Enhanced Regular Corecursion for Data Streams*

Davide Ancona, Pietro Barbieri, and Elena Zucca

DIBRIS, University of Genova

Abstract. We propose a simple calculus for processing *data streams* (infinite flows of data series), represented by finite sets of equations built on stream operators. Furthermore, functions defining streams are *regularly corecursive*, that is, cyclic calls are detected, avoiding non-termination as happens with ordinary recursion in the call-by-value evaluation strategy. As we illustrate by several examples, the combination of such two mechanisms provides a good compromise between expressive power and decidability. Notably, we provide an algorithm to check that the stream returned by a function call is represented by a *well-defined set of equations* which actually admits a unique solution, hence access to an arbitrary element of the returned stream will never diverge.

Keywords: Operational semantics, stream programming, regular terms.

1 Introduction

Applications often deal with data structures which are conceptually infinite, among those *data streams* (infinite flows of data series) are a mainstream example: as we venture deeper into the Internet of Things (IoT) era, stream processing is becoming increasingly important. Indeed, all main IoT platforms provide embedded and integrated engines for real time analysis of potentially infinite flowing data series; such a process occurs before the data is stored for efficiency and, as often happens in Computer Science, there is a trade-off between the expressive power of the language, the efficiency of its implementation and the decidability of properties important to guarantee reliability and tractability.

Another related important problem is data stream generation, which is essential to test complex distributed IoT systems; the deterministic simulation of sensor data streams through a suitable language offers a practical solution to IoT testing and favors early detection of some kinds of bugs that can be fixed more easily before the deployment of the whole system.

A well-established solution to data stream generation and processing is *lazy evaluation*, as supported, e.g., in Haskell, and most stream libraries offered by mainstream languages, as `java.util.stream`. In this approach, conceptually infinite data streams are the result of a function or method call, which is evaluated

according to the call-by-need strategy. For instance, in Haskell we can define `one_two = 1:2:one_two`, or even represent the list of natural numbers as `from 0`, where `from n = n:from(n+1)`. However, such a great expressive power comes at a cost; let us consider, for instance, the definition `bad_stream = 0:tail bad_stream`. The Haskell compiler does not complain about this definition, and no problem arises at runtime as long as the manipulation of `bad_stream` requires only its first element to be accessed; anyway, any operation which needs to inspect `bad_stream` at a deeper level is deemed to diverge. Unfortunately, it is not decidable to check, even at runtime, whether the stream returned by a Haskell function is *well-defined*, that is, all of its elements can be computed[1]; indeed, the full expressive power of Haskell can be used to define streams by means of recursive functions. For similar reasons, it is not decidable to check at runtime whether the streams returned by two Haskell functions are equal.

More recently, a complementary approach has been considered in different programming paradigms — functional [11], logic [17,1,7], and object-oriented [2] — based on the following two ideas:

- Infinite streams can be finitely represented by *finite sets of equations* involving only the stream constructor, e.g., $x = 1 : 2 : x$. Such a representation corresponds to what has been called by Courcelle in its seminal paper [6] a *regular*, a.k.a. *rational*, tree, that is, a tree with possibly infinite depth but a finite set of subtrees.
- Functions are *regularly corecursive*, that is, execution keeps track of pending function calls, so that, when the same call is considered the second time, this is detected, avoiding non-termination as happens with ordinary recursion in the call-by-value evaluation strategy.

In this way, the Haskell stream `one_two` can be equivalently obtained by the call[2] `one_two()`, with the function `one_two` defined by `one_two() = 1:2:one_two()`. Indeed, with regular corecursion the result of this call is the value corresponding to the unique solution of the equation $x = 1 : 2 : x$. On the other hand, since the expressive power is limited to regular streams, it is not possible to define a corecursive function whose call returns the stream of natural numbers, as happens for the `from 0` Haskell example. However, there exist procedures for checking well-defined streams and their equality, even with tractable algorithms.

In this paper, we propose a simple calculus of numeric streams which supports regular corecursion and goes beyond regular streams by extending equations with other typical stream operators besides the stream constructor: tail and pointwise operators can be contained in stream equations and are therefore not evaluated.

In this way, we are able to achieve a good compromise between expressive power and decidability. Notably:

- the extended shape of equations allows the definition of functions which return non-regular streams; for instance, it is possible to obtain the stream of

[1] This is what is also known as a productive corecursive definition [5].

[2] Differently from Haskell, for simplicity in our calculus functions are uncurried, hence they take as arguments possibly empty tuples, delimited by parentheses.

natural numbers as `from(0)`, by defining `from(n)=n:(from(n)[+]repeat(1))`, with `[+]` the pointwise addition on numeric streams and `repeat` the function defined by `repeat(n)=n:repeat(n)`;
- there exists a decidable procedure to dynamically check whether the stream returned by a corecursive function is well-defined;
- however, it is not possible to express *all* streams computable with the lazy evaluation approach, but only those which have a specific structure (that is, can be expressed as the unique solution of a set of equations built with the above mentioned operators).

In Sect. 2 we define the calculus, in Sect. 3 we show examples, and in Sect. 4 we provide an operational characterization of well-defined streams, proved sufficient and necessary for an access to an arbitrary index to never diverge. In Sect. 5 we discuss related and further work. An extended version including more examples of derivations and complete proofs can be found at `http://arxiv.org/abs/2108.00281`.

2 Stream calculus

Fig. 1 shows the syntax of the calculus.

$\overline{fd} ::= fd_1 \dots fd_n$	program
$fd ::= f(\overline{x}) = se$	function declaration
$e ::= se \mid ne \mid be$	expression
$se ::= x \mid \texttt{if}\ be\ \texttt{then}\ se_1\ \texttt{else}\ se_2 \mid ne : se \mid se\hat{} \mid se_1[op]se_2 \mid f(\overline{e})$	stream expression
$ne ::= x \mid se(ne) \mid ne_1\ op\ ne_2 \mid 0 \mid 1 \mid 2 \mid \dots$	numeric expression
$be ::= x \mid \texttt{true} \mid \texttt{false} \mid \dots$	boolean expression
$op ::= + \mid - \mid * \mid /$	numeric operation

Fig. 1. Stream calculus: syntax

A program is a sequence of (mutually recursive) function declarations, for simplicity assumed to only return streams. Stream expressions are variables, conditional expressions, expressions built by stream operators, and function calls. We consider the following stream operators: constructor (prepending a numeric element), tail, and pointwise arithmetic operations. Numeric expressions include the access to the i-th[3] element of a stream. We use $\overline{fd}$ to denote a sequence $fd_1, \dots, fd_n$ of function declarations, and analogously for other sequences.

The operational semantics, given in Fig. 2, is based on two key ideas:

1. (some) infinite streams are represented in a finite way
2. evaluation keeps trace of already considered function calls

[3] For simplicity, here indexing and numeric expressions coincide, even though indexes are expected to be natural numbers, while values in streams can range over a larger numeric domain.

$$
\begin{array}{lll}
c & ::= f(\overline{v}) & \text{(evaluated) call} \\
v & ::= s \mid n \mid b & \text{value} \\
s & ::= x \mid n : s \mid s\hat{} \mid s_1[op]s_2 & \text{(open) stream value} \\
i, n & ::= 0 \mid 1 \mid 2 \mid ... & \text{index, numeric value} \\
b & ::= \texttt{true} \mid \texttt{false} & \text{boolean value} \\
\tau & ::= c_1 \mapsto x_1 \ \ldots \ c_n \mapsto x_n \quad (n \geq 0) & \text{call trace} \\
\rho & ::= x_1 \mapsto s_1 \ldots x_n \mapsto s_n \quad (n \geq 0) & \text{environment}
\end{array}
$$

$$
\text{(VAL)}\ \frac{}{v, \rho, \tau \Downarrow (v, \rho)} \qquad \text{(IF-T)}\ \frac{be, \rho, \tau \Downarrow (\texttt{true}, \rho) \quad se_1, \rho, \tau \Downarrow (s, \rho')}{\texttt{if}\ be\ \texttt{then}\ se_1\ \texttt{else}\ se_2, \rho, \tau \Downarrow (s, \rho')} \qquad \text{(IF-F)}\ \frac{be, \rho, \tau \Downarrow (\texttt{false}, \rho) \quad se_2, \rho, \tau \Downarrow (s, \rho')}{\texttt{if}\ be\ \texttt{then}\ se_1\ \texttt{else}\ se_2, \rho, \tau \Downarrow (s, \rho')}
$$

$$
\text{(CONS)}\ \frac{ne, \rho, \tau \Downarrow (n, \rho) \quad se, \rho, \tau \Downarrow (s, \rho')}{ne : se, \rho, \tau \Downarrow (n : s, \rho')} \qquad \text{(TAIL)}\ \frac{se, \rho, \tau \Downarrow (s, \rho')}{se\hat{}, \rho, \tau \Downarrow (s\hat{}, \rho')} \qquad \text{(PW)}\ \frac{se_1, \rho, \tau \Downarrow (s_1, \rho_1) \quad se_2, \rho, \tau \Downarrow (s_2, \rho_2)}{se_1[op]se_2, \rho, \tau \Downarrow (s_1[op]s_2, \rho_1 \sqcup \rho_2)}
$$

$$
\text{(ARGS)}\ \frac{e_i, \rho, \tau \Downarrow (v_i, \rho_i) \quad \forall i \in 1..n \qquad f(\overline{v}), \widehat{\rho}, \tau \Downarrow (s, \rho')}{f(\overline{e}), \rho, \tau \Downarrow (s, \rho')} \quad \begin{array}{l} \overline{e} = e_1, \ldots, e_n \text{ not of shape } \overline{v} \\ \overline{v} = v_1, \ldots, v_n \\ \widehat{\rho} = \bigsqcup_{i \in 1..n} \rho_i \end{array}
$$

$$
\text{(INVK)}\ \frac{se[\overline{v}/\overline{x}], \rho, \tau\{f(\overline{v}) \mapsto x\} \Downarrow (s, \rho')}{f(\overline{v}), \rho, \tau \Downarrow (x, \rho'\{x \mapsto s\})} \quad \begin{array}{l} f(\overline{v}) \notin dom(\tau_{\approx\rho}) \\ x \text{ fresh} \\ fbody(f) = (\overline{x}, se) \\ wd(\rho', x, s) \end{array} \qquad \text{(COREC)}\ \frac{}{f(\overline{v}), \rho, \tau \Downarrow (x, \rho)}\ \tau_{\approx\rho}(f(\overline{v})) = x
$$

$$
\text{(AT)}\ \frac{se, \rho, \tau \Downarrow (s, \rho') \quad ne, \rho, \tau \Downarrow (i, \rho)}{se(ne), \rho, \tau \Downarrow (n, \rho)}\ at_{\rho'}(s, i) = n
$$

$$
\text{(AT-VAR)}\ \frac{at_\rho(\rho(x), i) = n'}{at_\rho(x, i) = n'} \qquad \text{(AT-CONS-0)}\ \frac{}{at_\rho(n : s, 0) = n} \qquad \text{(AT-CONS-N)}\ \frac{at_\rho(s, i-1) = n'}{at_\rho(n : s, i) = n'}\ i > 0
$$

$$
\text{(AT-TAIL)}\ \frac{at_\rho(s, i+1) = n}{at_\rho(s\hat{}, i) = n} \qquad \text{(AT-PW)}\ \frac{at_\rho(s_1, i) = n_1 \quad at_\rho(s_2, i) = n_2}{at_\rho(s_1[op]s_2, i) = n_1\ op\ n_2}
$$

Fig. 2. Stream calculus: operational semantics

To obtain (1), our approach is inspired by *capsules* [10], which are essentially expressions supporting cyclic references. That is, the *result* of the evaluation of a stream expression is a pair (s, ρ), where s is an *(open) stream value*, built on top of stream variables, numeric values, the stream constructor, the tail destructor and the pointwise arithmetic operators, and ρ is an *environment* mapping a finite set of variables into stream values. In this way, cyclic streams can be obtained: for instance, $(x, x \mapsto n : x)$ denotes the stream constantly equal to n.

We denote by $vars(\rho)$ the set of variables occurring in ρ, by $fv(\rho)$ the set of its free variables, that is, $vars(\rho) \setminus dom(\rho)$, and say that ρ is *closed* if $fv(\rho) = \emptyset$, *open* otherwise, and analogously for a result (v, ρ).

To obtain point (2) above, evaluation has an additional parameter which is a *call trace*, a map from function calls where arguments are values (dubbed *calls* for short in the following) into variables.

Altogether, the semantic judgment has shape $e, \rho, \tau \Downarrow (v, \rho')$, where e is the expression to be evaluated, ρ the current environment defining possibly cyclic stream values that can occur in e, τ the call trace, and (v, ρ') the result. The semantic judgments should be indexed by an underlying (fixed) program, omitted for sake of simplicity. Rules use the following auxiliary definitions:

- $\rho \sqcup \rho'$ is the union of two environments, which is well-defined if they have disjoint domains; $\rho\{x \mapsto s\}$ is the environment which gives s on x, coincides with ρ elsewhere; we use analogous notations for call traces.
- $se[\overline{v}/\overline{x}]$ is obtained by parallel substitution of variables $\overline{x}$ with values $\overline{v}$.
- $fbody(f)$ returns the pair of the parameters and the body of the declaration of f, if any, in the assumed program.

Moreover, the rules are parametric in the following other judgments, for which different definitions will be discussed in Sect. 4:

- $wd(\rho, x, s)$, that is, by adding the association $x \mapsto s$ to the (well-defined) environment ρ, we still get a well-defined environment.
- $v \approx_\rho v'$, that is, the two values are equivalent in the environment[4] ρ. Then, $\tau_{\approx\rho}$ is the extension of τ *up to equivalence* in ρ: $\tau_{\approx\rho}(f(v_1, \ldots, v_n)) = x$ iff there exist $v'_1, \ldots, v'_n$ such that $\tau(f(v'_1, \ldots, v'_n)) = x$ and $v_i \approx_\rho v'_i$ for $i \in 1..n$.

Intuitively, a closed result (s, ρ) is well-defined if it denotes a unique stream (infinite sequence of numeric values), and a closed environment ρ is well-defined if, for each $x \in dom(\rho)$, (x, ρ) is well-defined. In other words, the corresponding set of equations admits a unique solution. For instance, the environment $\{x \mapsto x\}$ is not well-defined, since it is undetermined (any stream satisfies the equation $x = x$); the environment $\{x \mapsto x[+]y, y \mapsto 1 : y\}$ is not well-defined as well, since it is undefined (the two equations $x = x \mapsto x[+]y, y = 1 : y$ admit no solutions for x). Finally, two stream values s and s' such that the results (s, ρ) and (s', ρ) are closed and well-defined are equivalent if they denote the same stream.

These notions can be generalized to open results and environments, assuming that free variables denote unique streams, as will be formalized in Sect. 4.

Rules for values and conditional are straightforward. In rules (CONS), (TAIL) and (PW), arguments are evaluated, while the stream operator is applied without any further evaluation; the fact that the tail and pointwise operators are treated as the stream constructor $\_ : \_$ is crucial to get results which denote non-regular streams as shown in Sect. 3. However, when non-constructors are allowed to occur in values, ensuring well-defined results become more challenging, because the usual simple syntactic constraints that can be safely used for constructors [5] no longer work (see more details in Sect. 4 and 5).

The rules for function call use a mechanism of cycle detection, similar to that in [2]. They are given in a modular way. That is, evaluation of arguments is handled by a separate rule (ARGS).

[4] This equivalence is assumed to be the identity on numeric and boolean values.

Rule (INVK) is applied when a call is considered for the first time, as expressed by the first side condition. The body is retrieved by using the auxiliary function *fbody*, and evaluated in a call trace where the call has been mapped into a fresh variable. Then, it is checked that adding the association from such variable to the result of the evaluation of the body keeps the environment well-defined. If the check succeeds, then the final result consists of the variable associated with the call and the updated environment. For simplicity, here execution is stuck if the check fails; an implementation should raise a runtime error instead.

Rule (COREC) is applied when a call is considered for the second time, as expressed by the first side condition (note that cycle detection takes place up to equivalence in the environment). The variable x is returned as result. However, there is no associated value in the environment yet; in other words, the result (x, ρ) is open at this point. This means that x is undefined until the environment is updated with the corresponding value in rule (INVK). However, x can be safely used as long as the evaluation does not require x to be inspected; for instance, x can be safely passed as an argument to a function call.

For instance, if we consider `f()=g() g()=1:f()`, then the judgment $\texttt{f()}, \emptyset, \emptyset \Downarrow (x, \rho)$, with $\rho = \{x \mapsto y, y \mapsto 1 : x\}$, is derivable; however, while the final result (x, ρ) is closed, the derivation contains also judgments with open results, as, e.g., $\texttt{f()}, \emptyset, \{\texttt{f()} \mapsto x, \texttt{g()} \mapsto y\} \Downarrow (x, \emptyset)$ and $\texttt{g()}, \emptyset, \{\texttt{f()} \mapsto x\} \Downarrow (y, \{y \mapsto 1 : x\})$.

As another example, if we consider `f()=g(2:f()) g(s)=1:s`, then the derivation of the judgment $\texttt{f()}, \emptyset, \emptyset \Downarrow (x, \rho)$ with $\rho = \{x \mapsto y, y \mapsto 1 : 2 : x\}$ is built on top of the derivation of $\texttt{g(2:}x\texttt{)}, \emptyset, \{\texttt{f()} \mapsto x\} \Downarrow (y, \{y \mapsto 1 : 2 : x\})$, corresponding to the evaluation of `g(2:`x`)` where x is an operand of the stream constructor whose result is passed as argument to the call to `g`, despite x is not defined yet.

Finally, rule (AT) computes the i-th element of a stream expression. After evaluation of the arguments, the numeric result is obtained by the auxiliary judgment $at_\rho(s, i) = n$, inductively defined in the bottom part of the figure. If the stream value is a variable, rule (AT-VAR), then the evaluation is propagated to the associated stream value in the environment, if any. If, instead, the variable is free in the environment, then execution is stuck; again, an implementation should raise a runtime error instead. If the stream value is built by the constructor, then the result is the first element of the stream if the index is 0, rule (AT-CONS-0); otherwise, the evaluation is recursively propagated to its tail with the predecessor index, rule (AT-CONS-N). Conversely, if the stream is built by the tail operator, rule (AT-TAIL), then the evaluation is recursively propagated to the stream argument with the successor index. Finally, if the stream is built by a pointwise operation, rule (AT-PW), then the evaluation is recursively propagated to the operands with the same index and then the corresponding arithmetic operation is computed on the results.

Derivations of examples in this section can be found in the extended version.

3 Examples

First we show some simple examples, to explain how regular corecursion works. Then we provide some more significant examples.

Consider the following function declarations:

```
repeat(n) = n:repeat(n)
one_two() = 1:two_one()
two_one() = 2:one_two()
```

With the standard semantics of recursion, the calls, e.g., `repeat(0)` and `one_two()` lead to non-termination. Thanks to regular corecursion, instead, these calls terminate, producing as result $(x, \{x \mapsto \texttt{0} : x\})$, and $(x, \{x \mapsto \texttt{1} : y, y \mapsto \texttt{2} : x\})$, respectively. Indeed, when initially invoked, the call `repeat(0)` is added in the call trace with an associated fresh variable, say x. In this way, when evaluating the body of the function, the recursive call is detected as cyclic, the variable x is returned as its result, and, finally, the stream value $\texttt{0} : x$ is associated in the environment with the result x of the initial call. The evaluation of `one_two()` is analogous, except that another fresh variable y is generated for the intermediate call `two_one()`. The formal derivations are given below.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{repeat(0)}, \emptyset, \{\texttt{repeat(0)} \mapsto x\} \Downarrow (x, \emptyset)}\,(\textsc{corec})}{\texttt{0} : \texttt{repeat(0)}, \emptyset, \{\texttt{repeat(0)} \mapsto x\} \Downarrow (\texttt{0} : x, \emptyset)}\,(\textsc{value})\,(\textsc{cons})}{\texttt{repeat(0)}, \emptyset, \emptyset \Downarrow (x, \{x \mapsto \texttt{0} : x\})}\,(\textsc{invk})$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{one\_two}(), \emptyset, \{\texttt{one\_two}() \mapsto x,\ \texttt{two\_one}() \mapsto y\} \Downarrow (x, \emptyset)}\,(\textsc{corec})}{\texttt{2} : \texttt{one\_two}(), \emptyset, \{\texttt{one\_two}() \mapsto x,\ \texttt{two\_one}() \mapsto y\} \Downarrow (\texttt{2} : x, \emptyset)}\,(\textsc{value})\,(\textsc{cons})}{\texttt{two\_one}(), \emptyset, \{\texttt{one\_two}() \mapsto x\} \Downarrow (y, \{y \mapsto \texttt{2} : x\})}\,(\textsc{value})\,(\textsc{invk})}{\texttt{1} : \texttt{two\_one}(), \emptyset, \{\texttt{one\_two}() \mapsto x\} \Downarrow (\texttt{1} : y, \{y \mapsto \texttt{2} : x\})}\,(\textsc{cons})}{\texttt{one\_two}(), \emptyset, \emptyset \Downarrow (x, \{x \mapsto \texttt{1} : y,\ y \mapsto \texttt{2} : x\})}\,(\textsc{invk})$$

For space reasons, we did not report the application of rule (VALUE). In both derivations, note that rule (COREC) is applied, without evaluating the body once more, when the cyclic call is detected.

The following examples show function definitions whose calls return non-regular streams, notably, the natural numbers, the natural numbers raised to the power of a number, the factorials, the powers of a number, the Fibonacci numbers, and the stream obtained by pointwise increment by one.

```
nat() = 0:(nat()[+]repeat(1))
nat_to_pow(n) =                    //nat_to_pow(n)(i)=i^n
  if n <= 0 then repeat(1) else nat_to_pow(n-1)[*]nat()
fact() = 1:((nat()[+]repeat(1))[*]fact())
pow(n) = 1:(repeat(n)[*]pow(n)) //pow(n)(i)=n^i
fib() = 0:1:(fib()[+]fib()^)
incr(s) = s[+]repeat(1)
```

The definition of `nat` uses regular corecursion, since the recursive call `nat()` is cyclic. Hence the call `nat()` returns $(x, \{x \mapsto 0 : (x[+]y), y \mapsto 1 : y\})$. The definition of `nat_to_pow` is a standard inductive one where the argument strictly decreases in the recursive call. Hence, the call, e.g., `nat_to_pow(2)`, returns $(x_2, \{x_2 \mapsto x_1[*]x, x_1 \mapsto x_0[*]x, x_0 \mapsto y, y \mapsto 1 : y, x \mapsto 0 : (x[+]y'), y' \mapsto 1 : y'\})$. The definitions of `fact`, `pow`, and `fib` are regularly corecursive. For instance, the call `fact()` returns $(z, z \mapsto (x[+]y)[*]z, x \mapsto 0 : (x[+]y'), y \mapsto 1 : y, y' \mapsto 1 : y')$. The definition of `incr` is non-recursive, hence always converges, and the call `incr(`s`)` returns $(x, \{x \mapsto s[+]y, y \mapsto 1 : y\})$. The following alternative definition

```
incr_reg(s) = (s(0)+1):incr_reg(s^)
```

relies, instead, on regular corecursion. Note the difference: the latter version ensures termination only for regular streams, as in `incr_reg(one_two())`, since, eventually, in the recursive call, the expression `s^` turns out to denote the initial stream; however, the computation does not terminate for non-regular streams, as in `incr_reg(nat())`, which, however, converges with `incr`.

The following function computes the stream of partial sums of the first $i+1$ elements of a stream s, that is, `sum(`s`)(`i`)`$= \sum_{k=0}^{i} s(k)$:

```
sum(s) = s(0):(s^[+]sum(s))
```

Such a function is useful for computing streams whose elements approximate a series with increasing precision; for instance, the following function returns the stream of partial sums of the first $i+1$ elements of the Taylor series of the exponential function:

```
sum_expn(n) = sum(pow(n)[/]fact())
```

Function `sum_expn` calls `sum` with the argument `pow(n)[/]fact()` corresponding to the stream of all terms of the Taylor series of the exponential function; hence, by accessing the i-th element of the stream, we have the following approximation:

$$\texttt{sum\_expn}(n)(i) = \sum_{k=0}^{i} \frac{n^k}{k!} = 1 + n + \frac{n^2}{2!} + \frac{n^3}{3!} + \frac{n^4}{4!} + \cdots + \frac{n^i}{i!}$$

Lastly, we present a couple of examples showing how it is possible to define primitive operations provided by IoT platforms for real time analysis of data streams; we start with `aggr(n,s)`, which allows aggregation (by addition) of contiguous data in the stream `s` w.r.t. a frame of length `n`:

```
aggr(n,s) = if n<=0 then repeat(0) else s[+]aggr(n-1,s^)
```

For instance, `aggr(3,`s`)` returns the stream s' s.t. $s'(i) = s(i)+s(i+1)+s(i+2)$. On top of `aggr`, we can easily define `avg(n,s)` to compute the stream of average values of `s` in the frame of length `n`:

```
avg(n,s) = aggr(n,s)[/]repeat(n)
```

4 Well-defined environments and equivalent streams

In the semantic rules, we have left unspecified two notions: *well-defined environments*, and *equivalent streams*. We provide now a formal definition in abstract terms. Then, we provide an operational definition of well-defined environments.

Semantically, a stream σ is an infinite sequence of numeric values, that is, a function which returns, for each index $i \geq 0$, the i-th element $\sigma(i)$. Given a result (s, ρ), we get a stream by instantiating variables in s with streams, in a way consistent with ρ, and evaluating operators. To make this formal, we need some preliminary definitions.

A *substitution* θ is a function from a finite set of variables to streams. We denote by $[\![s]\!]\theta$ the stream obtained by applying θ to s, and evaluating operators, as formally defined below.

$$[\![x]\!]\theta = \theta(x)$$

$$([\![n : s]\!]\theta)(i) = \begin{cases} n & i = 0 \\ ([\![s]\!]\theta)(i-1) & i \geq 1 \end{cases}$$

$$([\![s\hat{}]\!]\theta)(i) = [\![s]\!]\theta(i+1) \qquad i \geq 0$$

$$([\![s_1[op]s_2]\!]\theta)(i) = [\![s_1]\!]\theta(i)\ op\ [\![s_2]\!]\theta(i) \qquad i \geq 0$$

Given an environment ρ and a substitution θ with domain $vars(\rho)$, the substitution $\rho[\theta]$ is defined by:

$$\rho[\theta](x) = \begin{cases} [\![\rho(x)]\!]\theta & x \in dom(\rho) \\ \theta(x) & x \in fv(\rho) \end{cases}$$

Then, a *solution* of ρ is a substitution θ with domain $vars(\rho)$ such that $\rho[\theta] = \theta$.

A closed environment ρ is *well-defined* if it has exactly one solution, denoted $\mathsf{sol}(\rho)$. For instance, $\{x \mapsto 1 : x\}$ and $\{y \mapsto 0 : (y[+]x),\ x \mapsto 1 : x\}$ are well-defined, since their unique solutions map x to the infinite stream of ones, and y to the stream of natural numbers, respectively. Instead, for $\{x \mapsto 1[+]x\}$ there are no solutions. Lastly, an environment can be undetermined: for instance, a substitution mapping x into an arbitrary stream is a solution of $\{x \mapsto x\}$.

An open environment ρ is well-defined if, for each θ with domain $fv(\rho)$, it has exactly one solution θ' such that $\theta \subseteq \theta'$. For instance, the open environment $\{y \mapsto 0 : (y[+]x)\}$ is well-defined.

Given a closed result (s, ρ), with ρ well-defined, we define its semantics by $[\![s]\!]\rho = [\![s]\!]\theta$ for $\theta = \mathsf{sol}(\rho)$. Then, two stream values s and s' are *semantically equivalent in* ρ if $[\![s]\!]\rho = [\![s']\!]\rho'$.

We now consider the non-trivial problem of ensuring that a closed environment ρ is well-defined; if environments would be allowed to contain only the stream constructor, then it would suffice to require all non-free variables to be *guarded* by the stream constructor [5]. For instance, the environment $\{x \mapsto 1 : x\}$ satisfies such a syntactic condition, and is well-defined, while in the non well-defined environment $\{x \mapsto x\}$ the variable x is not guarded by the constructor.

However, when non constructors as the tail and pointwise operators come into play, the fact that variables are guarded by the stream constructor no longer ensures that the environment is well-defined; for instance, the environment $\rho = \{x \mapsto 0 : x\hat{}\}$ corresponding to the definition of `bad_stream` in Sect. 1 is not well-defined since it admits infinite solutions (all streams starting with 0), although variable x is guarded by the stream constructor. A more complex check is needed: roughly, more constructors than tail operators should have been traversed when we find a cyclic reference, as formalized in Fig. 3.

$m ::= x_1 \mapsto n_1 \dots x_n \mapsto n_k \quad (n \geq 0)$ map from variables to natural numbers

$$(\text{MAIN})\ \frac{\mathsf{wd}(x, \rho\{x \mapsto v\}, \emptyset)}{wd(\rho, x, v)} \qquad (\text{WF-VAR})\ \frac{\mathsf{wd}(\rho(x), \rho, m\{x \mapsto 0\})}{\mathsf{wd}(x, \rho, m)}\ x \notin dom(m)$$

$$(\text{WF-COREC})\ \frac{}{\mathsf{wd}(x, \rho, m)}\ \begin{array}{l} x \in dom(m) \\ m(x) > 0 \end{array} \qquad (\text{WF-FV})\ \frac{}{\mathsf{wd}(x, \rho, m)}\ x \notin dom(\rho)$$

$$(\text{WF-CONS})\ \frac{\mathsf{wd}(s, \rho, m^{+1})}{\mathsf{wd}(n : s, \rho, m)} \qquad (\text{WF-TAIL})\ \frac{\mathsf{wd}(s, \rho, m^{-1})}{\mathsf{wd}(s\hat{}, \rho, m)} \qquad (\text{WF-PW})\ \frac{\mathsf{wd}(s_1, \rho, m) \quad \mathsf{wd}(s_2, \rho, m)}{\mathsf{wd}(s_1 [op] s_2, \rho, m)}$$

Fig. 3. Operational definition of well-defined environments

The judgment $wd(\rho, x, s)$ used in the side condition of rule (INVK) holds if $\mathsf{wd}(x, \rho\{x \mapsto v\}, \emptyset)$ holds. The judgment $\mathsf{wd}(s, \rho, \emptyset)$ means that a result is well-defined. That is, restricting the domain of ρ to the variables reachable from s (those either occurring in s, or, transitively, in values associated with reachable variables) we get a well-defined environment; thus, $wd(\rho, x, s)$ holds if adding the association of s with x preserves well-definedness of ρ.

The additional argument m in the judgment $\mathsf{wd}(s, \rho, m)$ is a map from variables to natural numbers. We write m^{+1} and m^{-1} for the maps $\{(x, m(x) + 1) \mid x \in dom(m)\}$, and $\{(x, m(x) - 1) \mid x \in dom(m)\}$, respectively.

In rule (MAIN), this map is initially empty. In rule (WF-VAR), a variable x defined in the environment is added in the map, with initial value 0, the first time it is found. In rule (WF-COREC), when it is found the second time, it is checked that more constructors than tail operators have been traversed. In rule (WF-FV), a free variable is considered well-defined.[5] In rules (WF-CONS), (WF-TAIL), and (WF-PW), the value associated with a variable is incremented/decremented by one each time a constructor and tail operator are traversed, respectively.

As an example of derivation of well-definedness and access to the i-th element, in Fig. 4 we consider the result $(x, \{x \mapsto 0 : (x\ [+]\ y), y \mapsto 1 : y\})$, obtained by evaluating the call `nat()` with `nat` defined as in Sect. 3.

[5] Indeed, non-well-definedness can only be detected on closed results.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathsf{wd}(x,\rho,\{x\mapsto 1\})}\,\text{(WF-COREC)}}{\mathsf{wd}(x\ [+]\ y,\rho,\{x\mapsto 1\})}\,\text{(WF-PW)}\qquad \dfrac{\dfrac{\dfrac{}{\mathsf{wd}(y,\rho,\{x\mapsto 1,y\mapsto 1\})}\,\text{(WF-COREC)}}{\mathsf{wd}(1:y,\rho,\{x\mapsto 1,y\mapsto 0\})}\,\text{(WF-CONS)}}{\mathsf{wd}(y,\rho,\{x\mapsto 1\})}\,\text{(WF-VAR)}}{\mathsf{wd}(0:(x\ [+]\ y),\rho,\{x\mapsto 0\})}\,\text{(WF-CONS)}}{\mathsf{wd}(x,\rho,\emptyset)}\,\text{(WF-VAR)}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{at_\rho(0:(x\ [+]\ y),0)=0}\,\text{(AT-CONS-0)}}{\vdots}}{at_\rho(x,i-1)=i-1}\,\text{(AT-VAR)}\qquad \dfrac{\dfrac{\dfrac{}{at_\rho(1:y,0)=1}\,\text{(AT-CONS-0)}}{\vdots}}{at_\rho(y,i-1)=1}\,\text{(AT-VAR)}}{at_\rho(x\ [+]\ y,i-1)=i}\,\text{(AT-OP)}}{at_\rho(0:(x\ [+]\ y),i)=i}\,\text{(AT-CONS-N)}}{at_\rho(x,i)=i}\,\text{(AT-VAR)}$$

Fig. 4. Derivations for $\rho = (x, \{x \mapsto 0 : (x\ [+]\ y),\ y \mapsto 1 : y\})$.

In the extended version we show the derivation for a trickier example, that is, the result $(x, \{x \mapsto 0 : 1 : (2 : x\hat{})\hat{}\})$. Its semantics is the stream $0, 1, 1, 1, \ldots$.

We show now that well-definedness of a result is a necessary and sufficient condition for termination of access to an arbitrary index. To formally express and prove this statement, we introduce some definitions and notations.

First of all, since the numeric value obtained as result is not relevant for the following technical treatment, for simplicity we will write $at_\rho(s, i)$ rather than $at_\rho(s, i) = n$. We call *derivation* an either finite or infinite proof tree.

We write $\mathsf{wd}(s', \rho, m') \vdash \mathsf{wd}(s, \rho, m)$ to mean that $\mathsf{wd}(s', \rho, m')$ is a premise of a (meta-)rule where $\mathsf{wd}(s, \rho, m)$ is the consequence, and $\vdash^\star$ for the reflexive and transitive closure of this relation. Moreover, $\mathsf{wd}(x, \rho, m') \vdash^\star_{\mathcal{X}} \mathsf{wd}(s, \rho, m)$, with $x \notin \mathcal{X}$, means that in the path there can be nodes of shape $\mathsf{wd}(y, \rho, \_)$ only for $y \in \mathcal{X}$ and non-repeated. We use analogous notations for the judgment $at_\rho(s, i)$.

Lemma 1.

1. *If* $at_\rho(x, i') \vdash^\star_{\mathcal{X}} at_\rho(s, i)$, *then* $at_\rho(x, i' + k) \vdash^\star_{\mathcal{X}} at_\rho(s, i + k)$, *for each* $k \geq 0$.
2. *A judgment* $\mathsf{wd}(s, \rho, \emptyset)$ *has no derivation iff the following condition holds:*
   (WF-STUCK) $\mathsf{wd}(x, \rho, m') \vdash^\star_{\mathcal{X}'} \mathsf{wd}(\rho(x), \rho, m\{x \mapsto 0\}) \vdash \mathsf{wd}(x, \rho, m) \vdash^\star_{\mathcal{X}} \mathsf{wd}(s, \rho, \emptyset)$
   for some $x \in dom(\rho)$, $\mathcal{X}', \mathcal{X}$, *and* m', m *s.t.*
   $x \notin dom(m), m'(x) = k \leq 0$.
3. *The derivation of* $at_\rho(s, j)$ *is infinite iff the following condition holds:*
   (AT-∞) $at_\rho(x, i + k) \vdash^\star_{\mathcal{X}'} at_\rho(\rho(x), i) \vdash at_\rho(x, i) \vdash^\star_{\mathcal{X}} at_\rho(s, j)$
   for some $x \in dom(\rho)$, $\mathcal{X}', \mathcal{X}$, *and* $i, k \geq 0$.

Lemma 2. *For* $x \in dom(m)$, *the following conditions are equivalent:*

1. $at_\rho(x, i') \vdash^\star_{\mathcal{X}} at_\rho(s, i)$ *for some* i', i
2. $\mathsf{wd}(x, \rho, m') \vdash^\star_{\mathcal{X}} \mathsf{wd}(s, \rho, m)$ *for some* m' *such that* $m'(x) = m(x) + i - i'$.

Proof.

1⇒ 2 By induction on the length of the path in $at_\rho(x,i') \vdash^\star_{\mathcal{X}} at_\rho(s,i)$.

Base The length of the path is 0, hence we have $at_\rho(x,i) \vdash^\star_\emptyset at_\rho(x,i)$. We also have $\mathsf{wd}(x,\rho,m) \vdash^\star_\emptyset \mathsf{wd}(x,\rho,m)$, and $m(x) = m(x) + i - i$, as requested.

Inductive step By cases on the rule applied to derive $at_\rho(s,i)$. We show the most significant cases.

(at-var) We have $at_\rho(y,i)$, with $y \neq x$ since the length of the path is > 0, and $at_\rho(x,i') \vdash^\star_{\chi\setminus\{y\}} at_\rho(\rho(y),i)$.
Moreover, we can derive $\mathsf{wd}(y,\rho,m)$ by rule (WF-VAR), and by inductive hypothesis we also have $\mathsf{wd}(x,\rho,m') \vdash^\star_{\chi\setminus\{y\}} \mathsf{wd}(\rho(y),\rho,m\{y\mapsto 0\})$, and $m'(x) = m\{y\mapsto 0\}(x) + i - i'$, hence we get the thesis.

(at-cons-0) Empty case, since the derivation for $at_\rho(n : s, 0)$ does not contain a node $at_\rho(x,i')$.

(at-cons) We have $at_\rho(n : s, i)$, and $at_\rho(x,i') \vdash^\star_{\mathcal{X}} at_\rho(s, i-1)$. Moreover, we can derive $\mathsf{wd}(n : s,\rho,m)$ by rule (WF-CONS), and by inductive hypothesis we also have $\mathsf{wd}(x,\rho,m') \vdash^\star_{\mathcal{X}} \mathsf{wd}(s,\rho,m^{+1})$, with $m'(x) = m^{+1}(x) + (i-1) - i'$, hence we get the thesis.

2⇒ 1 By induction on the length of the path in $\mathsf{wd}(x,\rho,m') \vdash^\star \mathsf{wd}(s,\rho,m)$.

Base The length of the path is 0, hence we have $\mathsf{wd}(x,\rho,m) \vdash^\star_\emptyset \mathsf{wd}(x,\rho,m)$. We also have, for an arbitrary i, $at_\rho(x,i) \vdash^\star_\emptyset at_\rho(x,i)$, and $m(x) = m(x) + i - i$, as requested.

Inductive step By cases on the rule applied to derive $\mathsf{wd}(s,\rho,m)$. We show the most significant cases.

(wf-var) We have $\mathsf{wd}(y,\rho,m)$, with $y \notin dom(m)$, $y \neq x$ since $x \in dom(m)$, and $\mathsf{wd}(x,\rho,m') \vdash^\star_{\mathcal{X}\setminus\{y\}} \mathsf{wd}(\rho(y),\rho,m\{y \mapsto 0\})$. By inductive hypothesis we have $at_\rho(x,i') \vdash^\star_{\mathcal{X}\setminus\{y\}} at_\rho(\rho(y),i)$ for some i', i such that $m'(x) = m(x) + i - i'$. Moreover, since $y \in dom(\rho)$, $at_\rho(\rho(y),i) \vdash at_\rho(y,i)$ by rule (AT-VAR), hence we get $at_\rho(x,i') \vdash^\star_{\chi} at_\rho(y,i)$.

(wf-corec) Empty case, since the derivation for $\mathsf{wd}(y,\rho,m)$ would not contain a node $\mathsf{wd}(x,\rho,m)$.

(wf-fv) Empty case, since the derivation for $\mathsf{wd}(y,\rho,m)$ would not contain a node $\mathsf{wd}(x,\rho,m)$.

(wf-cons) We have $\mathsf{wd}(n : s,\rho,m)$, and $\mathsf{wd}(x,\rho,m') \vdash^\star_{\mathcal{X}} \mathsf{wd}(s,\rho,m^{+1})$. By inductive hypothesis we have $at_\rho(x,i') \vdash^\star_{\mathcal{X}} at_\rho(s,i)$ for some i', i such that $m'(x) = m^{+1}(x) + i - i'$. Moreover, $at_\rho(s,i) \vdash at_\rho(n\colon s, i+1)$ by rule (AT-CONS-N), hence we get $at_\rho(x,i') \vdash^\star_{\chi} at_\rho(n : s, i+1)$ with $m'(x) = m(x) + i + 1 - i'$, as requested.

Lemma 3. *For $x \notin dom(m)$, the following conditions are equivalent:*

1. $at_\rho(x,i') \vdash^\star_{\mathcal{X}} at_\rho(s,i)$ *for some* i', i
2. $\mathsf{wd}(x,\rho,m') \vdash^\star_{\mathcal{X}} \mathsf{wd}(s,\rho,m)$ *for some* m' *such that* $x \notin dom(m')$.

Proof. Easy variant of the proof of Lemma 2.

Theorem 1. $\mathsf{wd}(s,\rho,\emptyset)$ *is derivable iff, for all* j, $at_\rho(s,j)$ *either has no derivation or a finite derivation.*

Proof. We prove that $at_\rho(s,j)$ has an infinite derivation for some j iff $\mathsf{wd}(s,\rho,\emptyset)$ has no derivation.

$\Rightarrow$ By Lemma 1-(3),we have that the following condition holds:

(AT-∞) $at_\rho(x,i+k) \vdash^\star_{\mathcal{X}'} at_\rho(\rho(x),i) \vdash at_\rho(x,i) \vdash^\star_{\mathcal{X}} at_\rho(s,j)$
for some $x \in dom(\rho)$, $\mathcal{X}', \mathcal{X}$, and $i,k \geq 0$.

Then, starting from the right, by Lemma 3 we have $\mathsf{wd}(x,\rho,m) \vdash^\star_{\mathcal{X}} \mathsf{wd}(s,\rho,\emptyset)$ for some m such that $x \notin dom(m)$; by rule (WF-VAR) we have $\mathsf{wd}(\rho(x),\rho,m\{x \mapsto 0\}) \vdash \mathsf{wd}(x,\rho,m)$, and finally by Lemma 2 we have:

(WF-STUCK) $\mathsf{wd}(x,\rho,m') \vdash^\star_{\mathcal{X}'} \mathsf{wd}(\rho(x),\rho,m\{x \mapsto 0\}) \vdash \mathsf{wd}(x,\rho,m) \vdash^\star_{\mathcal{X}} \mathsf{wd}(s,\rho,\emptyset)$
for $x \in dom(\rho)$, $\mathcal{X}', \mathcal{X}$, and m', m s.t. $x \notin dom(m), m'(x) = k \leq 0$.

hence we get the thesis.

$\Leftarrow$ By Lemma 1-(2),we have that the condition (WF-STUCK) above holds. Then, starting from the left, by Lemma 2 we have $at_\rho(x,i') \vdash^\star_{\mathcal{X}'} at_\rho(\rho(x),i)$ for some i', i such that $i - i' = k \leq 0$; by rule (AT-VAR) we have $at_\rho(\rho(x),i) \vdash at_\rho(x,i)$, and by Lemma 3 we have $at_\rho(x,j') \vdash^\star_{\mathcal{X}} at_\rho(s,j)$ for some j', j. If $i = j' + h$, $h \geq 0$, then by Lemma 1-(1) we have

$$at_\rho(x,i+k) \vdash^\star_{\mathcal{X}'} at_\rho(\rho(x),i) \vdash at_\rho(x,i) \vdash^\star_{\mathcal{X}} at_\rho(s,j+h)$$

If $j' = i + h$, $h \geq 0$, then by Lemma 1-(1) we have

$$at_\rho(x,i+k+h) \vdash^\star_{\mathcal{X}'} at_\rho(\rho(x),i) \vdash at_\rho(x,i+h) \vdash^\star_{\mathcal{X}} at_\rho(s,j).$$

In both cases, the derivation of $at_\rho(s,j)$ is infinite.

5 Related and future work

As mentioned in Sect. 1, our approach extends regular corecursion, where the semantics keeps track of method/function calls. Regular corecursion originated from *co-SLD resolution* [16,17,1,3], where already considered goals (up to unification), called *coinductive hypotheses*, are considered successful. Language constructs that support this programming style have also been proposed in the functional [11] and object-oriented [4,2] paradigm.

There have been a few attempts of extending the expressive power of regular corecursion. Notably, *structural resolution* [12,13] is a proposed operational semantics for logic programming where infinite derivations that cannot be built in finite time are generated lazily, and only partial answers are shown. Another approach is the work on infinite trees [6], where Courcelle introduces algebraic trees and equations as generalizations of regular ones.

For the operators considered in the calculus and some examples, our main sources of inspiration have been the works of Rutten [14], where a coinductive calculus of streams of real numbers is defined, and Hinze [9], where a calculus of generic streams is defined in a constructive way and implemented in Haskell.

The problem of ensuring well-defined corecursive definitions has been also considered in the context of type theory and proof assistants. We have shown in Sect. 4 that simple guarded definitions [5] do not work properly in case values are allowed to contain non constuctors as the tail operator; a more complex approach based on a type system has been proposed by Sacchini [15] for an extension of the calculus of constructions which is more expressive than that considered here;

however, as opposed to what happens with the judgment *wd* defined in Sect. 4, corecursive calls to the result of an application of tail are never well-typed even in case of well-defined streams as happens for the definition of `fib` as given in Sect. 3.

Lastly, D'Angelo et al. presented LOLA [8], a specification language for runtime monitoring that manipulates streams. The general idea behind the framework is to generate a set of output streams, starting from a given set of input streams. The main difference with respect to our work is that LOLA only allows streams with a finite number of elements. In this framework, well-formedness is checked by relying on a dependency graph, which keeps track of relations between the processed streams. The vertices of this graph are the streams, while the edges represent the dependencies between them. Each edge is weighted with a value ω to point the fact that a stream depends on another one shifted by ω positions. Then, the well-formedness constraint is that each closed-walk inside the graph must have a total weight different from 0. These syntactic constraints appear to be very similar to the approach we used for predicate *wd*.

Our main technical result is Theorem 1, stating that passing the well-definedness check performed at runtime for each function call is necessary and sufficient to prevent non-termination in accessing elements in the resulting stream at an arbitrary index. In future work, we plan to also prove soundness of the operational well-definedness with respect to its abstract definition. Completeness does not hold, as shown by the example `zeros() = repeat(0) [*] zeros()` which is not well-defined operationally, but admits as unique solution the stream of all zeros. On the other hand, the simplest operational characterization of equivalence of stream values is syntactic equivalence. This works for all the examples presented in this paper, but is, again, not complete with respect to the abstract definition, as illustrated below:

```
first(s) = s(0):first(s) // works with syntactic equivalence
first2(s) = s(0):first2(s(0):s^) // does not work
```

Indeed, we get an infinite derivation for `first2(repeat(1))` with call traces of increasing shape $\{\texttt{first2}(x) \mapsto y_1, \texttt{first2}(\texttt{1:}x\texttt{^}) \mapsto y_2, \texttt{first2}(\texttt{1:(1:}x\texttt{^)^}) \mapsto y_3, \ldots\}$ in the environment $\{x \mapsto 1 : x\}$. In future work we plan to investigate more expressive operational characterizations of equivalence.

References

1. Davide Ancona. Regular corecursion in Prolog. *Computer Languages, Systems & Structures*, 39(4):142–162, 2013.
2. Davide Ancona, Pietro Barbieri, Francesco Dagnino, and Elena Zucca. Sound regular corecursion in coFJ. In Robert Hirschfeld and Tobias Pape, editors, *ECOOP'20 - Object-Oriented Programming*, volume 166 of *LIPIcs*, pages 1:1–1:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.
3. Davide Ancona and Agostino Dovier. A theoretical perspective of coinductive logic programming. *Fundamenta Informaticae*, 140(3-4):221–246, 2015.
4. Davide Ancona and Elena Zucca. Corecursive Featherweight Java. In *FTfJP'12 - Formal Techniques for Java-like Programs*, pages 3–10. ACM Press, 2012.

5. Thierry Coquand. Infinite objects in type theory. In *Types for Proofs and Programs, International Workshop TYPES'93, Nijmegen, The Netherlands, May 24-28, 1993, Selected Papers*, pages 62–78, 1993.
6. Bruno Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25:95–169, 1983.
7. Francesco Dagnino, Davide Ancona, and Elena Zucca. Flexible coinductive logic programming. *Theory and Practice of Logic Programming*, 20(6):818–833, 2020. Issue for ICLP 2020.
8. Ben D'Angelo, Sriram Sankaranarayanan, César Sánchez, Will Robinson, Bernd Finkbeiner, Henny B. Sipma, Sandeep Mehrotra, and Zohar Manna. LOLA: runtime monitoring of synchronous systems. In *12th International Symposium on Temporal Representation and Reasoning (TIME 2005)*, pages 166–174, 2005.
9. Ralf Hinze. Concrete stream calculus: An extended study. *Journal of Functional Programming*, 20(5–6):463–535, 2010.
10. Jean-Baptiste Jeannin and Dexter Kozen. Computing with capsules. *Journal of Automata, Languages and Combinatorics*, 17(2-4):185–204, 2012.
11. Jean-Baptiste Jeannin, Dexter Kozen, and Alexandra Silva. CoCaml: Functional programming with regular coinductive types. *Fundamenta Informaticae*, 150:347–377, 2017.
12. Ekaterina Komendantskaya, Patricia Johann, and Martin Schmidt. A productivity checker for logic programming. In Manuel V. Hermenegildo and Pedro López-García, editors, *Logic-Based Program Synthesis and Transformation - LOPSTR 2016, Revised Selected Papers*, volume 10184 of *Lecture Notes in Computer Science*, pages 168–186. Springer, 2016.
13. Ekaterina Komendantskaya, John Power, and Martin Schmidt. Coalgebraic logic programming: from semantics to implementation. *J. Log. Comput.*, 26(2):745–783, 2016.
14. Jan J. M. M. Rutten. A coinductive calculus of streams. *Mathematical Structures in Computer Science*, 15(1):93–147, 2005.
15. Jorge Luis Sacchini. Type-based productivity of stream definitions in the calculus of constructions. In *Symposium on Logic in Computer Science, LICS 2013*, pages 233–242, 2013.
16. Luke Simon. *Extending logic programming with coinduction.* PhD thesis, University of Texas at Dallas, 2006.
17. Luke Simon, Ajay Bansal, Ajay Mallya, and Gopal Gupta. Co-logic programming: Extending logic programming with coinduction. In Lars Arge, Christian Cachin, Tomasz Jurdzinski, and Andrzej Tarlecki, editors, *Automata, Languages and Programming, 34th International Colloquium, ICALP 2007*, volume 4596 of *Lecture Notes in Computer Science*, pages 472–483. Springer, 2007.