Non-regular corecursive streams

Pietro Barbieri
Joint work with: Davide Ancona and Elena Zucca

DIBRIS, University of Genova

T-LADIES kick-off meeting
July 6-7, 2022

Problem description

(Conceptually) infinite structures are hard to manage

E.g.: streams in IoT contexts, infinite trees, ...

Problem description

(Conceptually) infinite structures are hard to manage

E.g.: streams in IoT contexts, infinite trees, . . .

Main issues

- Representation

Problem description

(Conceptually) infinite structures are hard to manage

E.g.: streams in IoT contexts, infinite trees, . . .

Main issues

- Representation
- Manipulation

Problem description

(Conceptually) infinite structures are hard to manage

E.g.: streams in IoT contexts, infinite trees, . . .

Main issues

- Representation
- Manipulation
- Identification of ill-formed definitions

Aim of the work

Design a calculus to:

- Finitely represent infinite streams

Aim of the work

Design a calculus to:

- Finitely represent infinite streams
- Study properties of entire streams

Aim of the work

Design a calculus to:

- Finitely represent infinite streams
- Study properties of entire streams
- Statically check the correctness of stream definitions

Aim of the work

Design a calculus to:

- Finitely represent infinite streams
- Study properties of entire streams
- Statically check the correctness of stream definitions

Possible application: testing of IoT systems

- Generation of complex streams

Aim of the work

Design a calculus to:

- Finitely represent infinite streams
- Study properties of entire streams
- Statically check the correctness of stream definitions

Possible application: testing of IoT systems

- Generation of complex streams
- Possibility of relying on common stream processing functions

State of the art

State of the art: lazy evaluation

- Well-established solution for data stream generation and processing

State of the art: lazy evaluation

- Well-established solution for data stream generation and processing
- Haskell examples:

State of the art: lazy evaluation

- Well-established solution for data stream generation and processing
- Haskell examples:
  - `one_two = 1:2:one_two`

State of the art: lazy evaluation

- Well-established solution for data stream generation and processing
- Haskell examples:
  - `one_two = 1:2:one_two`
  - `from n = n:from(n+1)`

State of the art: lazy evaluation

- Well-established solution for data stream generation and processing
- Haskell examples:
    - `one_two = 1:2:one_two`
    - `from n = n:from(n+1)`
- Operations that inspect the whole structure diverge

State of the art: lazy evaluation

- Well-established solution for data stream generation and processing
- Haskell examples:
  - `one_two = 1:2:one_two`
  - `from n = n:from(n+1)`
- Operations that inspect the whole structure diverge
  - `one_two == one_two`

State of the art: lazy evaluation

- Well-established solution for data stream generation and processing
- Haskell examples:
  - `one_two = 1:2:one_two`
  - `from n = n:from(n+1)`
- Operations that inspect the whole structure diverge
  - `one_two == one_two`
- Moreover, ill-formed definitions allowed:

State of the art: lazy evaluation

- Well-established solution for data stream generation and processing
- Haskell examples:
  - `one_two = 1:2:one_two`
  - `from n = n:from(n+1)`
- Operations that inspect the whole structure diverge
  - `one_two == one_two`
- Moreover, ill-formed definitions allowed:
  - `bad_stream = bad_stream`

State of the art: lazy evaluation

- Well-established solution for data stream generation and processing
- Haskell examples:
  - `one_two = 1:2:one_two`
  - `from n = n:from(n+1)`
- Operations that inspect the whole structure diverge
  - `one_two == one_two`
- Moreover, ill-formed definitions allowed:
  - `bad_stream = bad_stream`
- Well-definedness of streams not decidable in Haskell

State of the art: Regular corecursion

- Infinite streams finitely represented by sets of equations built only on the stream constructor

State of the art: Regular corecursion

- Infinite streams finitely represented by sets of equations built only on the stream constructor
- Regular (cyclic) streams are supported

State of the art: Regular corecursion

- Infinite streams finitely represented by sets of equations built only on the stream constructor
- Regular (cyclic) streams are supported
- Functions are regularly corecursive:

State of the art: Regular corecursion

- Infinite streams finitely represented by sets of equations built only on the stream constructor
- Regular (cyclic) streams are supported
- Functions are regularly corecursive:
    - Execution keeps track of pending function calls

State of the art: Regular corecursion

- Infinite streams finitely represented by sets of equations built only on the stream constructor
- Regular (cyclic) streams are supported
- Functions are regularly corecursive:
  - Execution keeps track of pending function calls
  - Non-termination avoided

State of the art: Regular corecursion

- Infinite streams finitely represented by sets of equations built only on the stream constructor
- Regular (cyclic) streams are supported
- Functions are regularly corecursive:
  - Execution keeps track of pending function calls
  - Non-termination avoided
- `one_two() = 1:2:one_two()` $\longrightarrow$ `x = 1:2:x`

State of the art: Regular corecursion

- Infinite streams finitely represented by sets of equations built only on the stream constructor
- Regular (cyclic) streams are supported
- Functions are regularly corecursive:
  - Execution keeps track of pending function calls
  - Non-termination avoided
- `one_two() = 1:2:one_two()` $\longrightarrow$ `x = 1:2:x`
- `one_two() == one_two())` $\longrightarrow$ `true`

State of the art: Regular corecursion

- Infinite streams finitely represented by sets of equations built only on the stream constructor
- Regular (cyclic) streams are supported
- Functions are regularly corecursive:
  - Execution keeps track of pending function calls
  - Non-termination avoided
- `one_two() = 1:2:one_two()` $\longrightarrow$ `x = 1:2:x`
- `one_two() == one_two())` $\longrightarrow$ `true`
- However, fails to model non-regular streams

State of the art: Regular corecursion

- Infinite streams finitely represented by sets of equations built only on the stream constructor
- Regular (cyclic) streams are supported
- Functions are regularly corecursive:
  - Execution keeps track of pending function calls
  - Non-termination avoided
- `one_two() = 1:2:one_two()` $\longrightarrow$ `x = 1:2:x`
- `one_two() == one_two())` $\longrightarrow$ `true`
- However, fails to model non-regular streams
  - No value for `from(0)`

Non-regular corecursive streams

Our approach

- Keeps the benefits of regular corecursion

Our approach

- Keeps the benefits of regular corecursion
- Functions can also return non-regular streams

Our approach

- Keeps the benefits of regular corecursion
- Functions can also return non-regular streams
    - `repeat(n) = n:repeat(n)`
      `from(n)=n:(from(n)[+]repeat(1))`

  pointwise addition [+] on streams allowed in equations similarly as the stream constructor _:_

Our approach

- Keeps the benefits of regular corecursion
- Functions can also return non-regular streams
    - `repeat(n) = n:repeat(n)`
      `from(n)=n:(from(n)[+]repeat(1))`

  pointwise addition `[+]` on streams allowed in equations similarly as the stream constructor `_:_`
- Decidable procedure to check whether a stream is well-defined

Our approach

- Keeps the benefits of regular corecursion
- Functions can also return non-regular streams
    - `repeat(n) = n:repeat(n)`
      `from(n)=n:(from(n)[+]repeat(1))`

  pointwise addition `[+]` on streams allowed in equations similarly as the stream constructor `_:_`
- Decidable procedure to check whether a stream is well-defined
- Only well-defined streams accepted at runtime

Our approach

- Keeps the benefits of regular corecursion
- Functions can also return non-regular streams
  - `repeat(n) = n:repeat(n)`
    `from(n)=n:(from(n)[+]repeat(1))`

  pointwise addition `[+]` on streams allowed in equations similarly as the stream constructor `_:_`
- Decidable procedure to check whether a stream is well-defined
- Only well-defined streams accepted at runtime
- Decidable procedure to check the equality of two streams

Syntax of the calculus

$$
\begin{array}{lcl}
\overline{fd} & ::= & fd_1 \ldots fd_n \\
fd & ::= & f(\overline{x}) = se \\
e & ::= & se \mid ne \mid be \\
se & ::= & x \mid \texttt{if } be \texttt{ then } se_1 \texttt{ else } se_2 \mid ne : se \mid se\hat{} \mid se_1\, op\; se_2 \mid f(\overline{e}) \\
ne & ::= & x \mid se(ne) \mid ne_1\; nop\; ne_2 \mid 0 \mid 1 \mid 2 \mid \ldots \\
be & ::= & x \mid \texttt{true} \mid \texttt{false} \mid \ldots \\
op & ::= & [nop] \mid \| \\
nop & ::= & + \mid - \mid * \mid /
\end{array}
$$

- Program = sequence of mutually recursive function declarations
- Functions can only return streams
- Expressions can be: streams, numeric values, booleans

Simple examples

- `one_two() = 1:2:one_two()`

Simple examples

- `one_two() = 1:2:one_two()`

  $\texttt{one\_two()} \longrightarrow (\texttt{x},\ \{\texttt{x} \mapsto \texttt{1:2:x}\})$

Simple examples

- `one_two() = 1:2:one_two()`

  $\texttt{one\_two()} \longrightarrow (\texttt{x},\ \{\texttt{x} \mapsto \texttt{1:2:x}\})$

- `repeat(n) = n:repeat(n)`

Simple examples

- `one_two() = 1:2:one_two()`

  $\texttt{one\_two()} \longrightarrow (\texttt{x},\ \{\texttt{x} \mapsto \texttt{1:2:x}\})$

- `repeat(n) = n:repeat(n)`

  $\texttt{repeat(1)} \longrightarrow (\texttt{y},\ \{\texttt{y} \mapsto \texttt{1:y}\})$

Simple examples

- one_two() = 1:2:one_two()

  one_two() $\longrightarrow$ (x, $\{$x $\mapsto$ 1:2:x$\}$)

- repeat(n) = n:repeat(n)

  repeat(1) $\longrightarrow$ (y, $\{$y $\mapsto$ 1:y$\}$)

- incr(s) = s[+]repeat(1)

Simple examples

- one_two() = 1:2:one_two()

  one_two() $\longrightarrow$ (x, {x $\mapsto$ 1:2:x})

- repeat(n) = n:repeat(n)

  repeat(1) $\longrightarrow$ (y, {y $\mapsto$ 1:y})

- incr(s) = s[+]repeat(1)

  incr(one_two()) $\longrightarrow$ (x[+]y,{x $\mapsto$ 1:2:x, y $\mapsto$ 1:y})

Simple examples

- `one_two() = 1:2:one_two()`

  `one_two()` $\longrightarrow$ `(x, {x` $\mapsto$ `1:2:x})`

- `repeat(n) = n:repeat(n)`

  `repeat(1)` $\longrightarrow$ `(y, {y` $\mapsto$ `1:y})`

- `incr(s) = s[+]repeat(1)`

  `incr(one_two())` $\longrightarrow$ `(x[+]y,{x` $\mapsto$ `1:2:x, y` $\mapsto$ `1:y})`

  `incr(one_two())(0)`

Simple examples

- `one_two() = 1:2:one_two()`

  $\texttt{one\_two()} \longrightarrow (\texttt{x}, \{\texttt{x} \mapsto \texttt{1:2:x}\})$

- `repeat(n) = n:repeat(n)`

  $\texttt{repeat(1)} \longrightarrow (\texttt{y}, \{\texttt{y} \mapsto \texttt{1:y}\})$

- `incr(s) = s[+]repeat(1)`

  $\texttt{incr(one\_two())} \longrightarrow (\texttt{x[+]y}, \{\texttt{x} \mapsto \texttt{1:2:x}, \texttt{y} \mapsto \texttt{1:y}\})$

  $\texttt{incr(one\_two())(0)} \longrightarrow \texttt{(x[+]y)(0)}$

Simple examples

- `one_two() = 1:2:one_two()`

  `one_two()` $\longrightarrow$ `(x, {x` $\mapsto$ `1:2:x})`

- `repeat(n) = n:repeat(n)`

  `repeat(1)` $\longrightarrow$ `(y, {y` $\mapsto$ `1:y})`

- `incr(s) = s[+]repeat(1)`

  `incr(one_two())` $\longrightarrow$ `(x[+]y,{x` $\mapsto$ `1:2:x, y` $\mapsto$ `1:y})`

  `incr(one_two())(0)` $\longrightarrow$ `(x[+]y)(0)` $\longrightarrow$ `x(0)+y(0)`

Simple examples

- one_two() = 1:2:one_two()

  one_two() $\longrightarrow$ (x, $\{$x $\mapsto$ 1:2:x$\}$)

- repeat(n) = n:repeat(n)

  repeat(1) $\longrightarrow$ (y, $\{$y $\mapsto$ 1:y$\}$)

- incr(s) = s[+]repeat(1)

  incr(one_two()) $\longrightarrow$ (x[+]y,$\{$x $\mapsto$ 1:2:x, y $\mapsto$ 1:y$\}$)

  incr(one_two())(0) $\longrightarrow$ (x[+]y)(0) $\longrightarrow$ x(0)+y(0) $\longrightarrow$ (1:2:x)(0)+(1:y)(0)

Simple examples

- `one_two() = 1:2:one_two()`

  $\texttt{one\_two()} \longrightarrow (\texttt{x}, \{\texttt{x} \mapsto \texttt{1:2:x}\})$

- `repeat(n) = n:repeat(n)`

  $\texttt{repeat(1)} \longrightarrow (\texttt{y}, \{\texttt{y} \mapsto \texttt{1:y}\})$

- `incr(s) = s[+]repeat(1)`

  $\texttt{incr(one\_two())} \longrightarrow (\texttt{x[+]y}, \{\texttt{x} \mapsto \texttt{1:2:x}, \texttt{y} \mapsto \texttt{1:y}\})$

  $\texttt{incr(one\_two())(0)} \longrightarrow \texttt{(x[+]y)(0)} \longrightarrow \texttt{x(0)+y(0)} \longrightarrow \texttt{(1:2:x)(0)+(1:y)(0)} \longrightarrow 2$

Main ingredients of the calculus:

- Operational semantics: evaluation keeps track of already considered function calls, streams represented in a finite way [AnconaBarbieriZucca@ICTCS21]

Main ingredients of the calculus:

- Operational semantics: evaluation keeps track of already considered function calls, streams represented in a finite way [AnconaBarbieriZucca@ICTCS21]
- Well-definedness check to guarantee safe access to streams [AnconaBarbieriZucca@FLOPS22], [Submitted journal paper]

Main ingredients of the calculus:

- Operational semantics: evaluation keeps track of already considered function calls, streams represented in a finite way [AnconaBarbieriZucca@ICTCS21]
- Well-definedness check to guarantee safe access to streams [AnconaBarbieriZucca@FLOPS22], [Submitted journal paper]
- Decidable procedure to check the equality of two streams [AnconaBarbieriZucca@ICTCS22], [Ongoing work]

Semantics

- Shape of the judgment: $e, \rho, \tau \Downarrow (v, \rho')$

Semantics

- Shape of the judgment: $e, \rho, \tau \Downarrow (v, \rho')$
  - e expression to be evaluated

Semantics

- Shape of the judgment: $e, \rho, \tau \Downarrow (v, \rho')$
  - e expression to be evaluated
  - $\rho ::= x_1 \mapsto s_1 \ldots x_n \mapsto s_n$ environment

Semantics

- Shape of the judgment: $e, \rho, \tau \Downarrow (v, \rho')$
  - e expression to be evaluated
  - $\rho ::= x_1 \mapsto s_1 \ldots x_n \mapsto s_n$ environment
  - $\tau ::= f_1(\overline{v}_1) \mapsto x_1 \ \ldots \ f_n(\overline{v}_n) \mapsto x_n$ call trace

Semantics

- Shape of the judgment: $e, \rho, \tau \Downarrow (v, \rho')$
  - e expression to be evaluated
  - $\rho ::= x_1 \mapsto s_1 \ldots x_n \mapsto s_n$ environment
  - $\tau ::= f_1(\overline{v}_1) \mapsto x_1 \ \ldots \ f_n(\overline{v}_n) \mapsto x_n$ call trace
  - (v, ρ') result

Semantics

- Shape of the judgment: $e, \rho, \tau \Downarrow (v, \rho')$
  - e expression to be evaluated
  - $\rho ::= x_1 \mapsto s_1 \ldots x_n \mapsto s_n$ environment
  - $\tau ::= f_1(\overline{v}_1) \mapsto x_1 \;\; \ldots \;\; f_n(\overline{v}_n) \mapsto x_n$ call trace
  - (v, ρ') result
- Values:

Semantics

- Shape of the judgment: $e, \rho, \tau \Downarrow (v, \rho')$
  - e expression to be evaluated
  - $\rho ::= x_1 \mapsto s_1 \ldots x_n \mapsto s_n$ environment
  - $\tau ::= f_1(\overline{v}_1) \mapsto x_1 \quad \ldots \quad f_n(\overline{v}_n) \mapsto x_n$ call trace
  - (v, ρ') result
- Values:
  - $v ::= s \mid n \mid b$ value
  - $s ::= x \mid n : s \mid s\hat{} \mid s_1[op]s_2$ (open) stream value

Semantics

- Shape of the judgment: $e, \rho, \tau \Downarrow (v, \rho')$
  - e expression to be evaluated
  - $\rho ::= x_1 \mapsto s_1 \ldots x_n \mapsto s_n$ environment
  - $\tau ::= f_1(\bar{v}_1) \mapsto x_1 \;\ldots\; f_n(\bar{v}_n) \mapsto x_n$ call trace
  - (v, ρ') result
- Values:
  - $v ::= s \mid n \mid b$ value
  - $s ::= x \mid n : s \mid s\hat{} \mid s_1[op]s_2$ (open) stream value
  - $n ::= 0 \mid 1 \mid 2 \mid \ldots$ index, numeric value
  - $b ::= \mathtt{true} \mid \mathtt{false}$ boolean value

Advanced examples

Examples: non-regular streams

```
nat() = 0:(nat()[+]repeat(1))
```

- stream of natural numbers

Examples: non-regular streams

```
nat() = 0:(nat()[+]repeat(1))
```

- stream of natural numbers

```
nat_to_pow(n) = if n <= 0 then repeat(1)
    else nat_to_pow(n-1)[*]nat()
```

- nat_to_pow(n)(x)= x^n

Examples: non-regular streams

```
nat() = 0:(nat()[+]repeat(1))
```

- stream of natural numbers

```
nat_to_pow(n) = if n <= 0 then repeat(1)
    else nat_to_pow(n-1)[*]nat()
```

- nat_to_pow(n)(x)= x^n

```
pow(n) = 1:(repeat(n)[*]pow(n))
```

- pow(n)(x)= n^x

Examples: non-regular streams

```
nat() = 0:(nat()[+]repeat(1))
```

- stream of natural numbers

```
nat_to_pow(n) = if n <= 0 then repeat(1)
    else nat_to_pow(n-1)[*]nat()
```

- nat_to_pow(n)(x)= x^n

```
pow(n) = 1:(repeat(n)[*]pow(n))
```

- pow(n)(x)= n^x

```
fact() = 1:((nat()[+]repeat(1))[*]fact())
```

- factorial

Examples: non-regular streams

```
nat() = 0:(nat()[+]repeat(1))
```

- stream of natural numbers

```
nat_to_pow(n) = if n <= 0 then repeat(1)
    else nat_to_pow(n-1)[*]nat()
```

- nat_to_pow(n)(x)= x^n

```
pow(n) = 1:(repeat(n)[*]pow(n))
```

- pow(n)(x)= n^x

```
fact() = 1:((nat()[+]repeat(1))[*]fact())
```

- factorial

```
fib() = 0:1:(fib()[+]fib()^)
```

- stream of Fibonacci numbers

Examples: common functions on streams

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

Examples: common functions on streams

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

- stream of partial sums of the first `i+1` elements of `s`
- $\mathtt{sum(s)(i)} = \sum_{\mathtt{k}=0}^{\mathtt{i}} \mathtt{s(k)}$

Examples: common functions on streams

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

- stream of partial sums of the first `i+1` elements of `s`
- $\mathtt{sum(s)(i)} = \sum_{\mathtt{k}=\mathtt{0}}^{\mathtt{i}} \mathtt{s(k)}$

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

Examples: common functions on streams

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

- stream of partial sums of the first `i+1` elements of `s`
- $\texttt{sum(s)(i)} = \sum_{\texttt{k}=0}^{\texttt{i}} \texttt{s(k)}$

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

- stream of all terms of the Taylor series of the exponential function
- $\texttt{sum\_expn(n)(i)} = \sum_{\texttt{k}=0}^{\texttt{i}} \frac{\texttt{n}^{\texttt{k}}}{\texttt{k}!} = 1 + \texttt{n} + \frac{\texttt{n}^2}{2!} + \frac{\texttt{n}^3}{3!} + \frac{\texttt{n}^4}{4!} + \cdots + \frac{\texttt{n}^{\texttt{i}}}{\texttt{i}!}$

Examples: common functions on streams

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

- stream of partial sums of the first `i+1` elements of `s`
- $\mathtt{sum(s)(i)} = \sum_{k=0}^{i} \mathtt{s(k)}$

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

- stream of all terms of the Taylor series of the exponential function
- $\mathtt{sum\_expn(n)(i)} = \sum_{k=0}^{i} \frac{n^k}{k!} = 1 + n + \frac{n^2}{2!} + \frac{n^3}{3!} + \frac{n^4}{4!} + \cdots + \frac{n^i}{i!}$

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

- $\mathtt{aggr(3,s)} = s'$ s.t. $s'(i) = s(i) + s(i+1) + s(i+2)$

Examples: common functions on streams

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

- stream of partial sums of the first `i+1` elements of `s`
- $\texttt{sum(s)(i)} = \sum_{k=0}^{i} s(k)$

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

- stream of all terms of the Taylor series of the exponential function
- $\texttt{sum\_expn(n)(i)} = \sum_{k=0}^{i} \frac{n^k}{k!} = 1 + n + \frac{n^2}{2!} + \frac{n^3}{3!} + \frac{n^4}{4!} + \cdots + \frac{n^i}{i!}$

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

- $\texttt{aggr(3,s)} = s'$ s.t. $s'(i) = s(i) + s(i+1) + s(i+2)$

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

- stream of average values of `s` in the window of length `n`

Well-definedness of streams

Well-definedness

Definition

Well-defined environment ρ: for each $x \in dom(\rho)$, access to element $x(k)$ terminates for all $k \in \mathbb{N}$.

Well-definedness

Definition

Well-defined environment ρ: for each $x \in dom(\rho)$, access to element $x(k)$ terminates for all $k \in \mathbb{N}$.

Examples

$$\begin{Bmatrix} \mathsf{x} & \mapsto & \mathsf{1:2:x} \\ \mathsf{y} & \mapsto & \mathsf{x\hat{}} \end{Bmatrix}$$

Well-definedness

Definition

Well-defined environment ρ: for each $x \in \rho$, access to element $x(k)$ terminates for all $k \in \mathbb{N}$.

Examples

$$\left\{ \begin{array}{lll} \mathtt{x} & \mapsto & \mathtt{1:2:x} \\ \mathtt{y} & \mapsto & \mathtt{x\hat{}} \end{array} \right\}$$

Well-defininedness

Definition

Well-defined environment ρ: for each $x \in \rho$, access to element $x(k)$ terminates for all $k \in \mathbb{N}$.

Examples

$$\left\{\begin{array}{lll} \mathtt{x} & \mapsto & \mathtt{1:2:x} \\ \mathtt{y} & \mapsto & \mathtt{x\hat{}} \end{array}\right\} \qquad \left\{\begin{array}{lll} \mathtt{x} & \mapsto & \mathtt{1:y} \\ \mathtt{y} & \mapsto & \mathtt{y} \end{array}\right\}$$

Well-definedness

Definition

Well-defined environment ρ: for each $x \in \rho$, access to element $x(k)$ terminates for all $k \in \mathbb{N}$.

Examples

$$\left\{\begin{array}{lll} \mathtt{x} & \mapsto & \mathtt{1:2:x} \\ \mathtt{y} & \mapsto & \mathtt{x\hat{}} \end{array}\right\} \qquad \left\{\begin{array}{lll} \mathtt{x} & \mapsto & \mathtt{1:y} \\ \mathtt{y} & \mapsto & \mathtt{y} \end{array}\right\}$$

Equality of streams

Equality

- Stream operators in equations $=$ non-trivial equational theory

Equality

- Stream operators in equations = non-trivial equational theory
- Syntactic equality between cyclic terms provides a too weak notion

Equality

- Stream operators in equations = non-trivial equational theory
- Syntactic equality between cyclic terms provides a too weak notion

Semantic definition

$s_1 \equiv s_2$ iff, for each $k \in \mathbb{N}$, $s_1(k) = s_2(k)$

An algorithm: examples

Environment $\rho = \{\mathsf{x} \mapsto 1 : \mathsf{x}\}$

$$\mathsf{x} \equiv \mathsf{x}\hat{}$$

An algorithm: examples

Environment $\rho = \{\mathsf{x} \mapsto 1 : \mathsf{x}\}$

$$\mathsf{x} \equiv \mathsf{x}\hat{}$$

$$\downarrow$$

$$\mathsf{x} \equiv (1 : \mathsf{x})\hat{}$$

An algorithm: examples

Environment $\rho = \{\mathsf{x} \mapsto 1 : \mathsf{x}\}$

$$\mathsf{x} \equiv \mathsf{x}\hat{}$$

$$\downarrow$$

$$\mathsf{x} \equiv (1 : \mathsf{x})\hat{}$$

$$\downarrow$$

$$\mathsf{x} \equiv \mathsf{x}$$

An algorithm: examples

Environment $\rho = \{\mathsf{x} \mapsto 1:\mathsf{x},\ \mathsf{y} \mapsto 1:1:\mathsf{y}\ \}$

$$\mathsf{x} \equiv \mathsf{y}$$

An algorithm: examples

Environment $\qquad \rho = \{\mathsf{x} \mapsto 1:\mathsf{x},\ \mathsf{y} \mapsto 1:1:\mathsf{y}\ \}$

$$\begin{array}{c} \mathsf{x} \equiv \mathsf{y} \\ \downarrow \\ 1:\mathsf{x} \equiv 1:1:\mathsf{y} \end{array}$$

An algorithm: examples

Environment $\rho = \{\mathsf{x} \mapsto 1 : \mathsf{x},\ \mathsf{y} \mapsto 1 : 1 : \mathsf{y}\ \}$

$$\begin{array}{c} \mathsf{x} \equiv \mathsf{y} \\ \downarrow \\ 1 : \mathsf{x} \equiv 1 : 1 : \mathsf{y} \\ \downarrow \\ \mathsf{x} \equiv 1 : \mathsf{y} \end{array}$$

An algorithm: examples

Environment $\rho = \{\mathsf{x} \mapsto \mathsf{1:x},\ \mathsf{y} \mapsto \mathsf{1:1:y}\ \}$

$$
\begin{array}{c}
\mathsf{x} \equiv \mathsf{y} \\
\downarrow \\
\mathsf{1:x} \equiv \mathsf{1:1:y} \\
\downarrow \\
\mathsf{x} \equiv \mathsf{1:y} \\
\downarrow \\
\mathsf{1:x} \equiv \mathsf{1:y}
\end{array}
$$

An algorithm: examples

Environment $\rho = \{ x \mapsto 1 : x, \; y \mapsto 1 : 1 : y \}$

$$
\begin{array}{c}
x \equiv y \\
\downarrow \\
1 : x \equiv 1 : 1 : y \\
\downarrow \\
x \equiv 1 : y \\
\downarrow \\
1 : x \equiv 1 : y \\
\downarrow \\
x \equiv y
\end{array}
$$

Relevant tasks and future work

Task 1.1 (Adaptation)

- Only streams of naturals with arithmetic operators considered in the calculus

Relevant tasks and future work

Task 1.1 (Adaptation)

- Only streams of naturals with arithmetic operators considered in the calculus

Aims:

- Make the calculus parametric

Relevant tasks and future work

Task 1.1 (Adaptation)

- Only streams of naturals with arithmetic operators considered in the calculus

Aims:

- Make the calculus parametric
- Indeed, smoothly extending the approach to other data types (booleans, pairs, records, . . .)

Relevant tasks and future work

Task 1.1 (Adaptation)

- Only streams of naturals with arithmetic operators considered in the calculus

Aims:

- Make the calculus parametric
- Indeed, smoothly extending the approach to other data types (booleans, pairs, records, . . .)
- e.g., an `if_then_else_` stream operator whose first argument is a stream of booleans

Task 3.2 (Integration of static and dynamic verification)

- Untyped calculus

Task 3.2 (Integration of static and dynamic verification)

- Untyped calculus
- The well-definedness check takes place at runtime

Relevant tasks and future work

Task 3.2 (Integration of static and dynamic verification)

- Untyped calculus
- The well-definedness check takes place at runtime

Aims:

- Design a static type system to filter out early errors

Relevant tasks and future work

Task 3.2 (Integration of static and dynamic verification)

- Untyped calculus
- The well-definedness check takes place at runtime

Aims:

- Design a static type system to filter out early errors
- Reduce runtime overhead identifying ill-formed definitions ahead

Relevant tasks and future work

Task 4.4 (Application scenarios)

- Possibility to generate and manipulate a wide variety of streams

Relevant tasks and future work

Task 4.4 (Application scenarios)

- Possibility to generate and manipulate a wide variety of streams
- IoT relevant operations supported

Relevant tasks and future work

Task 4.4 (Application scenarios)

- Possibility to generate and manipulate a wide variety of streams
- IoT relevant operations supported

Aims:

- Integration with stream programming:
- Stream generation (sink streams) already supported

Relevant tasks and future work

Task 4.4 (Application scenarios)

- Possibility to generate and manipulate a wide variety of streams
- IoT relevant operations supported

Aims:

- Integration with stream programming:
- Stream generation (sink streams) already supported
- Source streams, pipeline to be investigated

Thank You!

Examples

Example of equality

Environment $\rho = \{\texttt{x} \mapsto \texttt{0:1:(x} \parallel \texttt{x)},\ \texttt{y} \mapsto \texttt{0:1:((2:y)} \parallel \texttt{y}\hat{)}\ \}$

$$\texttt{0:1:(x} \parallel \texttt{x)} \equiv \texttt{0:1:((2:y)} \parallel \texttt{y}\hat{)}$$

Example of equality

Environment $\rho = \{\mathtt{x} \mapsto \mathtt{0:1:(x} \parallel \mathtt{x),\ y} \mapsto \mathtt{0:1:((2:y)} \parallel \mathtt{y}\hat{)}\ \}$

$$\mathtt{0:1:(x} \parallel \mathtt{x)} \equiv \mathtt{0:1:((2:y)} \parallel \mathtt{y}\hat{)}$$

$$\downarrow^{*}$$

$$\mathtt{(x} \parallel \mathtt{x)} \equiv \mathtt{((2:y)} \parallel \mathtt{y}\hat{)}$$

Example of equality

Environment $\quad \rho = \{\texttt{x} \mapsto \texttt{0:1:(x} \parallel \texttt{x),}\ \texttt{y} \mapsto \texttt{0:1:((2:y)} \parallel \texttt{y}\hat{\texttt{)}}\ \}$

$$\texttt{0:1:(x} \parallel \texttt{x)} \equiv \texttt{0:1:((2:y)} \parallel \texttt{y}\hat{\texttt{)}}$$

$$\downarrow^{*}$$

$$\texttt{(x} \parallel \texttt{x)} \equiv \texttt{((2:y)} \parallel \texttt{y}\hat{\texttt{)}}$$

$$\downarrow$$

$$\texttt{(x} \parallel \texttt{x)} \equiv \texttt{(y} \parallel \texttt{y)}$$

Example of equality

Environment $\quad \rho = \{\texttt{x} \mapsto \texttt{0:1:(x} \parallel \texttt{x)},\ \texttt{y} \mapsto \texttt{0:1:((2:y)} \parallel \texttt{y}\hat{)}\ \}$

$$\texttt{0:1:(x} \parallel \texttt{x)} \equiv \texttt{0:1:((2:y)} \parallel \texttt{y}\hat{)}$$

$$\downarrow^{*}$$

$$\texttt{(x} \parallel \texttt{x)} \equiv \texttt{((2:y)} \parallel \texttt{y}\hat{)}$$

$$\downarrow$$

$$\texttt{(x} \parallel \texttt{x)} \equiv \texttt{(y} \parallel \texttt{y)}$$

$$\downarrow$$

$$\texttt{x} \equiv \texttt{y}$$

Example of equality

Environment $\quad \rho = \{\texttt{x} \mapsto \texttt{0:1:(x} \parallel \texttt{x)},\ \texttt{y} \mapsto \texttt{0:1:((2:y)} \parallel \texttt{y}\hat{\texttt{)}}\ \}$

$$\texttt{0:1:(x} \parallel \texttt{x)} \equiv \texttt{0:1:((2:y)} \parallel \texttt{y}\hat{\texttt{)}}$$

$$\downarrow^{*}$$

$$\texttt{(x} \parallel \texttt{x)} \equiv \texttt{((2:y)} \parallel \texttt{y}\hat{\texttt{)}}$$

$$\downarrow$$

$$\texttt{(x} \parallel \texttt{x)} \equiv \texttt{(y} \parallel \texttt{y)}$$

$$\downarrow$$

$$\texttt{x} \equiv \texttt{y}$$

Semantics of the calculus

Rules (1)

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

$$(\text{IF-F})\ \frac{be, \rho, \tau \Downarrow (\texttt{false}, \rho) \quad se_2, \rho, \tau \Downarrow (s, \rho')}{\texttt{if}\ be\ \texttt{then}\ se_1\ \texttt{else}\ se_2, \rho, \tau \Downarrow (s, \rho')} \qquad (\text{CONS})\ \frac{ne, \rho, \tau \Downarrow (n, \rho) \quad se, \rho, \tau \Downarrow (s, \rho')}{ne : se, \rho, \tau \Downarrow (n : s, \rho')}$$

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

Rules (2)

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

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

$$(\textsc{corec})\ \frac{}{f(\overline{v}), \rho, \tau \Downarrow (x, \rho)} \quad \tau(f(\overline{v})) = x$$

Well-definedness

Well-definedness: an algorithm

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

$$\text{(MAIN)}\ \frac{\mathrm{wd}_{\rho\{x \mapsto v\}}(x, \emptyset)}{\color{red}{wd(\rho, x, v)}} \qquad \text{(WD-VAR)}\ \frac{\mathrm{wd}_\rho(\rho(x), m\{x \mapsto 0\})}{\mathrm{wd}_\rho(x, m)}\ x \notin dom(m) \qquad \text{(WD-CONS)}\ \frac{\mathrm{wd}_\rho(s, m^{+1})}{\mathrm{wd}_\rho(n : s, m)}$$

$$\text{(WD-COREC)}\ \frac{}{\mathrm{wd}_\rho(x, m)}\ \begin{array}{l} x \in dom(m) \\ m(x) > 0 \end{array} \qquad \text{(WD-FV)}\ \frac{}{\mathrm{wd}_\rho(x, m)}\ x \notin dom(\rho) \qquad \text{(WD-TAIL)}\ \frac{\mathrm{wd}_\rho(s, m^{-1})}{\mathrm{wd}_\rho(s\hat{}, m)}$$

$$\text{(WD-NOP)}\ \frac{\mathrm{wd}_\rho(s_1, m) \quad \mathrm{wd}_\rho(s_2, m)}{\mathrm{wd}_\rho(s_1[op\,]s_2, m)} \qquad \text{(WD-}\|\text{)}\ \frac{\mathrm{wd}_\rho(s_1, m) \quad \mathrm{wd}_\rho(s_2, m^{+1})}{\mathrm{wd}_\rho(s_1 \| s_2, m)}$$

Idea: more constructors than tail operators traversed when a cyclic reference is found

On well-definedness

- `zeros()= repeat(0)[*] zeros()`
- Not well-defined operationally but admits a unique solution

On well-definedness

- A closed result (s, ρ) is well-defined if it denotes a unique stream
- A closed environment ρ is well-defined if, for each $x \in dom(\rho)$, (x, ρ) is well-defined.
- $=$ the corresponding set of equations admits a unique solution
  - $\{x \mapsto 1 : x\}$ well-defined
  - $\{x \mapsto x\}$ not well-defined
  - $\{x \mapsto x[+]y, y \mapsto 1 : y\}$ not well-defined