Enhancing Expressivity of Checked Corecursive Streams

Davide Ancona(✉), Pietro Barbieri, and Elena Zucca

DIBRIS, University of Genova, Genoa, Italy
davide.ancona@unige.it

Abstract. We propose a novel approach to stream definition and manipulation. Our solution is based on two key ideas. Regular corecursion, which avoids non termination by detecting cyclic calls, is enhanced, by allowing in equations defining streams other operators besides the stream constructor. In this way, some non-regular streams are definable. Furthermore, execution includes a runtime check to ensure that the stream generated by a function call is well-defined, in the sense that access to an arbitrary index always succeeds. We extend the technique beyond the simple stream operators considered in previous work, notably by adding an *interleaving* combinator which has a non-trivial recursion scheme.

Keywords: Operational semantics · Stream programming · Runtime checking

1 Introduction

Applications often deal with data structures which are conceptually infinite; among those *data streams* (unbounded sequences of data) are a paradigmatic example, important in several application domains as the Internet of Things. Lazy evaluation is a well-established and widely-used solution to data stream generation and processing, supported, e.g., in Haskell, and in most stream libraries offered by mainstream languages, as `java.util.stream`. In this approach, data streams can be defined as the result of an arbitrary function. For instance, in Haskell we can write

```
one_two = 1:2:one_two   -- 1:2:1:2:1: ...
from n = n:from(n+1)   -- n:n+1:n+2: ...
```

Functions which only need to inspect a finite portion of the structure, e.g., getting the i-th element, can be correctly implemented, thanks to the lazy evaluation strategy as exemplified below.

```
get_elem 3 (one_two)   -- evaluates to 2
get_elem 3 (from 5)   -- evaluates to 7
```

More recently, another approach has been proposed [2,11,14,19], called *regular corecursion*, which exploits the fact that streams as `one_two` above are

M. Hanus and A. Igarashi (Eds.): FLOPS 2022, LNCS 13215, pp. 1–18, 2022.
https://doi.org/10.1007/978-3-030-99461-7_1

periodic, a.k.a. *regular* following the terminology in [8], meaning that the term `1:2:1:2:1: ...` is infinite but has a finite number of subterms. Regular streams can be actually represented at runtime by a finite set of equations involving only the stream constructor, in the example $x = 1 : 2 : x$. Furthermore, function definitions are *corecursive*, meaning that they do not have the standard inductive semantics; indeed, even though the evaluation strategy is call-by-value, thanks to the fact that pending function calls are tracked, cyclic calls are detected, avoiding in this case non-termination.

For instance, with regular corecursion[1] we have:

```
one_two() = 1:2:one_two()
from(n) = n:from(n+1)
get_elem(3,one_two())   -- evaluates to 2
get_elem (3,from(5))   -- leads to non-termination
```

Despite their differences, in both approaches programmers are allowed to write intuitively ill-formed definitions such as `bad_stream() = bad_stream()`; any access to indexes of the stream returned by this function leads to non-termination both with lazy evaluation and regular corecursion. However, while in the regular case it is simple to reject the result of calling `bad_stream` by checking a guardedness syntactic condition, the Haskell compiler does not complain if one calls such a function. In this paper, we propose a novel approach to stream generation and manipulation, providing, in a sense, a middle way between those described above. Our solution is based on two key ideas:

- Corecursion is enhanced, by allowing in stream equations other typical operators besides the stream constructor; in this way, some non-regular streams are supported. For instance, we can define `from(n)=n:(from(n)[+]repeat(1))`, with `[+]` the pointwise addition and `repeat` defined by `repeat(n)=n:repeat(n)`.
- Execution includes a runtime check which rejects the stream generated by a function call if it is ill-formed, in the sense that access to an index could possibly diverge. For instance, the call `bad_stream()` raises a runtime error.

In this way we achieve a convenient trade-off between expressive power and reliability; indeed, we do not have the full expressive power of Haskell, where we can manipulate streams generated as results of arbitrary functions, but, clearly, the well-definedness check described above would be not decidable. On the other hand, we significantly augment the expressive power of regular corecursion, allowing several significant non-regular streams, at the price of making the well-definedness check non-trivial, but still decidable.

The main formal results are (1) Theorem 1 stating the soundness of the runtime check; (2) Theorem 2 stating that the optimized definition of the runtime check in Sect. 5 is equivalent to the simpler one given in Sect. 4. In particular, for contribution (1) the interleaving operator requires a more involved proof in comparison with [3] (see Sect. 6), while for (2) we show that the optimized definition improves the time complexity from $O(N^2)$ to $O(N \log N)$.

[1] Here we use the syntax of our calculus, where, differently from Haskell, functions are uncurried, that is, take as arguments possibly empty tuples delimited by parentheses.

In Sect. 2 we formally define the calculus, in Sect. 3 we show examples, in Sect. 4 we define the well-formedness check, and in Sect. 5 its optimized version. Finally, in Sect. 6 we discuss related and further work. More examples of derivations and omitted proofs can be found in the extended version [4].

2 Stream Calculus

Figure 1 shows the syntax of the calculus.

$$
\begin{array}{lll}
\overline{fd} & ::= fd_1 \ldots fd_n & \text{program} \\
fd & ::= f(\overline{x}) = se & \text{function declaration} \\
e & ::= se \mid ne \mid be & \text{expression} \\
se & ::= x \mid \texttt{if}\ be\ \texttt{then}\ se_1\ \texttt{else}\ se_2 \mid ne : se \mid se\hat{}\ \mid se_1\, op\ se_2 \mid f(\overline{e}) & \text{stream expression} \\
ne & ::= x \mid se(ne) \mid ne_1\, nop\, ne_2 \mid 0 \mid 1 \mid 2 \mid \ldots & \text{numeric expression} \\
be & ::= x \mid \texttt{true} \mid \texttt{false} \mid \ldots & \text{boolean expression} \\
op & ::= [nop] \mid \| & \text{binary stream operator} \\
nop & ::= + \mid - \mid * \mid / & \text{numeric operator}
\end{array}
$$

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, conditionals, expressions built by stream operators, and function calls. We consider the following stream operators: constructor (prepending a numeric element), tail, pointwise arithmetic operators, and interleaving. Numeric expressions include the access to the i-th[2] element of a stream. We use $\overline{fd}$ to denote a sequence $fd_1, \ldots, fd_n$ 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 can be represented in a finite way
2. evaluation keeps trace of already considered function calls

To obtain (1), our approach is inspired by *capsules* [13], which are expressions supporting cyclic references. That is, the *result* 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, the pointwise arithmetic and the interleaving operators, and ρ is an *environment* mapping 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 $dom(\rho)$ the domain of ρ, 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).

[2] For simplicity, here indexing and numeric expressions coincide.

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

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

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

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

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

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

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

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

$$\text{(AT-}\|\text{-EVEN)}\frac{at_\rho(s_1, i) = n}{at_\rho(s_1 \| s_2, 2i) = n} \qquad \text{(AT-}\|\text{-ODD)}\frac{at_\rho(s_2, i) = n}{at_\rho(s_1 \| s_2, 2i+1) = n}$$

Fig. 2. Stream calculus: operational semantics

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 $\overline{v}$.
- $\mathit{fbody}(f)$ returns the pair of the parameters and the body of the declaration of f, if any, in the assumed program.

Intuitively, a closed result (s, ρ) is well-defined if it denotes a unique stream, and a closed environment ρ is well-defined if, for each $x \in \mathit{dom}(\rho)$, (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). This notion 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 (op), arguments are evaluated and the stream operator is applied without any further evaluation. That is, we treat all these operators as constructors.

The rules for function call are based on a mechanism of cycle detection [2]. Evaluation of arguments is handled by a separate rule (args), whereas the following two rules handle (evaluated) calls.

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 of such variable with the result of the evaluation of the body keeps the environment well-defined, as expressed by the judgment $\mathit{wd}(\rho, x, s)$, which will be defined in Sect. 4. 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. An example of stuck derivation is shown in [4].

Rule (corec) is applied when a call is considered for the second time, as expressed by the first side condition. 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 the program `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 happens for $\texttt{f()}, \emptyset, \{\texttt{f()} \mapsto x, \texttt{g()} \mapsto y\} \Downarrow (x, \emptyset)$ and $\texttt{g()}, \emptyset, \{\texttt{f()} \mapsto x\} \Downarrow (y, \{y \mapsto 1 : x\})$. The full derivation can be found in [4].

Finally, rule (at) computes the i-th element of a stream expression. After evaluating the arguments, the result is obtained by the auxiliary judgment $at_\rho(s, i) = n$, whose straightforward definition is at the bottom of the figure.

Rules (AT-)$\|$(-EVEN) and (AT-)$\|$(-ODD) define the behaviour of the interleaving operator, which merges two streams together by alternating their elements.

When evaluating $at_\rho(s, i)$, if s is a variable free in the environment, then execution is stuck; again, an implementation should raise a runtime error instead.

3 Examples

First we show some simple examples, to explain how corecursive definitions work. 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 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 `0` $: x$ is associated in the environment with the result x of the initial call. In the sequel, we will use `[k]` as a shorthand for `repeat(`k`)`. 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.

$$\dfrac{\dfrac{(\text{VALUE})\quad \dfrac{}{\texttt{repeat(0)}, \emptyset, \{\texttt{repeat(0)} \mapsto x\} \Downarrow (x, \emptyset)}\,(\text{COREC})}{\texttt{0} : \texttt{repeat(0)}, \emptyset, \{\texttt{repeat(0)} \mapsto x\} \Downarrow (\texttt{0} : x, \emptyset)}\,(\text{CONS})}{\texttt{repeat(0)}, \emptyset, \emptyset \Downarrow (x, \{x \mapsto \texttt{0} : x\})}\,(\text{INVK})$$

$$\dfrac{\dfrac{(\text{VALUE})\quad \dfrac{\dfrac{(\text{VALUE})\quad \dfrac{}{\texttt{one\_two()}, \emptyset, \{\texttt{one\_two()} \mapsto x,\ \texttt{two\_one()} \mapsto y\} \Downarrow (x, \emptyset)}\,(\text{COREC})}{\texttt{2} : \texttt{one\_two()}, \emptyset, \{\texttt{one\_two()} \mapsto x,\ \texttt{two\_one()} \mapsto y\} \Downarrow (\texttt{2} : x, \emptyset)}\,(\text{CONS})}{\texttt{two\_one()}, \emptyset, \{\texttt{one\_two()} \mapsto x\} \Downarrow (y, \{y \mapsto \texttt{2} : x\})}\,(\text{INVK})}{\texttt{1} : \texttt{two\_one()}, \emptyset, \{\texttt{one\_two()} \mapsto x\} \Downarrow (\texttt{1} : y, \{y \mapsto \texttt{2} : x\})}\,(\text{CONS})}{\texttt{one\_two()}, \emptyset, \emptyset \Downarrow (x, \{x \mapsto \texttt{1} : y,\ y \mapsto \texttt{2} : x\})}\,(\text{INVK})$$

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

The definition of `nat` uses 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 corecursive. For instance, the call `fact()` returns $(z, z \mapsto 1 : ((x[+]y)[*]z), x \mapsto 0 : (x[+]y'), y \mapsto 1 : y, y' \mapsto 1 : y')$. 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 next few examples show applications of the interleaving operator.

```
dup_occ() = 0:1:(dup_occ() || dup_occ())
```

Function `dup_occ()` generates the stream which alternates sequences of occurrences of `0` and `1`, with the number of repetitions of the same number duplicated at each step, that is, `(0:1:0:0:1:1:0:0:0:0...)`.

A more involved example shows a different way to generate the stream of all powers of `2` starting from 2^1:

```
pow_two=2:4:8:((pow_two^^[*]pow_two)||(pow_two^^[*]pow_two^))
```

The following definition is an instance of a schema generating the infinite sequence of labels obtained by a breadth-first visit of an infinite complete binary tree where the labels of children are defined in terms of that of their parent.

```
bfs_index() = 1:((bfs_index()[*][2])||(bfs_index()[*][2][+][1]))
```

In particular, the root is labelled by `1`, and the left and right child of a node with label `i` are labelled by `2*i` and `2*i+1`, respectively. Hence, the generated stream is the sequence of natural numbers starting from `1`, as it happens in the array implementation of a binary heap.

In the other instance below, the root is labelled by `0`, and children are labelled with `i+1` if their parent has label `i`. That is, nodes are labelled by their level.

```
bfs_level() = 0:((bfs_level()[+][1])||(bfs_level()[+][1]))
```

In this case, the generated stream is more interesting; indeed, `bfs_level()(n) = floor(log`$_2$`(n+1))`.

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 terms of the Taylor series of the exponential; hence, by accessing the i-th element of the stream, we have the following approximation of the series:

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

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 data in windows of length `n`:

```
aggr(n,s) = if n<=0 then [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 windows of length `n`:

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

4 Well-Definedness Check

A key feature of our approach is the runtime check ensuring that the stream generated by a function call is well-defined, see the side condition $wd(\rho', x, s)$ in (INVK); in this section we formally define the corresponding judgment and prove its soundness. Before doing this, we provide, for reference, a formal abstract definition of well-definedness.

Intuitively, an environment is well-defined if each variable in its domain denotes a unique stream. 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.

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

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

$$
\begin{array}{l}
([\![s\,\hat{}\,]\!]\theta)(i) = [\![s]\!]\theta(i+1) \qquad i \geq 0\\
([\![s_1[\mathit{nop}]s_2]\!]\theta)(i) = [\![s_1]\!]\theta(i)\ \mathit{nop}\ [\![s_2]\!]\theta(i) \qquad i \geq 0\\
([\![s_1 \| s_2]\!]\theta)(2i) = [\![s_1]\!]\theta(i) \qquad i \geq 0\\
([\![s_1 \| s_2]\!]\theta)(2i+1) = [\![s_2]\!]\theta(i) \qquad i \geq 0
\end{array}
$$

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

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

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

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

In Fig. 3 we provide the operational characterization of well-definedness. The judgment $\mathit{wd}(\rho, x, s)$ used in the side condition of rule (INVK) holds if $\mathsf{wd}_{\rho'}(x, \emptyset)$ holds, with $\rho' = \rho\{x \mapsto v\}$. The judgment $\mathsf{wd}_{\rho}(s, \emptyset)$ means well-definedness of a result. That is, restricting the domain of ρ to the variables reachable from s (that is, either occurring in s, or, transitively, in values associated with reachable variables) we get a well-defined environment; thus, $\mathit{wd}(\rho, x, s)$ holds if adding the association of s with x preserves well-definedness of ρ.

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

$$
\textsc{(main)}\ \frac{\mathsf{wd}_{\rho'}(x, \emptyset)}{\mathit{wd}(\rho, x, v)}\ \rho' = \rho\{x \mapsto v\}
\qquad
\textsc{(wd-var)}\ \frac{\mathsf{wd}_{\rho}(\rho(x), m\{x \mapsto 0\})}{\mathsf{wd}_{\rho}(x, m)}\ x \notin \mathit{dom}(m)
$$

$$
\textsc{(wd-corec)}\ \frac{}{\mathsf{wd}_{\rho}(x, m)}\ \begin{array}{l} x \in \mathit{dom}(\rho)\\ m(x) > 0 \end{array}
\qquad
\textsc{(wd-delay)}\ \frac{\mathsf{wd}_{\rho}(\rho(x), m\{x \mapsto 0\})}{\mathsf{wd}_{\rho}(x, m)}\ m(x) > 0
$$

$$
\textsc{(wd-fv)}\ \frac{}{\mathsf{wd}_{\rho}(x, m)}\ x \notin \mathit{dom}(\rho)
\qquad
\textsc{(wd-cons)}\ \frac{\mathsf{wd}_{\rho}(s, m^{+1})}{\mathsf{wd}_{\rho}(n : s, m)}
\qquad
\textsc{(wd-tail)}\ \frac{\mathsf{wd}_{\rho}(s, m^{-1})}{\mathsf{wd}_{\rho}(s\,\hat{}\,, m)}
$$

$$
\textsc{(wd-nop)}\ \frac{\mathsf{wd}_{\rho}(s_1, m) \quad \mathsf{wd}_{\rho}(s_2, m)}{\mathsf{wd}_{\rho}(s_1[\mathit{nop}]s_2, m)}
\qquad
\textsc{(wd-}\|\textsc{)}\ \frac{\mathsf{wd}_{\rho}(s_1, m) \quad \mathsf{wd}_{\rho}(s_2, m^{+1})}{\mathsf{wd}_{\rho}(s_1 \| s_2, m)}
$$

Fig. 3. Operational definition of well-definedness

The additional argument m in the judgment $\mathsf{wd}_\rho(s, m)$ is a map from variables to integer 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 (WD-VAR), when a variable x defined in the environment is found the first time, it is added in the map with initial value 0 before propagating the check to the associated value. In rule (WD-COREC), when it is found the second time, it is checked that constructors and right operands of interleave are traversed more times than tail operators, and if it is the case the variable is considered well-defined. Rule (WD-DELAY), which is only added for the purpose of the soundness proof and should be not part of an implementation[3], performs the same check but then considers the variable occurrence as it is was the first, so that success of well-definedness is delayed. Note that rules (WD-VAR), (WD-COREC), and (WD-DELAY) can only be applied if $x \in dom(\rho)$; in rule (WD-COREC), this explicit side condition could be omitted since satisfied by construction of the proof tree.

In rule (WD-FV), a free variable is considered well-defined.[4] In rules (WD-CONS) and (WD-TAIL) the value associated with a variable is incremented/decremented by one, respectively, before propagating the check to the subterm. In rule (WD-NOP) the check is simply propagated to the subterms. In rule (WD-)$\|$, the check is also propagated to the subterms, but on the right-hand side the value associated with a variable is incremented by one before propagation; this reflects the fact that, in the worst case, $at_\rho(s_1 \| s_2, i) = at_\rho(s_1, i)$, and this happens only for $i = 0$, while for odd indexes i we have that $at_\rho(s_1 \| s_2, i) = at_\rho(s_2, i - k)$, with $k \geq 1$; more precisely, $k = 1$ only when $i = 1$; for all indexes $i > 1$ (both even and odd), $k > 1$. For instance, the example `s() = 1:(s()`$\|$`s()^)`, which has the same semantics as `[1]`, would be considered not well-defined if we treated the interleaving as the pointwise arithmetic operators.

Note that the rules in Fig. 3 can be immediately turned into an algorithm which, given a stream value s, always terminates either successfully (finite proof tree), or with failure (no proof tree can be constructed). On the other hand, the rules in Fig. 2 defining the $at_\rho(s, i) = n$ judgment can be turned into an algorithm which can possibly diverge (infinite proof tree).

Two examples of derivation of well-definedness and access to the i-th element can be found in [4] for the results obtained by evaluating the calls `nat()` and `bfs_level()`, respectively, with `nat` and `bfs_level` defined as in Sect. 3. Below we show an example of failing derivation:

As depicted in Fig. 4, the check succeeds for the left-hand component of the interleaving operator, while the proof tree cannot be completed for the other side. Indeed, the double application of the tail operator makes undefined access to stream elements with index greater than 1, since the evaluation of $at_\rho(x, 2)$ infinitely triggers the evaluation of itself.

[3] Indeed, it does not affect derivability, see Lemma 4 in the following.

[4] Non-well-definedness can only be detected on closed results.

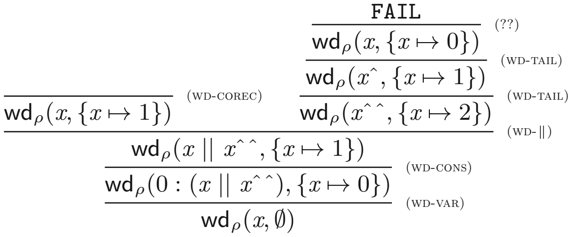

FAIL
(??)
$\mathsf{wd}_\rho(x, \{x \mapsto 0\})$
(WD-TAIL)
$\mathsf{wd}_\rho(x\hat{}, \{x \mapsto 1\})$
(WD-COREC)
$\mathsf{wd}_\rho(x, \{x \mapsto 1\})$
(WD-TAIL)
$\mathsf{wd}_\rho(x\hat{}\hat{}, \{x \mapsto 2\})$
(WD-||)
$\mathsf{wd}_\rho(x \mid\mid x\hat{}\hat{}, \{x \mapsto 1\})$
(WD-CONS)
$\mathsf{wd}_\rho(0 : (x \mid\mid x\hat{}\hat{}), \{x \mapsto 0\})$
(WD-VAR)
$\mathsf{wd}_\rho(x, \emptyset)$

Fig. 4. Failing derivation for $\rho = \{x \mapsto 0 : (x \mid\mid x\hat{}\hat{})\}$

To formally express and prove that well-definedness of a result implies termination of access to an arbitrary index, we introduce some definitions and notations. First of all, since the 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 $\mathsf{wd}_\rho(s', m') \vdash \mathsf{wd}_\rho(s, m)$ to mean that $\mathsf{wd}_\rho(s', m')$ is a premise of a (meta-)rule where $\mathsf{wd}_\rho(s, m)$ is the conclusion, and $\vdash^\star$ for the reflexive and transitive closure of this relation.

Lemma 1.

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

Lemma 2. *If $at_\rho(x, i') \vdash^\star at_\rho(s', i)$, and $\mathsf{wd}_\rho(s', m) \vdash^\star \mathsf{wd}_\rho(s, \emptyset)$ with $\mathsf{wd}_\rho(s, \emptyset)$ derivable, and $x \in dom(m)$, then*

$$\mathsf{wd}_\rho(x, m') \vdash^\star \mathsf{wd}_\rho(s', m) \text{ for some } m' \text{ such that } m'(x) - m(x) \leq i - i'.$$

Proof. The proof is by induction on the length of the path in $at_\rho(x, i') \vdash^\star at_\rho(s', i)$.

Base. The length of the path is 0, hence we have $at_\rho(x, i) \vdash^\star at_\rho(x, i)$. We also have $\mathsf{wd}_\rho(x, m) \vdash^\star \mathsf{wd}_\rho(x, m)$, and we get the thesis since $m(x) = m(x) + i - i$.

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

(at-var) We have $at_\rho(x, i') \vdash^\star at_\rho(\rho(y), i) \vdash at_\rho(y, i)$. There are two cases:

- If $y \notin dom(m)$ (hence $y \neq x$), we have $\mathsf{wd}_\rho(\rho(y), m\{y \mapsto 0\}) \vdash \mathsf{wd}_\rho(y, m)$ by rule (WD-VAR), the premise is derivable, hence by inductive hypothesis we have $\mathsf{wd}_\rho(x, m') \vdash^\star \mathsf{wd}_\rho(\rho(y), m\{y \mapsto 0\})$, and $m'(x) \leq m\{y \mapsto 0\}(x) + i - i' = m(x) + i - i'$, hence we get the thesis.
- If $y \in dom(m)$, then it is necessarily $m(y) > 0$, otherwise, by Lemma 1-(1), $\mathsf{wd}_\rho(s, \emptyset)$ would not be derivable. Hence, we have $\mathsf{wd}_\rho(\rho(y), m\{y \mapsto 0\}) \vdash \mathsf{wd}_\rho(y, m)$ by rule (WD-DELAY), hence by

inductive hypothesis we have $\mathsf{wd}_\rho(x, m') \vdash^\star \mathsf{wd}_\rho(\rho(y), m\{y \mapsto 0\})$, and $m'(x) \leq m\{y \mapsto 0\}(x) + i - i'$. There are two subcases:

- If $y \neq x$, then $m\{y \mapsto 0\}(x) = m(x)$, and we get the thesis as in the previous case.
- If $y = x$, then $m\{x \mapsto 0\}(x) = 0$, hence $m'(x) \leq i - i' \leq m(x) + i - i'$, since $m(x) > 0$.

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

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

(at-tail) This case is symmetric to the previous one.

(at-nop) We have $at_\rho(s_1[op]s_2, i)$, and either $at_\rho(x, i') \vdash^\star at_\rho(s_1, i)$, or $at_\rho(x, i') \vdash^\star at_\rho(s_2, i)$. Assume the first case holds, the other is analogous. Moreover, we can derive $\mathsf{wd}_\rho(s_1[op]s_2, m)$ by rule (wd-nop), and by inductive hypothesis we also have $\mathsf{wd}_\rho(x, m') \vdash^\star \mathsf{wd}_\rho(s_1, m)$, with $m'(x) \leq m(x) + i - i'$, hence we get the thesis.

(at-$\|$-even) We have $at_\rho(s_1\|s_2, 2i)$ and $at_\rho(x, i') \vdash^\star at_\rho(s_1, i)$. By inductive hypothesis, we have $\mathsf{wd}_\rho(x, m') \vdash^\star \mathsf{wd}_\rho(s_1, m)$, with $m'(x) \leq m(x)+i-i'$. Moreover, $\mathsf{wd}_\rho(s_1, m) \vdash \mathsf{wd}_\rho(s_1\|s_2, m)$ holds by rule (wd-)$\|$, hence we have $\mathsf{wd}_\rho(x, m') \vdash^\star \mathsf{wd}_\rho(s_1\|s_2, m)$ with $m'(x) \leq m(x) + 2i - i'$ and, thus, the thesis.

(at-$\|$-odd) We have $at_\rho(s_1\|s_2, 2i+1)$ and $at_\rho(x, i') \vdash^\star at_\rho(s_2, i)$. By inductive hypothesis, we have $\mathsf{wd}_\rho(x, m') \vdash^\star \mathsf{wd}_\rho(s_2, m^{+1})$, with $m'(x) \leq m^{+1}(x) + i - i'$. Moreover, $\mathsf{wd}_\rho(s_2, m) \vdash \mathsf{wd}_\rho(s_1\|s_2, m)$ holds by rule (wd-)$\|$, hence we have $\mathsf{wd}_\rho(x, m') \vdash^\star \mathsf{wd}_\rho(s_1\|s_2, m)$ with $m'(x) \leq m(x) + 2i + 1 - i'$ and, thus, the thesis.

Lemma 3. *If $at_\rho(x, i') \vdash^\star at_\rho(s, i)$, and $\mathsf{wd}_\rho(s, \emptyset)$ derivable, then*

$\mathsf{wd}_\rho(x, m) \vdash^\star \mathsf{wd}_\rho(s, \emptyset)$ *for some m such that $x \notin dom(m)$.*

Proof. Easy variant of the proof of Lemma 2.

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

Proof. Assume by contradiction that $at_\rho(s, j)$ has an infinite derivation for some j, and $\mathsf{wd}_\rho(s, \emptyset)$ is derivable. By Lemma 1–(2), the following condition holds:

$$\text{(AT-}\infty\text{)}\quad at_\rho(x, i+k) \vdash^\star at_\rho(\rho(x), i) \vdash at_\rho(x, i) \vdash^\star at_\rho(s, j)$$
$$\text{for some } x \in dom(\rho), \text{ and } i, k \geq 0.$$

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

$$\text{(WD-STUCK)}\quad \mathsf{wd}_\rho(x, m') \vdash^\star \mathsf{wd}_\rho(\rho(x), m\{x \mapsto 0\}) \vdash \mathsf{wd}_\rho(x, m) \vdash^\star \mathsf{wd}_\rho(s, \emptyset)$$
$$\text{for some } x \in dom(\rho), \text{ and } m', m \text{ s.t. } x \notin dom(m), m'(x) \leq -k \leq 0.$$

hence this is absurd by Lemma 1-(1).

5 An Optimized Algorithm for Well-Definedness

The definition of well-definedness in Fig. 3 can be easily turned into an algorithm, since, omitting rule (WD-DELAY), at each point in the derivation there is at most one applicable rule. Now we will discuss its time complexity, assuming that insertion, update and lookup are performed in constant time. It is easy to see that when we find a stream constructor we need to perform an update of the map ρ for every variable in its domain. If we consider the following environment:

$$\rho = (x_0, \{x_0 \mapsto 0 : x_1, x_1 \mapsto 0 : x_2, x_2 \mapsto 0 : x_3, x_3 \mapsto 0 : x_4,\ \dots\ , x_n \mapsto 0 : x_0\})$$

we get the derivation presented in Fig. 5. Here, the number of constructor occurrences for which we have to perform an update of all variables in the domain of the map is linearly proportional to the number N of nodes in the derivation tree; since the domain is increased by one for each new variable, and the total number of variables is again linearly proportional to N, it is easy to see that we have a time complexity quadratic in N.

$$\begin{array}{cl}
\vdots & \\ \hline
\mathsf{wd}_\rho(x_3, \{x_0 \mapsto 3, x_1 \mapsto 2, x_2 \mapsto 1\}) & (\textsc{wd-var}) \\ \hline
\mathsf{wd}_\rho(0 : x_3, \{x_0 \mapsto 2, x_1 \mapsto 1, x_2 \mapsto 0\}) & (\textsc{wd-cons}) \\ \hline
\mathsf{wd}_\rho(x_2, \{x_0 \mapsto 2, x_1 \mapsto 1\}) & (\textsc{wd-var}) \\ \hline
\mathsf{wd}_\rho(0 : x_2, \{x_0 \mapsto 1, x_1 \mapsto 0\}) & (\textsc{wd-cons}) \\ \hline
\mathsf{wd}_\rho(x_1, \{x_0 \mapsto 1\}) & (\textsc{wd-var}) \\ \hline
\mathsf{wd}_\rho(0 : x_1, \{x_0 \mapsto 0\}) & (\textsc{wd-cons}) \\ \hline
\mathsf{wd}_\rho(x_0, \emptyset) & (\textsc{wd-var})
\end{array}$$

Fig. 5. wd worst case

We propose now an optimized version of the well-definedness check, having a time complexity of $O(N \log N)$. On the other hand, the version we provided in Fig. 3 is more abstract, hence more convenient for the proof of Theorem 1.

In the optimized version, given in Fig. 6, the judgment has shape $\mathsf{owd}_\rho(s, \mathsf{m}, \pi)$, where π represents a path in the proof tree where each element corresponds to a visit of either the constructor or the right operand of interleave (value 1 for both) or the tail operator (value -1), and m associates with each variable an index (starting from 0) corresponding to the point in the path π where the variable was found the first time. The only operation performed on a path π is the addition $\pi \cdot b$ of an element b at the end.

In rule (MAIN), both the map and the path are initially empty. In rule (OWD-VAR), a variable x defined in the environment, found for the first time, is added in the map with as index the length of the current path. In rule (OWD-COREC),

$$\begin{array}{lll} \mathsf{m} ::= x_1 \mapsto i_1 \ldots x_n \mapsto i_k & (i \geq 0) & \text{map from variables to indexes} \\ \pi \ ::= b_1 b_2 \ldots b_n & & \text{sequence of either 1 or -1} \end{array}$$

$$\text{(MAIN)}\ \frac{\mathsf{owd}_{\rho'}(x, \emptyset, \epsilon)}{wd(\rho, x, v)}\ \rho' = \rho\{x \mapsto v\} \qquad \text{(OWD-VAR)}\ \frac{\mathsf{owd}_{\rho}(\rho(x), \mathsf{m}\{x \mapsto i\}, \pi)}{\mathsf{owd}_{\rho}(x, \mathsf{m}, \pi)}\ \begin{array}{l} x \notin dom(\mathsf{m}) \\ i = length(\pi) \end{array}$$

$$\text{(OWD-COREC)}\ \frac{}{\mathsf{owd}_{\rho}(x, \mathsf{m}, \pi)}\ \begin{array}{l} x \in dom(\mathsf{m}) \\ sum(\mathsf{m}(x), \pi) > 0 \end{array} \qquad \text{(OWD-FV)}\ \frac{}{\mathsf{owd}_{\rho}(x, \mathsf{m}, \pi)}\ x \notin dom(\rho)$$

$$\text{(OWD-CONS)}\ \frac{\mathsf{owd}_{\rho}(s, \mathsf{m}, \pi \cdot 1)}{\mathsf{owd}_{\rho}(n : s, \mathsf{m}, \pi)} \qquad \text{(OWD-TAIL)}\ \frac{\mathsf{owd}_{\rho}(s, \mathsf{m}, \pi \cdot (-1))}{\mathsf{owd}_{\rho}(s\hat{}, \mathsf{m}, \pi)}$$

$$\text{(OWD-NOP)}\ \frac{\mathsf{owd}_{\rho}(s_1, \mathsf{m}, \pi) \quad \mathsf{owd}_{\rho}(s_2, \mathsf{m}, \pi)}{\mathsf{owd}_{\rho}(s_1[nop]s_2, \mathsf{m}, \pi)} \qquad \text{(OWD-}\|\text{)}\ \frac{\mathsf{owd}_{\rho}(s_1, \mathsf{m}, \pi) \quad \mathsf{owd}_{\rho}(s_2, \mathsf{m}, \pi \cdot 1)}{\mathsf{owd}_{\rho}(s_1 \| s_2, \mathsf{m}, \pi)}$$

$$\text{(SUM-0)}\ \frac{sum(\pi) = n}{sum(0, \pi) = n} \qquad \text{(SUM-N)}\ \frac{sum(n-1, b_2 \ldots b_n) = n'}{sum(n, b_1 b_2 \ldots b_n) = n'}\ n > 0$$

$$\text{(SUM-B)}\ \frac{}{sum(\epsilon) = 0} \qquad \text{(SUM-I)}\ \frac{sum(b_2 \ldots b_n) = n}{sum(b_1 b_2 \ldots b_n) = b_1 + n}$$

Fig. 6. Optimized operational definition of well-definedness

when the same variable is found the second time, the auxiliary function *sum* checks that more constructors and right operands of interleave have been traversed than tail operators (see below). In rule (OWD-FV), a free variable is considered well-defined as in the corresponding rule in Fig. 3. In rules (OWD-CONS), (OWD-TAIL) and (OP-WD), the value corresponding to the traversed operator is added at the end of the path (1 for the constructor and the right operand of interleave, −1 for the tail operator). Lastly, rules (OWD-NOP) behaves in a similar way as in Fig. 3. The semantics of the auxiliary function *sum* is straightforward: starting from the point in the path where the variable was found the first time, the sum of all the elements is returned.

Let us now consider again the example above:

$$\rho = (x_0, \{x_0 \mapsto 0 : x_1, x_1 \mapsto 0 : x_2, x_2 \mapsto 0 : x_3, x_3 \mapsto 0 : x_4,\ \ldots\ , x_n \mapsto 0 : x_0\})$$

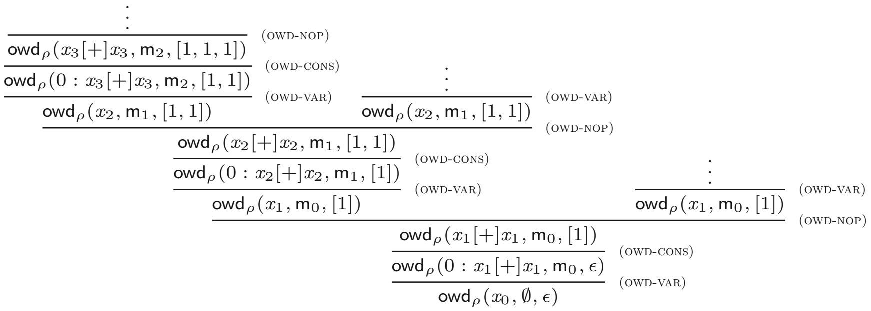
By the new predicate owd, we get a derivation tree of the same shape as in Fig. 5. However, *sum* is applied to the path π only at the leaves, and the length of π is linearly proportional to the depth of the derivation tree, which coincides with the number N of nodes in this specific case; hence, the time complexity to compute $sum(0, \pi)$ (that is, $sum(\mathsf{m}(x_0), \pi)$) is linear in N. Finally, since for inner nodes only constant time operations are performed[5] (addition at the end of the path, and map insertion and lookup), the overall time complexity is linear in N.

[5] This holds for any valid derivation tree and not for this specific case.

As worst case in terms of time complexity for the predicate owd, consider

$$\rho_i = (x_0, \{x_0 \mapsto 0 : x_1[+]x_1, x_1 \mapsto 0 : x_2[+]x_2, x_2 \mapsto 0 : x_3[+]x_3, \dots\ , x_i \mapsto 0 : x_0\})$$

The derivation tree for this environment is shown in Fig. 7, where m_i abbreviates the map $\{x_0 \mapsto 0, x_1 \mapsto 1, \dots, x_i \mapsto i\}$.



$\vdots$
(OWD-NOP)
$\mathsf{owd}_\rho(x_3[+]x_3, \mathsf{m}_2, [1,1,1])$
(OWD-CONS)
$\mathsf{owd}_\rho(0 : x_3[+]x_3, \mathsf{m}_2, [1,1])$
(OWD-VAR)
$\mathsf{owd}_\rho(x_2, \mathsf{m}_1, [1,1])$
$\vdots$
(OWD-VAR)
$\mathsf{owd}_\rho(x_2, \mathsf{m}_1, [1,1])$
(OWD-NOP)
$\mathsf{owd}_\rho(x_2[+]x_2, \mathsf{m}_1, [1,1])$
(OWD-CONS)
$\mathsf{owd}_\rho(0 : x_2[+]x_2, \mathsf{m}_1, [1])$
(OWD-VAR)
$\mathsf{owd}_\rho(x_1, \mathsf{m}_0, [1])$
$\vdots$
(OWD-VAR)
$\mathsf{owd}_\rho(x_1, \mathsf{m}_0, [1])$
(OWD-NOP)
$\mathsf{owd}_\rho(x_1[+]x_1, \mathsf{m}_0, [1])$
(OWD-CONS)
$\mathsf{owd}_\rho(0 : x_1[+]x_1, \mathsf{m}_0, \epsilon)$
(OWD-VAR)
$\mathsf{owd}_\rho(x_0, \emptyset, \epsilon)$

Fig. 7. owd worst case

As already noticed, for inner nodes only constant time operations are performed, and the length of the paths in the leaves is linearly proportional to the depth D of the derivation tree; however, in this worst case the number of leaves is not just one, but is linearly proportional to the total number N of nodes in the derivation tree, hence the depth D is linearly proportional to $\log N$. Therefore the overall time complexity is $O(N \cdot D)$, that is, $O(N \cdot \log N)$.

We now show that the optimized version of the judgment has the same semantics as its counterpart presented in Sect. 4. First of all we formally state that, in Fig. 3, rule (WD-DELAY) does not affect derivability.

Lemma 4. *A judgment* $\mathsf{wd}_\rho(s, \emptyset)$ *has a derivation iff it has a derivation which does not use rule* (WD-DELAY)*.*

Proof. The right-to-left implication is obvious. If $\mathsf{wd}_\rho(s, \emptyset)$ uses rule (WD-DELAY), all the (first in their path) occurrences of the rule can be replaced by rule (WD-COREC), still getting a derivation.

Then, we define a relation between the auxiliary structures used in the two judgments:

For all m and (m, π), $m \bowtie (\mathsf{m}, \pi)$ holds iff
$dom(m) = dom(\mathsf{m})$ and, for all $x \in dom(m)$, $m(x) = sum(\mathsf{m}(x), \pi)$.

In this way, we have the following generalization, whose straightforward proof can be found in [4].

Theorem 2. *If* $\mathsf{m} \bowtie (\mathsf{m}, \pi)$*, then, for all* s*,* $\mathsf{wd}_\rho(s, m)$ *is derivable iff* $\mathsf{owd}_\rho(s, \mathsf{m}, \pi)$ *is derivable.*

Corollary 1. $\mathsf{wd}_\rho(s, \emptyset)$ *is derivable iff* $\mathsf{owd}_\rho(s, \emptyset, \epsilon)$ *is derivable.*

6 Related and Future Work

As mentioned in Sect. 1, our approach extends regular corecursion, which originated from *co-SLD resolution* [1,6,18,19], where already considered goals (up to unification), called *coinductive hypotheses*, are successfully solved. Language constructs that support this programming style have also been proposed in the functional [14] and object-oriented [2,7] paradigm.

There have been a few attempts of extending the expressive power of regular corecursion. Notably, *structural resolution* [15,16] is an 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 in [8], introducing algebraic trees and equations as generalizations of regular ones. Such proposals share, even though with different techniques and in a different context, our aim of extending regular corecursion; on the other hand, the fact that corecursion is *checked* is, at our knowledge, a novelty of our work.

For the operators considered in the calculus and some examples, our main sources of inspiration have been the works of Rutten [17], where a coinductive calculus of streams of real numbers is defined, and Hinze [12], where a calculus of generic streams is defined in a constructive way and implemented in Haskell.

In this paper, as in all the above mentioned approaches derived from co-SLD resolution, the aim is to provide an *operational* semantics, designed to directly lead to an implementation. That is, even though streams are infinite objects (terms where the constructor is the only operator, defined coinductively), evaluation handles *finite* representations, and is defined by an *inductive* inference system. Coinductive approaches can be adopted to obtain more abstract semantics of calculi with infinite terms. For instance, [9] defines a coinductive semantics of the infinitary lambda-calculus where, roughly, the semantics of terms with an infinite reduction sequence is the infinite term obtained as limit. In coinductive logic programming, co-SLD resolution is the operational counterpart of a coinductive semantics where a program denotes a set of infinite terms. In [2], analogously, regular corecursion is shown to be sound with respect to an abstract coinductive semantics using *flexible coinduction* [5,10], see below.

Our calculus is an enhancement of that presented in [3], with two main significant contributions: (1) the interleaving operator, challenging since it is based on a non-trivial recursion schema; (2) an optimized definition of the runtime well-definedness check, as a useful basis for an implementation. Our main technical results are Theorem 1, stating that passing the runtime well-definedness check performed for a function call prevents non-termination in accessing elements in the resulting stream, and Theorem 2, stating that the optimized version is equivalent.

Whereas in [3] the well-definedness check was also a necessary condition to guarantee termination, this is not the case here, due to the interleaving operator. Consider, for instance, the following example: $\rho = \{s \mapsto (s\hat{}\,\|s)\|0{:}s\}$. The judgment $\mathsf{wd}_\rho(s, \emptyset)$ is not derivable, in particular because of $s\hat{}$, since $\mathsf{wd}_\rho(s, \{s \mapsto -1\})$ is not derivable and, hence, $\mathsf{wd}_\rho(s\hat{}, \{s \mapsto 0\})$, $\mathsf{wd}_\rho(s\hat{}\,\|s, \{s \mapsto 0\})$, and $\mathsf{wd}_\rho((s\hat{}\,\|s)\|0{:}s, \{s \mapsto 0\})$. However, $at_\rho(s, i)$ is well-defined for all indexes i; indeed, $at_\rho(s,1) = 0$ is derivable, $at_\rho(s,0) = k$ is derivable iff $at_\rho(s,1) = k$ is derivable, and, for all $i > 1$, $at_\rho(s,i) = k$ is derivable iff $at_\rho(s,j) = k$ is derivable for some $j < i$, hence $at_\rho(s,i) = 0$ is derivable for all i. We leave for future work the investigation of a complete check.

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() = [0] [*] zeros()` which is not well-formed operationally, but admits as unique solution the stream of all zeros.

Finally, in the presented calculus a cyclic call is detected by rule (COREC) if it is syntactically the same of some in the call trace. Although such a rule allows cycle detection for all the examples presented in this paper, it is not complete with respect to the abstract notion where expressions denoting the same stream are equivalent, as illustrated by the following alternative definition of function `incr` as presented in Sect. 3:

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

If syntactic equivalence is used to detect cycles, then the call `incr_reg([0])` diverges, since the terms passed as argument to the recursive calls are all syntactically different; as an example, consider the arguments x and $x\hat{}$ passed to the initial call and to the first recursive call, respectively, in the environment $\rho = \{x \mapsto 0 : x\}$; they are syntactically different, but denote the same stream.

In future work we plan to investigate more expressive operational characterizations of equivalence.

Other interesting directions for future work are the following.

- Investigate additional operators and the expressive power of the calculus.
- Design a static type system to prevent runtime errors such as the non-well-definedness of a stream.
- Extend corecursive definition to *flexible* corecursive definitions [10,11] where programmers can define specific behaviour when a cycle is detected. In this way we could get termination in cases where lazy evaluation diverges. For instance, assuming to allow also booleans results for functions, we could define the predicate `allPos`, checking that all the elements of a stream are positive, specifying as result `true` when a cycle is detected; in this way, e.g., `allPos(one_two)` would return the correct result.

References

1. Ancona, D.: Regular corecursion in Prolog. Comput. Lang. Syst. Struct. **39**(4), 142–162 (2013)
2. Ancona, D., Barbieri, P., Dagnino, F., Zucca, E.: Sound regular corecursion in coFJ. In: Hirschfeld, R., Pape, T. (eds.) ECOOP'20 - Object-Oriented Programming. LIPIcs, vol. 166, pp. 1:1–1:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020)
3. Ancona, D., Barbieri, P., Zucca, E.: Enhanced regular corecursion for data streams. In: ICTCS'21 - Italian Conference on Theoretical Computer Science (2021)

4. Ancona, D., Barbieri, P., Zucca, E.: Enhancing expressivity of checked corecursive streams (extended version) (2022). https://arxiv.org/abs/2202.06868
5. Ancona, D., Dagnino, F., Zucca, E.: Generalizing inference systems by coaxioms. In: Yang, H. (ed.) ESOP 2017. LNCS, vol. 10201, pp. 29–55. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54434-1_2
6. Ancona, D., Dovier, A.: A theoretical perspective of coinductive logic programming. Fund. Inform. **140**(3–4), 221–246 (2015)
7. Ancona, D., Zucca, E.: Corecursive featherweight Java. In: FTfJP'12 - Formal Techniques for Java-like Programs, pp. 3–10. ACM Press (2012)
8. Courcelle, B.: Fundamental properties of infinite trees. Theoret. Comput. Sci. **25**, 95–169 (1983)
9. Czajka, L.: A new coinductive confluence proof for infinitary lambda calculus. Log. Methods Comput. Sci. **16**(1) (2020)
10. Dagnino, F.: Flexible coinduction. Ph.D. thesis, DIBRIS, University of Genova (2021)
11. Dagnino, F., Ancona, D., Zucca, E.: Flexible coinductive logic programming. Theory Pract. Logic Program. **20**(6), 818–833 (2020). Issue for ICLP 2020
12. Hinze, R.: Concrete stream calculus: an extended study. J. Funct. Program. **20**(5–6), 463–535 (2010)
13. Jeannin, J.-B., Kozen, D.: Computing with capsules. J. Autom. Lang. Comb. **17**(2–4), 185–204 (2012)
14. Jeannin, J.-B., Kozen, D., Silva, A.: CoCaml: functional programming with regular coinductive types. Fund. Inform. **150**, 347–377 (2017)
15. Komendantskaya, E., Johann, P., Schmidt, M.: A productivity checker for logic programming. In: Hermenegildo, M.V., Lopez-Garcia, P. (eds.) LOPSTR 2016. LNCS, vol. 10184, pp. 168–186. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63139-4_10
16. Komendantskaya, E., Power, J., Schmidt, M.: Coalgebraic logic programming: from semantics to implementation. J. Log. Comput. **26**(2), 745–783 (2016)
17. Rutten, J.J.M.M.: A coinductive calculus of streams. Math. Struct. Comput. Sci. **15**(1), 93–147 (2005)
18. Simon, L.: Extending logic programming with coinduction. Ph.D. thesis, University of Texas at Dallas (2006)
19. Simon, L., Bansal, A., Mallya, A., Gupta, G.: Co-logic programming: extending logic programming with coinduction. In: Arge, L., Cachin, C., Jurdziński, T., Tarlecki, A. (eds.) ICALP 2007. LNCS, vol. 4596, pp. 472–483. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73420-8_42