Behavioral Rewrite Systems and Behavioral Productivity

Grigore Roşu[1,2] and Dorel Lucanu[2]

[1] University of Illinois at Urbana-Champaign, USA, grosu@illinois.edu
[2] Alexandru Ioan Cuza University, Iaşi, Romania, dlucanu@info.uaic.ro

Abstract. This paper introduces *behavioral rewrite systems*, where rewriting is used to evaluate experiments, and *behavioral productivity*, which says that each experiment can be fully evaluated, and investigates some of their properties. First, it is shown that, in the case of (infinite) streams, behavioral productivity generalizes and may bring to a more basic rewriting setting the existing notion of stream productivity defined in the context of infinite rewriting and lazy strategies; some arguments are given that in some cases one may prefer the behavioral approach. Second, a behavioral productivity criterion is given, which reduces the problem to conventional term rewrite system termination, so that one can use off-the-shelf termination tools and techniques for checking behavioral productivity in general, not only for streams. Finally, behavioral productivity is shown to be equivalent to a proof-theoretic (rather than model-theoretic) notion of behavioral well-specifiedness, and its difficulty in the arithmetic hierarchy is shown to be Π^0_2-complete. All new concepts are exemplified over streams, infinite binary trees, and processes.

1 Introduction

Behavioral abstraction, or the process of understanding how a system behaves under a given set of relevant observations or experiments, is a fundamental problem in formal methods: like information hiding, behavioral abstraction provides the capability to abstract away from internal implementation details to better capture and reason about the actual system behavior.

Behavioral equivalence, also informally called *indistinguishability under experiments* in the literature [18, 13, 2, 19], is an important example of behavioral abstraction. CafeOBJ [4], an executable specification language developed under the leadership and vision of Kokichi Futatsugi, was one of the first systems that provided explicit support for specifying and verifying behavioral equivalence.

We briefly explain behavioral equivalence using a very simple example. The two (infinite) processes represented in Figure 1 can be behaviorally specified by the following terminating term rewriting system R (behavioral specifications typically use equations, but we here tacitly use rewriting instead):

$$out(a) \to 0 \qquad out(b) \to 1 \qquad next(a) \to b \qquad next(b) \to a$$
$$out(s_i) \to i \bmod 2 \qquad next(s_i) \to s_{i+1}$$

Each state has an output value, represented in the figure by a pair *state*/*output*. The output is modeled by the operation $out(state)$ and the transitions are modeled by the operation $next(state)$. We can observe that the states a and s_0 are behaviorally indistinguishable by experimenting with them:

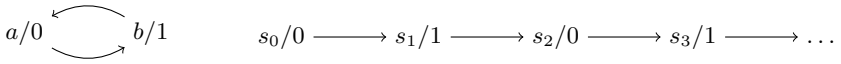

Fig. 1. Two behavioraly equivalent processes.

1. We first check if the output values for the two states are equal: $out(a) \xrightarrow{*}_R 0$ and $out(s_0) \xrightarrow{*}_R 0$.
2. We check the equality of the output values after one transition: $out(next(a)) \xrightarrow{*}_R 1$ and $out(next(s_0)) \xrightarrow{*}_R 1$.
3. We check the equality of the output values after two transitions: $out(next^2(a)) \xrightarrow{*}_R 0$ and $out(next^2(s_0)) \xrightarrow{*}_R 0$. And so on.

So, for each experiment $out(next^i(*))$ respectively applied on the two states, we obtain $out(next^i(a)) \xrightarrow{*}_R v$ and $out(next^i(s_0)) \xrightarrow{*}_R v$ for certain $v \in \{0, 1\}$ and thus conclude that a and s_0 are indistinguishable under experiments. Obviously, a and s_1 are distinguishable under experiments (e.g., $out(s_1) \xrightarrow{*}_R 1$).

Lazy rewriting is an alternative, more operational approach to study infinite-behavior objects. The idea here is to use lazy rewriting to only extract as much information from the infinite behavior of an object or data-structure as needed in the given context, this way avoiding the infinite nature of the object or data-structure. In this approach, the notion of *productivity* [5, 10, 24] plays a crucial rule. It captures the intuition of *unlimited progress*, that is, that the term under analysis can be continuously evaluated (or rewritten) in such a way that its infinite behavior is uniquely determined as the limit of this evaluation process.

Both behavioral equivalence and productivity were proposed in the early 1980's, the former by Reichel [18] and the later by Dijkstra [5]. Since then, attracted by the benefits and elegance of each of the two approaches, there have been many related approaches, reasoning techniques and tool prototypes proposed for each of them, e.g. [2, 4, 7–10, 13, 14, 16, 17, 19, 22, 24–26, 28, 29] among many others. However, up to now, in spite of common intuitions and ultimate goals, these two approaches to infinite behavior have lived separate lives. In this paper we make a first step towards bringing the two approaches closer. To make this possible, we first introduce the general notion of a behavioral rewrite system, and then formally define our notion of behavioral productivity for such systems. Note that almost any two papers in the aforementioned lists defines different variants of behavioral equivalence or productivity. We are not attempting to consolidate all these different variations in this paper. Instead, our objective is to capture the *essence* of these important concepts in order to highlight their relationships. We believe that our results can be adapted to each particular approach, but this is beyond our scope here.

A *behavioral rewrite system* (BRS) is a term rewrite system (TRS) together with a set of *derivative* operations (or *observers*) which are used to formally define *experiments*, where the rewriting relation is used to compute the results of the experiments. The usual productivity makes sense for those TRS's defining infinite data structures[3]: R is productive for a ground term t, intended to represent an infinite data structure $[\![t]\!]$, if the rewriting relation $\rightarrow_R$ can be used to

[3] Again, formal definitions of productivity differ from paper to paper, mixing the conceptual notion with operational or technical limitations (e.g., requiring the terms

obtain any approximation of $[\![t]\!]$ starting from t. We propose *behavioral productivity* as another example of a behavioral abstraction: it says that each experiment applied on t can be computed in finite time by means of ordinary rewriting. Behavioral productivity captures the idea that the behavior of a given term can be gradually "produced"; since the set of experiments that need to be applied on the term in order to potentially yield the term's behavior is recursively enumerable, behavioral productivity means, in fact, that each of the experiments applied on the term can be "evaluated", or in our rewrite context, can be rewritten to a data term. For the above process example, R is behaviorally productive for a state s if and only if each experiment of the form $out(next^i(*))$ can be rewritten to a data value term (here 0 or 1) when applied in any state. It is easy to see that R is behaviorally productive for all states of the two processes. However, if we add a new state c and only the transition $next(b) \to c$, then we observe that R is not behaviorally productive for c because $out(next^i(c))$ is irreducible.

We show that for streams, behavioral productivity for BRS's generalizes the productivity for TRS's in several ways: it can be defined for a larger class of specifications, there are non-productive TRS's for which their behavioral versions are behaviorally productive, and it can be defined for non-ground terms as well. Behavioral productivity plays for coinductive specifications a role which is dual to that played by sufficient completeness for inductive specifications. We show that if a BRS (not necessarily defining streams) is productive for a term t, then it behaviorally well-specifies the object represented by t; moreover, under mild conditions the two notions coincide. We also show that the problem of saying whether a given BRS is behaviorally productive is Π^0_2-complete. Our main practical result in this paper is a criterion that reduces the checking of behavioral productivity of a "coinductive" BRS to the termination of the rewriting relation (in the usual sense) of its underlying TRS. That means that one can use off-the-shelf termination techniques and tools developed and continuously improved by the rewriting community (see, e.g., [12, 6]) to test for behavioral productivity.

All results reported in this paper lead us to the belief that behavioral rewrite systems may be more suitable than term rewrite systems when we want to analyze the behavioral properties of infinite data structures/processes. We summarize the arguments supporting this idea: 1) behavioral productivity is uniformly defined for all BRS's , 2) a TRS can be associated with various BRS's and hence we can capture various definitions for the productivity of TRS's , 3) some anomalies like "productive for t but does not well-specify t" are avoided, and 4) productivity can be defined for a larger class of terms.

Section 2 introduces the notation used in the paper and recalls the definitions of stream productivity and of behavioral specifications. Section 3 introduces behavioral rewrite systems and behavioral productivity. Section 4 discusses how behavioral productivity captures stream productivity as a special instance. Two main properties of the behavioral productivity are studied in the next two sections: Section 5 shows that the termination in the standard sense of the term

to only be streams, or requiring orthogonality of the TRS, or both). We drop all those limitations here and focus on the essence of the concept.

rewriting relation of a coinductive BRS yields behavioral productivity; Section 6 shows that behavioral productivity implies behavioral well-definedness and that, under some reasonable conditions, the two notions coincide. The hardness of of the behavioral productivity problem is studied in Section 8.

2 Background, Preliminary Notions, and Notations

A *many-sorted signature* (S, Σ), or just *signature* Σ, is a set of *sorts* S together with a set Σ of *operations* $\sigma : s_1 \times \cdots \times s_n \to s$, where $s_1, \ldots, s_n, s \in S$. We let $T_\Sigma(X)$ denote the set of *Σ-terms* built with operation symbols in Σ and with variables in the S-indexed set X. A *Σ-context for sort $s \in S$* is a Σ-term $C \in T_\Sigma(X \cup \{* : s\})$ having precisely one occurrence of the special variable $*$ of sort s; to emphasize that C is such a context we may write $C[*{:}s]$, and if t is a term of sort s then we let $C[t]$ denote the term obtained replacing $*$ by t in C.

Fix a set $\mathcal{X}$ of S-sorted variables. A *Σ-rewrite rule* is a triple $(\forall X)\, l \to r$, where $X \subseteq \mathcal{X}$ is an S-indexed set of *variables* and $l, r \in T_\Sigma(X)$ such that l is not a variable and each variable in r also occurs in l. We often simply write $l \to r$ for a rewrite rule and then X is the set of the variables occurring in l. A *term rewriting system (TRS)* is a pair (Σ, R), where Σ is a many-sorted signature and R is a set of Σ-rewrite rules. The *rewrite relation* $\to_R$ is defined as usual: $t \to_R t'$ for $t, t' \in T_\Sigma(\mathcal{X})$ iff there exists a Σ-context C, a rule $(\forall X)\, l \to r$, and a substitution $\theta : X \to T_\Sigma(\mathcal{X})$ such that $t = C[\theta(l)]$ and $t' = C[\theta(r)]$. We let $\overset{*}{\to}_R$ denote the reflexive and transitive closure of $\to_R$, and $\leftarrow_R$ the inverse of $\to_R$.

A *Σ-equation* is a triple $(\forall X)\, t = t'$, where X has the same meaning as that for rewrite rules and $t, t' \in T_\Sigma(X)$. A *many-sorted equational specification* is a pair (Σ, E), where Σ is a many-sorted signature and E a set of Σ- equations.

2.1 Stream Productivity

Although there are attempts to define productivity more generally, e.g. [10], so far productivity was mainly used for streams (or infinite lists), with lazy (infinite) rewriting [5, 24, 7–10, 26, 28, 29]. Here we remind the reader this particular but conventional notion of productivity. To clearly emphasize its limited scope to stream rewrite systems and to distinguish it from our more general notion of behavioral productivity, we will call it *stream productivity* from here on. Also, to avoid the difficult task of unifying the various definitions of streams and productivity in the papers listed above, we adopt the least restricted definition that we were able to find in the literature, which in our view best captures the intuition underlying the concept originally proposed by Dijskstra in [5]. This is the definition proposed in [10], but without the orthogonally requirement. Orthogonality ensures unique normal forms, so well-definedness, but that seems to be unnecessary for productivity. In our view, productivity is the capability of producing an element of the stream on any position, not necessarily of producing a unique such element. Adding the orthogonality restriction brings no technical

difficulty, but since some of the papers above do not require it, we find it more appropriate to keep our notions and results as unrestricted as possible.

A *stream-TRS* [10] is a TRS (Σ, R) having a special sort *Data* for stream elements, a sort *Stream* for streams, and an "implicit" operation $\_:\_ : \mathit{Data} \times \mathit{Stream} \to \mathit{Stream}$ that allows to regard a stream as its "head" element followed by its "tail" stream. A stream-TRS may contain operations together with rewrite rules defining the data and may contain operations together with rewrite rules defining the streams of interest and desired operations on them. For instance, the constant stream $\mathit{zeros} := 0 : 0 : 0 : \ldots$ containing only 0's, the sub-stream consisting only of the elements on odd positions of a stream, and the stream obtained by zipping two streams can be defined by the following rewrite rules:

$$\mathit{zeros} \to 0 : \mathit{zeros} \qquad \mathit{odd}(B_1 : B_2 : S) \to B_1 : \mathit{odd}(S) \qquad \mathit{zip}(B : S, S') \to B : \mathit{zip}(S', S)$$

where B, B_1, B_2 are variables of sort *Data* and S, S' are variables of sort *Stream*. The above TRS is non-terminating, so termination is not the right concept for stream-TRS's. Stream productivity aims at capturing the notion of *unlimited progress*. Informally, a stream is productive iff it can be continuously evaluated (or rewritten), element by element. Formally, a stream-TRS (Σ, R) is ***stream productive*** [10] for the stream (ground term) s iff $\mathit{Prod}_R(s) = \infty$, where the *stream production function* Prod_R is defined by $\mathit{Prod}_R(s) = \sup\{n \mid s \xrightarrow{*}_R d_1 : d_2 : \ldots : d_n : t\}$, where d_i are data terms. In practice, to avoid non-termination, stream productivity is typically used in combination with lazy rewriting.

2.2 Behavioral Signatures, Experiments and Behavioral Equivalence

Here we recall several folklore behavioral concepts, following for uniformity the notation and approach in [22] but without claiming any novelty or ownership. These concepts have been introduced under various names and notations in earlier works by fathers of behavioral specification, such as Sannella, Tarlecki, Wirsing, Reichel, Goguen, Futatsugi, Bidoit, Hennicker, to only mention a few.

A *behavioral signature* is a pair (Σ, Δ), where Σ is a signature and Δ is a set of Σ-contexts, which we call *derivatives*. If $\delta[*{:}h] \in \Delta$ then the sort h is called a *hidden sort*. Let $H \subseteq S$ be the set of all hidden sorts of (Σ, Δ). The remaining sorts in $V = S - H$ are called *data, or visible sorts*. A *data operation* is an operation in Σ taking and returning only visible sorts; we let $\Sigma{\restriction_V} \subseteq \Sigma$ denote the sub-signature of data sorts and operations. A Σ-sentence/-equation is called a *data sentence/equation* iff it is a $\Sigma{\restriction_V}$-sentence/-equation. A Σ-equation $(\forall X)\, t = u$ is called a *hidden equation* when the sort of t, u is hidden. A *behavioral (equational) specification* $\mathcal{B}$ is a pair $((\Sigma, \Delta), E)$, where (Σ, Δ) is a behavioral signature and E is a set of Σ-equations. A Δ-*experiment* is a Δ-context (i.e., one formed only with contexts in Δ) of visible result sort. If Δ is clear, we may write experiment for Δ-experiment and context for Δ-context.

In the case of streams, the most straightforward choice for derivatives is $\Delta = \{\mathit{hd}[*{:}\mathit{Stream}],\ \mathit{tl}[*{:}\mathit{Stream}]\}$, which is the one we will consider for our stream examples in the rest of the paper. However, the following are also possible choices for derivatives, as they allow to reach any element of a stream:

$\{hd[*{:}Stream],\ hd(tl[*{:}Stream]),\ tl(tl[*{:}Stream])\}$,
$\{hd[*{:}Stream],\ odd[*{:}Stream],\ even[*{:}Stream]\}$,
$\{hd[*{:}Stream],\ zip(tl([*{:}Stream]), S)\}$, $\{hd(tl^n[*{:}Stream]) \mid n \in Nat\}$, etc.
Many examples of derivative sets are discussed in [19], there called *cobases*.

Given a behavioral signature (Σ, Δ), the Δ-experiments allow us to "observe" hidden terms, and thus state and prove *behavioral properties*. A common behavioral property is the *behavioral equivalence*, stating that two hidden terms are behaviorally equivalent iff they are indistinguishable under experiments: $\mathcal{B}$ *behaviorally satisfies* hidden equation e, written $\mathcal{B} \Vdash e$, iff $\mathcal{B} \vdash C[e]$ for each experiment C, where $C[t = u]$ is $C[t] = C[u]$. Another behavioral property is the *behavioral productivity*, which is the main concept introduced and investigated in this paper, stating that a hidden term has "producible" behaviors, that is, it rewrites to some data element under each experiment.

3 Behavioral Rewrite Systems and Productivity

In this section we introduce our main notions in this paper and discuss them by means of examples. Our first notion, that of a *behavioral rewrite system*, has the same relationship to behavioral equational specifications as rewrite systems have to equational specifications: they orient the equations into rewrite rules.

Definition 1. *A* **behavioral rewrite system (BRS)** $\mathcal{B}$ *is a pair* $((\Sigma, \Delta), R)$*, where* (Σ, Δ) *is a behavioral signature and* (Σ, R) *is a TRS.* $\mathcal{B}$ **terminates** *iff* (Σ, R) *terminates as a TRS, and is* **coinductive** *iff* $\delta[f(\overline{x})]$ *is* not *a normal form for any* $f : \overline{s} \to h$ *in* $\Sigma - \Delta$ *and* $\delta[*{:}h]$ *in* Δ *(*$\overline{x}$ *are variables of sort* $\overline{s}$*).*

Our notion of coinductive rewrite system is reminiscent of earlier beahavioral concepts, such as "observer completeness" in [1] and "cobasis" in [21]. It is in fact dual to the folklore notion of "inductive rewrite system", that is, one where there is a subset of operations called "constructors" such that $f(\gamma(\overline{x}))$ is rewritable (typically the left-hand side term of some rewrite rule) for any non-constructor (or defined) operator f and any constructor γ; such rules guarantee that, in the presence of termination, each non-constructor operation is fully defined in terms of constructors. Dually, the fact that the terms $\delta[f(\overline{x})]$ in a coinductive rewrite system are rewritable will guarantee that, in the presence of termination, each non-derivative operation can be fully observed (Theorem 3). We next introduce our general notion of behavioral productivity, inspired from the more particular but insightful notion of stream productivity [5, 10, 24]:

Definition 2. *BRS* $\mathcal{B} = ((\Sigma, \Delta), R)$ *is* **productive for a hidden term** t *iff for each* Δ*-experiment* C *there is some* $(\Sigma\restriction_V \cup \Delta)$*-term* d *such that* $C[t] \xrightarrow{*}_R d$*.* $\mathcal{B}$ *is* **productive** *iff it is productive for any hidden term, and it is* **ground productive** *iff it is productive for any ground hidden term.*

Therefore, productivity for a hidden term t means that a result of applying any experiment C on t can be eventually "produced", or in other words,

since the experiments can typically be easily enumerated, that a behavior of t can be incrementally approximated without getting stuck on any particular experiment. Following the duality induction/coinduction above, note that productivity plays the dual role of sufficient completeness [15]. Indeed, sufficient completeness implies that a term $u[x]$ (for simplicity, suppose that u has only one variable, x) can be shown equal to a constructor ground term under any substitution of its variable x by a constructor ground term c, that is, all the non-constructor operations in $u[c]$ can be eventually eliminated; dually, productivity implies that a term t can be shown equal to a derivative term under any derivation (or experiment) C of it, that is, all the non-derivative operations in $C[t]$ can be eventually eliminated. This duality between productivity and sufficient completeness is technically irrelevant in this paper, therefore we do not bother to formalize it, but we find it interesting and thus worthwhile noting.

Note that the "result" $(\Sigma\restriction_V \cup \Delta)$-term d in Definition 2 can use the operations in Δ only as Δ-experiments applied to variables of hidden sort, because no other non-data operations are allowed in d and because the sort of d is visible. In particular, if t has no hidden variables then d is a $\Sigma\restriction_V$-term. To avoid confusion with "stream productivity", we take the freedom to tacitly call our productivity for BRS's *behavioral productivity* whenever we feel that clarifies the presentation.

In the sequel we illustrate the notions above on several examples.

Example 1. (*Streams*) A behavioral rewrite system of bit streams (or infinite lists) may include a sort *Bit* with two constants 0 and 1, a sort *Stream* for bit streams with operations $hd : Stream \to Bit$ (head) and $tl : Stream \to Stream$ (tail), and as many stream operations and defining rewriting rules as desired. For instance, the constant stream *zeros*, the sub-stream on odd positions *odd*, and the stream merging *zip* can be behaviorally defined as:

$$\begin{array}{lll} hd(zeros) \to 0 & hd(odd(S)) \to hd(S) & hd(zip(S,S')) \to hd(S) \\ tl(zeros) \to zeros & tl(odd(S)) \to odd(tl(tl(S))) & tl(zip(S,S')) \to zip(S', tl(S)) \end{array}$$

One can also define a stream operation returning the sub-stream on even positions as $even(S) \to odd(tl(S))$. The set Δ of derivatives consists of the two contexts $hd[*{:}Stream]$ and $tl[*{:}Stream]$. Thus, *Stream* is a hidden sort and *Bit* is visible. The stream experiments are contexts of the form $hd(tl^i[*{:}Stream])$, where $i \geq 0$. It is not hard to check that this behavioral rewrite system for streams is terminating, coinductive, and behaviorally productive. For instance, *zeros* is behaviorally productive because $hd(tl^i(zeros)) = 0$ for any $i > 0$. Also, $odd(S)$ is productive because $hd(tl^i(odd(S))) \xrightarrow{*} hd(tl^{2i}(S))$ for any $i > 0$.

Example 2. (*Non-Deterministic Streams*) Consider now a bit stream *random*, which can produce any sequence of 0 and 1 bits. It can be defined as follows:

$$hd(random) \to 0 \qquad hd(random) \to 1 \qquad tl(random) \to random$$

This BRS terminates and is both coinductive and productive. The stream *random* is *not* productive according to existing formal definitions of stream productivity [10], as those require, in our view unjustified, that the stream elements are not only producible, but also *unique*. We believe that productivity and unique normal forms of experiments are orthogonal issues, so we do not mix them.

Example 3. (*Non-Terminating Streams*) Let us extend the stream BRS in Example 1 with a new constant *ones*, an operation $\_:\_ : Bit \times Stream \to Stream$, and the rewrite rules:

$$ones \to 1 : ones \qquad hd(B : S) = B \qquad tl(B : S) = S$$

Obviously, the resulting BRS is not terminating. It is coinductive, however, because $hd(ones) \to hd(1 : ones)$ and $tl(ones) \to tl(1 : ones)$. It is also productive, because there is some rewriting sequence $hd(tl^i(ones)) \xrightarrow{*} 1$ for each $i \geq 0$; even though a lazy strategy may be needed in order to generate such rewriting sequences, it is important to note that for each experiment on *ones* there is some finite rewriting sequence computing it, so the BRS is productive on *ones*.

Example 4. (*Non-Coinductive Streams*) Let us replace the two defining rules of *zeros* in the stream BRS in Example 1 with the following three rules:

$$hd(zeros) \to 0 \qquad hd(tl(zeros)) = 0 \qquad tl(tl(zeros)) = zeros$$

Obviously, the resulting BRS is not coinductive, because $tl(zeros)$ is now a normal form. It remains both terminating and productive, though. However, if we instead choose Δ to be $\{hd[*{:}Stream],\ hd(tl[*{:}Stream]),\ tl(tl[*{:}Stream])\}$, then the stream BRS becomes also coinductive.

Example 5. (*Non-Productive Streams*) All the example BRS's above were productive. One can also have non-productive BRS's ; however, since Theorem 3 tells us that coinductive and terminating BRS's are also productive, it must be the case that any non-productive BRS must either not be coinductive or not terminate. We show an example of each. An example of non-coinductive non-productive BRS can be obtained from the BRS in Example 1 by dropping any of its rules. The other case is trickier. Let us extend the stream BRS in Example 1 with a stream a (constant of sort *Stream*) defined with the rules $hd(a) \to 0$ and $tl(a) \to odd(a)$. Then the resulting BRS is not productive for a, because there is no way to "evaluate" $hd(tl^2(a))$: indeed, $hd(tl^2(a)) \to hd(tl(odd(a))) \to hd(odd(tl^2(a))) \to hd(tl^2(a)) \to \ldots$. This rewrite sequence shows that this stream BRS is also non-terminating. Note, however, that it is coinductive.

Let us next also discuss some non-stream examples of BRS's .

Example 6. (*Infinite Binary Trees*) A BRS defining infinite binary trees over bits consists of a definition of bits (similar to that of streams), a sort *Tree* for infinite binary trees together with the operations $root : Tree \to Bit$ (the root of the tree), $left : Tree \to Tree$ (the left subtree), $right : Tree \to Tree$ (the right subtree), and other operations over trees and their defining rewriting rules. Here are several examples of such operations inspired from [25]:

$root(ones) \to 1$	$root(\neg T) \to \overline{root(T)}$	$root(thue) \to 0$
$left(ones) \to ones$	$left(\neg T) \to \neg left(T)$	$left(thue) \to thue$
$right(ones) \to ones$	$right(\neg T) \to \neg right(T)$	$right(thue) \to thue + ones$

$$root(T1 + T2) \to root(T1) \oplus root(T2)$$
$$left(T1 + T2) \to left(T1) + left(T2)$$
$$right(T1 + T2) \to right(T1) + right(T2)$$

where the addition ($\oplus$) and the negation ($\bar{\cdot}$) over bits are defined as usual: $0 \oplus B \to B$, $1 \oplus 1 \to 0$, $\overline{0} \to 1$, $\overline{1} \to 0$. The set Δ of derivatives consists of three contexts: $root(*{:}Tree)$, $left(*{:}Tree)$, and $right(*{:}Tree)$. So, the sort *Tree* is hidden and the sort *Bit* is visible. This behavioral rewrite system for infinite binary trees is terminating, coinductive and productive. Moreover, one can show that, for example, the infinite binary trees $thue + ones$ and $\neg thue$ are indistinguishable under Δ-experiments; indeed, the behavioral prover CIRC [16] can prove them behaviorally equivalent, but this is beyond our scope in this paper.

Example 7. (*Processes*) The BRS defining the processes presented in Section 1 consists of a visible sort *Int* for integers, visible operations over integers and their defining rewriting rules, a hidden sort *State* for the states, two hidden constants a, b of sort *State* describing the states of the first process, an operation (generalized hidden constant) $s : Int \to State$ for the states of the second process, and two operations $out : State \to Int$ and $next : State \to State$, which together with their rewrite rules describe the behaviors of the two processes. The set of derivatives Δ consists of two contexts: $out(*{:}State)$ and $next(*{:}State)$. It is easy to see that this BRS of processes is terminating, coinductive and productive.

Example 8. (*Non-Deterministic, or Non-Confluent Processes*) Here is an example of BRS which is productive but is not confluent. Add to the BRS in Example 7 one more hidden constant of sort *State*, say c, together with the following rules:

$$next(a) \to c \qquad next(c) \to a \qquad out(c) \to 1$$

The resulting BRS is productive (for each of a, b, c) but is not confluent (the critical pair $b \leftarrow next(a) \to c$ is not join-able). However, $out(next^i(b)) \xrightarrow{*}_R v_i$ and $out(next^i(c)) \xrightarrow{*}_R v_i$ for some $v_i \in \{0, 1\}$, so b and c are indistinguishable under experiments. Hence each experiment on a is uniquely determined, in spite of the lack of confluence of this BRS.

The various examples above showed that neither termination, nor coinductivity, nor confluence is a requirement for productivity. As shown in Section 5, termination and coinductivity imply productivity; confluence, however, appears to play no role for productivity.

4 Behavioral Productivity Generalizes Stream Productivity

In this section we discuss the relationship between productivity in the usual sense of stream-TRS definitions and behavioral productivity of stream BRS definitions, essentially showing that nothing is lost wrt productivity when using the latter. On the contrary, the BRS approach to define streams has the benefit that one can use our termination-based technique in Theorem 3 to prove stream productivity.

We start by defining a transformation, given in Definition 3, which shows the immediate correspondence between stream productivity and behavioral productivity, namely that the former falls as a special case of the latter for particular behavioral rewrite systems, namely ones of streams.

Definition 3. *Let $\mathcal{R} = (\Sigma, R)$ be a stream-TRS (see Section 2.1). We let $\mathcal{B}_0(\mathcal{R})$ be the BRS $((\Sigma \cup \{hd, tl\}, \{hd, tl\}), R \cup \{hd(B : S) \to B,\ tl(B : S) \to S\})$.*

To avoid interfering with the head/tail operations that may already be defined and used in the original stream-TRS $\mathcal{R}$, we assume that the hd/tl added to the BRS $\mathcal{B}_0(\mathcal{R})$ are fresh. To achieve this, one may need to rename the potentially homonymous operations in $\mathcal{R}$.

Theorem 1. *Let $\mathcal{R}$ be a stream-TRS and let s be a ground stream term. Then $s \xrightarrow{*}_{\mathcal{R}} d_1 : d_2 : \ldots : d_n : t$ iff $hd(s) \xrightarrow{*}_{\mathcal{B}_0(\mathcal{R})} d_1$, $\ldots$, $hd(tl^{n-1}(s)) \xrightarrow{*}_{\mathcal{B}_0(\mathcal{R})} d_n$, and $tl^n(s) \xrightarrow{*}_{\mathcal{B}_0(\mathcal{R})} t$. Therefore, $\mathcal{R}$ is productive for s if and only if $\mathcal{B}_0(\mathcal{R})$ is behaviorally productive for s.*

Proof. Straightforward by induction on n, noting that $s \xrightarrow{*}_{\mathcal{R}} h : t$ if and only if $hd(s) \xrightarrow{*}_{\mathcal{B}_0(\mathcal{R})} h$ and $tl(s) \xrightarrow{*}_{\mathcal{B}_0(\mathcal{R})} t$. □

The transformation $\mathcal{R} \mapsto \mathcal{B}_0(\mathcal{R})$ above is so trivial that it should not be surprising that $\mathcal{B}_0(\mathcal{R})$, in spite of capturing stream productivity as an instance of the more general concept of behavioral productivity, does not add much behavioral value; in particular, it is not coinductive and, if the original stream-TRS $\mathcal{R}$ does not terminate, $\mathcal{B}_0(\mathcal{R})$ does not terminate either. Therefore, our termination-based technique in Theorem 3 cannot be applied to prove stream productivity if we follow this simplistic approach.

We next give a converse transformation, from stream-BRS's to stream-TRS's, also trivial and also productivity preserving:

Definition 4. *Let $\mathcal{B}$ be a stream-BRS. We let $\mathcal{R}_0(\mathcal{B})$ be the stream-TRS obtained from $\mathcal{B}$ by adding the lazy constructor*[4] *$\_:\_$ and the rule $S \to hd(S) : tl(S)$.*

Like in the previous transformation, to avoid interfering with the stream construct that may already be defined and used in the original stream-BRS $\mathcal{B}$, we assume that the $\_ : \_$ added to the TRS $\mathcal{R}_0(\mathcal{B})$ is fresh. If one does not like the fact that the rule added to $\mathcal{R}_0(\mathcal{B})$ has a variable (S, of sort *Stream*) as left hand side, then one can instantiate the rule above for the stream operations defined in $\mathcal{B}$. This rule, however, is not problematic for lazy rewriting, because it is not applied indefinitely under the hd/tl operations that it generates.

Theorem 2. *Let $\mathcal{B}$ be a stream-BRS. Then $s \xrightarrow{*}_{\mathcal{R}_0(\mathcal{B})} d_1 : d_2 : \ldots : d_n : t$ iff $hd(s) \xrightarrow{*}_{\mathcal{B}} d_1$, $\ldots$, $hd(tl^{n-1}(s)) \xrightarrow{*}_{\mathcal{B}} d_n$, and $tl^n(s) \xrightarrow{*}_{\mathcal{B}} t$. Therefore, $\mathcal{B}$ is behaviorally productive for a ground stream term s if and only if $\mathcal{R}_0(\mathcal{B})$ is stream productive for s.*

[4] We refer the reader to [10] for precise stream-TRS definitions and terminology.

Proof. Straightforward again, by induction on n, noting that $s \xrightarrow{*}_{\mathcal{R}_0(\mathcal{B})} h : t$ if and only if $hd(s) \xrightarrow{*}_{\mathcal{B}} h$ and $tl(s) \xrightarrow{*}_{\mathcal{B}} t$. □

As explained in Section 2.1, existing variants of stream-TRS and stream productivity definitions are more restricted than ours. If one wants to adapt our results above to those variants, then one needs to add similar restrictions to the corresponding stream-BRSes. For example, if one strongly believes that or absolutely needs that stream-TRSes must be orthogonal (in order to ensure unique normal forms), as it is the case in several stream-TRS variants, then one can require that same orthogonality restriction on the stream-BRS.

In addition to its theoretical significance, the transformation above may also have practical value. Supposing that one prefers to use lazy rewriting to define streams, one may admittedly be reluctant to use our "behavioral style" because, even if one proves productivity using behavioral techniques (e.g., Theorem 3), one still cannot directly use the BRS in one's lazy rewriting framework. The transformation $\mathcal{B} \mapsto \mathcal{R}_0(\mathcal{B})$ above says that all one needs to do to take advantage of *both* our behavioral approach and one's lazy rewriting framework is to define one's streams as a BRS $\mathcal{B}$, prove it behaviorally productive, then transform it into the stream-TRS $\mathcal{R}_0(\mathcal{B})$ by adding the lazy construct and rule as in Definition 4, and finally use it in one's lazy rewrite framework knowing that it is productive.

While we agree that the stream-TRS definitional style is compact, elegant, and the required lazy rewriting strategy to evaluate them is well supported by several programming languages, we conclude this section by warning the reader that the more conventional stream-TRS style may sometimes, rather unexpectedly, lead to situations of what one may call *accidental non-productivity*. Consider, for example, the following stream-TRS from [29]:

$$zeros \to 0 : zeros \qquad f(x : s) \to g(f(s)) \qquad g(x : s) \to zeros$$

This stream-TRS follows the lazy definitional style and it is easy to see that $f(zeros)$ can only be the stream $zeros$, which is productive. However, unfortunately, $f(zeros)$ is *not productive in the original sense*, because $f(zeros) \to f(0{:}zeros) \to g(f(zeros)) \to^* g^2(f(zeros)) \to \ldots$ and there is no way to produce a first 0 element. The problem here is that the lazy stream construct in the definition of g prevents the rule from matching, because $f(zeros)$ cannot be split in a head and tail. Such a situation would have not appeared if one followed a behavioral rewriting style, aiming at defining a terminating and coinductive BRS like the following, which is immediately productive by Theorem 3:

$$\begin{array}{lll} hd(zeros) \to 0 & hd(f(s)) \to hd(g(f(tl(s)))) & hd(g(s)) \to 0 \\ tl(zeros) \to zeros & tl(f(s)) \to tl(g(f(tl(s)))) & tl(g(s)) \to zeros \end{array}$$

One could argue that accidental non-productive situations like above are not an artifact of lazy TRS rewriting as we are implying, but instead desirable. Even if one agrees with that, we think that one may still want to eliminate accidental non-productivity whenever possible, preferably even through automatic equivalent TRS-transformations. We believe that the behavioral rewriting approach

proposed in this paper could help with this aspect, but our results in this direction are preliminary and so are only informally discussed in Section 7.

The idea is to devise more involved (stream-semantics preserving) transformations $\mathcal{R} \mapsto \mathcal{B}_i(\mathcal{R})$ (for different i indexes - $i = 0$ is the most basic, starting point transformation) from stream-TRS's into BRS's , more precisely ones where $\mathcal{B}_i(\mathcal{R})$ may be behaviorally productive also in situations where $\mathcal{R}$ is not necessarily productive. Then one can use Theorem 3 and termination tools to check the behavioral productivity of $\mathcal{B}_i(\mathcal{R})$, and finally report back the equivalent stream-TRS $\mathcal{R}_0(\mathcal{B}_i(\mathcal{R}))$ which is now stream productive. One can also devise different transformations $\mathcal{B} \mapsto \mathcal{R}_i(\mathcal{B})$ that make the resulting stream-TRS follow the more common style that one uses when defining stream-TRS's (e.g., replacing pairs of behavioral rules $hd(l) \to h$ and $tl(l) \to t$ by lazy rules $l \to h : t$,etc.). Such transformations are beyond our scope in this paper.

5 Termination and Coinductivity Imply Productivity

Productivity is an inherently difficult problem (see Section 8) and there are no tools available that can check productivity in general. It is therefore important to reduce the problem of checking productivity to other problems with better tool support. In this section we give a practical criterion that reduces behavioral productivity to termination in the standard sense, so that one can use off-the-shelf termination tools to check productivity.

Theorem 3. *Let $\mathcal{B} = ((\Sigma, \Delta), R)$ be a BRS such that $\Sigma - (\Sigma\restriction_V \cup \Delta)$ contains only operations of hidden result sort. Then $\mathcal{B}$ terminating and coinductive implies $\mathcal{B}$ productive.*

Proof. Let t be a hidden term and let C be a Δ-experiment for t. If t is a $(\Sigma\restriction_V \cup \Delta)$-term then we are done. If t contains some operation in $\Sigma - (\Sigma\restriction_V \cup \Delta)$, which by hypothesis must be of hidden result sort, then since the result sort of $C[t]$ is visible it must be the case that $C[t]$ contains a subterm of the form $\delta[f(\overline{u})]$ for some $f : \overline{s} \to h$ in $(\Sigma\restriction_V \cup \Delta)$, some $\delta[*{:}h]$ in Δ, and some tuple term $\overline{u}$ of tuple sort $\overline{s}$. Hence, by coinductivity, $C[t]$ cannot be in normal form, so it can be rewritten to some other term of visible sort. If the resulting term contains any operation in $\Sigma - (\Sigma\restriction_V \cup \Delta)$, then we can apply the same arguments above and reduce it to another term of visible sort. Since R terminates, eventually the resulting term will contain no operations in $\Sigma - (\Sigma\restriction_V \cup \Delta)$, which proves that $\mathcal{B}$ is productive for t. □

The condition "$\Sigma - (\Sigma\restriction_V \cup \Delta)$ contains only operations of hidden result sort" in Theorem 3 is, unfortunately, necessary. Indeed, consider a stream BRS defining a stream a with rules $hd(a) \to vis(a)$ and $tl(a) \to a$, where $vis : Stream \to Bit$ is some artificially included operation of visible result in $\Sigma - (\Sigma\restriction_V \cup \Delta)$. This BRS obviously terminates and is coinductive, but it is not productive because $hd(a)$ cannot be evaluated.

Fortunately, both this condition and the coinductivity of a BRS are trivial syntactic checks. The only non-trivial hypothesis of Theorem 3 is the termination, but since all that is required is standard termination of a TRS, this theorem allows for the use of off-the-shelf termination tools (e.,g., [12, 6]) for checking productivity of behavioral rewrite systems. Note that this would not be possible if we allowed rules of the form $zeros \to 0 : zeros$; the point here is that such non-terminating rules are unnecessary, because they can be replaced with their coinductive variants and then productivity can be checked using conventional termination techniques and tools.

Theorem 3 is reminiscent of a recent result by Zantema [29] which states that, for some restricted variants of stream rewrite systems, termination implies well-definedness; however, well-definedness of streams is formalized as a rather intricate, model-theoretical concept in [29], while our formalization is based on simple proof-theoretical/operational arguments.

Finally, Theorem 3 may find applications in deciding that certain classes of stream TRS-es are productive, provided that one can decide termination for the corresponding BRS-es; it would be interesting to see whether one can find this way an alternative proof for the decidability of productivity result by Endrullis et al. [10] for the particular class of stream constant specifications.

6 Behavioral Productivity Means Well-Specified Behavior

Behavioral productivity suggests, intuitively, well-specified behavior, that is, behavior which is not under-specified. However, it is not immediate what it means for a term to be well-specified in our general behavioral context. To avoid the complications and diversity that come with particular choices of models over behavioral signatures (see [19] for several of them), we prefer to take an operational, or proof-theoretical approach here: we say that a term t is well-specified iff it is indistinguishable by means of experiments (and rewriting) from a clone t' of it using cloned operations defined the same way as the original operations. This notion of behavioral well-specifiedness is somehow dual to constructor-based well-definedness. In this section we give an alternative but equivalent way to understand productivity by means of well-specified behavior.

Definition 5. *Given BRS $\mathcal{B}$, let $\mathcal{B} \Vdash\!\!\mapsto t = t'$ denote the* ***behavioral join equivalence*** *of $\mathcal{B}$: for any experiment C there is some term u with $C[t] \to^* u \leftarrow^* C[t']$.*

Consider the stream term *zeros* in Example 1. BRS `STREAM` behaviorally well-specifies *zeros* because one can show that $\texttt{STREAM} \Vdash\!\!\mapsto zeros = zeros'$ for any other stream $zeros'$ specified the same way as *zeros* (i.e., `STREAM` includes $hd(zeros') \to 0$ and $tl(zeros') \to zeros'$). Similarly, `STREAM` well-specifies the stream operation *odd* in Example 1, because one can behaviorally prove $\texttt{STREAM} \Vdash\!\!\mapsto (\forall S)\ odd(S) = odd'(S)$ for any operation odd' defined the same way as *odd* (i.e., $hd(odd'(S)) \to hd(S)$ and $tl(odd'(S)) \to odd'(tl^2(S))$). The CIRC tool [16] can prove these properties automatically by circular coinduction. However, `STREAM` does not well-specifies a constant stream a specified without any rule, because there is no way

to show that $a = a'$ for another constant a'. Also, it does not well-specifies a constant stream a specified with rules $hd(a) \to 0$ and $tl(a) \to odd(a)$, since $hd(tl^2(a)) \to hd(tl(odd(a))) \to hd(odd(tl^2(a))) \to hd(tl^2(a)) \to \ldots$ and similarly for a clone a' of a, with no chance to show that $hd(tl^2(a)) = hd(tl^2(a'))$.

Interestingly and perhaps intriguingly at first, the *random* stream in Example 2 defined as $hd(random) \to 0$, $hd(random) \to 1$, and $tl(random) \to random$ is in fact well-specified. It is non-deterministic, but that is intended in its specification; its non-determinism is not a consequence of under- or lack of specification.

As it is usually the case with "equality" relations defined in terms of joint rewriting, one should be aware of the fact that non-confluent rewriting might lead to equalities which are not semantically valid. For example in our case here, since *random* can rewrite its bits to either 0 or 1, it is behaviorally join equivalent to any other stream, in particular to *zeros*. To avoid such phenomena, we advice the reader to only use the notion of behavioral join equivalence in combination with term cloning, which is described below.

Definition 6. *Given behavioral specification $\mathcal{B} = ((\Sigma, \Delta), R)$, let $\mathcal{B}'$ extend $\mathcal{B}$ by adding to Σ a copy σ' of each $\sigma \in \Sigma - (\Sigma\restriction_V \cup \Delta)$ and to R a copy $l' \to r'$ of each $l \to r \in R$, where l' (resp. r') is obtained by replacing each $\sigma \in \Sigma - (\Sigma\restriction_V \cup \Delta)$ in l (resp. r) with σ'. $\mathcal{B}$* **(behaviorally) well-specifies term** *t iff $\mathcal{B}' \Vdash\!\mapsto t = t'$, where t' is obtained by replacing each $\sigma \in \Sigma - (\Sigma\restriction_V \cup \Delta)$ in t with σ'.*

Hence, $\mathcal{B}'$ "clones" each operation which is not a data operation or a derivative, as well as all the rules referring to those operations. Behavioral well-specifiedness of a term t states that t is behaviorally equivalent to its corresponding clone t', so from a behavioral point of view, t can have only one meaning.

Theorem 4. *$\mathcal{B} = ((\Sigma, \Delta), R)$ productive for t implies $\mathcal{B}$ well-defines t. Conversely, if the rules in R "do not introduce" operations in $\Sigma - (\Sigma\restriction_V \cup \Delta)$, that is, if for each $(l \to r) \in R$ it is the case that if l does not contain operations in $\Sigma - (\Sigma\restriction_V \cup \Delta)$ then r does not contain operations in $\Sigma - (\Sigma\restriction_V \cup \Delta)$ either, then $\mathcal{B}$ well-defines t implies $\mathcal{B}$ productive for t.*

Proof. Suppose that $\mathcal{B} = ((\Sigma, \Delta), R)$ is productive on term t and let $\mathcal{B}' = ((\Sigma', \Delta), R')$ and t' be the clone extension of $\mathcal{B}$ and the clone of t, respectively, as explained in Definition 6. Let C be a Δ-experiment for t. Since $\mathcal{B}$ is productive, there is some $(\Sigma\restriction_V \cup \Delta)$-term d such that $C[t] \xrightarrow{*}_R d$. We get $C[t'] \xrightarrow{*}_{R'} d$ using the copies of the rules used in the above reduction. Hence, $\mathcal{B} \vdash C[t] = C[t']$. Since the experiment C is arbitrary, $\mathcal{B}' \Vdash\!\mapsto t = t'$.

Suppose now that $\mathcal{B}$ well-defines t and let $\mathcal{B}'$ and t' be the clone extension of $\mathcal{B}$ and the clone of t, respectively, as explained in Definition 6. Let C be a Δ-experiment for t. Since $\mathcal{B}$ well-defines t, there is some term u such that $C[t] \to_R^* u$ and $C[t'] \to_R^* u$. Since $C[t] \to_R^* u$ and the rules of R do not introduce operations in $\Sigma - (\Sigma\restriction_V \cup \Delta)$, it follows that u cannot contain any clone operation in $\Sigma' - (\Sigma\restriction_V \cup \Delta)$. For the same reason, since $C[t'] \to_R^* u$, it follows that u cannot contain any operation in $\Sigma - (\Sigma\restriction_V \cup \Delta)$. The only possibility is then that u is a $(\Sigma\restriction_V \cup \Delta)$-term, which proves that $\mathcal{B}$ is productive for t. □

Note that all the productivities in Section 3 follow by Theorem 4.

7 Towards More Pragmatic Transformations

We have the following situation: on the one hand, term rewriting systems are more compact and elegant for specifying infinite data structures or systems; on the other hand, behavioral rewrite systems are more suitable for analyzing the behavioral well-definedness (which is implied by the behavioral productivity). The question is whether we can have the advantages of both approaches. We strongly believe that the answer is yes, provided that we are able to define appropriate mechanisms to safely translate from one approach to the other. In this section we discuss some initial steps towards such mechanisms.

The transformation $\mathcal{R} \mapsto \mathcal{B}_0(\mathcal{R})$ taking a stream-TRS into a BRS (see Definition 3) typically yields neither terminating nor coinductive BRS's , so it is not very practical. However, we have seen in Section 4 that there are non-productive stream-TRS's $\mathcal{R}$ for which we can find behavioral versions $\mathcal{B}(\mathcal{R})$ which are productive. So, we may aim at finding transformations $\mathcal{R} \mapsto \mathcal{B}(\mathcal{R})$ which avoid the accidental non-productivity. A partial positive answer is given by the algorithm described in [29, 28]. That algorithm works fine only on a particular subclass of stream-TRS's and associates a BRS $\mathcal{B}_1(\mathcal{R})$ with a stream-TRS $\mathcal{R}$ such that $\mathcal{R}$ is well-defined (has a unique model) if and only if $\mathcal{B}_1(\mathcal{R})$ is terminating. The conditions on $\mathcal{R}$ ensures the fact $\mathcal{B}_1(\mathcal{R})$ is coinductive and, by Theorem 3, we obtain that if $\mathcal{B}_1(\mathcal{R})$ is terminating then it is productive. We may further assume that we have a transformation $\mathcal{R}_1$ associating a stream-TRS $\mathcal{R}_1(\mathcal{B})$ with each coinductive and terminating stream-BRS $\mathcal{B}$ such that the productivity is preserved. Then the composition of the two transformations $\mathcal{R} \mapsto \mathcal{R}_1(\mathcal{B}_1((R)))$ may transform a non-productive stream-TRS into a productive one. For instance, if $\mathcal{R}_1(\mathcal{B})$ includes rules of the form $f(x : s) \to h : t$ with h and t $\mathcal{B}$-normal forms of $hd(f(x : s))$ and respectively $tl(f(x : s))$, then the stream-TRS considered in Section 4 is transformed in

$$zeros \to 0 : zeros \qquad f(x : s) \to 0 : zeros \qquad g(x : s) \to 0 : zeros$$

which is productive (the anomalies are away).

Not only the streams can be specified as TRS's with infinite rewriting. For instance, the infinite binary trees defined in Example 6 are specified by the following TRS:

$$\begin{aligned} ones &\to 1/ones, ones\backslash \\ \neg B/T_1, T_2\backslash &\to \overline{B}/\neg T_1, \neg T_2\backslash \\ B/T_1, T_2\backslash + B'/T_1'', T_2'\backslash &\to B \oplus B'/T_1 + T_1', T_2 + T_2'\backslash \\ thue &\to 0/thue,\ thue + ones\backslash \end{aligned}$$

where $\_/\_, \_\backslash : Bit\ Tree\ Tree \to Tree$ is a constructor. Similarly, the two processes defined in Section 1 can be specified by the rewrite rules

$$a \to 0; b \quad b \to 1; a \quad s_{2i} \to 0; s_{2i+1} \quad s_{2i+1} \to 0; s_{2i+2}$$

where $\_; \_ : Int\ State \to State$ is a constructor.

The algorithm defining the transformation $\mathcal{B}_1$ can be adapted, e.g., for trees or for processes. Like for streams, only a subclass of tree-TRS's or process-TRS's can be transformed with such an algorithm; these subclasses can be defined by

adapting the conditions from Definition 1 in [29]. Unfortunately, a transformation which can be applied in the general case may be hard or impossible to define. The main reason is given by the fact that it is hard or impossible to formally state at this level of generality what it means for a BRS to be a "correct behavioral version" of a given TRS.

We suggest the following methodology in order to have both the compactness and the elegance of the TRS definitional style, as well as the behavioral well-definedness/productivity for a given class of specifications:

1. formally define when a BRS is a correct behavioral version (e.g., specifies the same data structure or system) of a given TRS from your class;
2. define a transformation $\mathcal{B}$ which associate a BRS $\mathcal{B}(\mathcal{R})$ with a given TRS $\mathcal{R}$ from your class and prove that $\mathcal{B}(\mathcal{R})$ is a correct version of $\mathcal{R}$;
3. when checking if a given TRS $\mathcal{R}$ is behaviorally productive, show that $\mathcal{B}(\mathcal{R})$ is coinductive and terminating;
4. eventually, define a transformation $\mathcal{R}$ which associate a TRS with each coinductive and terminating TRS in order to have a way to transform non-productive TRS's into productive ones.

8 Behavioral Productivity is a Π^0_2-Complete Problem

Behavioral equivalence is known to be a Π^0_2-complete problem, both for streams [20] and in general [3]. Also, a series of recent results show that many rewriting problems, including termination and stream productivity, are Π^0_2-complete [9, 7, 26, 11]. Here we show that the behavioral productivity problem is no exception.

Π^0_2 is the class, or degree, in the arithmetic hierarchy consisting of predicates $\pi(z)$ of the form $(\forall x)(\exists y)\ r(x, y, z)$, where r is a recursive (or decidable) predicate and x, y, z range over natural numbers (or, equivalently, over recursively enumerable domains). Π^0_2 contains predicates which are strictly harder than recursively enumerable or co-recursively enumerable. A canonical Π^0_2-complete problem is $\textsc{Totality}(M) := (\forall x)(\exists n)\ \textsc{Stop}(x, n, M)$, asking whether computational device (Turing machine, program, rewrite system, etc.) M stops on all its inputs; here $\textsc{Stop}(x, n, M)$ is the recursive predicate saying that machine M stops in at most n steps on input x. The reader is referred to [23] for more details on the arithmetic hierarchy and the class Π^0_2.

To see why, for example, the terminating problem for a rewrite system is Π^0_2-complete [9, 26], consider TRS's computing r.e. functions (see, e.g., Section 3.2 in Terese book [27]) instead of Turing machines and interpret $\textsc{Stop}(x, n, M)$ by "the TRS M finds in at most n steps" all normal forms of the term x; then Totality becomes exactly the terminating problem for TRS's .

Theorem 5. *The behavioral productivity problem is Π^0_2-complete.*

Proof. We first show the membership to the class Π^0_2. Let $\mathcal{B} = ((\Sigma, \Delta), R)$ be a BRS and let t be a hidden term. The predicate $\textsc{Search}(t \xrightarrow{?} u, n, R)$, telling that there is a reduction $t \xrightarrow{*}_R u$ of length at most n, is recursive. The set of

$(\Sigma\restriction_V \cup \Delta)$-terms d is r.e. and therefore the predicate $(\exists d)C[t] \xrightarrow{*}_R d$ is equivalent to $(\exists\langle n, d\rangle)\textsc{Search}(C[t]\xrightarrow{?}d,\, n, R)$. Then the productivity problem is equivalent to $(\forall C)(\exists\langle n, d\rangle)\textsc{Search}(C[t]\xrightarrow{?}d,\, n, R)$.

The Π^0_2-hardness of the behavioral productivity problem over behavioral rewrite systems is proved using the reduction given by the transformation $\mathcal{R} \mapsto \mathcal{B}_0(\mathcal{R})$, defined over stream-TRS's in Section 4: The productivity problem for stream-TRS's is Π^0_2-hard [9] and this problem is reduced to the productivity problem for the stream behavioral specifications by Theorem 1. We can now conclude with the main result. □

9 Conclusion

This paper investigates the role of term rewriting in behavioral reasoning. The notion of term rewriting system is extended to that of behavioral rewrite system, and a proper notion of productivity, called behavioral productivity, is given for the new systems. Various aspects of the new notion are largely exemplified on streams, infinite binary trees and processes. It is shown that behavioral productivity plays a similar role for coinductive specifications to that played by sufficient completeness for inductive specifications. Behavioral productivity generalizes the existing notion of productivity defined over stream rewriting systems. Two main properties of the proposed behavioral approach are proved (under mild conditions): termination yields behavioral productivity, and behavioral productivity is equivalent to behavioral well-specification. The former property allows us to use the existing tools for rewrite termination [12, 6] for checking behavioral productivity. It was also shown that behavioral productivity has the same complexity as many other rewriting-related problems, namely it is Π^0_2-complete.

Even if behavioral productivity is defined for behavioral rewrite systems, it can be extended to term rewrite systems by means of algorithms similar to that described in [29], which associate behavioral versions to term rewrite systems. Finding such algorithms for more general cases than that of streams is one of the future work directions.

Productivity was defined for the first time for streams [5]. Then it was extended for infinite data structures used in functional programming [24]. See, e.g., [8] for a review of the main approaches dealing with productivity. Recently, productivity was intensively studied in the context of term rewriting systems [10, 26]. Behavioral specifications were first introduced in [18]. Then behavioral reasoning was intensively studied in different algebraic/logic frameworks [13, 2, 4, 19, 17, 14]. The behavioral rewrite systems introduced in this paper are an instance of the parametric definition given in [22].

Acknowledgment. The work presented in this paper was supported by the Romanian Contract 161/15.06.2010, SMISCSNR 602-12516 (DAK), and by the USA grants NSF CCF-1218605, NSA H98230-10-C-0294, DARPA HACMS (SRI subcontract) 19-000222, and Rockwell Collins 4504813093.

References

1. M. Bidoit and R. Hennicker. Observer complete definitions are behaviourally coherent. In *OBJ/CAFEOBJ/MAUDE AT FORMAL METHODS '99*, pages 83–94. THETA, 1999.
2. M. Bidoit, R. Hennicker, and A. Kurz. Observational logic, constructor-based logic, and their duality. *Theoretical Computer Science*, 3(298):471–510, 2003.
3. S. Buss and G. Roşu. Incompleteness of behavioral logics. In *Proceeding of CMCS'00*, volume 33 of *ENTCS*, pages 61–79. Elsevier, 2000.
4. R. Diaconescu and K. Futatsugi. *CafeOBJ Report*, volume 6 of *AMAST Series in Computing*. World Scientific, 1998.
5. E. W. Dijkstra. On the productivity of recursive definitions. EWD749, Sept. 1980.
6. F. Durán, S. Lucas, and J. Meseguer. Mtt: The maude termination tool (system description). In *IJCAR*, pages 313–319, 2008.
7. J. Endrullis, J. Geuvers, and H. Zantema. Degrees of undecidability in term rewriting. In R. K. E. Grädel, editor, *Computer Science Logic (23rd international workshop, CSL 2009, 18th annual conference of the EACSL*, volume 5771 of *LNCS*, pages 255–270. Springer, 2009.
8. J. Endrullis, C. Grabmayer, and D. Hendriks. Data-oblivious stream productivity. In *LPAR '08: Proceedings of the 15th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, pages 79–96, Berlin, Heidelberg, 2008. Springer-Verlag.
9. J. Endrullis, C. Grabmayer, and D. Hendriks. Complexity of fractran and productivity. In *Proceedings of the 22nd International Conference on Automated Deduction*, CADE-22, pages 371–387, Berlin, Heidelberg, 2009. Springer-Verlag.
10. J. Endrullis, C. Grabmayer, D. Hendriks, A. Isihara, and J. W. Klop. Productivity of stream definitions. *Theor. Comput. Sci.*, 411(4-5):765–782, 2010.
11. J. Endrullis, D. Hendriks, and R. Bakhshi. On the Complexity of Equivalence of Specifications of Infinite Objects. In *Proc. ACM SIGPLAN Int. Conf. on Functional Programming (ICFP 2013)*, pages 153–164. ACM, 2012.
12. J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Automated Termination Proofs with AProVE. In *Proceedings of the 15th International Conference on Rewriting Techniques and Applications (RTA-04)*, volume 3091 of *Lecture Notes in Computer Science*, pages 210–220, 2004.
13. J. Goguen and R. Diaconescu. Towards an algebraic semantics for the object paradigm. In *Recent Trends in Data Type Specification*, number 785 in LNCS, pages 91–126, 1994.
14. D. Hausmann, T. Mossakowski, and L. Schröder. Iterative Circular Coinduction for CoCASL in Isabelle/HOL. In *Proceedings of FASE'05*, volume 3442 of *LNCS*, pages 341–356. Springer, 2005.
15. D. Kapur, P. Narendran, D. J. Rosenkrantz, and H. Zhang. Sufficient-completeness, ground-reducibility and their complexity. *Acta Inf.*, 28(4):311–350, 1991.
16. D. Lucanu, E.-I. Goriac, G. Caltais, and G. Roşu. CIRC : A behavioral verification tool based on circular coinduction. In *CALCO 2009*, volume 5728 of *LNCS*, pages 433–442. Springer, 2009.
17. T. Mossakowski, L. Schröder, M. Roggenbach, and H. Reichel. Algebraic-coalgebraic specification in CoCASL. *J. Log. Alg. Program.*, 67(1-2):146–197, 2006.
18. H. Reichel. Behavioural equivalence – a unifying concept for initial and final specifications. In *the 3rd Hungarian Comp. Sci. Conference*. Akademiai Kiado, 1981.
19. G. Roşu. *Hidden Logic*. PhD thesis, University of California at San Diego, 2000.

20. G. Roşu. Equality of streams is a Π^0_2-complete problem. In *Proceedgins of ICFP'06*, pages 184–191. ACM, 2006.
21. G. Roşu and J. Goguen. Hidden congruent deduction. In R. Caferra and G. Salzer, editors, *Automated Deduction in Classical and Non-Classical Logics*, volume 1761 of *Lecture Notes in Computer Science*, pages 251–266. Springer, 2000. Papers from First Order Theorem Proving 98 (FTP98), Vienna, November 1998.
22. G. Roşu and D. Lucanu. Circular Coinduction – A Proof Theoretical Foundation. In *CALCO 2009*, volume 5728 of *LNCS*, pages 127–144. Springer, 2009.
23. H. Rogers. *Theory of Recursive Functions and Effective Computability*. The MIT Press, paperback edition, 1987.
24. B. A. Sijtsma. On the productivity of recursive list definitions. *ACM Trans. Program. Lang. Syst.*, 11(4):633–649, 1989.
25. A. Silva and J. Rutten. Behavioural differential equations and coinduction for binary trees. In *WoLLIC 2007*, volume 4576 of *LNCS*, pages 322–336, 2007.
26. J. G. Simonsen. The Π^0_2-completeness of most of the properties of rewriting systems you care about (and productivity). In *Proceedings of RTA'09*, volume 5595 of *LNCS*, pages 335–349, 2009.
27. Terese, editor. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2003.
28. H. Zantema. A tool proving well-definedness of streams using termination tools. In *CALCO 2009*, volume 5728 of *Lecture Notes in Computer Science*, pages 449–456. Springer, 2009.
29. H. Zantema. Well-definedness of streams by termination. In *RTA 2009*, volume 5595 of *LNCS*, pages 164–178. Springer, 2009.