Global and local space properties of stream programs

Marco Gaboardi, Romain Péchoux

Global and local space properties of stream programs⋆

Marco Gaboardi[1] and Romain Péchoux[2]

[1] Dip. di scienze dell'Informazione, Università di Bologna & INRIA project Focus
[2] Université Nancy 2, Nancy-université & INRIA project Carte, Loria
`gaboardi@cs.unibo.it,pechoux@loria.fr`

Abstract. In this paper, we push forward the approach proposed in [1] aiming at studying semantic interpretation criteria for the purpose of ensuring safety and complexity properties of programs working on streams. The paper improves the previous results by considering global and local upper bounds properties of both theoretical and practical interests guaranteeing that the size of each output stream element is bounded by a function in the maximal size of the input stream elements. Moreover, in contrast to previous studies, these properties also apply to a wide class of stream definitions, that is functions that do not have streams in the input but produce an output stream.

1 Introduction

A wider interest for infinite data structures and, particularly, streams has emerged in the last past two decades. Indeed, advances in computer networking combined with the creation of modern operating systems have made streaming practical and affordable for ordinary computers, thus leading streaming to become one of the most used network technologies. This technological jump has coincided with a renewal of interest in theoretical infinitary models and studies.

Stream-like language and properties Several formal frameworks have been designed for the manipulation of infinite objects including infinitary rewriting [2], infinitary lambda-calculus [3] and computable analysis, which provides different models of computation over real numbers [4]. Important properties of these models such as infinitary weak and strong normalizations and complexity classes definitions and characterizations, among others, have been deeply studied in the literature. Another interesting approach to deal with infinite data is the use of laziness in functional programming languages [5]. In languages like Haskell, streams are list expressions whose elements are evaluated on demand. In this way streams can be treated by finitary means.
In parallel, several studies have emerged on the underlying theories. Many efforts have been made on studying tools and techniques, as co-induction and bisimulation, to prove stream program equivalence [6, 7]. Other studies have been made

---

to develop techniques for ensuring productivity, a notion introduced in [8]. A stream definition is productive if it can be effectively evaluated in a unique constructor infinite normal form. Productivity is in general undecidable, so, many restricted languages and restricting criteria have been studied to ensure it [9–13]. Besides program equivalence and productivity, other stream program properties, in particular complexity-related properties, surprisingly have received little attention. Some interesting considerations about space complexity properties of streams programs, as buffering and overflow, have been made in [11, 14], suggesting that space complexity aspects of streams are of real practical interest. The sized type based tools and techniques developed in [11] had a wide diffusion in the study of program properties. For instance, they have been used in [15] to analyze complexity properties of strict programs working on lists. Nevertheless, a general study of practical stream program properties based on size analysis is lacking.

From a theoretical point of view, the fact that usual tools of complexity theory, well behaving on inductive data types, cannot be directly applied to streams suggests that an extensive study of complexity properties of stream programs is necessary. Some preliminaries results in this direction have been obtained in [16] where the polynomial time complexity of infinite data type programs is studied using a type-based restricted language with co-inductive types.

Contribution In [1], we considered a small stream Haskell-like first order language and we started the study of some *space properties*, i.e. properties about the size of stream elements. We presented a new method that use semantic interpretations in order to ensure I/O and synchrony upper bounds properties for functions working on streams. The semantic interpretations used there are extensions of the notions of quasi-interpretation [17] and sup-interpretation [18], introduced to obtain upper bounds on finitary term rewriting systems.

The method introduced in [1] is promising and is well adapted to purely operational reasoning, nevertheless the properties studied there are limited to properties about *functions* working on input streams and they do not consider *definitions* of streams, i.e. functions that do not have streams in input and that produce an output stream. For example, such properties do not hold even for simple examples of stream programs that include stream definitions like the program computing the Fibonacci sequence or like the following program:

```
ones :: [Nat]              nats :: Nat → [Nat]
ones = 1 : ones            nats x = x : (nats (x + 1))
```

In the present work, we generalize the method of [1], in order to study *properties of stream programs*, i.e. properties of both functions working on streams and stream definitions. In particular, we study space properties of streams like `nats` or `ones`. We design criteria to ensure a *global* and a *local* space upper bound properties. Consider the following stream program:

```
repeat :: Nat → [Nat]          zip :: [α] → [α] → [α]
repeat x = x : (repeat x)      zip (x : xs) ys = x : (zip ys xs)
```

It is easy to verify that the size of every element of a stream `s` built only using `repeat` and `zip` is bounded by a constant k, i.e. the maximal natural number $\underline{n}$ in a subterm `repeat` $\underline{n}$ in `s`. In particular, it means that every stream `s` built only using `repeat` and `zip` is globally bounded by a constant k. In order to generalize this property, we study a *Global Upper Bound* (GUB) property ensuring that the size of stream elements is bounded by a function in the maximal size of the input elements. Analogously, consider the following stream program:

```
nats :: Nat → [Nat]              sad :: [Nat] → [Nat] → [Nat]
nats x = x : (nats (x + 1))      sad (x : xs) (y : ys) = (add x y) : (sad xs ys)
```

Clearly, every stream `s` built using `nats` and `sad` is not globally bound. Nevertheless it is easy to verify that for every such an `s` there exists a function f such that every element `a` of `s` in the *local* position n has a size bounded by $f(n)$. In order to generalize this property, we study a *Local Upper Bound* (LUB) property ensuring that the size of the n-th evaluated element of a stream is bounded by a function in its index n and the maximal size of the input. The above properties, very natural from a complexity theory point of view, are also of practical interest since they can be used to prove upper bounds on classical stream examples. For this reason, we study two distinct criteria guaranteeing them.

Previous works The stream language introduced in the paper is the same than in [1]. From a technical point of view, in order to ensure the global upper bound property, we need to simply adapt the interpretation tools developed in [1]. The result is simple but gives an interesting insight on the way global properties of infinite data types can be given in a finitary way. Conversely, in order to ensure the local upper bound we need an extension of the usual semantic interpretation. In particular, we introduce the new notion of *parametrized semantic interpretation*, i.e. semantic interpretation where functions depend on external parameters. The parametrized semantic interpretations allow us to ensure the local upper bound. Since parametrized interpretations and non parametrized interpretations that we use in this paper are extensions of quasi-interpretations, we also know that their synthesis (i.e. find an interpretation of a given program) is decidable for polynomials of bounded degree over real numbers [19, 20]. Consequently, they seem to be a pertinent tool for future developments.

Outline of the paper In Section 2, we introduce the language considered and notations. In Section 3, we recall some basic notions about interpretations. In Section 4, we study the global upper bound properties and the semantic interpretation criteria to ensure it. In Section 5, we introduce the local upper bound property, we generalize semantic interpretations to the parametrized ones and we study criteria to ensure it. Finally, in Section 6, we conclude and we sketch some further directions that we would like to investigate.

2 Preliminaries

2.1 Syntax of the first order sHask language

We consider the first order Haskell-like language sHask computing on stream data presented in [1]. Consider three disjoint sets $\mathcal{X}$, $\mathcal{C}$ and $\mathcal{F}$ representing the set of *variables*, the set of *constructor symbols* and, respectively, the set of *function symbols*. A sHask program consists in a set of definitions described in the the grammar of Table 1, where $x \in \mathcal{X}$, $c \in \mathcal{C}$ and $f \in \mathcal{F}$. Throughout the paper, we use the identifier t to represent a symbol in $\mathcal{C} \cup \mathcal{F}$ and the notation $\bar{d}$, for some identifier d, to represent the sequence $d_1, \ldots, d_n$, whenever n is clear from the context. We will also use the notation $t \overrightarrow{e}$ as a shortcut notation for the application $t\ e_1\ \cdots\ e_n$, whenever t is a symbol of arity n. Notice that, as usual, application associates to the left. Moreover, we distinguish a special *error symbol* **Err** in $\mathcal{C}$ of arity 0 corresponding to pattern matching failure.

The language sHask includes a Case operator to carry out pattern matching and first order program definitions. Note that Case operator can appear only in

$$
\begin{array}{ll}
p ::= x \mid c\ p_1\ \cdots\ p_n & \text{(Patterns)} \\
e ::= x \mid t\ e_1\ \cdots\ e_n & \text{(Expressions)} \\
v ::= c\ e_1\ \cdots\ e_n & \text{(Values)} \\
\underline{v} ::= c\ \underline{v}_1\ \cdots\ \underline{v}_n & \text{(CValues)} \\
d ::= f\ x_1\ \cdots\ x_n\ = \text{Case } \bar{e} \text{ of } \overline{p_1} \to e_1 \ldots \overline{p_m} \to e_m & \text{(Definitions)}
\end{array}
$$

Table 1. sHask syntax

definitions. In this, our grammar presentation differs from the one in [1], but there we also have some additional conditions that turn the definitions to be exactly the same considered here. We will use set of definitions $f\ \overrightarrow{p_1} = e_1, \ldots, f\ \overrightarrow{p_k} = e_k$ as *syntactic sugar* for an expression of the shape $f\ \overrightarrow{x} = \text{Case } \bar{x} \text{ of } \overline{p_1} \to e_1, \ldots, \overline{p_k} \to e_k$.

Finally, we suppose that all the free variables contained in the expression e_i of a case expression appear in the patterns $\overline{p_i}$, that no variable occurs twice in $\overline{p_i}$ and that patterns are non-overlapping. It entails that programs are confluent. For simplicity, we only consider well-typed first order programs dealing with lists

$$
\frac{}{x :: A}\ (\text{Var}) \qquad \frac{}{\textbf{Err} :: A}\ (\text{E}) \qquad \frac{\bar{e} :: \bar{A} \quad \overline{p_1} :: \bar{A} \quad \cdots \quad \overline{p_m} :: \bar{A} \quad e_1 :: A \quad \cdots \quad e_m :: A}{\text{Case } \bar{e} \text{ of } \overline{p_1} \to e_1, \ldots, \overline{p_m} \to e_m :: A}\ (\text{Case})
$$

$$
\frac{}{t :: A_1 \to \cdots \to A_n \to A}\ (\text{Tb}) \qquad \frac{t :: A_1 \to \cdots \to A_n \to A \quad e_1 :: A_1 \quad \cdots \quad e_n :: A_n}{t\ e_1\ \cdots\ e_n :: A}\ (\text{Ts})
$$

Table 2. sHask type system

that do not contain other lists. We assure this property by a typing restriction similar to the one of [14].

Definition 1. *Let $\mathcal{S}$ be the set of* basic and value types *defined by the following grammar:*

$$\sigma ::= \alpha \mid \texttt{Nat} \mid \sigma \times \sigma \qquad \text{(basic types)}$$
$$\texttt{A} ::= a \mid \sigma \mid \texttt{A} \times \texttt{A} \mid [\sigma] \qquad \text{(value types)}$$

where α is a basic type variable*, a is a* value type variable*, $\texttt{Nat}$ is a* constant type *representing natural numbers, $\times$ and $[\ ]$ are* type constructors*.*

Notice that the above definition can be extended to standard algebraic data types. In the sequel, we use α, β to denote basic type variables, a, b to denote value type variables, σ, τ to denote basic types and $\texttt{A}, \texttt{B}$ to denote value types. As in Haskell, we allow restricted polymorphism, i.e. a basic type variable α and a value type variable a represent every basic type and respectively every value type. As usual, $\rightarrow$ associates to the right. For notational convenience, we will use $\overrightarrow{\texttt{A}} \rightarrow \texttt{B}$ as an abbreviation for $\texttt{A}_1 \rightarrow \cdots \rightarrow \texttt{A}_n \rightarrow \texttt{B}$.

Every function and constructor symbol $\texttt{t}$ of arity n come equipped with a type $\texttt{A}_1 \rightarrow \cdots \rightarrow \texttt{A}_n \rightarrow \texttt{A}$. Well typed symbols, patterns and expressions are defined using the type system in Table 2. Note that the symbol **Err** can be typed with every value type $\texttt{A}$ in order to get type preservation in the evaluation mechanism. Moreover, it is worth noting that the type system, in order to allow only first order function definitions, assigns functional types to constructors and function symbols, but only value types to expressions. Typing judgments are of the shape $\texttt{t} :: \overrightarrow{\texttt{A}} \rightarrow \texttt{B}$, for some symbol $\texttt{t}$ and some type $\overrightarrow{\texttt{A}} \rightarrow \texttt{B}$. In our examples, we will only consider three standard data types: numerals, lists and pairs, encoded by the constructor symbols 0 and infix $+1$, $\texttt{nil}$ and infix : and, respectively, $(\_, \_)$. In this work, we are specifically interested in studying stream properties. So we pay attention to particular classes programs working on $[\alpha]$, the type of both finite and infinite lists of type α. In what follows we use a terminology slightly different from the one used in [1]. A function symbol $\texttt{f}$ is called a *stream function* if $\texttt{f} :: [\sigma_1] \rightarrow \cdots \rightarrow [\sigma_n] \rightarrow \overrightarrow{\tau} \rightarrow [\sigma]$ with $n > 0$. In the case where $\texttt{f} :: \overrightarrow{\tau} \rightarrow [\sigma]$, the function symbol $\texttt{f}$ is called a *stream constructor*. Given a definition $\texttt{f} \overrightarrow{\texttt{p}} = \texttt{e}$ we say that it is a *function definition* if $\texttt{f}$ is a stream function, otherwise if $\texttt{f}$ is a stream constructor we say that it is a *stream definition*. In what follows, we will in general talk about properties of function symbols to stress that such properties holds both for functions and stream definitions.

Example 1. Consider the following program:

```
zip :: [α] → [α] → [α]              nats :: Nat → [Nat]
zip (x : xs) ys = x : (zip ys xs)   nats x = x : (nats (x + 1))
```

$\texttt{zip}$ is a stream function and $\texttt{nats}$ is a stream constructor.

2.2 $\texttt{sHask}$ lazy operational semantics

The $\texttt{sHask}$ language has a standard lazy operational semantics, where sharing is not considered. The semantics is described by the rules of Table 3 using substitutions, where a substitution σ (sometimes denoted $\{\texttt{e}_1/x_1, \ldots, \texttt{e}_n/x_n\}$) is a

mapping from variables to expressions. The computational domain is the set of `Values` defined in Table 1. `Values` are either particular expressions with a constructor symbol at the outermost position or the symbol **Err** corresponding to pattern matching errors. Note that the intended meaning of the notation $e \Downarrow v$ is that the expression e *evaluates* to the value $v \in$ `Values`. As usual in lazy semantics the evaluation does not explore the entire results but stops once the requested information is found.

Example 2. Consider again the program defined in Example 1. We have the following evaluations: $\texttt{nats } 0 \ \Downarrow \ 0 : (\texttt{nats } (0+1)))$ and $\texttt{zip nil } (\texttt{nats } 0) \ \Downarrow \textbf{Err}$.

$$\frac{\textbf{c} \in \mathcal{C}}{\textbf{c } e_1 \ \cdots \ e_n \Downarrow \textbf{c } e_1 \ \cdots \ e_n} \ (\text{val}) \qquad \frac{e\{e_1/x_1, \cdots, e_n/x_n\} \Downarrow v \quad \texttt{f } x_1 \ \cdots \ x_n = e}{\texttt{f } e_1 \ \cdots \ e_n \ \Downarrow v} \ (\text{fun})$$

$$\frac{\texttt{Case } e^1 \texttt{ of } p_1^1 \to \ldots \to \texttt{Case } e^m \texttt{ of } p_1^m \to d_1 \Downarrow v \quad v \neq \textbf{Err}}{\texttt{Case } \overline{e} \texttt{ of } \overline{p_1} \to d_1, \ldots, \overline{p_n} \to d_n \Downarrow v} \ (c_b)$$

$$\frac{\texttt{Case } e^1 \texttt{ of } p_1^1 \to \ldots \to \texttt{Case } e^m \texttt{ of } p_1^m \to d_1 \Downarrow \textbf{Err} \quad \texttt{Case } \overline{e} \texttt{ of } \overline{p_2} \to d_2, \ldots, \overline{p_n} \to d_n \Downarrow v}{\texttt{Case } \overline{e} \texttt{ of } \overline{p_1} \to d_1, \ldots, \overline{p_n} \to d_n \Downarrow v} \ (c)$$

$$\frac{e \Downarrow \textbf{c } e_1 \ \cdots \ e_n \quad \texttt{Case } e_1 \texttt{ of } p_1 \to \ldots \texttt{Case } e_n \texttt{ of } p_n \to d \Downarrow v}{\texttt{Case } e \texttt{ of } \textbf{c } p_1 \ \cdots \ p_n \to d \Downarrow v} \ (\text{pm})$$

$$\frac{e \Downarrow v \quad v \neq \textbf{c } e_1, \cdots, e_n}{\texttt{Case } e \texttt{ of } \textbf{c } p_1, \cdots, p_n \to d \Downarrow \textbf{Err}} \ (\text{pm}_e) \qquad \frac{e'\{e/x\} \Downarrow v}{\texttt{Case } e \texttt{ of } x \to e' \Downarrow v} \ (\text{pm}_b)$$

Table 3. `sHask` lazy operational semantics

2.3 Preliminary notions

In this section, we introduce some useful programs and notions in order to study stream properties by operational finitary means. First, we define the usual Haskell list indexing function !! which returns the n-th element of a list.

```
!! :: [α] → Nat → α
(x : xs) !!   0   =   x
(x : xs) !! (y + 1) = xs !! y
```

Second, we define a program `eval` that forces the (possibly diverging) full evaluation of expressions to constructor values in `CValues` described in Table 1, which are expressions composed only by constructors. We define `eval` for every value type `A` as:

```
eval :: A → A
eval  (c e1 · · · en) = Ĉ (eval e1) · · · (eval em)
```

where $\hat{\textbf{C}}$ is a function symbol representing the *strict* version of the primitive constructor **c**. For example in the case where **c** is $+1$ we can define $\hat{\textbf{C}}$ as the program

6

$\mathtt{succ} :: \mathtt{Nat} \to \mathtt{Nat}$ defined by $\mathtt{succ}\ 0 = 0 + 1$ and $\mathtt{succ}\ (\mathtt{x} + 1) = (\mathtt{x} + 1) + 1$. When we want to stress that an expression $\mathtt{e}$ is completely evaluated (i.e. evaluated to a constructor value) we denote it by $\underline{\mathtt{e}}$. A relevant set of completely evaluated expressions is the set $\mathbb{N} = \{\underline{\mathtt{n}} \mid \underline{\mathtt{n}} :: \mathtt{Nat}\}$ of *canonical numerals*. In general we write $\underline{0}, \underline{1}, \ldots$ for concrete instances of canonical numerals. Finally we introduce a notion of *size* for expressions.

Definition 2 (Size). *The* size *of a expression* $\mathtt{e}$ *is defined as*

$$|\mathtt{e}| = 0 \qquad\qquad\qquad \textit{if } \mathtt{e} \textit{ is a variable or a symbol of arity } 0$$

$$|\mathtt{e}| = \sum_{i \in \{1, \ldots, n\}} |\mathtt{e}_i| + 1 \qquad \textit{if } e = \mathtt{t}\ \mathtt{e}_1\ \cdots\ \mathtt{e}_n, \ \mathtt{t} \in \mathcal{C} \cup \mathcal{F}.$$

Note that for each $\underline{\mathtt{n}} \in \mathbb{N}$ we have $|\underline{\mathtt{n}}| = n$. Let $F(\overline{\mathtt{e}})$ denote the componentwise application of F to the sequence $\overline{\mathtt{e}}$ (i.e. $F(\mathtt{e}_1, \cdots, \mathtt{e}_n) = F(\mathtt{e}_1), \ldots, F(\mathtt{e}_n)$). For example, given a sequence $\overline{\mathtt{s}} = \mathtt{s}_1, \cdots, \mathtt{s}_n$, we use the notation $|\overline{\mathtt{s}}|$ for $|\mathtt{s}_1|, \ldots, |\mathtt{s}_n|$.

3 Interpretations

Now we introduce interpretations, our main tool in order to ensure stream properties. Throughout the paper, $\geq$ and $>$ denote the natural ordering on real numbers and its restriction.

Definition 3 (Assignment). *An assignment of a symbol* $\mathtt{t} \in \mathcal{F} \cup \mathcal{C}$ *of arity* n *is a function* $(\!|\mathtt{t}|\!) : (\mathbb{R}^+)^n \to \mathbb{R}^+$. *For each variable* $\mathtt{x} \in \mathcal{X}$, *we define* $(\!|\mathtt{x}|\!) = X$, *with* X *a fresh variable ranging over* $\mathbb{R}^+$. *This allows us to extend assignments* $(\!|-|\!)$ *to expressions canonically. Given an expression* $\mathtt{t}\ \mathtt{e}_1\ \ldots\ \mathtt{e}_n$ *with* m *variables, its assignment is a function* $(\mathbb{R}^+)^m \to \mathbb{R}^+$ *defined canonically by:*

$$(\!|\mathtt{t}\ \mathtt{e}_1\ \ldots\ \mathtt{e}_n|\!) = (\!|\mathtt{t}|\!)((\!|\mathtt{e}_1|\!), \cdots, (\!|\mathtt{e}_n|\!))$$

A program assignment is an assignment $(\!|-|\!)$ *defined for each symbol of the program. An assignment is* monotonic *if for any symbol* $\mathtt{t}$, $(\!|\mathtt{t}|\!)$ *is an increasing function, that is for every symbol* $\mathtt{t}$ *and all* X_i, Y_i *of* $\mathbb{R}^+$ *such that* $X_i \geq Y_i$, *we have* $(\!|\mathtt{t}|\!)(\ldots, X_i, \ldots) \geq (\!|\mathtt{t}|\!)(\ldots, Y_i, \ldots)$.

Now we define the notion of additive assignments which guarantees that the assignment of a constructor value remains affinely bounded by its size.

Definition 4. *An assignment of a symbol* $\mathbf{c}$ *of arity* n *is additive if*

$$(\!|\mathbf{c}|\!)(X_1, \cdots, X_n) = \sum_{i=1}^{n} X_i + \alpha_{\mathbf{c}}$$

with $\alpha_{\mathbf{c}} \geq 1$, *whenever* $n > 0$, *and* $(\!|\mathbf{c}|\!) = 0$ *otherwise. The assignment* $(\!|-|\!)$ *of a program is called* additive *assignment if each constructor symbol of* $\mathcal{C}$ *has an additive assignment.*

Additive assignments have the following interesting property.

Lemma 1. *Given an additive assignment $(\!|-|\!)$, there is a constant α such that for each constructor value $\underline{v}$, the following inequalities are satisfied:*

$$|\underline{v}| \leq (\!|\underline{v}|\!) \leq \alpha \times |\underline{v}|$$

Proof. The proof is in [21].

Definition 5 (Interpretation). *A program admits an interpretation $(\!|-|\!)$ if $(\!|-|\!)$ is a monotonic assignment such that for each definition of the shape $\mathtt{f}\ \overrightarrow{\mathtt{p}} = \mathtt{e}$ we have $(\!|\mathtt{f}\ \overrightarrow{\mathtt{p}}|\!) \geq (\!|\mathtt{e}|\!)$.*

Example 3. The following assignment $(\!|\mathtt{zip}|\!)(X, Y) = X + Y$ and $(\!|\mathtt{:}|\!)(X, Y) = X + Y + 1$ is an additive interpretation of the program $\mathtt{zip}$ of Example 1:

$$
\begin{aligned}
(\!|\mathtt{zip\ (x:xs)\ ys}|\!) &= (\!|\mathtt{zip}|\!)((\!|\mathtt{x:xs}|\!), (\!|\mathtt{ys}|\!)) &&\text{By canonical extension}\\
&= (\!|\mathtt{x:xs}|\!) + (\!|\mathtt{ys}|\!) &&\text{By definition of } (\!|\mathtt{zip}|\!)\\
&= (\!|\mathtt{x}|\!) + (\!|\mathtt{xs}|\!) + (\!|\mathtt{ys}|\!) + 1 &&\text{By definition of } (\!|\mathtt{:}|\!)\\
&= (\!|\mathtt{x:(zip\ ys\ xs)}|\!) &&\text{Using the same reasoning}
\end{aligned}
$$

Let $\rightarrow$ be the rewrite relation induced by giving an orientation from left to right to the definitions and let $\rightarrow^*$ be its contextual, transitive and reflexive closure. We start by showing some properties on monotonic assignments:

Proposition 1. *Given a program admitting the interpretation $(\!|-|\!)$, then for every closed expression $\mathtt{e}$ we have:*

1. *If $\mathtt{e} \rightarrow^* \mathtt{d}$ then $(\!|\mathtt{e}|\!) \geq (\!|\mathtt{d}|\!)$*
2. *If $\mathtt{e} \Downarrow \mathtt{v}$ then $(\!|\mathtt{e}|\!) \geq (\!|\mathtt{v}|\!)$*
3. *If $\mathtt{eval\ e} \Downarrow \underline{v}$ then $(\!|\mathtt{e}|\!) \geq (\!|\underline{v}|\!)$*

Proof. 1. By induction on the derivation length and can be found in [21].
2. It is a direct consequence of point 1 of this proposition because the lazy semantics is just a particular rewrite strategy.
3. By induction on the size of constructor values using point 2 of this proposition and monotonicity. □

It is important to relate the size of an expression and its interpretation.

Lemma 2. *Given a program having an assignment $(\!|-|\!)$, there is a function $G : \mathbb{R}^+ \rightarrow \mathbb{R}^+$ such that for each expression $\mathtt{e}$ we have: $(\!|\mathtt{e}|\!) \leq G(|\mathtt{e}|)$.*

Proof. By induction on the size of expressions, it can be found in [1]. □

In the following sections, we study global and local stream properties related to constructor value size upper bounds. Moreover, we introduce criteria that use interpretations to ensure them. Although these properties are mostly undecidable, the criteria we discuss are decidable when considering restricted classes of functions, for example polynomials over real numbers of bounded degree and coefficients, see [19, 20] for a more detailed discussion.

4 Global upper bound (GUB)

In studying space properties of programs it is natural to relate the output element size to the input size. Ensuring this provides interesting information about the complexity of functions computed by the corresponding program. However, when lazy languages and infinite data are considered, the standard complexity measure, i.e. the whole size of input, is nonsense. In these situations, a good compromise is to take the maximal size of a stream element as a parameter and to bound the maximal output element size by a function in the maximal input size. This is what we call a global upper bound because it bounds the size of output elements globally. Notice that, in general, such a definition has a meaning if the input stream has a maximal element size. This trivially holds when there is no input stream.

Definition 6. *Given a* sHask *program, the function symbol* $\mathtt{f} :: \overrightarrow{[\sigma]} \to \overrightarrow{\tau} \to [\sigma]$ *has a* global upper bound *if there is a function* $G : \mathbb{R}^+ \to \mathbb{R}^+$ *such that for every expression* $\mathtt{s}_i :: [\sigma_i]$ *and* $\mathtt{e}_i :: \tau_i$ *of the program:*

$$\forall \underline{\mathtt{n}} \in \mathtt{N}\ s.t.\ \mathtt{eval}((\mathtt{f}\ \overrightarrow{\mathtt{s}}\ \overrightarrow{\mathtt{e}})\ !!\ \underline{\mathtt{n}}) \Downarrow \underline{\mathtt{v}},\ G(\max(|\overline{\mathtt{s}}|, |\overline{\mathtt{e}}|)) \geq |\underline{\mathtt{v}}|$$

A program has a global upper bound *if every function symbol in it has a global upper bound.*

Note that in the definition above we consider both stream functions and stream constructors, this means that we can both have globally bounded function and stream definitions.

Example 4. The program consisting in the `zip` function definition of Example 1 together with the stream definition $\mathtt{ones} = \underline{\mathtt{1}} : \mathtt{ones}$ has a global upper bound. Let `e` be `zip ones ones`. We know that every element of `ones` has size bounded by the constant 1. Since for each $\underline{\mathtt{n}} \in \mathtt{N}$, $\mathtt{eval}(\mathtt{e}\ !!\ \underline{\mathtt{n}}) \Downarrow \underline{\mathtt{1}}$, by taking $G(X) = X + 1$, we obtain $|\underline{\mathtt{1}}| = 1 + |\underline{\mathtt{0}}| = 1 \leq 1 = G(0) \leq G(|\mathtt{ones}|)$.

Now, we define a criterion using interpretations in order to ensure the global upper bound.

Definition 7. *A program is* GUB *if it admits an interpretation* $(\!|-|\!)$ *that is additive but on the constructor* : *where* $(\!|:|\!)$ *is defined by* $(\!|:|\!)(X,Y) = \max(X,Y)$.

Lemma 3. *If a program is* GUB, $\forall \mathtt{e} :: [\sigma], \forall \underline{\mathtt{n}} \in \mathtt{N}\ s.t.\ \mathtt{e}\ !!\ \underline{\mathtt{n}} \Downarrow \mathtt{v}$ *and* $\mathtt{v} \neq$ **Err**, $(\!|\mathtt{e}|\!) \geq (\!|\mathtt{v}|\!)$.

Proof. We proceed by induction on $\underline{\mathtt{n}} \in \mathtt{N}$.
Let $\underline{\mathtt{n}} = \mathtt{0}$ and $\mathtt{e}\ !!\ \mathtt{0} \Downarrow \mathtt{v}$ and $\mathtt{v} \neq$ **Err**. It is easy to verify that necessarily $\mathtt{e} \Downarrow \mathtt{hd} : \mathtt{tl}$ because programs are well typed. By definition of GUB and by Proposition 1(2) we know that there is an interpretation such that $(\!|\mathtt{e}|\!) \geq (\!|\mathtt{hd} : \mathtt{tl}|\!) = (\!|:|\!)((\!|\mathtt{hd}|\!), (\!|\mathtt{tl}|\!)) \geq (\!|\mathtt{hd}|\!)$, because $(\!|:|\!)(X,Y) = \max(X,Y)$. Applying Proposition 1(2) again, we know that if $\mathtt{hd} \Downarrow \mathtt{v}$ then $(\!|\mathtt{hd}|\!) \geq (\!|\mathtt{v}|\!)$. So we have $(\!|\mathtt{e}|\!) \geq (\!|\mathtt{v}|\!)$ and then the conclusion.

Now, let $\underline{n} = \underline{n}' + 1$ and $e \mathbin{!!} (\underline{n}' + 1) \Downarrow v$ and $v \neq \mathbf{Err}$. It is easy to verify that necessarily $e \Downarrow \mathtt{hd} : \mathtt{tl}$ and again we have $(\!|e|\!) \geq (\!|\mathtt{hd} : \mathtt{tl}|\!) \geq (\!|\mathtt{tl}|\!)$. Moreover $e \mathbin{!!} (\underline{n}'+1) \Downarrow v$ implies by definition that $\mathtt{tl} \mathbin{!!} \underline{n}' \Downarrow v$ and by induction hypothesis we have $(\!|\mathtt{tl}|\!) \geq (\!|v|\!)$. So we can conclude $(\!|e|\!) \geq (\!|v|\!)$ and so the conclusion. $\qquad\square$

Theorem 1. *If a program is* GUB *then it has a global upper bound.*

Proof. It suffices to show that every function symbol has a global upper bound. For simplicity, we suppose that for each function symbol we have only one definition. The general case with several definitions follows directly. Consider a function symbol $\mathtt{f} :: \overrightarrow{[\sigma]} \to \overrightarrow{\tau} \to [\sigma]$ and a definition $\mathtt{f} \ \overrightarrow{\mathtt{p}_s} \ \overrightarrow{\mathtt{p}_b} = e$.
Let $\underline{n} \in \mathbb{N}$ and σ be a substitution and suppose $\mathtt{eval}((\mathtt{f} \ \overrightarrow{\mathtt{p}_s}\sigma \ \overrightarrow{\mathtt{p}_b}\sigma) \mathbin{!!} \underline{n}) \Downarrow \underline{v}$. It follows that $(\mathtt{f} \ \overrightarrow{\mathtt{p}_s}\sigma \ \overrightarrow{\mathtt{p}_b}\sigma) \mathbin{!!} \underline{n} \Downarrow v$, for some v such that $\mathtt{eval}\ v \Downarrow \underline{v}$. By Lemma 3, $(\!|\mathtt{f} \ \overrightarrow{\mathtt{p}_s}\sigma \ \overrightarrow{\mathtt{p}_b}\sigma|\!) \geq (\!|v|\!)$. By Proposition 1(3) $(\!|v|\!) \geq (\!|\underline{v}|\!)$ and, by Lemma 1, $(\!|\underline{v}|\!) \geq |\underline{v}|$. Hence we can conclude $(\!|\mathtt{f} \ \overrightarrow{\mathtt{p}_s}\sigma \ \overrightarrow{\mathtt{p}_b}\sigma|\!) \geq |\underline{v}|$.
By Lemma 2 and monotonicity, we know that there is a function $F : \mathbb{R}^+ \to \mathbb{R}^+$ such that $(\!|\mathtt{f}|\!) \ (F(|\overline{\mathtt{p}_s\sigma}|), F(|\overline{\mathtt{p}_b\sigma}|)) \geq (\!|\mathtt{f}|\!) \ ((\!|\overline{\mathtt{p}_s\sigma}|\!), (\!|\overline{\mathtt{p}_b\sigma}|\!)) = (\!|\mathtt{f} \ \overrightarrow{\mathtt{p}_s}\sigma \ \overrightarrow{\mathtt{p}_b}\sigma|\!)$. So we obtain a global upper bound by taking $G(X) = (\!|\mathtt{f}|\!)(F(\overline{X}), F(\overline{X}))$. $\qquad\square$

Example 5. The program consisting in the $\mathtt{zip}$ function definition of Example 1 together with the stream definition $\mathtt{ones} = \underline{1} : \mathtt{ones}$ is GUB wrt the following assignment $(\!|\mathtt{zip}|\!)(X,Y) = \max(X,Y)$, $(\!|\mathtt{ones}|\!) = 1$, $(\!|\underline{0}|\!) = 0$, $(\!|+1|\!)(X) = X + 1$ and $(\!|:|\!)(X,Y) = \max(X,Y)$. Indeed, we let the reader check that $(\!|\mathtt{ones}|\!) \geq (\!|\underline{1} : \mathtt{ones}|\!)$. Consequently, it admits a global upper bound. For example, taking $G(X) = X + 1$ and $F(X) = (\!|\mathtt{zip}|\!)(X,X)$ we know that $(\!|\mathtt{ones}|\!) \leq G(|\mathtt{ones}|)$ and we obtain that for all $\underline{n} \in \mathbb{N}$ such that $\mathtt{eval}((\mathtt{zip}\ \mathtt{ones}\ \mathtt{ones})\mathbin{!!}\underline{n}) \Downarrow \underline{v}_{\underline{n}}$, we have $F(G(|\mathtt{ones}|)) = F(G(0)) = F(1) = 1 \geq |\underline{v}_{\underline{n}}|$ (Indeed for all $\underline{n}$, we have $\underline{v}_{\underline{n}} = \underline{1}$).

Example 6 (Thue-Morse sequence). The following program computes the Thue-Morse sequence:

```
morse :: [Nat]                          tail :: [α]
morse = 0 : (zip (inv morse) (tail morse))   tail x : xs =  xs

inv :: [Nat] → [Nat]        zip :: [α] → [α] → [α]
inv 0 : xs = 1 : xs         zip (x : xs) ys = x : (zip ys xs)
inv 1 : xs = 0 : xs
```

Clearly this program has a global upper bound. Moreover, it is GUB with respect to the following interpretation: $(\!|\underline{0}|\!) = 0$, $(\!|+1|\!)(X) = 1$, $(\!|:|\!)(X,Y) = (\!|\mathtt{zip}|\!) = \max(X,Y)$, $(\!|\mathtt{inv}|\!)(X) = \max(1,X)$, $(\!|\mathtt{tail}|\!)(X) = X$ and $(\!|\mathtt{morse}|\!) = 1$. For instance, for the first rule, we have that for each $L \in \mathbb{R}$:

$$
\begin{aligned}
(\!|\mathtt{morse}|\!) = 1 &\geq \max(0,1,1,1) \\
&= \max((\!|\underline{0}|\!), \max(1, (\!|\mathtt{morse}|\!), (\!|\mathtt{morse}|\!))) \\
&= \max((\!|\underline{0}|\!), \max((\!|\mathtt{inv}\ \mathtt{morse}|\!), (\!|\mathtt{tail}\ \mathtt{morse}|\!))) \\
&= \max((\!|\underline{0}|\!), (\!|(\mathtt{zip}\ (\mathtt{inv}\ \mathtt{morse})\ (\mathtt{tail}\ \mathtt{morse}))|\!)) \\
&= (\!|\underline{0} : (\mathtt{zip}\ (\mathtt{inv}\ \mathtt{morse})\ (\mathtt{tail}\ \mathtt{morse}))|\!)
\end{aligned}
$$

5 Local upper bound (LUB)

Previous upper bounds are very useful in practice but also very restrictive. Simple examples like the stream definition of `nats` do not admit any global upper bound (and it is not GUB because we should demonstrate that $(\!|\mathtt{nats}\ \mathtt{x}|\!) \geq_? (\!|\mathtt{x} : (\mathtt{nats}\ (\mathtt{x} + 1))|\!) = \max((\!|\mathtt{x}|\!), (\!|\mathtt{nats}|\!)((\!|\mathtt{x}|\!) + k))$, for some $k \geq 1$) just because they compute streams with unbounded element size. However we would like to establish some properties over such kind of programs depending on other parameters. Clearly, in functional programming we never expect a stream to be fully evaluated. A Haskell programmer will evaluate some elements of a stream `s` using some function like !! or `take`. In this case, it may be possible to derive an upper bound on the size of the elements using the input index n of the element we want to reach. For example, we know that the size of the complete evaluation of $(\mathtt{nats}\ \underline{0})\ \mathord{!\!!}\ \underline{n}$ is bounded by the size of $\underline{n}$.

From these observations it is easy to argue that we need another kind of space property, that we call local because it does not only rely on the maximal size of the input stream elements but also on their index in the output stream.

Definition 8. *Given a* `sHask` *program, the function symbol* $\mathtt{f} :: \overrightarrow{[\sigma]} \to \overrightarrow{\tau} \to [\sigma]$ *has a* local upper bound *if there is a function* $G : \mathbb{R}^+ \to \mathbb{R}^+$ *such that for every expression* $\mathtt{s}_i :: [\sigma_i], \mathtt{e}_i :: \tau_i$ *of the program:*

$$\forall \underline{n} \in \mathbb{N}\ s.t.\ \mathtt{eval}((\mathtt{f}\ \overrightarrow{\mathtt{s}}\ \overrightarrow{\mathtt{e}})\ \mathord{!\!!}\ \underline{n}) \Downarrow \underline{v},\ G(\max(|\overline{\mathtt{s}}|, |\overline{\mathtt{e}}|, |\underline{n}|)) \geq |\underline{v}|$$

A program has a local upper bound *if every function symbol in it has a local upper bound.*

Note that also in the definition above, like in the case of GUB, we consider both stream functions and stream constructors, this means that we can both have locally bounded function and stream definitions.

Example 7. Consider the stream definition of `nats` of Example 1. The output stream has unbounded elements size. However we know that $\forall \underline{n} \in \mathbb{N}$ and $\forall \mathtt{e} :: \mathtt{Nat}$ if $(\mathtt{nats}\ \mathtt{e})\ \mathord{!\!!}\ \underline{n} \Downarrow \mathtt{v}$ then $\mathtt{v} = ((\mathtt{e} \underbrace{+1) + \cdots) + 1}_{n\ \mathrm{times}}$. Consequently, taking $F(X) = 2 \times X$, we obtain that $\forall \underline{n} \in \mathbb{N}$:

$$|\mathtt{v}| = |\underline{n}| + |\mathtt{e}| \leq 2 \times \max(|\underline{n}|, |\mathtt{e}|) = F(\max(|\underline{n}|, |\mathtt{e}|))$$

Now we define a criterion ensuring the fact that the size of an output element may depend on its index. For that purpose, we need to introduce a variation on assignments.

Definition 9. *A program assignment is parametrized by some variable* $L \in \mathbb{R}$, *denoted* $(\!|-|\!)_L$, *if for each symbol* $\mathtt{t}$ *of arity* n, $(\!|\mathtt{t}|\!)_L$ *is a function from* $(\mathbb{R}^+)^n \times \mathbb{R}$ *to* $\mathbb{R}$. *We use the notation* $(\!|-|\!)_r$, *whenever some* $r \in \mathbb{R}$ *is substituted to* L. *The parametrized assignment of a variable* $\mathtt{x}$ *is defined by a fresh variable* X *ranking over* $\mathbb{R}^+$. *Each parametrized assignment is extended to expressions as follows:*

$$(\!|\mathtt{t}\ \mathtt{e}_1 \cdots \mathtt{e}_n|\!)_L = (\!|\mathtt{t}|\!)_L((\!|\mathtt{e}_1|\!)_L, \ldots, (\!|\mathtt{e}_n|\!)_L) \qquad if\ \mathtt{t} \neq :$$
$$(\!|\mathtt{hd} : \mathtt{tl}|\!)_L = (\!|:|\!)_L((\!|\mathtt{hd}|\!)_L, (\!|\mathtt{tl}|\!)_{L-1}) \qquad otherwise$$

11

A parametrized assignment is monotonic if it is monotonic with respect to each of its arguments, including the parameter $L \in \mathbb{R}$.

We extend the notion of additive interpretations to the parametrized ones so that an additive symbol $\mathbf{c}$ of arity n, for some $k \geq 1$, has the interpretation

$$(\!|\mathbf{c}|\!)(X_1, \cdots, X_n)_L = \sum_{i=1}^{n} X_i + L + k$$

A program admits a parametrized interpretation $(\!|-|\!)_L$ if there is a monotonic parametrized assignment $(\!|-|\!)_L$ such that for each definition of the shape $\mathbf{f} \ \overrightarrow{\mathbf{p}} = \mathbf{e}$ we have $(\!|\mathbf{f} \ \overrightarrow{\mathbf{p}}|\!)_L \geq (\!|\mathbf{e}|\!)_L$.

Proposition 2. *Given a program admitting the parametrized interpretation $(\!|-|\!)_L$, then for every closed expression $\mathbf{e}$ and every $r \in \mathbb{R}$ we have:*

1. *If $\mathbf{e} \rightarrow^* \mathbf{d}$ then $(\!|\mathbf{e}|\!)_r \geq (\!|\mathbf{d}|\!)_r$*
2. *If $\mathbf{e} \Downarrow \mathbf{v}$ then $(\!|\mathbf{e}|\!)_r \geq (\!|\mathbf{v}|\!)_r$*
3. *If $\mathtt{eval} \ \mathbf{e} \Downarrow \underline{\mathbf{v}}$ then $(\!|\mathbf{e}|\!)_r \geq (\!|\underline{\mathbf{v}}|\!)_r$*

Proof. 1.We show that this result holds for every expression $\mathbf{d}$ such that $\mathbf{e} \rightarrow^* \mathbf{d}$, by induction on the reduction length n. It trivially holds for $n = 0$. Suppose it holds for n and take a reduction of length $n + 1$: $\mathbf{e} \rightarrow^{n+1} \mathbf{d}$. Define a context $\mathsf{C}[\diamond]$ to be a non case expression with one hole $\diamond$ and let $\mathsf{C}[\mathbf{e}]$ denote the result of filling the hole $\diamond$ with $\mathbf{e}$. We know that there are a context $\mathsf{C}[\diamond]$, a substitution σ and a definition $l = r$ such that $\mathbf{e} \rightarrow^n \mathsf{C}[l\sigma] \rightarrow \mathsf{C}[r\sigma] = \mathbf{d}$. By induction hypothesis, $(\!|\mathbf{e}|\!)_r \geq (\!|\mathsf{C}[l\sigma]|\!)_r$. Now let $(\!|\mathsf{C}|\!)_r$ be a function in $\mathbb{R} \rightarrow \mathbb{R}$ satisfying $(\!|\mathsf{C}|\!)_r(X) = (\!|\mathsf{C}[\diamond]|\!)_r$ for each $X \in \mathbb{R}^+$ such that $X = (\!|\diamond|\!)_u$, for all $u \in \mathbb{R}$. We know that there is some $r' \in \mathbb{R}$ such that $(\!|\mathsf{C}[l\sigma]|\!)_r = (\!|\mathsf{C}|\!)_r((\!|l\sigma|\!)_{r'})$. The real number r' just depends on the structure of the context $\mathsf{C}[\diamond]$ (indeed it is equal to r minus the number of times where the expression $l\sigma$ appears as a subterm of the rightmost argument of the constructor symbol : in the context $\mathsf{C}[\diamond]$). By definition of parametrized interpretations, we also know that for all L (and in particular for r'), $(\!|l\sigma|\!)_L \geq (\!|r\sigma|\!)_L$. So we have $(\!|\mathsf{C}|\!)_r((\!|l\sigma|\!)_{r'}) \geq (\!|\mathsf{C}|\!)_r((\!|r\sigma|\!)_{r'}) = (\!|\mathbf{d}|\!)_r$, by monotonicity.

2. It follows for every $\mathbf{v}$ such that $\mathbf{e} \Downarrow \mathbf{v}$, from the fact that the lazy semantics is just a particular rewrite strategy.

3. We prove it using the same reasoning as in the proof of Proposition1.3. There are two cases to consider. If $\mathbf{e} \Downarrow \mathbf{c} \ \overrightarrow{\mathbf{e}}$, with $\mathbf{c} \neq :$, then we can show easily that $\mathtt{eval} \ \mathbf{e} \Downarrow \mathbf{c} \ \overrightarrow{\underline{\mathbf{v}}}$, for some $\underline{\mathbf{v}}_i$ such that $\mathtt{eval} \ \mathbf{e}_i \Downarrow \underline{\mathbf{v}}_i$. Since $(\!|\mathbf{e}|\!)_r \geq (\!|\mathbf{c} \ \overrightarrow{\mathbf{e}}|\!)_r$, by (2), and $(\!|\mathbf{e}_i|\!)_r \geq (\!|\underline{\mathbf{v}}_i|\!)_r$, by induction hypothesis, we conclude that $(\!|\mathbf{e}|\!)_r \geq (\!|\mathbf{c} \ \overrightarrow{\mathbf{e}}|\!)_r = (\!|\mathbf{c}|\!)_r((\!|\overrightarrow{\mathbf{e}}|\!)_r) \geq (\!|\mathbf{c}|\!)_r((\!|\overrightarrow{\underline{\mathbf{v}}}|\!)_r) = (\!|\mathbf{c} \ \overrightarrow{\underline{\mathbf{v}}}|\!)_r$, by monotonicity of $(\!|\mathbf{c}|\!)_r$ and, by definition of canonical extension. Now if $\mathbf{e} \Downarrow \mathtt{hd} : \mathtt{tl}$ and $\mathtt{eval} \ \mathtt{hd} \Downarrow \underline{\mathbf{v}}$ and $\mathtt{eval} \ \mathtt{tl} \Downarrow \underline{\mathbf{w}}$ then $(\!|\mathbf{e}|\!)_r \geq (\!|\mathtt{hd} : \mathtt{tl}|\!)_r$, by (2), and $(\!|\mathtt{hd}|\!)_r \geq (\!|\underline{\mathbf{v}}|\!)_r$ and $(\!|\mathtt{tl}|\!)_{r-1} \geq (\!|\underline{\mathbf{w}}|\!)_{r-1}$, by induction hypothesis. We conclude that $(\!|\mathbf{e}|\!)_r \geq (\!|\mathtt{hd} : \mathtt{tl}|\!)_r = (\!|:|\!)_r((\!|\mathtt{hd}|\!)_r, (\!|\mathtt{tl}|\!)_{r-1}) \geq (\!|:|\!)_r((\!|\underline{\mathbf{v}}|\!)_r, (\!|\underline{\mathbf{w}}|\!)_{r-1}) = (\!|\underline{\mathbf{v}} : \underline{\mathbf{w}}|\!)_r$, by monotonicity of $(\!|:|\!)_r$. $\qquad \square$

Lemma 4. *Given a program admitting a monotonic parametrized assignment $(\!|-|\!)_L$, there is a function $G : \mathbb{R}^+ \times \mathbb{R}^+ \to \mathbb{R}^+$ such that for each expression* e *and every* $r \in \mathbb{R}^+$:

$$(\!|\mathsf{e}|\!)_r \leq G(|\mathsf{e}|, r)$$

Proof. Define $F(X, L) = \max(\max_{\mathsf{t} \in \mathcal{C} \cup \mathcal{F}} (\!|\mathsf{t}|\!)_L(X, \ldots, X), X)$ and $F^{n+1}(X, L) = F(F^n(X, L), L)$ and $F^0(X, L) = F(X, L)$. We prove by induction on the structure of e that $(\!|\mathsf{e}|\!)_r \leq F^{|\mathsf{e}|}(|\mathsf{e}|, r)$. If e is a variable, a constructor or a function symbol of arity 0, then conclusion follows directly by definition of F, i.e $(\!|\mathsf{e}|\!)_L \leq F(X, L)$. Now, consider $\mathsf{e} = \mathsf{t}\ \mathsf{d}_1 \cdots \mathsf{d}_n$ and suppose $|\mathsf{d}_j| = \max_{i=1}^n |\mathsf{d}_i|$. By induction hypothesis, we have $(\!|\mathsf{d}_i|\!)_r \leq F^{|\mathsf{d}_i|}(|\mathsf{d}_i|, r)$. We have two possibilities depending on the shape of t. If $\mathsf{t} \neq\ :$ then by induction hypothesis, definition and monotonicity of F:

$$
\begin{aligned}
(\!|\mathsf{e}|\!)_r = (\!|\mathsf{t}|\!)_r((\!|\mathsf{d}_1|\!)_r, \ldots, (\!|\mathsf{d}_n|\!)_r) &\leq (\!|\mathsf{t}|\!)_r(F^{|\mathsf{d}_1|}(|\mathsf{d}_1|, r), \ldots, F^{|\mathsf{d}_n|}(|\mathsf{d}_n|, r)) \\
&\leq (\!|\mathsf{t}|\!)_r(F^{|\mathsf{d}_j|}(|\mathsf{d}_j|, r), \ldots, F^{|\mathsf{d}_j|}(|\mathsf{d}_j|, r)) \leq F(F^{|\mathsf{d}_j|}(|\mathsf{d}_j|, r), r) \\
&\leq F^{|\mathsf{d}_j|+1}(|\mathsf{e}|, r) \leq F^{|\mathsf{e}|}(|\mathsf{e}|, r)
\end{aligned}
$$

Conversely in the case $\mathsf{t} =\ :$ by definition of parametrized interpretation, induction hypothesis, definition and monotonicity of F, we have:

$$
\begin{aligned}
(\!|\mathsf{e}|\!)_r = (\!|\mathsf{hd} : \mathsf{tl}|\!)_r = (\!|:|\!)_r((\!|\mathsf{hd}|\!)_r, (\!|\mathsf{tl}|\!)_{r-1}) &\leq (\!|:|\!)_r(F^{|\mathsf{hd}|}(|\mathsf{hd}|, r), F^{|\mathsf{tl}|}(|\mathsf{tl}|, r-1)) \\
&\leq (\!|:|\!)_r(F^{|\mathsf{hd}|}(|\mathsf{hd}|, r), F^{|\mathsf{tl}|}(|\mathsf{tl}|, r)) \leq F^{|\mathsf{e}|}(|\mathsf{e}|, r)
\end{aligned}
$$

Now the conclusion follow easily by taking $G(X, L) = F^X(X, L)$. $\qquad\square$

Definition 10. *A program is* LUB *if it admits an additive parametrized interpretation $(\!|-|\!)_L$ but on $:$ where $(\!|:|\!)_L$ is defined by $(\!|:|\!)_L(X, Y) = \max(X, Y)$.*

Example 8. Consider again the stream definition of $\mathtt{nats}$ of example 1 together with the following parametrized assignment $(\!|-|\!)_L$ defined by $(\!|\mathtt{nats}|\!)_L(X) = X + L$, $(\!|+1|\!)_L(X) = X + 1$, $(\!|0|\!)_L = 0$ and $(\!|:|\!)_L(X, Y) = \max(X, Y)$. We check that:

$$
\begin{aligned}
(\!|\mathtt{nats}(\mathtt{x})|\!)_L = (\!|\mathtt{nats}|\!)_L((\!|\mathtt{x}|\!)_L) = X + L &\geq \max(X, (X+1) + (L-1)) \\
&= \max((\!|\mathtt{x}|\!)_L, (\!|\mathtt{nats}(\mathtt{x}+1)|\!)_{L-1}) = (\!|\mathtt{x} : (\mathtt{nats}\ (\mathtt{x}+1))|\!)_L
\end{aligned}
$$

It is a parametrized interpretation and, consequently, $\mathtt{nats}$ is LUB.

Now, we show an intermediate result for parametrized interpretations.

Lemma 5. *For every* $\underline{\mathsf{n}} \in \mathbb{N}$ *and* $\mathsf{e} :: [\sigma]$, *if* $\mathsf{e}\ !!\ \underline{\mathsf{n}} \Downarrow \mathsf{v}$ *and* $\mathsf{v} \neq \mathbf{Err}$ *then*

$$(\!|\mathsf{e}|\!)_n \geq (\!|\mathsf{v}|\!)_0$$

Proof. We proceed by induction on $\underline{\mathsf{n}} \in \mathbb{N}$.
Let $\underline{\mathsf{n}} = 0$ and $\mathsf{e}\ !!\ 0 \Downarrow \mathsf{v}$. It is easy to verify that necessarily $\mathsf{e} \Downarrow \mathsf{hd} : \mathsf{tl}$. By definition of $(\!|:|\!)$ and by Proposition 2.2 we have $(\!|\mathsf{e}|\!)_0 \geq (\!|\mathsf{hd} : \mathsf{tl}|\!)_0 \geq (\!|\mathsf{hd}|\!)_0$. By Proposition 2.3, if $\mathtt{eval}\ \mathsf{hd} \Downarrow \mathsf{v}$ then $(\!|\mathsf{hd}|\!)_0 \geq (\!|\mathsf{v}|\!)_0$. So we have $(\!|\mathsf{e}|\!)_0 \geq (\!|\mathsf{v}|\!)_0$ and

then the conclusion.

Consider the case $\underline{n} = \underline{n}'+1$ and $e \,!!\, (\underline{n}'+1) \Downarrow v$. It is easy to verify that necessarily $e \Downarrow \mathtt{hd : tl}$, hence we have $(\!|e|\!)_n \geq (\!|\mathtt{hd : tl}|\!)_n = \max((\!|\mathtt{hd}|\!)_n, (\!|\mathtt{tl}|\!)_{n-1}) \geq (\!|\mathtt{tl}|\!)_{n-1}$, by Proposition 2.2. Moreover $e \,!!\, (\underline{n}'+1) \Downarrow v$ implies by definition that $\mathtt{tl} \,!!\, \underline{n}' \Downarrow v$ and by induction hypothesis we have $(\!|\mathtt{tl}|\!)_{n-1} \geq (\!|v|\!)_0$. So we have $(\!|e|\!)_n \geq (\!|v|\!)_0$ and then the conclusion. $\qquad\square$

Theorem 2. *If a program is* LUB *then it admits a local upper bound.*

Proof. It suffices to show that every function symbol has a local upper bound. For simplicity, we consider the case where for each function symbol we have only one definition. The general case with several definitions follows directly. Consider a stream function symbol $\mathtt{f} :: \overrightarrow{[\sigma]} \to \overrightarrow{\tau} \to [\sigma]$ defined by $\mathtt{f}\ \overrightarrow{\mathtt{p}_s}\ \overrightarrow{\mathtt{p}_b} = e$.

Let $\underline{n} \in \mathbb{N}$ and σ be a substitution and suppose $\mathtt{eval}((\mathtt{f}\ \overrightarrow{\mathtt{p}_s}\sigma\ \overrightarrow{\mathtt{p}_b}\sigma) \,!!\, \underline{n}) \Downarrow \underline{v}$. It is easy to verify that $(\mathtt{f}\ \overrightarrow{\mathtt{p}_s}\sigma\ \overrightarrow{\mathtt{p}_b}\sigma) \,!!\, \underline{n} \Downarrow v$, for some v such that $\mathtt{eval}\ v \Downarrow \underline{v}$. By Lemma 5, $(\!|\mathtt{f}\ \overrightarrow{\mathtt{p}_s}\sigma\ \overrightarrow{\mathtt{p}_b}\sigma|\!)_n \geq (\!|v|\!)_0$. By Proposition 2.3 and Lemma 1, $(\!|v|\!)_0 \geq (\!|\underline{v}|\!)_0 \geq |\underline{v}|$. Notice that Lemma 1 still holds because if $(\!|-|\!)_L$ is an additive parametrized assignment then $(\!|-|\!)_0$ is an additive assignment. Hence we can conclude $(\!|\mathtt{f}\ \overrightarrow{\mathtt{p}_s}\sigma\ \overrightarrow{\mathtt{p}_b}\sigma|\!)_n \geq |\underline{v}|$. By Lemma 4 and monotonicity:

$$(\!|\mathtt{f}|\!)_{|\underline{n}|}\ (G(|\overline{\mathtt{p}_s\sigma}|, |\underline{n}|), G(|\overline{\mathtt{p}_b\sigma}|, |\underline{n}|)) \geq (\!|\mathtt{f}|\!)_n\ ((\!|\overline{\mathtt{p}_s}\sigma|\!)_n, (\!|\overline{\mathtt{p}_b}\sigma|\!)_n) = (\!|\mathtt{f}\ \overrightarrow{\mathtt{p}_s}\sigma\ \overrightarrow{\mathtt{p}_b}\sigma|\!)_n$$

By taking $F(X) = (\!|\mathtt{f}|\!)_X(G(\overline{X}, X), G(\overline{X}, X))$, we have a local upper bound. $\quad\square$

Example 9. Consider the stream definition of $\mathtt{nats}$ of Example 1 together with the parametrized interpretation of Example 8. It is LUB, consequently, it admits a local upper bound. Taking $F(X) = (\!|\mathtt{nats}|\!)_X(X) = X + X$, we know that for each canonical numerals $\underline{m}, \underline{n} \in \mathbb{N}$ such that $\mathtt{eval}((\mathtt{nats}\ \underline{m}) \,!!\, \underline{n}) \Downarrow \underline{v}_{\underline{n}}$, we have $F(\max(|\underline{n}|, |\underline{m}|)) = 2 \times \max(m, n) \geq |\underline{v}_{\underline{n}}|$ (Indeed for all $\underline{n}$, we have $\underline{v}_{\underline{n}} = \underline{m} + \underline{n}$).

Example 10 (Fibonacci). The following computes the Fibonacci sequence:

```
fib  :: [Nat]                        add :: Nat → Nat → Nat
fib  =  0 : (1 : (sad fib (tail fib)))  add (x + 1) (y + 1) = ((add x y) + 1) + 1
                                     add (x + 1)    0    =       x + 1
tail :: [α]                          add    0    (y + 1) =       y + 1
tail x : xs =  xs
sad :: [Nat] → [Nat] → [Nat]
sad (x : xs) (y : ys) = (add x y) : (sad xs ys)
```

This program is LUB with respect to the following interpretation: $(\!|0|\!)_L = 0$, $(\!|+1|\!)_L(X) = X+L+1$, $(\!|:|\!)_L(X, Y) = \max(X, Y)$, $(\!|\mathtt{sad}|\!)_L(X, Y) = (\!|\mathtt{add}|\!)_L(X, Y) = X + Y$, $(\!|\mathtt{tail}|\!)_L(X) = X$ and $(\!|\mathtt{fib}|\!)_L = 2^L$. Indeed for the first rule, we have that for each $L \in \mathbb{R}$:

$$\begin{aligned}
(\!|\mathtt{fib}|\!)_L = 2^L &\geq \max(0, L, 2 \times 2^{L-2}) \\
&= \max((\!|0|\!)_L, \max((\!|1|\!)_{L-1}, 2 \times (\!|\mathtt{fib}|\!)_{L-2})) \\
&= \max((\!|0|\!)_L, \max((\!|1|\!)_{L-1}, (\!|\mathtt{sad\ fib\ (tail\ fib)}|\!)_{L-2})) \\
&= \max((\!|0|\!)_L, (\!|1 : \mathtt{sad\ fib\ (tail\ fib)}|\!)_{L-1}) \\
&= (\!|0 : (1 : \mathtt{sad\ fib\ (tail\ fib)})|\!)_L
\end{aligned}$$

14

We let the reader check the inequalities for the other definitions. We obtain that the function 2^L is a parametrized upper bound on the Fibonacci sequence: for each canonical numerals $\underline{n} \in \mathbb{N}$ s.t. $\mathtt{eval}(\mathtt{fib\ !!\ } \underline{n}) \Downarrow \underline{v_{\underline{n}}}$, we have $2^{|\underline{n}|} \geq |\underline{v_{\underline{n}}}|$.

5.1 Combining the criteria

One issue of interest is to study what happens if we consider locally bounded streams (like in the case of LUB) and if we want to obtain a global upper bound without any reference to the index as illustrated by the following example.

Example 11. We can compute the componentwise positive minus on streams:

```
min :: Nat → Nat → Nat        smin :: [Nat] → [Nat] → [Nat]
min  0      x   =   0          smin (x : xs) (y : ys) = (min x y) : (smin xs ys)
min (x + 1)  0   =  x + 1
min (x + 1) (y + 1) = min x y
```

The size of the result is always bounded by the maximal size of the first input stream elements even if the sizes of the second input stream elements are not bounded. Consequently, if the first input only contains GUB symbols, whatever the second input is we know that the result will be bounded. In this particular case, we might ask the program to be LUB together with the restrictions:

- $(\![\mathtt{smin}]\!)_L(X, Y)$ is constant in Y and L,
- the first argument of $\mathtt{smin}$ only applies to expressions $\mathtt{e}$ such that $(\![\mathtt{e}]\!)_L$ is constant in L

By Proposition 2.3 and by Lemmata 5 and 1, if $\mathtt{eval}((\mathtt{smin\ e\ d})!!\underline{n}) \Downarrow \underline{v}$ we know that $(\![\mathtt{smin\ e\ d}]\!)_n \geq (\![\underline{v}]\!)_0 \geq |\underline{v}|$. By the above restrictions, we obtain $(\![\mathtt{smin}]\!)_n((\![\mathtt{e}]\!)_n, (\![\mathtt{d}]\!)_n) = (\![\mathtt{smin}]\!)_0((\![\mathtt{e}]\!)_0, 0) \geq |\underline{v}|$ (substituting arbitrarily the constant 0 to n) and, consequently, we obtain an upper bound independent from $\underline{n}$. We claim it can be generalized easily to every LUB program. For example, we may show that $\mathtt{smin\ ones\ (nats\ 0)}$ has a global upper bound using this methodology.

6 Conclusion

In this paper, to continue the work started in [1], we have introduced two interpretation-based criteria over stream functions, named GUB and LUB respectively, such that:

- if a given program admits an interpretation satisfying the criterion GUB, then the output stream admits a global upper bound, i.e. the size of every of its elements can be bounded wrt to the input size
- if a given program admits an interpretation satisfying the criterion LUB, then the output stream admits a local upper bound, i.e. the size of the n-th element in it can be bounded wrt to the input size and its index n

These properties are part of a more general study about stream properties. However, a lot of questions are still open. In particular, we plan to investigate some other stream properties such as buffering [11] (which occurs when recursive calls store arguments of infinitely increasing size), memory leak [14] (which occurs when the program has to memorize an unbounded number of input stream elements between two outputs) or reachability (which occurs when some stream elements are never evaluated. Notice that although closely related, it is not the same as productivity) in future developments.

References

1. Gaboardi, M., Péchoux, R.: Upper bounds on stream I/O using semantic interpretations. In: CSL 2009. LNCS, vol. 5771, pp. 271–286. Springer, Heidelberg (2009)
2. Kennaway, J., Klop, J., Sleep, M., de Vries, F.: Transfinite Reductions in Orthogonal Term Rewriting Systems. In: RTA 1991. LNCS, vol. 488, pp. 1–12. Springer, Heidelberg (1989)
3. Kennaway, J., Klop, J., Sleep, M., de Vries, F.: Infinitary lambda calculus. TCS 175(1), 93–125 (1997)
4. Weihrauch, K.: A foundation for computable analysis. In: Foundations of Computer Science. LNCS, vol. 1337, pp. 185–199. Springer, Heidelberg (1997)
5. Henderson, P., Morris, J.: A lazy evaluator. In: POPL'76, pp. 95–103. ACM, New York (1976)
6. Pitts, A.M.: Operationally-based theories of program equivalence. In: Semantics and Logics of Computation. Cambridge University Press (1997)
7. Gordon, A.: Bisimilarity as a theory of functional programming. TCS 228(1-2), 5–47 (1999)
8. Dijkstra, E.W.: On the productivity of recursive definitions. EWD749 (1980)
9. Sijtsma, B.: On the productivity of recursive list definitions. ACM TOPLAS 11(4), 633–649 (1989)
10. Coquand, T.: Infinite objects in type theory. In: TYPES 1993. LNCS, vol. 806, pp. 62–78. Springer, Heidelberg (1994)
11. Hughes, J., Pareto, L., Sabry, A.: Proving the correctness of reactive systems using sized types. In: POPL'96, pp. 410–423. ACM, New York (1996)
12. Buchholz, W.: A term calculus for (co-) recursive definitions on streamlike data structures. Annals of Pure and Applied Logic 136(1-2), 75–90 (2005)
13. Endrullis, J., Grabmayer, C., Hendriks, D., Isihara, A., Klop, J.: Productivity of Stream Definitions. In: FCT 2007. LNCS, vol. 4639, pp. 274-287. Springer, Heidelberg (2007)
14. Frankau, S., Mycroft, A.: Stream processing hardware from functional language specifications. In: HICSS 2003. pp. 278–287. IEEE (2003)
15. Shkaravska, O., van Eekelen, M., Tamalet, A.: Collected Size Semantics for Functional Programs over Polymorphic Nested Lists. Technical Report ICIS–R09003, Radboud University Nijmegen (2009)
16. Burrell, M.J., Cockett, R., Redmond, B.F.: Pola: a language for PTIME programming. In: LCC 2009, LICS Workshop. (2009)
17. Bonfante, G., Marion, J.Y., Moyen, J.Y.: Quasi-interpretations, a way to control resources. TCS - Accepted.

18. Marion, J.Y., Péchoux, R.: Resource analysis by sup-interpretation. In: FLOPS 2006. LNCS, vol. 3945, pp. 163–176. Springer, Heidelberg (2006)
19. Amadio, R.: Synthesis of max-plus quasi-interpretations. Fundamenta Informaticae 65(1–2), 29–60 (2005)
20. Bonfante, G., Marion, J.Y., Moyen, J.Y., Péchoux, R.: Synthesis of quasi-interpretations. In: LCC 2005, LICS Workshop (2005) http://hal.inria.fr/.
21. Marion, J.Y., Péchoux, R.: Characterizations of polynomial complexity classes with a better intensionality. In: PPDP'08, pp. 79–88. ACM, New York (2008)