leq j$. For example, translating the I/O sequence $\overline{--+}$ corresponding to the stream function odd to a d-stream, we obtain $= \mathbf{2}:\mathbf{4}:\mathbf{6}\dots$, where $\mathbf{n} := \{0,\dots,n-1\}$. That is, to compute $(\mathsf{odd}(t))(1)$, all of the elements $t(0)$, $t(1)$, $t(2)$, $t(3)$ have to be defined. On the other hand, in the translation of unary stream functions into d-streams that we have in mind, odd would be translated to $\{1\}:\{3\}:\{5\}:\dots$, which says that to compute $(\mathsf{odd}(t))(1)$, we only need $t(3)$.

Embed the pebbleflow calculus of [21] in the calculus of d-streams. (Q5)

An alternative notion that can be of help for determining productivity is the following: A *path through* δ is a sequence of indices $d_1 \to d_2 \to \dots$ where $d_i \to d_{i+1}$ iff $d_{i+1} \in \delta_{\mathsf{f}}(d_i)$, paraphrased as d_i *depends on* d_{i+1}. Let $\mathsf{f} : A^\omega \to A^\omega$ be a stream function, and $\delta_{\mathsf{f}} \in \Delta$ be its d-stream. We conjecture:

all paths through δ_{f} *are finite if and only if* $\mathrm{lfp}(\mathsf{f})$ *is a productive stream* ,

where by a 'productive stream' we mean a stream all whose elements are defined. We show two applications of this theorem. First, reconsider S specified in (6.5), as the least fixed point of s, see (6.7). It is not hard to observe that all paths through δ_{s} are finite:

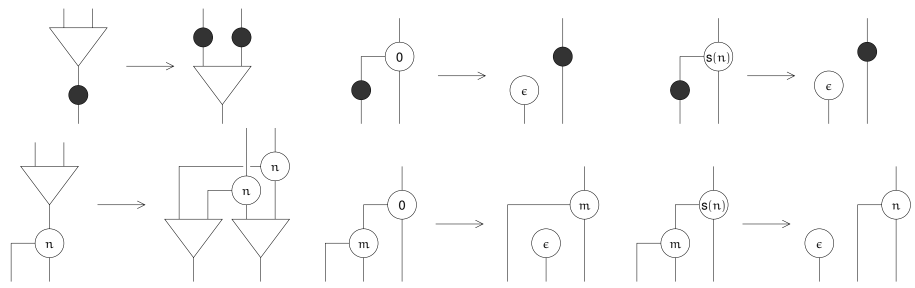


Figure 1: *Interaction rules for lazy pebbleflow.*

$$\begin{tikzcd}
\varnothing \;: & \{2\} \;: \arrow[r, bend right] & \{0\} \;: \arrow[ll, bend right] & \{1\} \;: \arrow[ll, bend right] & \{2\} \;: \arrow[ll, bend right] & \ldots \arrow[ll, bend right, dashed]
\end{tikzcd}$$

where an arrow from position i to position j indicates that i depends on j, i.e. $j \in \delta_{\mathsf{s}}(i)$. An example path through δ_{s} is $3 \to 1 \to 2 \to 0$. Hence, by application of the conjectured theorem, we conclude that the specification for S defines a productive stream.

Second, reconsider the specification (6.2), and note that E is the least fixed point of the function e defined by: $\mathsf{e}(\sigma) = 0 : 1 : \mathsf{even}(\sigma)$, with d-stream

$$\delta_{\mathsf{e}} = \varnothing : \varnothing : \delta_{\mathsf{even}} = \varnothing : \varnothing : \{0\} : \{2\} : \{4\} : \ldots$$

Now observe that $4 \in \delta_{\mathsf{e}}(4)$, and so $4 \to 4 \to \ldots$ is an infinite path through δ_{e}. It follows that E is not productive.

We note that termination problems can be encoded as productivity problems. For instance, it is easy to encode the well-known Collatz conjecture, also known as the '$3n+1$-problem'. It is also possible to encode Conway's Fractran [17] in the calculus of d-streams, implying that, like Fractran, the calculus of d-streams is Turing complete.

As we did in [21], here it may also turn out to be useful to visualize the evaluation of stream specifications by the flow of abstract 'pebbles' in cyclic networks (a particular kind of interaction nets [38]). To account for the extended calculus, we use 'stream dependent' pebbles. See Figure 1 for the interaction rules governing 'lazy pebbleflow'. In Figure 2 is displayed the initial part of an evaluation sequence of the net corresponding to the specification (6.5).

For the newly found classes of stream specifications we want to extend and modify the existing tool (see the footnote on page 5) which decides / recognizes productivity of these specifications.

We plan to carry out a precise analysis of the complexity of the productivity checkers we will develop. We expect the worst-case complexity of the existing decision algorithm for pure stream specifications [21, 20] to be exponential. This expectation is based on the observation that the period of the composition of rational stream functions is the least common multiple of the consumption

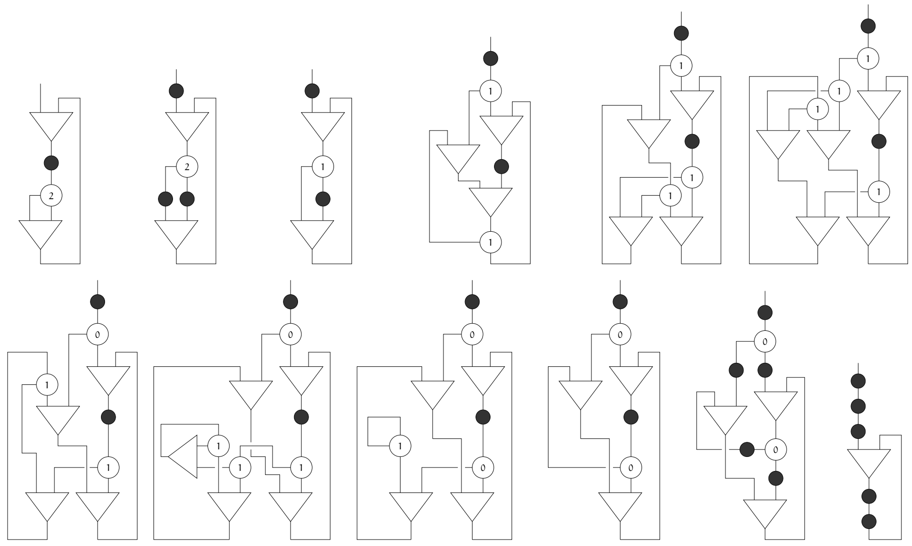


Figure 2: *Initial part of an evaluation sequence of the lazy pebbleflow net associated to* S *(top-left). The last net displayed (bottom-right) clearly produces an infinite chain of pebbles, showing productivity of the specification* (6.5).

(number of stream constructors to be matched) of their defining rules. Hence, it may be possible to construct an artificial example with stream functions that have distinct prime numbers as their consumption. However, we believe that such examples do not occur in practice; the ones we encountered in the literature mostly deal with consumptions ≤ 3.

Of potential future interest are connections with the world of typed systems, in particular typed lambda calculi, where the paradigm notion of an infinite object is that of a Böhm tree. Productivity is after translation to lambda calculus tantamount to the Böhm tree having a hereditary head normal form (being totally defined), for which property currently type systems are being developed.

Apply the obtained results to other coinductive
data structures, e.g. infinite trees, cyclic proof objects. (Q6)

We further envisage establishing stronger links with recent coalgebraic techniques that give an alternative, more general view on well-definedness and uniqueness properties of stream specifications.

6.5 Embedding in Existing Research

The Theoretical Computer Science Group at the VU, headed by Prof. Wan Fokkink, specializes in the development and application of formal techniques for the verification of distributed systems and protocols. There is a focus on protocol verification (Wan Fokkink), term rewriting (Jan Willem Klop, Roel de Vrijer, Femke van Raamsdonk), and coalgebra (Jan Rutten). Five PhD students and two postdocs take part in projects on system verification, process algebra, term rewriting, and coalgebra.

The research group at the VU, in cooperation with Utrecht University and CWI, hosts the NWO FOCUS/BRICKS-project Infinity (Infinite Objects, Computation, Modelling and Reasoning), of which the proposed project LaPro can be viewed as a spin-off. The objective of the Infinity project is the study of infinite objects from the combined perspective of term rewriting, proof theory and coalgebra. The project LaPro shifts the attention to lazy evaluation of recursive specifications, in particular in the context of functional programming. This cooperation between CWI, UU and VU provides also the broad context in which the proposed project LaPro will be embedded.

6.6 Application Perspective

The application domain of this proposal is functional programming, in particular giving meaning to non-terminating programs, and providing methods to show program correctness. We elaborate on this matter in Section 6.1.

A major application that we have in mind is an implementation of our decision algorithm in the Coq proof system [55]. In Coq only guarded recursion is allowed, which is far too restrictive for functional programming, see Section 6.2. Our effort admits a more flexible definitional mechanism of streams in Coq. More concretely, we plan to formalize in Coq the developed production calculus including a decision method on a syntactically defined class of stream specifcations.

Formalize questions Q1–Q3, Q5 in the Coq system. (Q7)

Then, by using the method of reflection [12] the formalized theory can be lifted to the native language of the proof assistant. This gives rise to a productivity checker for a (much) larger class of stream specifications. Well-definedness is assured by correctness of the decision algorithm.

7 Project Planning

In the first half of the project, we expect the postdoc to work out the production calculus (Q1), including a translation from lazy stream function specifications to d-streams (Q2). The postdoc identifies a subclass of lazy stream specifications (significantly extending the pure format) for which (i) the translation to d-streams is production preserving, and (ii) the image of this class under the translation is closed under, and allows for the computation of, composition, infimum and fixed points (Q3). As a start, the periodically increasing production moduli employed by [20] are embedded into d-streams (Q5). The development of the theory will go hand in hand with its formalization in Coq (Q7).

The next phase, starting at the end of the first year, is the interpretation of the algebra of d-streams into a suitable (co)algebra and prove the involved operations correct. A first step in this respect is to exploit Sijtsma's meta-theory of set-productivity (Q4).

In the last year of the project, the tool of [21] for automatic detection of productivity of stream specifications is extended to comply with the newly found classes. Furthermore the postdoc investigates the possibilities to generalize the obtained results for stream specifications to other coinductive data structures, e.g. cyclic proof objects (Q6).

8 Expected Use of Instrumentation

We do not apply for project related apparatus or software.

9 Literature

Key publications of the research team are [58, 50, 37, 21, 23].

[1] A. Abel. Termination and Productivity Checking with Continuous Types. In *TLCA*, pages 1–15, 2003.

[2] S. Abramsky and A. Jung. Domain Theory. In S. Abramsky, D. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 3, pages 1–168. Oxford University Press, 1994.

[3] J.-P. Allouche and J. Shallit. *Automatic Sequences: Theory, Applications, Generalizations*. Cambridge University Press, New York, 2003.

[4] R.M. Amadio and S. Coupet-Grimal. Analysis of a Guard Condition in Type Theory. In *FoSSaCS*, volume 1378 of *LNCS*, pages 48–62. Springer, 1998.

[5] Z.M. Ariola and J.W. Klop. Equational Term Graph Rewriting. *Fundamentae Informaticae*, 26(3-4):207–240, 1996.

[6] J.C.M. Baeten, J.A. Bergstra, and J.W. Klop. Decidability of Bisimulation Equivalence for Processes Generating Context-Free Languages. *J. ACM*, 40(3):653–682, 1993.

[7] G. Barthe, M.J. Frade, E. Giménez, L. Pinto, and T. Uustalu. Type-based Termination of Recursive Definitions. *Mathematical Structures in Computer Science*, 14:97–141, 2004.

[8] J. Bergstra, C. Middelburg, and Gh. Ştefănescu. Network Algebra for Synchronous and Asynchronous Dataflow. Technical Report P9508, University of Amsterdam, 1995.

[9] G. Berry and G. Gonthier. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[10] Y. Bertot. Filters on CoInductive Streams, an Application to Eratosthenes' Sieve. In *TLCA*, volume 3461 of *LNCS*, pages 102–115. Springer, 2005.

[11] S. Blom. *Term Graph Rewriting — syntax and semantics*. PhD thesis, Vrije Universiteit Amsterdam, 2001.

[12] S. Boutin. Using Reflection to Build Efficient and Certified Decision Procedures . In M. Abadi and T. Ito, editors, *TACS*, volume 1281 of *Lecture Notes in Computer Science*, pages 515–529. Springer, 1997.

[13] J.D. Brock and W.B. Ackerman. Scenarios: A Model of Non-Determinate Computation. In *ICFPC*, volume 107 of *Lecture Notes in Computer Science*, pages 252–259. Springer, 1981.

[14] W. Buchholz. A Term Calculus for (Co-)Recursive Definitions on Streamlike Data Structures. *Annals of Pure and Applied Logic*, 136(1-2):75–90, 2005.

[15] V. Capretta. General Recursion via Coinductive Types. *Logical Methods in Computer Science*, 1:1–28, 2005.

[16] P. Caspi and M. Pouzet. Synchronous Kahn Networks. In *International Conference on Functional Programming*, pages 226–238, 1996.

[17] J.H. Conway. FRACTRAN: A Simple Universal Programming Language for Arithmetic. In T.M. Cover and B. Gopinath, editors, *Open Problems in Communication and Computation*, pages 4–26. Springer–Verlag, 1987.

[18] Th. Coquand. Infinite Objects in Type Theory. In H. Barendregt and T. Nipkow, editors, *TYPES*, volume 806, pages 62–78. Springer–Verlag, Berlin, 1994.

[19] E.W. Dijkstra. On the Productivity of Recursive Definitions. `http://www.cs.utexas.edu/users/EWD/ewd07xx/EWD749.PDF`, 1980.

[20] J. Endrullis, C. Grabmayer, and D. Hendriks. Data-Oblivious Stream Productivity. In *Logic for Programming, Artificial intelligence and Reasoning 2008*, number 5330 in LNCS, pages 79–96. Springer, 2008.

[21] J. Endrullis, C. Grabmayer, D. Hendriks, A. Isihara, and J.W. Klop. Productivity of Stream Definitions. In *Fundamentals of Computation Theory 2007*, number 4639 in LNCS, pages 274–287. Springer, 2007.

[22] J. Endrullis, C. Grabmayer, D. Hendriks, A. Isihara, and J.W. Klop. Productivity of Stream Definitions. Technical Report Preprint 268, Logic Group Preprint Series, Department of Philosophy, Utrecht University, 2008. Accepted for publication in a forthcoming special issue of TCS. Available at `http://www.phil.uu.nl/preprints/lgps/`.

[23] J. Endrullis, C. Grabmayer, D. Hendriks, J.W. Klop, and R.C de Vrijer. Proving Infinitary Normalization. In *TYPES 2008*, 2009. Accepted for publication.

[24] M. Fernández and I. Mackie. Operational Equivalence for Interaction Nets. *Theoretical Computer Science*, 1–3(297):157–181, 2003.

[25] G. Di Gianantonio and M. Miculan. A Unifying Approach to Recursive and Corecursive Definitions. In *TYPES 2002*, number 2646 in LNCS, pages 148–161, 2003.

[26] E. Giménez. Codifying Guarded Definitions with Recursive Schemes. In *TYPES*, volume 996 of *LNCS*, pages 39–59. Springer, 1994.

[27] C. Grabmayer. *Relating Proof Systems for Recursive Types*. PhD thesis, Vrije Universiteit Amsterdam, 2005.

[28] N. Halbwachs. Synchronous Programming of Reactive Systems. In *Computer Aided Verification*, pages 1–16, 1998.

[29] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The Synchronous Data Flow Programming Language LUSTRE. In *Proc. of the IEEE*, volume 79, pages 1305–1320, 1991.

[30] M. Hasegawa. *Models of Sharing Graphs: A Categorical Semantics of* let *and* letrec. PhD thesis, University of Edinburgh, 1999.

[31] Y. Hirshfeld and F. Moller. Decidability Results in Automata and Process Theory. In G. Birtwistle and F. Moller, editors, *Logics for Concurrency*, volume 1043 of *LNCS*, pages 102–148. Springer Verlag, 1996.

[32] J. Hughes, L. Pareto, and A. Sabry. Proving the Correctness of Reactive Systems Using Sized Types. In *POPL '96*, pages 410–423, 1996.

[33] B. Jacobs and J. Rutten. A Tutorial on (Co)Algebras and (Co)Induction. *EATCS Bulletin*, 62:222–259, 1997.

[34] G. Kahn. The Semantics of a Simple Language for Parallel Programming. In *Information Processing*, pages 471–475, 1974.

[35] R. Kennaway, J.W. Klop, M.R. Sleep, and F.-J. de Vries. Transfinite Reductions in Orthogonal Term Rewriting Systems. *Inf. Comput.*, 119(1):18–38, 1995.

[36] R. Kennaway, J.W. Klop, M.R. Sleep, and F.-J. de Vries. Infinitary Lambda Calculus. *Theoretical Computer Science*, 175(1):93–125, 1997.

[37] J.W. Klop and R.C. de Vrijer. Infinitary Normalization. In S. Artemov, H. Barringer, A.S. d'Avila Garcez, L.C. Lamb, and J. Woods, editors, *We Will Show Them: Essays in Honour of Dov Gabbay*, volume 2, pages 169–192. College Publications, 2005.

[38] Y. Lafont. Interaction Nets. In *POPL '90*, pages 95–108. ACM Press, 1990.

[39] J. Matthews. Recursive Function Definition over Coinductive Types. In *TPHOLs '99*, number 1690 in LNCS, pages 73–90, 1999.

[40] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[41] R. Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Springer Verlag, 1999.

[42] M. Niqui. *Formalising Exact Arithmetic: Representations, Algorithms and Proofs*. PhD thesis, Radboud Universiteit Nijmegen, 2004.

[43] M. Niqui. Coinductive Field of Exact Real Numbers and General Corecursion. In *CMCS 2006*, ENTCS. Elsevier Science Publishers, 2006.

[44] M. Niqui. Productivity of Edalat–Potts Exact Arithmetic in Constructive Type Theory. *Theory of Computing Systems*, 2006.

[45] S.E. Panitz. Termination Proofs for a Lazy Functional Language by Abstract Reduction. Technical Report Frank 06, J.W. Goethe-Universität, 1996.

[46] S. Peyton Jones. *Haskell 98 Language and Libraries, The Revised Report*. Cambridge University Press, 2003.

[47] R. Plasmeijer and M. Eekelen. The Concurrent Clean Language Report (version 2.0). Technical report, University of Nijmegen, 2001.

[48] C. Raffalli. Data Types, Infinity and Equality in System AF_2. In E. Börger, Y. Gurevich, and K. Meinke, editors, *CSL '93*, volume 832 of *LNCS*, pages 280–294, 1993.

[49] J.J.M.M. Rutten. Behavioural Differential Equations: a Coinductive Calculus of Streams, Automata, and Power Series. *Theoretical Computer Science*, 308(1-3):1–53, 2003.

[50] J.J.M.M. Rutten. *On Streams and Coinduction*, volume 23 of *CRM Monograph series*, pages 1–92. American Mathematical Society, 2004.

[51] A. Salomaa. *Jewels of Formal Language Theory*. Pitman Publishing, 1981.

[52] D. Sangiorgi. On the Bisimulation Proof Method. *Mathematical Structures in Computer Science*, 8:447–479, 1998.

[53] D. Sangiorgi and D. Walker. *The Pi-calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.

[54] B.A. Sijtsma. On the Productivity of Recursive List Definitions. *ACM Transactions on Programming Languages and Systems*, 11(4):633–649, 1989.

[55] The Coq Development Team. *The Coq Proof Assistant Reference Manual, version 8.1*, 2006. `http://coq.inria.fr/`.

[56] A. Telford and D. Turner. Ensuring Streams Flow. In *AMAST*, pages 509–523, 1997.

[57] A. Telford and D. Turner. Ensuring the Productivity of Infinite Structures. Technical Report 14-97, The Computing Laboratory, University of Kent at Canterbury, 1997.

[58] Terese. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2003. Author's acronym includes J.W. Klop, V. van Oostrom and R.C. de Vrijer.

[59] D. Turner. An overview of miranda. *SIGPLAN Not.*, 21(12):158–166, 1986.

[60] T. Uustalu and V. Vene. The Essence of Dataflow Programming. In *CEFP*, pages 135–167, 2005.

[61] W.W. Wadge. An Extensional Treatment of Dataflow Deadlock. *Theoretical Computer Science*, 13:3–15, 1981.

[62] W.W. Wadge and E. Ashcroft. *LUCID, The Dataflow Programming Language*. Academic Press Professional, Inc., 1985.

10 Requested Budget

The budget is for a three-year postdoc on the basis of 1.0 fte.

appointment (inc. benchfee)	191.955
additional travelling budget	-
project related apparatus/software	-
other related activities	-
total	191.955