Upper Bounds on Stream I/O Using Semantic Interpretations

Marco Gaboardi* and Romain Péchoux†

* Dipartimento di Informatica - Università di Torino - gaboardi@di.unito.it

† Université Nancy 2 - Nancy Université - LORIA - romain.pechoux@loria.fr

Work partially supported by the French COMPLICE project
and by the Italian CONCERTO project.

Introduction

Setting and Motivations

- We deal with the problems of program resource control and resource usage certification.
- We are interested into studying such properties about programs working on infinite data structures, notably streams.
- Lazy Functional Programming Languages enable an elegant treatment of stream programs.
- Here, in order to prove properties of Lazy Functional programs dealing with streams, we apply static analysis methods inspired by Quasi-interpretation by Bonfante et al[2001] and Sup-interpretation by Marion and Péchoux[2006].
- In particular we give some criteria ensuring three distinct Input-Output upper bound properties.

Streams

- Streams are infinite lists

$$0 : 1 : 2 : 3 : 4 : 5 : 6 : \cdots : 65512 : 65513 \cdots$$

$$1 : 1 : 2 : 3 : 5 : 8 : 13 : \cdots : 46368 : 75025 : \cdots$$

$$(0,0) : (0,1) : (1,0) : (2,0) : (1,1) : (0,2) : \cdots : (64,0) : (63,1) : \cdots$$

- Lazy functional programming languages like Haskell allow us to easily define streams by expressions:

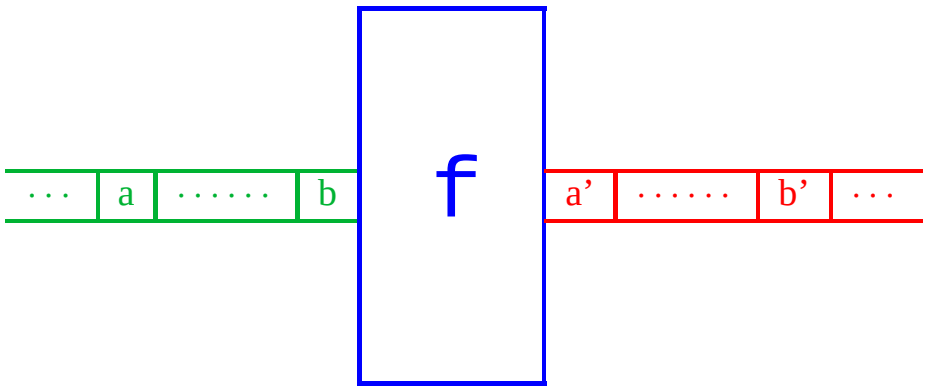
```
from x = x : (from (x + 1))

fibo = 1 : 1 : (sadd fibo (tail fibo))
```

Stream properties

- In the last years, many efforts have been made in the study of:
  - stream programs equivalence by means of denotational and operational semantics
  - stream productivity by means of several techniques: syntactical restrictions, type systems, data flow analysis
- Unfortunately only little efforts have been made in order to study other stream properties.
- Here we deal with some complexity properties of stream programs:
  - relating the number of output writings to the number of input readings.
  - relating the number of output writings to the size of input reading elements.
  - relating synchronously the size of output writing elements to the size of input reading elements.

Stream programs

- It is convenient to distinguish between different kinds of programs dealing with stream:
  - programs generating streams
  - programs working on input streams
- In this work we focus on programs working on input streams, we leave the study of programs generating streams for future developments.
- In particular, we look at programs as



Preliminaries

The sHask language

- A program P is a set of definitions described by the grammar:

$$
\begin{array}{lll}
\mathtt{p} & ::= \mathtt{x} \mid \mathbf{c}\ \mathtt{p}_1\ \cdots\ \mathtt{p}_n & \text{(Patterns)} \\
\mathtt{e} & ::= \mathtt{x} \mid \mathtt{t}\ \mathtt{e}_1\ \cdots\ \mathtt{e}_n \mid \mathtt{Case}\ \overline{\mathtt{e}}\ \mathtt{of}\ \overline{\mathtt{p}_1} \to \mathtt{e}_1, \ldots, \overline{\mathtt{p}_m} \to \mathtt{e}_m & \text{(Expressions)} \\
\mathtt{v} & ::= \mathbf{c}\ \mathtt{e}_1\ \cdots\ \mathtt{e}_n & \text{(Values)} \\
\mathtt{d} & ::= \mathtt{f}\ \mathtt{x}_1\ \cdots\ \mathtt{x}_n\ = \mathtt{e} & \text{(Definitions)}
\end{array}
$$

- There is a special error constructor symbol **Err** of arity 0 which corresponds to pattern matching failure. We also allow pattern matching on it.
- As syntactic sugar we write a definition of the shape $\mathtt{f}\ \overrightarrow{\mathtt{x}} = \mathtt{Case}\ \overline{\mathtt{x}}\ \mathtt{of}\ \overline{\mathtt{p}_1} \to \mathtt{e}_1, \ldots, \overline{\mathtt{p}_k} \to \mathtt{e}_k$ as a set of definitions

$$
\begin{array}{lcl}
\mathtt{f}\ \overrightarrow{\mathtt{p}_1} & = & \mathtt{e}_1 \\
 & \vdots & \\
\mathtt{f}\ \overrightarrow{\mathtt{p}_k} & = & \mathtt{e}_k
\end{array}
$$

Type system

- The set of types contains elements of the shape

$$\mathtt{A}_1 \to (\cdots \to (\mathtt{A}_n \to \mathtt{A}))$$

  where basic and value types are defined as

$$\begin{array}{lcl} \sigma & ::= & \alpha \mid \mathtt{Nat} \mid \sigma \times \sigma \\ \mathtt{A} & ::= & a \mid \sigma \mid \mathtt{A} \times \mathtt{A} \mid [\sigma] \end{array}$$

- The typing rules are

$$\frac{}{\mathtt{x} :: \mathtt{A}}\ (\mathtt{Var}) \qquad \frac{\overline{\mathtt{e}} :: \overline{\mathtt{A}} \quad \overline{\mathtt{p}_1} :: \overline{\mathtt{A}} \quad \cdots \quad \overline{\mathtt{p}_m} :: \overline{\mathtt{A}} \quad \mathtt{e}_1 :: \mathtt{A} \quad \cdots \quad \mathtt{e}_m :: \mathtt{A}}{\mathtt{Case}\ \overline{\mathtt{e}}\ \mathtt{of}\ \overline{\mathtt{p}_1} \to \mathtt{e}_1, \ldots, \overline{\mathtt{p}_m} \to \mathtt{e}_m :: \mathtt{A}}\ (\mathtt{Case})$$

$$\frac{}{\mathtt{t} :: \mathtt{A}_1 \to \cdots \to \mathtt{A}_n \to \mathtt{A}}\ (\mathtt{Tb}) \qquad \frac{\mathtt{t} :: \mathtt{A}_1 \to \cdots \to \mathtt{A}_n \to \mathtt{A} \quad \mathtt{e}_1 :: \mathtt{A}_1 \quad \cdots \quad \mathtt{e}_n :: \mathtt{A}_n}{\mathtt{t}\ \mathtt{e}_1 \cdots \mathtt{e}_n :: \mathtt{A}}\ (\mathtt{Ts})$$

- **Remarks:**
  - **Err** :: A for every A.
  - Streams of streams are forbidden.

Lazy Operational Semantics

- A natural way to deal with infinite data structure in a finitary way is by using a lazy evaluation $\Downarrow$.

$$\frac{\mathbf{c} \in \mathcal{C}}{\mathbf{c}\ \mathtt{e}_1\ \cdots\ \mathtt{e}_n \Downarrow \mathbf{c}\ \mathtt{e}_1\ \cdots\ \mathtt{e}_n}\ (\text{val}) \qquad \frac{\mathtt{e}\{\mathtt{e}_1/\mathtt{x}_1, \cdots, \mathtt{e}_n/\mathtt{x}_n\} \Downarrow \mathtt{v} \quad \mathtt{f}\ \mathtt{x}_1\ \cdots\ \mathtt{x}_n = \mathtt{e}}{\mathtt{f}\ \mathtt{e}_1\ \cdots\ \mathtt{e}_n \Downarrow \mathtt{v}}\ (\text{fun})$$

$$\frac{\mathtt{Case}\ \mathtt{e}^1\ \mathtt{of}\ \mathtt{p}_1^1 \to \ldots \to \mathtt{Case}\ \mathtt{e}^m\ \mathtt{of}\ \mathtt{p}_1^m \to \mathtt{d}_1 \Downarrow \mathtt{v} \quad \mathtt{v} \neq \mathbf{Err}}{\mathtt{Case}\ \overline{\mathtt{e}}\ \mathtt{of}\ \overline{\mathtt{p}_1} \to \mathtt{d}_1, \ldots, \overline{\mathtt{p}_n} \to \mathtt{d}_n \Downarrow \mathtt{v}}\ (\mathrm{c}_b)$$

$$\frac{\mathtt{Case}\ \mathtt{e}^1\ \mathtt{of}\ \mathtt{p}_1^1 \to \ldots \to \mathtt{Case}\ \mathtt{e}^m\ \mathtt{of}\ \mathtt{p}_1^m \to \mathtt{d}_1 \Downarrow \mathbf{Err} \quad \mathtt{Case}\ \overline{\mathtt{e}}\ \mathtt{of}\ \overline{\mathtt{p_2}} \to \mathtt{d_2}, \ldots, \overline{\mathtt{p_n}} \to \mathtt{d_n} \Downarrow \mathtt{v}}{\mathtt{Case}\ \overline{\mathtt{e}}\ \mathtt{of}\ \overline{\mathtt{p}_1} \to \mathtt{d}_1, \ldots, \overline{\mathtt{p}_n} \to \mathtt{d}_n \Downarrow \mathtt{v}}\ (\mathrm{c})$$

$$\frac{\mathtt{e} \Downarrow \mathbf{c}\ \mathtt{e}_1\ \cdots\ \mathtt{e}_n \qquad \mathtt{Case}\ \mathtt{e}_1\ \mathtt{of}\ \mathtt{p}_1 \to \ldots \to \mathtt{Case}\ \mathtt{e}_n\ \mathtt{of}\ \mathtt{p}_n \to \mathtt{d} \Downarrow \mathtt{v}}{\mathtt{Case}\ \mathtt{e}\ \mathtt{of}\ \mathbf{c}\ \mathtt{p}_1\ \cdots\ \mathtt{p}_n \to \mathtt{d} \Downarrow \mathtt{v}}\ (\text{pm})$$

$$\frac{\mathtt{e} \Downarrow \mathtt{v} \quad \mathtt{v} \neq \mathbf{c}\ \mathtt{e}_1\ \cdots\ \mathtt{e}_n}{\mathtt{Case}\ \mathtt{e}\ \mathtt{of}\ \mathbf{c}\ \mathtt{p}_1\ \cdots\ \mathtt{p}_n \to \mathtt{d} \Downarrow \mathbf{Err}}\ (\mathrm{pm}_e) \qquad \frac{\mathtt{e}'\{\mathtt{e}/\mathtt{x}\} \Downarrow \mathtt{v}}{\mathtt{Case}\ \mathtt{e}\ \mathtt{of}\ \mathtt{x} \to \mathtt{e}' \Downarrow \mathtt{v}}\ (\mathrm{pm}_b)$$

- For simplicity we do not consider sharing.

Some useful programs

In the sequel we need some programs:

$$\begin{array}{lll} \texttt{take} :: \texttt{Nat} \to [\alpha] \to [\alpha] & & \\ \texttt{take} \quad 0 \quad \texttt{s} & = & \texttt{nil} \\ \texttt{take} \quad (\texttt{x}+1) \quad \texttt{nil} & = & \texttt{nil} \\ \texttt{take} \quad (\texttt{x}+1) \quad (\texttt{y}:\texttt{ys}) & = & \texttt{y} : (\texttt{take}\ \texttt{x}\ \texttt{ys}) \end{array}$$

$$\begin{array}{lll} \texttt{lg} :: [\alpha] \to \texttt{Nat} & & \\ \texttt{lg} \quad \texttt{nil} & = & \underline{0} \\ \texttt{lg} \quad \mathbf{Err} & = & \underline{0} \\ \texttt{lg} \quad (\texttt{x}:\texttt{xs}) & = & (\texttt{lg}\ \texttt{xs}) + 1 \end{array}$$

$$\begin{array}{lll} !! :: [\alpha] \to \texttt{Nat} \to \alpha & & \\ (\texttt{x}:\texttt{xs}) \quad !! \quad 0 & = & \texttt{x} \\ (\texttt{x}:\texttt{xs}) \quad !! \quad (\texttt{y}+1) & = & \texttt{xs} \,!!\, \texttt{y} \end{array}$$

$$\begin{array}{lll} \texttt{lInd} :: \texttt{Nat} \to [\alpha] \to [\alpha] & & \\ \texttt{lInd} \quad \texttt{x} \quad \texttt{xs} & = & (\texttt{xs} \,!!\, \texttt{x}) : \texttt{nil} \end{array}$$

Moreover we need a program

$$\begin{array}{lll} \texttt{eval} :: \texttt{A} \to \texttt{A} & & \\ \texttt{eval} \quad (\mathbf{c}\ \texttt{e}_1 \cdots \texttt{e}_n) & = & \hat{\texttt{C}}\ (\texttt{eval}\ \texttt{e}_1) \cdots (\texttt{eval}\ \texttt{e}_m) \end{array}$$

where $\hat{\texttt{C}}$ is a program representing the *strict* version of the primitive constructor **c**.

Interpretation

Definition [Interpretation] A program P admits an interpretation if there is an assignment $(\!|-|\!)$:

- total: $\forall \mathtt{t}$, $(\!|\mathtt{t}|\!) : (\mathbb{R}^+)^n \to \mathbb{R}^+$
- monotonic: $X_i \geq Y_i \Rightarrow (\!|\mathtt{t}|\!)(\ldots, X_i, \ldots) \geq (\!|\mathtt{t}|\!)(\ldots, Y_i, \ldots)$,

such that for each definition in P of the shape $\mathtt{f}\ \overrightarrow{\mathtt{p}} = \mathtt{e}$:

$$(\!|\mathtt{f}\ \overrightarrow{\mathtt{p}}|\!) \geq (\!|\mathtt{e}|\!)$$

A program P admits an additive interpretation if $(\!|-|\!)$ is such that for every symbol $\boldsymbol{c}$ of arity n:

- $(\!|\boldsymbol{c}|\!) = 0 \quad$ if $n = 0$
- $(\!|\boldsymbol{c}|\!)(X_1, \cdots, X_n) = \sum_{i=1}^{n} X_i + \alpha_{\boldsymbol{c}}$, with $\alpha_{\boldsymbol{c}} \geq 1$ otherwise

Properties

Lemma [Interpretation respects evaluation] Given a program P admitting the interpretation $(\!|-|\!)$, for every closed expression $\mathtt{e}$ we have

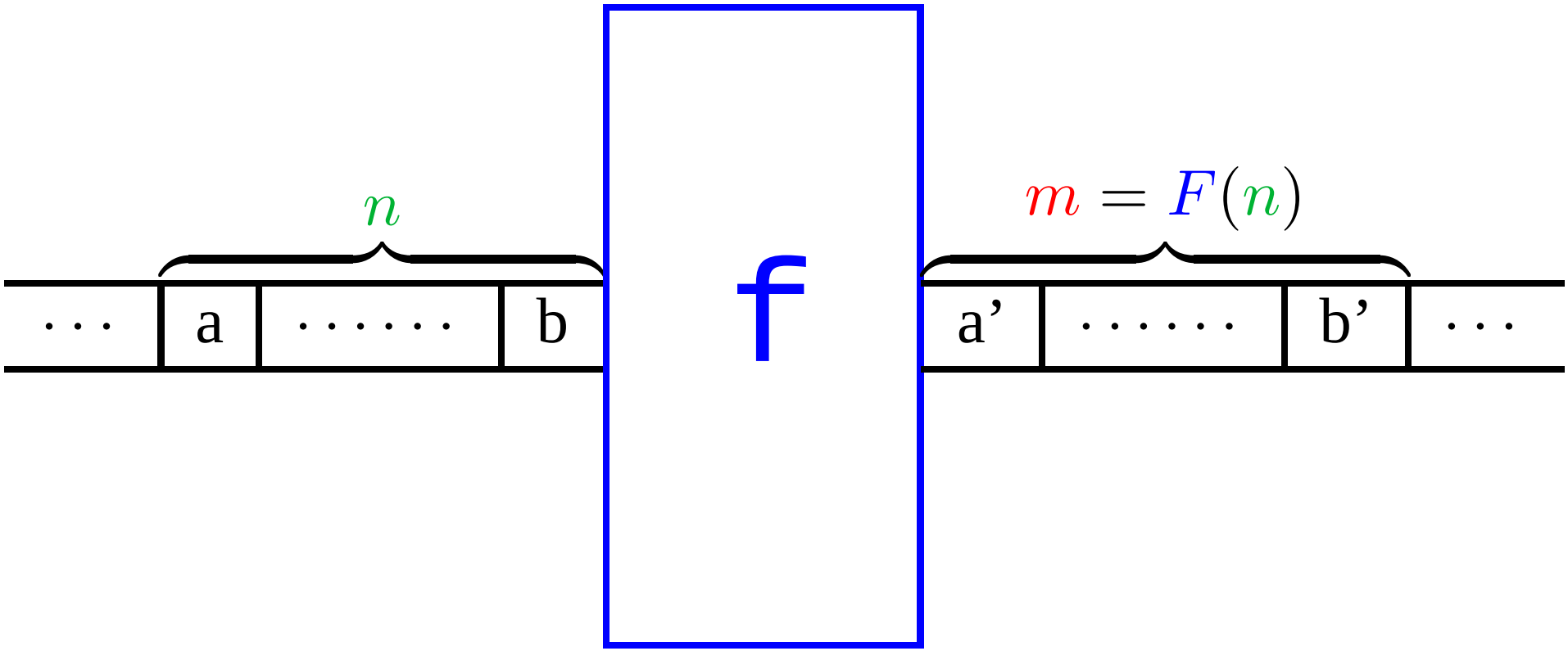
- if $\mathtt{e} \Downarrow \mathtt{v}$ then $(\!|\mathtt{e}|\!) \geq (\!|\mathtt{v}|\!)$
- if $\mathtt{eval\ e} \Downarrow \mathtt{v}$ then $(\!|\mathtt{e}|\!) \geq (\!|\mathtt{v}|\!)$

Lemma [Interpretation vs. expression size] Given a program P admitting the interpretation $(\!|-|\!)$, there is a function $G : \mathbb{R}^+ \to \mathbb{R}^+$ such that for each expression $\mathtt{e}$:

$$(\!|\mathtt{e}|\!) \leq G(|\mathtt{e}|)$$

Length Based I/O Upper Bound

Length Based I/O Upper Bound - Intuition



Length Based I/O Upper Bound - Formally

- **Definition**

  A stream function $\texttt{f} :: [\sigma'] \to \tau \to [\sigma]$ has a length based I/O upper bound if there is $F : \mathbb{R}^+ \to \mathbb{R}^+$ such that $\forall \texttt{s} :: [\sigma'], \forall \texttt{e} :: \tau$:

  $$\forall\, \underline{\texttt{n}} \in \mathbb{N},\ s.t.\ \texttt{eval}(\texttt{lg}(\texttt{f}\ (\texttt{take}\ \underline{\texttt{n}}\ \texttt{s})\ \texttt{e})) \Downarrow \underline{\texttt{m}},\ F(\max(|\underline{\texttt{n}}|, |\texttt{e}|)) \geq |\underline{\texttt{m}}|$$

- **Examples**

  $\texttt{double} :: [\texttt{Nat}] \to [\texttt{Nat}]$
  $\texttt{double}\ (\texttt{x} : \texttt{xs}) = \texttt{x} : (\texttt{x} : (\texttt{double}\ \texttt{xs}))$

  $\texttt{merge} :: [\alpha] \to [\alpha] \to [\alpha \times \alpha]$
  $\texttt{merge}\ (\texttt{x} : \texttt{xs})\ (\texttt{y} : \texttt{ys}) = (\texttt{x}, \texttt{y}) : (\texttt{merge}\ \texttt{xs}\ \texttt{ys})$

□ **Definition**

A program is LBUB if it admits an interpretation $(\!|-|\!)$ which satisfies $(\!|+1|\!)(X) = X + 1$ and which is additive but on the constructor symbol : where $(\!|:|\!)$ is defined as

$$(\!|:|\!)(X, Y) = Y + 1$$

□ **Lemma**

Given a LBUB program wrt the interpretation $(\!|-|\!)$:

– for every $\underline{\texttt{n}}$ :: `Nat` we have $(\!|\underline{\texttt{n}}|\!) = |\underline{\texttt{n}}|$.

– the interpretation can be extended to the program `lg` by $(\!|\texttt{lg}|\!)(X) = X$ and to the program `take` by $(\!|\texttt{take}|\!)(N, L) = N$

□ **Theorem**

If a program is LBUB then each stream function in it has a length based I/O upper bound.

The program $\texttt{double}$ is LBUB, it admits the additive interpretation

$$(\!|\texttt{double}|\!)(X) = 2X \qquad (\!|:|\!)(X, Y) = Y + 1$$

We have:

$$\begin{aligned}
(\!|\texttt{double}\ (\texttt{x} : \texttt{xs})|\!) &= (\!|\texttt{double}|\!)\ (\!|\texttt{x} : \texttt{xs}|\!) \\
&= 2(\!|\texttt{x} : \texttt{xs}|\!) \\
&= 2((\!|\texttt{xs}|\!) + 1) \\
&= (\!|\texttt{double}\ \texttt{xs}|\!) + 2 \\
&= (\!|\texttt{x} : (\texttt{x} : (\texttt{double}\ \texttt{xs}))|\!)
\end{aligned}$$

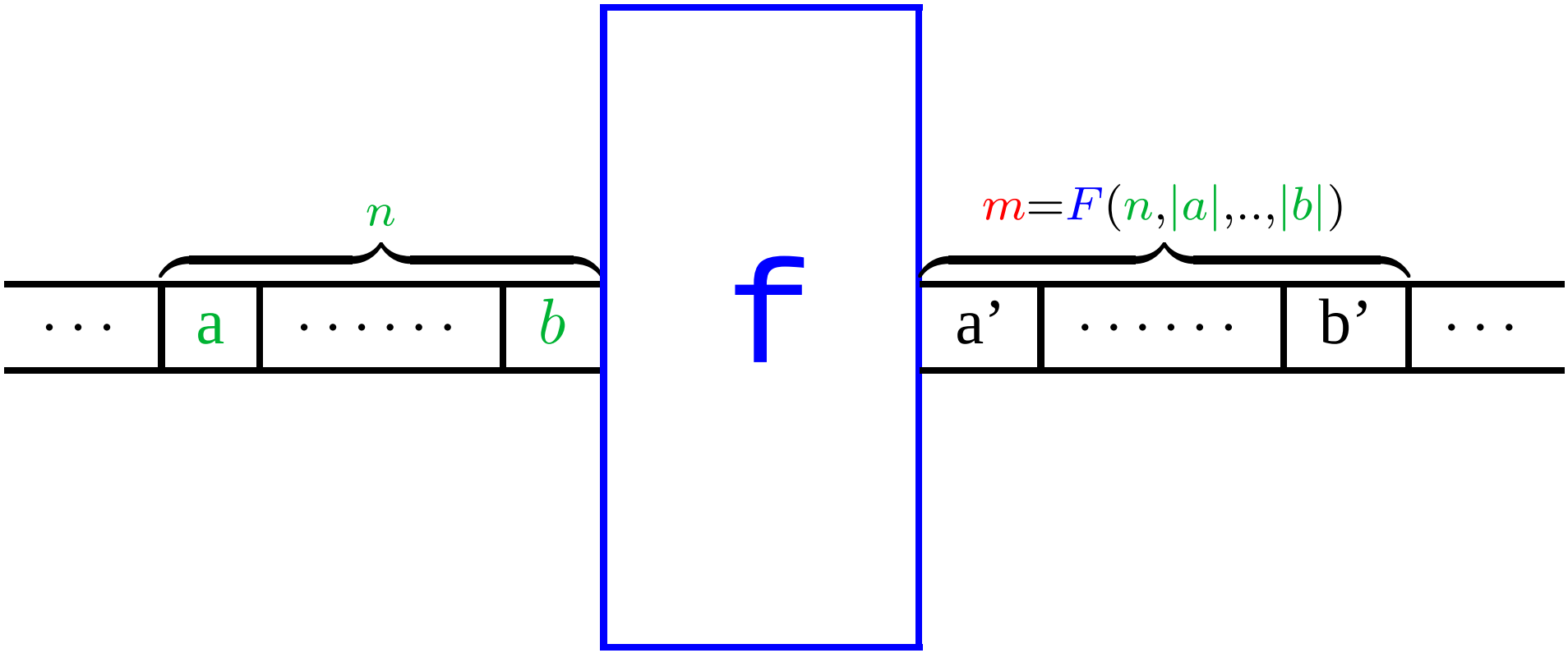
By defining $F(X) = (\!|\texttt{double}|\!)(X)$ we have that if

$$\texttt{eval}(\texttt{lg}(\texttt{double}\ (\texttt{take}\ \underline{\texttt{n}}\ \texttt{s}_1))) \Downarrow \underline{\texttt{m}}$$

then $F(|\underline{\texttt{n}}|) = 2n \geq |\underline{\texttt{m}}|$.

Size Based I/O Upper Bound

Size Based I/O Upper Bound - Intuition



Size Based I/O Upper Bound - Formally

- **Definition** A stream function $\texttt{f} :: [\sigma'] \to \tau \to [\sigma]$ has a size based I/O upper bound if there is $F : \mathbb{R}^+ \to \mathbb{R}^+$ such that $\forall \texttt{s} :: [\sigma'], \forall \texttt{e} :: \tau$:

  $$\forall\, \underline{\texttt{n}}_i \in \texttt{N},\ s.t.\ \texttt{eval}(\texttt{lg}(\texttt{f}\ (\texttt{take}\ \underline{\texttt{n}}\ \texttt{s})\ \texttt{e})) \Downarrow \underline{\texttt{m}},\ F(\max(|\underline{\texttt{n}}|, |\texttt{s}|, |\texttt{e}|)) \geq |\underline{\texttt{m}}|$$

- **Example**

```
app :: [α] → [α] → [α]
app  (x : xs)  ys  =  x : (app xs ys)
app   nil      ys  =      ys
```

```
upto :: Nat → [Nat]
upto    0     =        nil
upto  (x + 1) =  (x + 1) : (upto x)
```

```
extendupto :: [Nat] → [Nat]
extendupto (x : xs) = app (upto x) (extendupto xs)
```

Note that `extendupto` has not a length based I/O upper bound, nevertheless it has a size based I/O upper bound.

SBUB **Criterion**

- **Definition**
  A program is SBUB if it admits an additive interpretation $(\!|-|\!)$ satisfying

$$(\!|+1|\!)(X) = X + 1 \text{ and } (\!|:|\!)(X, Y) = X + Y + 1$$

- **Lemma**
  Given a SBUB program wrt the interpretation $(\!|-|\!)$:
  - for every $\underline{\texttt{n}}$ :: `Nat` we have $(\!|\underline{\texttt{n}}|\!) = |\underline{\texttt{n}}|$.
  - the interpretation can be extended to the program `lg` by $(\!|\texttt{lg}|\!)(X) = X$ and to the program `take` by $(\!|\texttt{take}|\!)(N, L) = L$

- **Theorem**
  If a program is SBUB then each stream function in it has a size based I/O upper bound.

The program `extendupto` is SBUB, it admits the additive interpretation

$$(\!|\texttt{upto}|\!)(X) = (\!|\texttt{extendupto}|\!)(X) = 2X^2 \qquad (\!|\texttt{app}|\!)(X,Y) = X + Y$$
$$(\!|:|\!)(X,Y) = X + Y + 1$$

We have:

$$\begin{aligned}
(\!|\texttt{extendupto}\ (\texttt{x} : \texttt{xs})|\!) &= (\!|\texttt{extendupto}|\!)\ (\!|\texttt{x} : \texttt{xs}|\!) \\
&= 2(\!|\texttt{x} : \texttt{xs}|\!)^2 \\
&= 2((\!|\texttt{x}|\!) + (\!|\texttt{xs}|\!) + 1)^2 \\
&= 2(\!|\texttt{x}|\!)^2 + 2(\!|\texttt{xs}|\!)^2 \\
&= (\!|\texttt{app}\ (\texttt{upto}\ \texttt{x})\ (\texttt{extendupto}\ \texttt{xs})|\!)
\end{aligned}$$

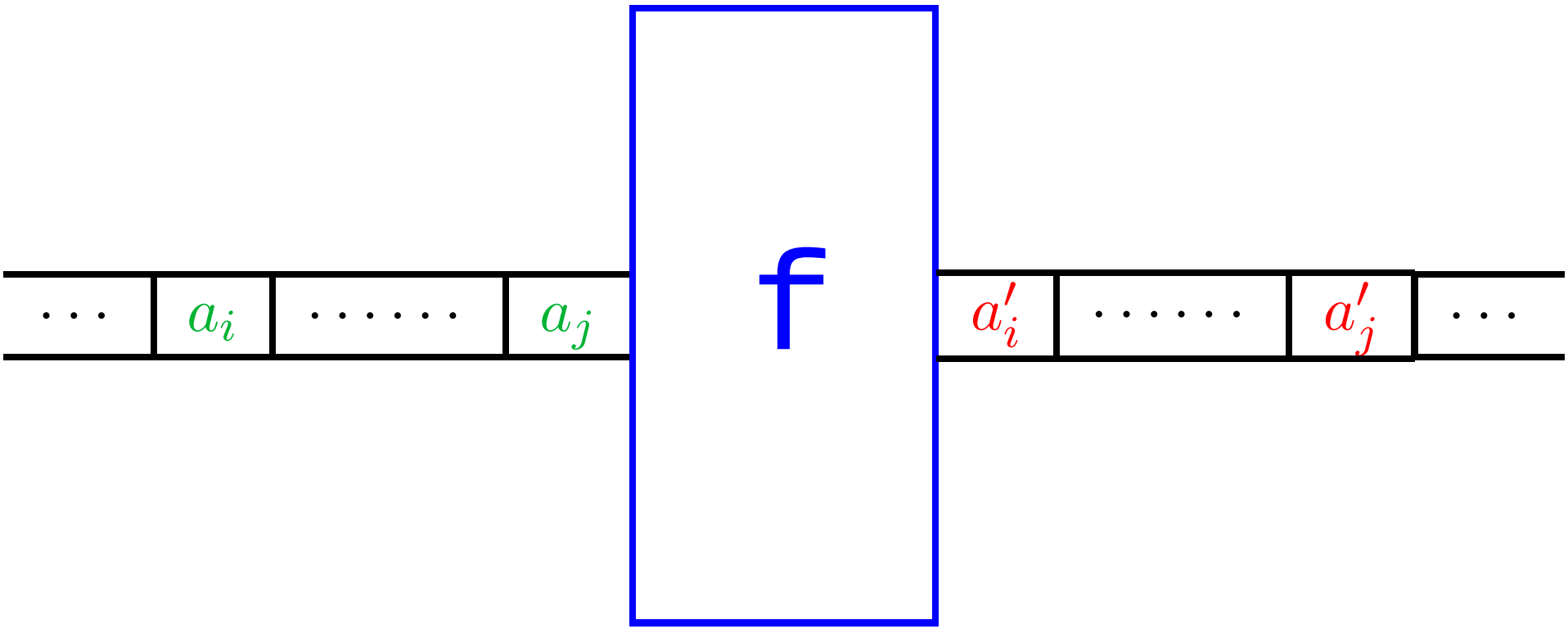
By defining $F(X) = (\!|\texttt{extendupto}|\!)(X)$ we have that if

$$\texttt{eval}(\texttt{lg}(\texttt{extendupto}\ (\texttt{take}\ \underline{\texttt{n}}\ \texttt{s}))) \Downarrow \underline{\texttt{m}}$$

then $F(\max(|\underline{\texttt{n}}|, |\texttt{s}|)) \geq |\underline{\texttt{m}}|$.

Synchrony Upper Bound

Synchrony Upper Bound - Intuition



where $|a'_i| = F(|a_i|), \ldots, |a'_j| = F(|a_j|)$.

Synchrony Upper Bound - Formally

- **Definition** A stream function $\mathtt{f} :: [\sigma'] \to \tau \to [\sigma]$ is said to be "Read, Write" if $\forall \mathtt{s} :: [\sigma']$, $\forall \mathtt{e} :: \tau$ and $\forall \underline{\mathtt{n}} \in \mathtt{N}$:

  If $\mathtt{eval}\ (\mathtt{lInd}\ \underline{\mathtt{n}}\ (\mathtt{f}\ \mathtt{s}\ \mathtt{e})) \Downarrow \mathtt{v}$ and $\mathtt{eval}\ (\mathtt{f}\ (\mathtt{lInd}\ \underline{\mathtt{n}}\ \mathtt{s})\ \mathtt{e}) \Downarrow \mathtt{v}'$ then $\mathtt{v} = \mathtt{v}'$

- **Definition** A "Read, Write" function $\mathtt{f} :: [\sigma'] \to \tau \to [\sigma]$ has a synchrony upper bound if there is $F : \mathbb{R}^+ \to \mathbb{R}^+$ such that $\forall \mathtt{s} :: [\sigma']$, $\forall \mathtt{e} :: \tau$ and $\underline{\mathtt{n}} \in \mathtt{N}$:

  If $\mathtt{eval}(\mathtt{s}\ !!\ \underline{\mathtt{n}}) \Downarrow \mathtt{w}$ and $\mathtt{eval}((\mathtt{f}\ \mathtt{s}\ \mathtt{e})\ !!\ \underline{\mathtt{n}}) \Downarrow \mathtt{v}$ then $F(\max(|\mathtt{w}|, |\mathtt{e}|)) \geq |\mathtt{v}|$

- **Example**

```
sadd :: [Nat] → [Nat] → [Nat]
sadd  (x : xs)  (y : ys)  =  (add x y) : (sadd xs ys)

add :: Nat → Nat → Nat
add  (x + 1)  (y + 1)  =  ((add x y) + 1) + 1
add  (x + 1)      0    =          x + 1
add      0    (y + 1)  =          y + 1
```

SUB **Criterion**

- **Definition** A stream function $\mathtt{f} :: [\sigma'] \to \tau \to [\sigma]$ is synchronously restricted if it can be written (and maybe extended) by definitions of the shape:

$$\begin{array}{llll} \mathtt{f} & (\mathtt{x} : \mathtt{xs}) & \overrightarrow{\mathtt{p}} & = \quad \mathtt{hd} : (\mathtt{f}\ \mathtt{xs}\ \overrightarrow{\mathtt{p}}) \\ \mathtt{f} & \mathtt{nil} & \overrightarrow{\mathtt{p}} & = \quad \mathtt{nil} \end{array}$$

  where $\mathtt{xs}$ do not appear in the expression $\mathtt{hd}$.

- **Lemma**
  Every synchronously restricted function is "Read, Write".

- **Definition** A program is SUB if it is synchronously restricted and it admits an additive interpretation $(\!|-|\!)$ but on : where

$$(\!|:|\!)(X, Y) = X$$

- **Theorem**
  If a program is SUB then each stream function in it admits a synchrony upper bound.

SUB **Example**

The program `sadd` is SUB because it is synchronously restricted and it admits the additive interpretation

$$(\!|\texttt{add}|\!)(X,Y) = X + Y \qquad (\!|\texttt{sadd}|\!)(X,Y) = X + Y \qquad (\!|:|\!)(X,Y) = X$$

We have:

$$\begin{array}{rcl}(\!|\texttt{sadd (x : xs) (y : ys)}|\!) & = & (\!|\texttt{sadd}|\!)\ (\!|\texttt{x : xs}|\!)\ (\!|\texttt{y : ys}|\!) \\ & = & (\!|\texttt{x : xs}|\!) + (\!|\texttt{y : ys}|\!) \\ & = & (\!|\texttt{x}|\!) + (\!|\texttt{y}|\!) \\ & = & (\!|\texttt{add x y}|\!) \\ & = & (\!|\texttt{(add x y) : (sadd xs ys)}|\!)\end{array}$$

By defining $F(X) = (\!|\texttt{sadd}|\!)(X,X)$ we have that if

$\texttt{eval}\,(\texttt{(x : xs)}!!\underline{\texttt{n}}) \Downarrow \texttt{w}$, $\texttt{eval}\,(\texttt{(y : ys)}!!\underline{\texttt{n}}) \Downarrow \texttt{w}'$ and $\texttt{eval}(\texttt{(sadd (x : xs) (y : ys)}!!\underline{\texttt{n}}) \Downarrow \texttt{v}$

then $F(\max(|\texttt{w}|, |\texttt{w}'|)) \geq |\texttt{v}|$.