Upper Bounds on Stream I/O Using Semantic Interpretations

Marco Gaboardi[1] and Romain Péchoux[2,⋆]

[1] Dipartimento di Informatica, Università di Torino
gaboardi@di.unito.it
[2] Computer Science Department, Trinity College, Dublin
pechouxr@tcd.ie

Abstract. This paper extends for the first time semantic interpretation tools to infinite data in order to ensure Input/Output upper bounds on first order Haskell like programs on streams. By I/O upper bounds, we mean temporal relations between the number of reads performed on the input stream elements and the number of output elements produced. We study several I/O upper bounds properties that are of both theoretical and practical interest in order to avoid memory overflows.

1 Introduction

Interpretations are a well-established verification tool for proving properties of first order functional programs, term rewriting systems and imperative programs.

In the mid-seventies, Manna and Ness [1] and Lankford [2] introduced polynomial interpretations as a tool to prove the termination of term rewriting systems. The introduction of abstract interpretations [3] has strongly influenced the development of program verification and static analysis techniques. From their introduction, interpretations have been studied with hundreds of variations.

One variation of interest is the notion of quasi-interpretation [4]. It consists in a polynomial interpretation with relaxed constraints (large inequalities, functions over real numbers). Consequently, it no longer applies to termination problems (since well-foundedness is lost) but it allows us to study program complexity in an elegant way. Indeed, the quasi-interpretation of a first order functional program provides an upper bound on the size of the output values in the input size.

The theory of quasi-interpretations has led by now to many theoretical developments [4], for example, characterizations of the classes of functions computable in polynomial time and polynomial space. Moreover, the decidability of finding a quasi-interpretation of a given program has been shown for some restricted class of polynomials [5,6]. This suggests that quasi-interpretations can be interestingly exploited also in practical developments.

Quasi-interpretations have been generalized to sup-interpretations which are intensionally more powerful [7], i.e. sup-interpretations capture the complexity

⋆ The financial support of Science Foundation Ireland and COMPLICE project, ANR BLANC, is gratefully acknowledged.

E. Grädel and R. Kahle (Eds.): CSL 2009, LNCS 5771, pp. 271–286, 2009.

of strictly more programs than quasi-interpretations do. This notion has led to a characterization of the NC^k complexity classes [8] which is a complementary approach to characterizations using function algebra presented in [9].

A new theoretical issue is whether interpretations can be used in order to infer resource properties on programs computing over infinite data. Here we approach this problem by considering lazy programs over stream data. Size upper bounds on stream data are meaningless. However, other interesting measures can be considered, e.g. size of stream elements and length of finite parts of streams. Here we consider some I/O properties with regard to such kind of measures. In particular, we would like to be able to obtain relations between the input reads and the output writes of a given program, where a read (or write) corresponds to the computation of a stream element in a function argument (resp. a stream element of the result). In this paper we identify three stream I/O upper bounds properties and we study criteria using interpretations in order to ensure them.

The first criterion, named Length Based I/O Upper Bound (LBUB), ensures that the number of output writes (the output length) is bounded by some function in the number of input reads. This criterion is respected by programs that need to read a finite amount of the input in order to produce a bounded amount of the output. The second criterion, named Size Based I/O Upper Bound (SBUB), ensures that the number of output writes is bounded by some function in the input reads size. It extends the previous criterion to programs where the output writes not only depend on the input structure but also on the input values. Finally, the last criterion, named Synchrony Upper Bound (SUB), ensures upper bounds on the output writes size depending on the input reads size in a synchronous framework, i.e. when the stream functions write exactly one element for one read performed.

The above criteria are interesting since they ensure upper bound properties corresponding to synchrony and asynchrony relations between program I/O. Moreover, besides the particular criteria studied, this work shows that semantic interpretation can be fruitfully exploited in studying programs dealing with infinite data types. Furthermore, we carry out the treatment of stream properties in a purely operational way. This shows that semantic interpretation are suitable for the usual equational reasoning on functional programs. From these conclusions we aim our work to be a new methodology in the study of stream functional languages properties.

Related works. Most of the works about stream properties considered stream definability and productivity, a notion dating back to [10]. Several techniques have been developed in order to ensure productivity, e.g syntactical [10,11,12], data-flows analysis [13,14], type-based [15,16,17,18]. Some of these techniques can be adapted to prove different properties of programs working on streams, e.g. in [17], the authors give different hints on how to use sized types to prove some kind of buffering properties. Unfortunately an extensive treatment using these techniques to prove other properties of programs working on streams is lacking.

Outline of the paper. In Section 2, we describe the considered first order stream language and its lazy semantic. In Section 3 we define the semantic interpretations and their basic properties. Then, in Section 4, we introduce the considered properties and criteria to ensure them. Finally, in the last section we conclude.

2 Preliminaries

2.1 The sHask Language

We consider a first order Haskell-like language, named sHask. Let $\mathcal{X}$, $\mathcal{C}$ and $\mathcal{F}$ be disjoint sets denoting respectively the set of *variables*, the set of *constructor symbols* and the set of *function symbols*. A sHask program is composed of a set of definitions described by the grammar in Table 1, where $\mathtt{c} \in \mathcal{C}$, $\mathtt{x} \in \mathcal{X}$, $\mathtt{f} \in \mathcal{F}$. We use the identifier $\mathtt{t}$ to denote a symbol in $\mathcal{C} \cup \mathcal{F}$. Moreover the notation $\overline{\mathtt{e}}$, for some identifier $\mathtt{e}$, is a short-hand for the sequence $\mathtt{e}_1, \ldots, \mathtt{e}_n$. As usual, application associates to the left, i.e. $\mathtt{t}\ \mathtt{e}_1\ \cdots\ \mathtt{e}_n$ corresponds to the expression $((\mathtt{t}\ \mathtt{e}_1)\cdots)\ \mathtt{e}_n$. In the sequel we will use the notation $\mathtt{t}\ \overrightarrow{\mathtt{e}}$ as a short for the application $\mathtt{t}\ \mathtt{e}_1\ \cdots\ \mathtt{e}_n$. The language sHask includes a Case operator to carry out pattern matching and first order function definitions. All the standard algebraic data types can be considered. Nevertheless, to be more concrete, in what follows we will consider as example three standard data types: numerals, lists and pairs. Analogously to Haskell, we denote by 0 and postfix $+1$ the constructors for numerals, by nil and infix : the constructors for lists and by $(\_,\_)$ the constructor for pairs.

Table 1. sHask syntax

$\mathtt{p} ::= \mathtt{x} \mid \mathtt{c}\ \mathtt{p}_1\ \cdots\ \mathtt{p}_n$	(Patterns)
$\mathtt{e} ::= \mathtt{x} \mid \mathtt{t}\ \mathtt{e}_1\ \cdots\ \mathtt{e}_n \mid \mathtt{Case}\ \overline{\mathtt{e}}\ \mathtt{of}\ \overline{\mathtt{p_1}} \to \mathtt{e}_1, \ldots, \overline{\mathtt{p}_m} \to \mathtt{e}_m$	(Expressions)
$\mathtt{v} ::= \mathtt{c}\ \mathtt{e}_1\ \cdots\ \mathtt{e}_n$	(Values)
$\mathtt{d} ::= \mathtt{f}\ \mathtt{x}_1\ \cdots\ \mathtt{x}_n\ = \mathtt{e}$	(Definitions)

Between the constants in $\mathcal{C}$ we distinguish a special *error* symbol **Err** of arity 0 which corresponds to pattern matching failure. In particular, **Err** is treated as the other constructors, so for example pattern matching is allowed on it. The set of Values contains the usual lazy values, i.e. expressions with a constructor as the outermost symbol.

In order to simplify our framework, we will put some syntactical restrictions on the shape of the considered programs. We restrict our study to outermost non nested case definitions, this means that no Case appears in the $\mathtt{e}_1, \cdots, \mathtt{e}_m$ of a definition of the shape $\mathtt{f}\ \overrightarrow{\mathtt{x}} = \mathtt{Case}\ \overline{\mathtt{e}}\ \mathtt{of}\ \overline{\mathtt{p_1}} \to \mathtt{e}_1, \ldots, \overline{\mathtt{p}_m} \to \mathtt{e}_m$ and we suppose that the function arguments and case arguments are the same, i.e. $\overline{\mathtt{x}} = \overline{\mathtt{e}}$.

The goal of this restriction is to simplify the considered framework. We claim that it is not a severe restriction since every program can be easily transformed in an equivalent one respecting this convention.

Finally, we suppose that all the free variables contained in the expression $\mathtt{e}_i$ of a case expression appear in the patterns $\overline{\mathtt{p}_i}$, that no variable occurs twice in $\overline{\mathtt{p}_i}$ and that patterns are non-overlapping. It entails that programs are confluent [19].

Haskell Syntactic Sugar. In the sequel we use the Haskell-like programming style. An expression of the shape $\mathtt{f}\ \overrightarrow{\mathtt{x}} = \mathtt{Case}\ \overline{\mathtt{x}}\ \mathtt{of}\ \overline{\mathtt{p}_1} \to \mathtt{e}_1, \ldots, \overline{\mathtt{p}_k} \to \mathtt{e}_k$ will be written as a set of definitions $\mathtt{f}\ \overrightarrow{\mathtt{p}_1} = \mathtt{e}_1, \ldots, \mathtt{f}\ \overrightarrow{\mathtt{p}_k} = \mathtt{e}_k$. Moreover, we adopt the standard Haskell convention for the parenthesis, e.g. we use $\mathtt{f}\ (\mathtt{x}+1)\ 0$ to denote $((\mathtt{f}(\mathtt{x}+1))0)$.

2.2 sHask Type System

Similarly to Haskell, we are interested only in well typed expressions. For simplicity, we consider only programs dealing with lists that do not contain other lists and we assure this property by a typing restriction similar to the one of [18].

Definition 1. *The* basic and value types *are defined by the following grammar:*

$$\begin{array}{ll} \sigma ::= \alpha \mid \mathtt{Nat} \mid \sigma \times \sigma & \mathtt{(basic\ types)} \\ \mathtt{A} ::= a \mid \sigma \mid \mathtt{A} \times \mathtt{A} \mid [\sigma] & \mathtt{(value\ types)} \end{array}$$

where α *is a* basic type variable*,* a *is a* value type variable*,* $\mathtt{Nat}$ *is a* constant type *representing natural numbers,* $\times$ *and* $[\ ]$ *are* type constructors*. The set of* types *contains elements of the shape* $\mathtt{A}_1 \to (\cdots \to (\mathtt{A}_n \to \mathtt{A}))$*, for every* $n \geq 0$*.*

Notice that the above definition can be extended to standard algebraic data types. In the sequel, we use σ, τ to denote basic types and $\mathtt{A}, \mathtt{B}$ to denote value types. As in Haskell, there is 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, $\to$ associates to the right, i.e. the notation $\mathtt{A}_1 \to \cdots \to \mathtt{A}_n \to \mathtt{A}$ corresponds to the type $\mathtt{A}_1 \to (\cdots \to (\mathtt{A}_n \to \mathtt{A}))$. Moreover, for notational convenience, we will use $\overrightarrow{\mathtt{A}} \to \mathtt{B}$ as an abbreviation for $\mathtt{A}_1 \to \cdots \to \mathtt{A}_n \to \mathtt{B}$ throughout the paper.

In what follows, we will be particularly interested in studying expressions of type $[\sigma]$, for some σ, i.e. the type of finite and infinite lists over σ, in order to study stream properties. Every function and constructor symbol $\mathtt{t}$ of arity n come equipped with a type $\mathtt{A}_1 \to \cdots \to \mathtt{A}_n \to \mathtt{A}$. 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 $\mathtt{A}$ 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 types to constant and function symbols, but only value types to expressions.

As usual, we use :: to denote typing judgments, e.g. $0 :: \mathtt{Nat}$ denotes the fact that 0 has type $\mathtt{Nat}$. A well typed definition is a function definition where we can assign the same value type $\mathtt{A}$ both to its left-hand and right-hand sides.

Table 2. sHask type system

$$\frac{}{\mathtt{x} :: \mathtt{A}}\ (\mathtt{Var}) \qquad \frac{\overline{\mathtt{e}} :: \overline{\mathtt{A}} \quad \overline{\mathtt{p_1}} :: \overline{\mathtt{A}} \quad \cdots \quad \overline{\mathtt{p}_m} :: \overline{\mathtt{A}} \quad \mathtt{e}_1 :: \mathtt{A} \quad \cdots \quad \mathtt{e}_m :: \mathtt{A}}{\mathtt{Case}\ \overline{\mathtt{e}}\ \mathtt{of}\ \overline{\mathtt{p_1}} \to \mathtt{e}_1, \ldots, \overline{\mathtt{p}_m} \to \mathtt{e}_m :: \mathtt{A}}\ (\mathtt{Case})$$

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

Stream terminology. In this work, we are specifically interested in studying stream properties. Since both finite lists over σ and streams over σ can be typed with type $[\sigma]$, we pay attention to particular classes of function working on $[\sigma]$, for some σ. Following the terminology of [14], a function symbol $\mathtt{f}$ is called a *stream function* if it is a symbol of type $\mathtt{f} :: \overrightarrow{[\sigma]} \to \overrightarrow{\tau} \to [\sigma]$.

Example 1. Consider the following programs:

$$\begin{array}{ll} \mathtt{merge} :: [\alpha] \to [\alpha] \to [\alpha \times \alpha] & \mathtt{nat} :: \mathtt{Nat} \to [\mathtt{Nat}] \\ \mathtt{merge}\ (\mathtt{x} : \mathtt{xs})\ (\mathtt{y} : \mathtt{ys}) = (\mathtt{x}, \mathtt{y}) : (\mathtt{merge}\ \mathtt{xs}\ \mathtt{ys}) & \mathtt{nat}\ \mathtt{x} = \mathtt{x} : (\mathtt{nat}\ (\mathtt{x} + 1)) \end{array}$$

`merge` and `nat` are two examples of stream functions

2.3 sHask Lazy Operational Semantics

We define a lazy operational semantics for the **sHask** language. The lazy semantics we give is an adaptation of the one in [20] to our first order Haskell-like

Table 3. sHask lazy operational semantics

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

$$\frac{\mathtt{Case}\ \mathtt{e}^1\ \mathtt{of}\ \mathtt{p}_1^1 \to \ldots \to \mathtt{Case}\ \mathtt{e}^m\ \mathtt{of}\ \mathtt{p}_1^m \to \mathtt{d}_1 \Downarrow \mathtt{v} \quad \mathtt{v} \neq \mathbf{Err}}{\mathtt{Case}\ \overline{\mathtt{e}}\ \mathtt{of}\ \overline{\mathtt{p_1}} \to \mathtt{d}_1, \ldots, \overline{\mathtt{p}_n} \to \mathtt{d}_n \Downarrow \mathtt{v}}\ (\mathrm{c}_b)$$

$$\frac{\mathtt{Case}\ \mathtt{e}^1\ \mathtt{of}\ \mathtt{p}_1^1 \to \ldots \to \mathtt{Case}\ \mathtt{e}^m\ \mathtt{of}\ \mathtt{p}_1^m \to \mathtt{d}_1 \Downarrow \mathbf{Err} \quad \mathtt{Case}\ \overline{\mathtt{e}}\ \mathtt{of}\ \overline{\mathtt{p_2}} \to \mathtt{d}_2, \ldots, \overline{\mathtt{p}_n} \to \mathtt{d}_n \Downarrow \mathtt{v}}{\mathtt{Case}\ \overline{\mathtt{e}}\ \mathtt{of}\ \overline{\mathtt{p_1}} \to \mathtt{d}_1, \ldots, \overline{\mathtt{p}_n} \to \mathtt{d}_n \Downarrow \mathtt{v}}\ (\mathrm{c})$$

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

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

language, where we do not consider sharing for simplicity. The semantics is defined by the rules of Table 3.

The computational domain is the set of `Values`. `Values` are particular expressions with a constructor symbol at the outermost position. Note that in particular **Err** is a value corresponding to pattern matching errors. As usual in lazy semantics, the evaluation does not explore the entire expression and stops once the requested information is found. The intended meaning of the notation $\mathtt{e} \Downarrow \mathtt{v}$ is that the expression `e` *evaluates* to the value $\mathtt{v} \in \mathtt{Values}$.

Example 2. Consider again the program defined in Example 1. It is easy to verify that: $\mathtt{nat}\ 0 \Downarrow 0 : (\mathtt{nat}\ (0+1))$ and $\mathtt{merge}\ (\mathtt{nat}\ 0)\ \mathtt{nil} \Downarrow \mathbf{Err}$.

2.4 Preliminary Notions

We are interested in studying stream properties by operational finitary means, for this purpose, we introduce some useful programs and notions.

First, we define the usual Haskell take and indexing programs `take` and !! which return the first n elements of a list and the n-th element of a list, respectively. As in Haskell, we use infix notation for !!.

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

Second, we define a program `eval` that forces the (possibly diverging) full evaluation of expressions to fully evaluated values, i.e. values with no function symbols. We define `eval` for every value type `A` as:

$$\begin{array}{l}
\mathtt{eval} :: \mathtt{A} \to \mathtt{A} \\
\mathtt{eval}\ \ (\mathbf{c}\ \mathtt{e}_1\ \cdots\ \mathtt{e}_n) = \hat{\mathtt{C}}\ (\mathtt{eval}\ \mathtt{e}_1)\ \cdots\ (\mathtt{eval}\ \mathtt{e}_m)
\end{array}$$

where $\hat{\mathtt{C}}$ is a program representing the *strict* version of the primitive constructor $\mathbf{c}$. For example in the case where $\mathbf{c}$ is $+1$ we can define $\hat{\mathtt{C}}$ as the program $\mathtt{succ} :: \mathtt{Nat} \to \mathtt{Nat}$ defined by:

$$\begin{array}{l}
\mathtt{succ}\ \ \ 0\ \ \ = 0+1 \\
\mathtt{succ}\ \ (\mathtt{x}+1) = (\mathtt{x}+1)+1
\end{array}$$

A set of fully evaluated values of particular interest is the set $\mathtt{N} = \{\underline{\mathtt{n}} \mid \underline{\mathtt{n}} = \underbrace{((\cdots(0+1)\cdots)+1)}_{n \text{ times}}$ and $\underline{\mathtt{n}} :: \mathtt{Nat}\}$ of *canonical numerals*.

Then, we define a program `lg` that returns the number of elements in a finite partial list:

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

Example 3. In order to illustrate the behaviour of `lg` consider the expression ((`succ` 0) : (`nil` !! 0)). We have $\texttt{eval}(\texttt{lg}\ ((\texttt{succ}\ 0) : (\texttt{nil}\ !!\ 0))) \Downarrow \underline{1}$.

Finally we introduce a notion of *size* for expressions:

Definition 2 (Size). *The* size *of an expression* e *is defined as*

$$|\mathtt{e}| = 0 \qquad \text{if } \mathtt{e} \text{ is a variable or a symbol of arity } 0$$
$$|\mathtt{e}| = \sum_{i \in \{1,\dots,n\}} |\mathtt{e}_i| + 1 \qquad \text{if } \mathtt{e} = \mathtt{t}\ \mathtt{e}_1\ \cdots\ \mathtt{e}_n,\ \mathtt{t} \in \mathcal{C} \cup \mathcal{F}.$$

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

3 Interpretations

In this section, we introduce the interpretation terminology. The interpretations we consider are inspired by the notions of quasi-interpretation [4] and sup-interpretation [7] and are used as a main tool in order to ensure stream properties. They basically consist in assignments over non negative real numbers following the terminology of [21]. 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* $\mathbb{R}^+$. *A program assignment is an assignment* $(\!|-|\!)$ *defined for each symbol of the program. An assignment is (weakly)* monotonic *if for any symbol* t, $(\!|\mathtt{t}|\!)$ *is an increasing (not necessarily strictly) function with respect to each variable, that is for every symbol* t *and all* X_i, Y_i *of* $\mathbb{R}^+$ *such that* $X_i \geq Y_i$, *we have* $(\!|\mathtt{t}|\!)(\dots, X_i, \dots) \geq (\!|\mathtt{t}|\!)(\dots, Y_i, \dots)$.

Notice that assignments are not defined on the `Case` construct since we only apply assignments to expressions without `Case`.

An assignment $(\!|-|\!)$ can be extended to expressions canonically. Given an expression $\mathtt{t}\ \mathtt{e}_1\ \dots\ \mathtt{e}_n$ with m variables, its assignment is a function $(\mathbb{R}^+)^m \to \mathbb{R}^+$ defined by:

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

Example 4. The function $(\!|-|\!)$ defined by $(\!|\mathtt{merge}|\!)(U, V) = U + V$, $(\!|(\_, \_)|\!)(U, V) = U + V + 1$ and $(\!|:|\!)(X, XS) = X + XS + 1$ is a monotonic assignment of the program `merge` of example 1.

Now we define the notion of additive assignments which guarantees that the size of a fully evaluated value is bounded by its assignment.

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

$$\begin{aligned} (\!|\mathbf{c}|\!)(X_1, \cdots, X_n) &= \textstyle\sum_{i=1}^{n} X_i + \alpha_{\mathbf{c}}, \text{ with } \alpha_{\mathbf{c}} \geq 1 && \text{if } n > 0, \\ (\!|\mathbf{c}|\!) &= 0 && \text{otherwise.} \end{aligned}$$

The assignment $(\!|-|\!)$ *of a program is called* additive *assignment if each constructor symbol of* $\mathcal{C}$ *has an additive assignment.*

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

Notice that if $(\!|\mathtt{t}|\!)$ is a subterm function (i.e. $\forall i \in \{1, n\}\ (\!|\mathtt{t}|\!)(X_1, \cdots, X_n) \geq X_i$), for every symbol $\mathtt{t}$, then the considered interpretation is called a quasi-interpretation in the literature (used for inferring upper bounds on values). Moreover, if $(\!|\mathtt{t}|\!)$ is a polynomial over natural numbers and the inequalities are strict then $(\!|-|\!)$ is called a polynomial interpretation (used for showing program termination).

Example 5. The assignment of example 4 is an additive interpretation of the program `merge`. Indeed, we have:

$$\begin{aligned} (\!|\mathtt{merge}\ (\mathtt{x} : \mathtt{xs})\ (\mathtt{y} : \mathtt{ys})|\!) &= (\!|\mathtt{merge}|\!)((\!|\mathtt{x} : \mathtt{xs}|\!), (\!|\mathtt{y} : \mathtt{ys}|\!)) && \text{By canonical extension} \\ &= (\!|\mathtt{x} : \mathtt{xs}|\!) + (\!|\mathtt{y} : \mathtt{ys}|\!) && \text{By definition of } (\!|\mathtt{merge}|\!) \\ &= (\!|\mathtt{x}|\!) + (\!|\mathtt{xs}|\!) + (\!|\mathtt{y}|\!) + (\!|\mathtt{ys}|\!) + 2 && \text{By definition of } (\!|:|\!) \\ &= (\!|(\mathtt{x}, \mathtt{y}) : (\mathtt{merge}\ \mathtt{xs}\ \mathtt{ys})|\!) && \text{Using the same reasoning} \end{aligned}$$

Let $\rightarrow$ be the rewrite relation induced by giving an orientation from left to right to the definitions and let $\rightarrow^*$ be its 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* $\mathtt{e}$ *such that* $\mathtt{e} \rightarrow^* \mathtt{d}$*, we have:* $(\!|\mathtt{e}|\!) \geq (\!|\mathtt{d}|\!)$

Proof. The proof is by induction on the derivation length [22]. □

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

Proof. The lazy semantics is just a particular rewrite strategy. □

Corollary 2. *Given a program admitting the interpretation* $(\!|-|\!)$*, then for every closed expression* $\mathtt{e}$ *such that* $\mathtt{eval}\ \mathtt{e} \Downarrow \mathtt{v}$*, we have:* $(\!|\mathtt{e}|\!) \geq (\!|\mathtt{v}|\!)$

Proof. By induction on the structure of expressions. □

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

Lemma 1. *Given a program having an interpretation* $(\!|-|\!)$ *then there is a function* $G : \mathbb{R}^+ \rightarrow \mathbb{R}^+$ *such that for each expression* $\mathtt{e}$*:* $(\!|\mathtt{e}|\!) \leq G(|\mathtt{e}|)$

Proof. By induction on the structure of expressions. □

4 Bounded I/O Properties and Criteria

In this section, we define distinct stream properties related to time and space and criteria using interpretations to ensure them. A naive approach would be to consider a time unit to be either a stream input read (the evaluation of a stream element in a function argument) or a stream output write (the evaluation of a stream element of the result). Since most of the interesting programs working on streams are non-terminating, this approach fails. In fact, we need a more concrete notion of what time should be. Consequently, by time we mean relations between input reads and output writes, that is the ability of a program to return a certain amount of elements in the output stream (that is to perform some number of writes) when fed with some input stream elements.

4.1 Length Based I/O Upper Bound (LBUB)

We focus on the relations that provide upper bounds on output writes. We here consider structural relations, that depend on the stream structure but not on the value of its elements. We point out an interesting property giving bounds on the number of generated outputs by a function in the length of the inputs. As already stressed, the complete evaluation of stream expressions does not terminates. So, in order to deal with streams by finitary means, we ask, inspired by the well known Bird and Wadler *Take Lemma* [23], the property to hold on all the finite fragments of the streams that produce a result.

Definition 6. *A stream function* $\mathtt{f} :: \overrightarrow{[\sigma]} \to \overrightarrow{\tau} \to [\sigma]$ *has a length based I/O upper bound if there is a function* $F : \mathbb{R}^+ \to \mathbb{R}^+$ *such that for every expression* $\mathtt{s}_i :: [\sigma_i]$ *and for every expression* $\mathtt{e}_i :: \tau_i$*, we have that:*

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

where $(\overrightarrow{\mathtt{take}\ \underline{\mathtt{n}}\ \mathtt{s}})$ *is a short for* $(\mathtt{take}\ \underline{\mathtt{n}}_1\ \mathtt{s}_1) \cdots (\mathtt{take}\ \underline{\mathtt{n}}_m\ \mathtt{s}_m)$.

Let us illustrate the length based I/O upper bound property by an example:

Example 6. The function `merge` of example 1 has a length based I/O upper bound. Indeed, consider $F(X) = X$, given two finite lists $\mathtt{s}, \mathtt{s}'$ of size n, n' such that $n \leq n'$, we know that $\mathtt{eval}(\mathtt{lg}\ (\mathtt{merge}\ \mathtt{s}\ \mathtt{s}'))$ evaluates to an expression $\underline{\mathtt{m}}$ such that $m = n$. Consequently, given two stream expressions $\mathtt{e}$ and $\mathtt{e}'$ such that $\mathtt{eval}(\mathtt{take}\ \underline{\mathtt{n}}'\ \mathtt{e}') \Downarrow \mathtt{s}'$ and $\mathtt{eval}(\mathtt{take}\ \underline{\mathtt{n}}\ \mathtt{e}) \Downarrow \mathtt{s}$, we have:

$$F(\max(|\underline{\mathtt{n}}|, |\underline{\mathtt{n}}'|)) = F(\max(n, n')) = n' \geq n = |\underline{\mathtt{m}}|$$

4.2 A Criterion for Length Based I/O Upper Bound

We here give a criterion ensuring that a given stream has a length based I/O upper bound. For simplicity, in the following sections, we suppose that the considered programs do not use the programs `lg` and `take`.

Definition 7. *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 by* $(\!|:|\!)(X, Y) = Y + 1$.

We start by showing some basic properties of LBUB programs.

Lemma 2. *Given a* LBUB *program, for every* $\underline{\mathtt{n}} :: \mathtt{Nat}$ *we have* $(\!|\underline{\mathtt{n}}|\!) = |\underline{\mathtt{n}}|$.

Proof. By an easy induction on canonical numerals. □

Lemma 3. *Given a* LBUB *program wrt the interpretation* $(\!|-|\!)$*, the interpretation can be extended to the program* `lg` *by* $(\!|\mathtt{lg}|\!)(X) = X$.

Proof. We check that the inequalities hold for every equation in the definition of `lg`. □

Lemma 4. *Given a* LBUB *program wrt the interpretation* $(\!|-|\!)$*, the interpretation can be extended to the program* `take` *by* $(\!|\mathtt{take}|\!)(N, L) = N$.

Proof. We check that the inequalities hold for every equation in the definition of `take`. □

Theorem 1. *If a program is* LBUB *then each stream function in it has a length based I/O upper bound.*

Proof. Given a LBUB program, if $\mathtt{eval}(\mathtt{lg}\ (\mathtt{f}\ (\overrightarrow{\mathtt{take}\ \underline{\mathtt{n}}\ \mathtt{s}})\ \overrightarrow{\mathtt{e}})) \Downarrow \underline{\mathtt{m}}$ then we know that $(\!|\mathtt{lg}\ (\mathtt{f}\ (\overrightarrow{\mathtt{take}\ \underline{\mathtt{n}}\ \mathtt{s}})\ \overrightarrow{\mathtt{e}})|\!) \geq (\!|\underline{\mathtt{m}}|\!)$, by Corollary 2. By Lemma 3, we obtain that $(\!|\mathtt{f}\ (\overrightarrow{\mathtt{take}\ \underline{\mathtt{n}}\ \mathtt{s}})\ \overrightarrow{\mathtt{e}}|\!) \geq (\!|\underline{\mathtt{m}}|\!)$. By Lemma 4, we know that $(\!|\mathtt{f}\ (\overrightarrow{\mathtt{take}\ \underline{\mathtt{n}}\ \mathtt{s}})\ \overrightarrow{\mathtt{e}}|\!) = (\!|\mathtt{f}|\!)((\!|\overline{\underline{\mathtt{n}}}|\!)\ (\!|\overline{\mathtt{e}}|\!))$. Applying Lemma 2, we obtain $|\underline{\mathtt{m}}| = (\!|\underline{\mathtt{m}}|\!) \leq (\!|\mathtt{f}|\!)(|\overline{\underline{\mathtt{n}}}|,\ (\!|\overline{\mathtt{e}}|\!))$. Finally, by Lemma 1, we know that there is a function $G : \mathbb{R}^+ \to \mathbb{R}^+$ such that $|\underline{\mathtt{m}}| \leq (\!|\mathtt{f}|\!)(|\overline{\underline{\mathtt{n}}}|,\ G(|\overline{\mathtt{e}}|))$. □

Example 7. The `merge` program of example 1 admits the following additive interpretation $(\!|\mathtt{merge}|\!)(X, Y) = \max(X, Y)$, $(\!|(\_, \_)|\!)(X, Y) = X+Y+1$ together with $(\!|:|\!)(X, Y) = Y+1$. Consequently, it is LBUB and, defining $F(X) = (\!|\mathtt{merge}|\!)(X, X)$ we know that for any two finite lists $\mathtt{s}_1$ and $\mathtt{s}_2$ of length m_1 and m_2, we have that if $\mathtt{eval}(\mathtt{lg}(\mathtt{merge}\ \mathtt{s}_1\ \mathtt{s}_2)) \Downarrow \underline{\mathtt{m}}$ then $F(\max(m_1, m_2)) \geq |\underline{\mathtt{m}}|$ (i.e. we are able to exhibit a precise upper bound).

4.3 Size Based I/O Upper Bound (SBUB)

The previous criterion guarantees an interesting homogeneous property on stream data. However, a wide class of stream programs with bounded relations between input reads and output writes do not enjoy it. The reason is just that some programs do not only take into account the structure of the input it reads, but also its value. We here point out a generalization of the LBUB property by considering an upper bound depending on the size of the stream expressions.

Example 8. Consider the following motivating example:

$$\begin{array}{ll}
\texttt{append} :: [\alpha] \to [\alpha] \to [\alpha] & \texttt{upto} :: \texttt{Nat} \to [\texttt{Nat}] \\
\texttt{append}\ (\texttt{x} : \texttt{xs})\ \texttt{ys} = \texttt{x} : (\texttt{append}\ \texttt{xs}\ \texttt{ys}) & \texttt{upto}\ 0 = \texttt{nil} \\
\texttt{append}\ \texttt{nil}\ \texttt{ys} = \texttt{ys} & \texttt{upto}\ (\texttt{x}+1) = (\texttt{x}+1) : (\texttt{upto}\ \texttt{x})
\end{array}$$

$$\begin{array}{l}
\texttt{extendupto} :: [\texttt{Nat}] \to [\texttt{Nat}] \\
\texttt{extendupto}\ (\texttt{x} : \texttt{xs}) = \texttt{append}\ (\texttt{upto}\ \texttt{x})\ (\texttt{extendupto}\ \texttt{xs})
\end{array}$$

The program `extendupto` has no length based I/O upper bound because for each number $\underline{\texttt{n}}$ it reads, it performs n output writes (corresponding to a decreasing sequence from $\underline{\texttt{n}}$ to $\underline{\texttt{1}}$).

Now we introduce a new property dealing with size, that allows us to overcome this problem.

Definition 8. *A stream function* $\texttt{f} :: \overrightarrow{[\sigma]} \to \overrightarrow{\tau} \to [\sigma]$ *has a size based I/O upper bound if there is a function* $F : \mathbb{R}^+ \to \mathbb{R}^+$ *such that, for every stream expression* $\texttt{s}_i :: [\sigma_i]$ *and for expression* $\texttt{e}_i :: \tau_i$*, we have that:*

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

where $(\overrightarrow{\texttt{take}\ \underline{\texttt{n}}\ \texttt{s}})$ *is a short for* $(\texttt{take}\ \underline{\texttt{n}}_1\ \texttt{s}_1) \cdots (\texttt{take}\ \underline{\texttt{n}}_m\ \texttt{s}_m)$.

Example 9. Since the program of example 8, performs n output writes for each number $\underline{\texttt{n}}$ it reads, it has a size based I/O upper bound.

Notice that this property informally generalizes the previous one, i.e. a size based I/O upper bounded program is also length based I/O program, just because size always bounds the length. But the length based criterion is still relevant for two reasons, first it is uniform (input reads and output writes are treated in the same way), second it provides more accurate upper bounds.

4.4 A Criterion for Size Based I/O Upper Bound

We give a criterion ensuring that a stream has a size based I/O upper bound.

Definition 9. *A program is* SBUB *if it admits an additive interpretation* $(\!|-|\!)$ *such that* $(\!|{+}1|\!)(X) = X + 1$ *and* $(\!|{:}|\!)(X, Y) = X + Y + 1$.

Lemma 5. *Given a* SBUB *program wrt the interpretation* $(\!|-|\!)$*, the interpretation can be extended to the program* `lg` *by* $(\!|\texttt{lg}|\!)(X) = X$.

Proof. We check that the inequalities hold for every definition of `lg`. □

Lemma 6. *Given a* SBUB *program wrt the interpretation* $(\!|-|\!)$*, the interpretation can be extended to the program* `take` *by* $(\!|\texttt{take}|\!)(N, L) = L$.

Proof. We check that the inequalities hold for every definition of `take`. □

Theorem 2. *If a program is* SBUB *then each stream function in it has a size based I/O upper bound.*

Proof. Given a SBUB program, then if $\mathtt{eval}(\mathtt{lg}(\mathtt{f}\ (\overrightarrow{\mathtt{take}\ \underline{\mathtt{n}}\ \mathtt{s}})\ \overrightarrow{\mathtt{e}}))\Downarrow \underline{\mathtt{m}}$, for some stream function $\mathtt{f}$, then $(\!|\mathtt{lg}(\mathtt{f}\ (\overrightarrow{\mathtt{take}\ \underline{\mathtt{n}}\ \mathtt{s}})\ \overrightarrow{\mathtt{e}})|\!) \geq (\!|\underline{\mathtt{m}}|\!)$, by Corollary 2. By Lemma 5, we obtain that $(\!|\mathtt{f}\ (\overrightarrow{\mathtt{take}\ \underline{\mathtt{n}}\ \mathtt{s}})\ \overrightarrow{\mathtt{e}}|\!) \geq (\!|\underline{\mathtt{m}}|\!)$. By Lemma 6, we know that $(\!|\mathtt{f}\ (\overrightarrow{\mathtt{take}\ \underline{\mathtt{n}}\ \mathtt{s}})\ \overrightarrow{\mathtt{e}}|\!) = (\!|\mathtt{f}|\!)((\!|\overline{\mathtt{s}}|\!)\ (\!|\overline{\mathtt{e}}|\!))$. Applying Lemma 2 (which still holds because the interpretation of $+1$ remains unchanged), we obtain $|\underline{\mathtt{m}}| = (\!|\underline{\mathtt{m}}|\!) \leq (\!|\mathtt{f}|\!)((\!|\overline{\mathtt{s}}|\!),\ (\!|\overline{\mathtt{e}}|\!))$. Finally, by Lemma 1, we know that there is a function $G : \mathbb{R}^+ \to \mathbb{R}^+$ such that $|\underline{\mathtt{m}}| \leq (\!|\mathtt{f}|\!)(G(|\overline{\mathtt{s}}|),\ G(|\overline{\mathtt{e}}|))$. □

Example 10. The program `extendupto` of example 8 admits the following additive interpretation $(\!|\mathtt{nil}|\!) = (\!|0|\!) = 0$, $(\!|\mathtt{append}|\!)(X, Y) = X + Y$, $(\!|\mathtt{upto}|\!)(X) = (\!|\mathtt{extendupto}|\!)(X) = 2 \times X^2$ together with $(\!|{+1}|\!)(X) = X + 1$ and $(\!|:|\!)(X, Y) = X + Y + 1$. Consequently, it is SBUB and, defining $F(X) = (\!|\mathtt{extendupto}|\!)(X)$ we know that for any finite list $\mathtt{s}$ of size n, if $\mathtt{eval}(\mathtt{lg}(\mathtt{extendupto}\ \mathtt{s}))\Downarrow \underline{\mathtt{m}}$ then $F(n) \geq |\underline{\mathtt{m}}|$, i.e. we are able to exhibit a precise upper bound. However notice that, as already mentioned, the bound is less tight than in previous criterion. The reason for that is just that size is an upper bound rougher than length.

4.5 Synchrony Upper Bound (SUB)

Sometimes we would like to be more precise about the computational complexity of the program. In this case a suitable property would be synchrony, i.e. a finite part of the input let produce a finite part of the output. Clearly not every stream enjoys this property. Synchrony between stream Input and Output is a non-trivial question that we have already tackled in the previous subsections by providing some upper bounds on the length of finite output stream parts. In this subsection, we consider the problem in a different way: we restrict ourselves to synchronous streams and we adapt the interpretation methodology in order to give upper bound on the output writes size with respect to input reads size.
In the sequel it will be useful to have the following program:

$\mathtt{lInd}\ ::\ \mathtt{Nat} \to [\alpha] \to [\alpha]$
$\mathtt{lInd}\ \mathtt{x}\ \ \mathtt{xs} = (\mathtt{xs}\ !!\ \mathtt{x}) : \mathtt{nil}$

We start to define the meaning of synchrony between input stream reads and output stream writes:

Definition 10. *A stream function* $\mathtt{f} :: \overrightarrow{[\sigma]} \to \overrightarrow{\tau} \to [\sigma]$ *is said to be of type "Read, Write" if for every expression* $\mathtt{s}_i :: \sigma_i$, $\mathtt{e}_i :: \tau_i$ *and for every* $\underline{\mathtt{n}} \in \mathtt{N}$*:*

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

This definition provides a one to one correspondence between input stream reads and output stream writes, because the stream function needs one input read in order to generate one output write and conversely, we know that it will not generate more than one output write (otherwise the two fully evaluated values cannot be matched).

Example 11. The following is an illustration of a "Read, Write" program:

$$\begin{array}{l}\mathtt{sadd} :: [\mathtt{Nat}] \to [\mathtt{Nat}] \to [\mathtt{Nat}]\\ \mathtt{sadd}\ (\mathtt{x} : \mathtt{xs})\ (\mathtt{y} : \mathtt{ys}) = (\mathtt{add}\ \mathtt{x}\ \mathtt{y}) : (\mathtt{sadd}\ \mathtt{xs}\ \mathtt{ys})\end{array}$$

$$\begin{array}{lll}\mathtt{add} :: \mathtt{Nat} \to \mathtt{Nat} \to \mathtt{Nat} & & \\ \mathtt{add}\ (\mathtt{x}+1)\ \ (\mathtt{y}+1) & = & ((\mathtt{add}\ \mathtt{x}\ \mathtt{y})+1)+1\\ \mathtt{add}\ (\mathtt{x}+1)\ \ \ \mathtt{0} & = & \mathtt{x}+1\\ \mathtt{add}\ \ \ \mathtt{0}\ \ \ (\mathtt{y}+1) & = & \mathtt{y}+1\end{array}$$

In this case, we would like to say that for each integer n, the size of the n-th output stream element is equal to the sum of the two n-th input stream elements.

Definition 11. *A "Read, Write" stream function* $\mathtt{f} :: \overrightarrow{[\sigma]} \to \overrightarrow{\tau} \to [\sigma]$ *has a* synchrony upper bound *if there is a function* $F : \mathbb{R}^+ \to \mathbb{R}^+$ *such that for every expression* $\mathtt{s}_i :: [\sigma_i]$, $\mathtt{e}_i :: \tau_i$ *and for every* $\underline{\mathtt{n}} \in \mathtt{N}$*:*

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

Another possibility would have been to consider a n to m correspondence between inputs and outputs. However such correspondences can be studied with slight changes over the one-one correspondence.

4.6 A Criterion for Synchrony Upper Bound

We begin to put some syntactical restriction on the considered programs so that each stream function symbol is "Read, Write" with respect to this restriction.

Definition 12. *A stream function* $\mathtt{f} :: \overrightarrow{[\sigma]} \to \overrightarrow{\tau} \to [\sigma]$ *is synchronously restricted if it can be written (and maybe extended) by definitions of the shape:*

$$\begin{array}{llllll}\mathtt{f} & (\mathtt{x}_1 : \mathtt{xs}_1) \cdots & (\mathtt{x}_n : \mathtt{xs}_n) & \overrightarrow{\mathtt{p}} = \mathtt{hd} : (\mathtt{f}\ \mathtt{xs}_1 \cdots \mathtt{xs}_n\ \overrightarrow{\mathtt{p}})\\ \mathtt{f} & \mathtt{nil} \quad \cdots & \mathtt{nil} & \overrightarrow{\mathtt{p}} = \mathtt{nil}\end{array}$$

where $\mathtt{xs}_1, \ldots, \mathtt{xs}_n$ *do not appear in the expression* hd*.*

Now we may show the following lemma.

Lemma 7. *Every synchronously restricted function is "Read, Write".*

Proof. By induction on numerals. □

Definition 13. *A program is* SUB *if it is synchronously restricted and admits an additive interpretation* $(\!|-|\!)$ *but on* : *where* $(\!|:|\!)$ *is defined by* $(\!|:|\!)(X, Y) = X$*.*

Fully evaluated values, i.e. values v containing only constructor symbols, have the following remarkable property.

Lemma 8. *Given an additive assignment* $(\!|-|\!)$*, for each fully evaluated value* v*:* $|\mathtt{v}| \leq (\!|\mathtt{v}|\!)$*.*

Proof. By structural induction on fully evaluated values. □

Theorem 3. *If a program is* SUB *then each stream function in it admits a synchrony upper bound.*

Proof. By Lemma 7, we can restrict our attention to a "Read, Write" stream function $\mathtt{f} :: \overrightarrow{[\sigma]} \to \overrightarrow{\tau} \to [\sigma]$ s.t. $\forall \underline{\mathtt{n}} \in \mathtt{N}$, we both have $\mathtt{eval}\ ((\mathtt{f}\ \overrightarrow{\mathtt{s}}\ \overrightarrow{\mathtt{e}})\ !!\ \underline{\mathtt{n}}) \Downarrow \mathtt{v}$ and $\mathtt{eval}(\mathtt{s}_i\ !!\ \underline{\mathtt{n}}) \Downarrow \mathtt{w}_i$.

We firstly prove that $\mathtt{eval}\ (\mathtt{f}\ \overrightarrow{(\mathtt{w} : \mathtt{nil})}\ \overrightarrow{\mathtt{e}}) \Downarrow \mathtt{v} : \mathtt{nil}$. By assumption, we have $\mathtt{eval}((\mathtt{f}\ \overrightarrow{\mathtt{s}}\ \overrightarrow{\mathtt{e}})\ !!\ \underline{\mathtt{n}}) \Downarrow \mathtt{v}$, so in particular we also have $\mathtt{eval}\ (((\mathtt{f}\ \overrightarrow{\mathtt{s}}\ \overrightarrow{\mathtt{e}})\ !!\ \underline{\mathtt{n}}) : \mathtt{nil}) \Downarrow \mathtt{v} : \mathtt{nil}$. By definition of "Read, Write" function, $\mathtt{eval}\ ((\mathtt{f}\ ((\overrightarrow{\mathtt{s}}\ !!\ \underline{\mathtt{n}}) : \mathtt{nil})\ \overrightarrow{\mathtt{e}}) \Downarrow \mathtt{v} : \mathtt{nil}$ and since by assumption $\mathtt{eval}(\mathtt{s}_i\ !!\ \underline{\mathtt{n}}) \Downarrow \mathtt{w}_i$ we can conclude $\mathtt{eval}\ (\mathtt{f}\ \overrightarrow{(\mathtt{w} : \mathtt{nil})}\ \overrightarrow{\mathtt{e}}) \Downarrow \mathtt{v} : \mathtt{nil}$. By Corollary 2, we have $(\!|\mathtt{f}\ \overrightarrow{(\mathtt{w} : \mathtt{nil})}\ \overrightarrow{\mathtt{e}}|\!) \geq (\!|\mathtt{v} : \mathtt{nil}|\!)$. By definition of SUB programs, we know that $(\!|:|\!)(X, Y) = X$ and, consequently, $(\!|\mathtt{f}|\!)((\!|\overline{\mathtt{w} : \mathtt{nil}}|\!), (\!|\overline{\mathtt{e}}|\!)) = (\!|\mathtt{f}|\!)((\!|\overline{\mathtt{w}}|\!), (\!|\overline{\mathtt{e}}|\!)) \geq (\!|\mathtt{v} : \mathtt{nil}|\!) = (\!|\mathtt{v}|\!)$. By Lemma 8, we have $(\!|\mathtt{f}|\!)((\!|\overline{\mathtt{w}}|\!), (\!|\overline{\mathtt{e}}|\!)) \geq |\mathtt{v}|$. Finally, by Lemma 1, there exists a function $G : \mathbb{R}^+ \to \mathbb{R}^+$ such that $(\!|\mathtt{f}|\!)(G(|\overline{\mathtt{w}}|), G(|\overline{\mathtt{e}}|)) \geq |\mathtt{v}|$. We conclude by taking $F(X) = (\!|\mathtt{f}|\!)(G(\overline{X}), G(\overline{X}))$. □

Example 12. The program of example 11 is synchronously restricted (it can be extended in such a way) and admits the following additive interpretation $(\!|0|\!) = 0$, $(\!|+1|\!)(X) = X+1$, $(\!|\mathtt{add}|\!)(X, Y) = X+Y$, $(\!|\mathtt{sadd}|\!)(X, Y) = X+Y$ and $(\!|:|\!)(X, Y) = X$. Consequently, the program is SUB and admits a synchrony upper bound. Moreover, taking $F(X) = (\!|\mathtt{sadd}|\!)(X, X)$, we know that if the k-th input reads evaluate to numbers $\underline{\mathtt{n}}$ and $\underline{\mathtt{m}}$ then $F(\max(|\underline{\mathtt{m}}|, |\underline{\mathtt{n}}|))$ is an upper bound on the k-th output size.

5 Conclusion

In this paper, we have applied interpretation methods for the first time to a lazy functional stream language, obtaining several criteria ensuring bound properties on the input read and output write elements of a program working on stream data. This shows that interpretations are a valid tool to well ensure stream function properties. Many interesting properties should be investigated, in particular memory leaks and overflows [17,18]. These questions are strongly related to the notions we have tackled in this paper. For example, consider the following:

$$\begin{array}{ll} \mathtt{odd} :: [\alpha] \to [\alpha] & \mathtt{memleak} :: [\alpha] \to [\alpha \times \alpha] \\ \mathtt{odd}\ (\mathtt{x} : \mathtt{y} : \mathtt{xs}) = \mathtt{x} : (\mathtt{odd}\ \mathtt{xs}) & \mathtt{memleak}\ \mathtt{s} = \mathtt{merge}\ (\mathtt{odd}\ \mathtt{s})\ \mathtt{s} \end{array}$$

The evaluation of an expression `memleak s` leads to a memory leak. Indeed, `merge` reads one stream element on each of its arguments in order to output one element and `odd` needs to read two stream input elements of `s` in order to output one element whereas `s` just makes one output for one input read. Consequently, there is a factor 2 of asynchrony between the two computations on `s`. Which means that `merge` needs to read $\mathtt{s}_n$ and $\mathtt{s}_{2\times n}$ (where $\mathtt{s}_i$ is the i-th element of

s) in order to compute the n-th output element. From a memory management perspective, it means that all the elements between $\mathtt{s}_n$ and $\mathtt{s}_{2\times n}$ have to be stored, leading the memory to a leak. We think interpretations could help the programmer to prevent such "bad properties" of programs. Moreover, we think that interpretations can be also exploited in the study of stream definitions, in particular in the context of stream productivity, but we leave this subject for further researchs.

References

1. Manna, Z., Ness, S.: On the termination of Markov algorithms. In: Third hawaii international conference on system science, pp. 789–792 (1970)
2. Lankford, D.: On proving term rewriting systems are noetherien. tech. rep (1979)
3. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of ACM POPL 1977, pp. 238–252 (1977)
4. Bonfante, G., Marion, J.Y., Moyen, J.Y.: Quasi-interpretations, a way to control resources. TCS (accepted)
5. Amadio, R.: Synthesis of max-plus quasi-interpretations. Fundamenta Informaticae 65(1-2) (2005)
6. Bonfante, G., Marion, J.Y., Moyen, J.Y., Péchoux, R.: Synthesis of quasi-interpretations. In: LCC 2005, LICS Workshop (2005), http://hal.inria.fr/
7. Marion, J.Y., Péchoux, R.: Sup-interpretations, a semantic method for static analysis of program resources. In: ACM TOCL 2009 (accepted, 2009), http://tocl.acm.org/
8. Marion, J.-Y., Péchoux, R.: A Characterization of NCk by First Order Functional Programs. In: Agrawal, M., Du, D.-Z., Duan, Z., Li, A. (eds.) TAMC 2008. LNCS, vol. 4978, pp. 136–147. Springer, Heidelberg (2008)
9. Bonfante, G., Kahle, R., Marion, J.-Y., Oitavem, I.: Recursion schemata for NC^k. In: Kaminski, M., Martini, S. (eds.) CSL 2008. LNCS, vol. 5213, pp. 49–63. Springer, Heidelberg (2008)
10. Dijkstra, E.W.: On the productivity of recursive definitions. EWD749 (1980)
11. Sijtsma, B.: On the productivity of recursive list definitions. ACM TOPLAS 11(4), 633–649 (1989)
12. Coquand, T.: Infinite objects in type theory. In: Barendregt, H., Nipkow, T. (eds.) TYPES 1993. LNCS, vol. 806, pp. 62–78. Springer, Heidelberg (1994)
13. Wadge, W.: An extensional treatment of dataflow deadlock. TCS 13, 3–15 (1981)
14. Endrullis, J., Grabmayer, C., Hendriks, D., Isihara, A., Klop, J.W.: Productivity of Stream Definitions. In: Csuhaj-Varjú, E., Ésik, Z. (eds.) FCT 2007. LNCS, vol. 4639, pp. 274–287. Springer, Heidelberg (2007)
15. Telford, A., Turner, D.: Ensuring streams flow. In: Johnson, M. (ed.) AMAST 1997. LNCS, vol. 1349, pp. 509–523. Springer, Heidelberg (1997)
16. 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)
17. Hughes, J., Pareto, L., Sabry, A.: Proving the correctness of reactive systems using sized types. In: Proceedings of ACM POPL 1996, pp. 410–423 (1996)

18. Frankau, S., Mycroft, A.: Stream processing hardware from functional language specifications. In: Proceeding of IEEE HICSS-36 (2003)
19. Huet, G.: Confluent reductions: Abstract properties and applications to term rewriting systems. Journal of the ACM 27(4), 797–821 (1980)
20. Launchbury, J.: A natural semantics for lazy evaluation. In: Proceedings of POPL 1993, pp. 144–154 (1993)
21. Dershowitz, N.: Orderings for term-rewriting systems. TCS 17(3), 279–301 (1982)
22. Marion, J.Y., Péchoux, R.: Characterizations of polynomial complexity classes with a better intensionality. In: Proceedings ACM PPDP 2008, pp. 79–88 (2008)
23. Bird, R., Wadler, P.: Introduction to Functional Programming. Prentice-Hall, Englewood Cliffs (1988)