On Bounding Space Usage of Streams Using Interpretation Analysis

Marco Gaboardi, Romain Péchoux

On Bounding Space Usage of Streams Using Interpretation Analysis

Marco Gaboardi[1,2]

Harvard University and University of Dundee

Romain Péchoux[1]

Université de Lorraine - INRIA project Carte, Loria, UMR 7503

Abstract

Interpretation methods are important tools in implicit computational complexity. They have been proved particularly useful to statically analyze and to limit the complexity of programs. However, most of these studies have been so far applied in the context of term rewriting systems over finite data.

In this paper, we show how interpretations can also be used to study properties of lazy first-order functional programs over streams. In particular, we provide some interpretation criteria useful to ensure two kinds of stream properties: *space upper bounds* and *input/output upper bounds*. Our space upper bounds criteria ensures global and local upper bounds on the size of each output stream element expressed in terms of the maximal size of the input stream elements. The input/output upper bounds criteria consider instead the relations between the number of elements read from the input stream and the number of elements produced on the output stream.

This contribution can be seen as a first step in the development of a methodology aiming at using interpretation properties to ensure space safety properties of programs working on streams.

Keywords: Stream Programs, Interpretations, Program Space Usage, Lazy Languages, Implicit Computational Complexity.

Email addresses: m.gaboardi@dundee.ac.uk (Marco Gaboardi), pechoux@loria.fr (Romain Péchoux)

URL: http://www.cs.unibo.it/~gaboardi (Marco Gaboardi), http://www.loria.fr/~pechoux (Romain Péchoux)

[1]Work partially supported by the projects ANR-14-CE25-0005 "ELICA", MIUR-PRIN'07 "CONCERTO" and INRIA Associated Team "CRISTAL".

[2]The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement n° 272487.

1. Introduction

The advances obtained in communication technology in the last two decades have posed new challenges to the software community. One of these challenges comes from the advancements achieved in computer networking where new software able to handle huge amount of data in an efficient way is required.

This situation has brought a renewed interest for stream-like data structures and for programs managing those data structures. Indeed, by representing discrete potentially infinite information flows, streams can be used to formalize and study situations as real-time data processing, network communication flows, audio and video signals flows, etc. Clearly, the problems that stream programs raise are different from the ones generally considered in the usual scenario where data are assumed to be finite. For this reason, several programming languages have been proposed with the aim of modeling stream-based computations, see [39] for a survey.

The aim of the present work is to contribute to the current scenario by developing some static analysis techniques useful to ensure basic properties of lazy functional programs working on streams. This is the first step of a more general investigation of complexity and efficiency properties of programs working on streams.

Stream-like languages and properties. Several formal frameworks have been designed for the manipulation of infinite objects including infinitary rewriting [23] and infinitary lambda-calculus [24]. Important properties of these models such as infinitary weak normalization and infinitary strong normalization have been deeply studied in the literature. However, little attention has been paid to space properties of such models. A different setting handling infinite data-structures is computable analysis, which provides several models of computation over real numbers [40]. In this setting a lot of work has been done to adapt the classical concept of complexity class and obtain implicit characterizations. However, even if streams can be considered as particular real numbers, the properties of interest for stream programs are usually different from the ones of interest in computable analysis.

A well-established approach to deal with infinite data, and in particular with streams, is by using *laziness* in functional programming languages [21]. In languages like Haskell, streams are expressions denoting infinite lists whose elements are evaluated on demand. In this way streams can be treated by finitary means. The practical diffusion of lazy programming languages has stimulated the development of tools and techniques in order to prove properties of programs in the presence of infinite data structures.

For example, on the side of program equivalence much attention has been paid to the study of co-induction and bisimulation techniques in languages working on streams [35, 20]. A property of stream definitions that has motivated many studies is productivity [14]. 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 restricted criteria have been studied to ensure it [38, 12, 22, 15].

Besides program equivalence and productivity, other stream program properties, in particular space-related properties, have received little attention. Such properties are studied in this paper through the use of interpretations, a static analysis tool.

Interpretations. Interpretation methods originate from the natural observation that, in order to reason about program properties, it is sometimes more convenient to interpret syntactic program constructions into the objects of an abstract domain and prove properties about the obtained abstract objects.

Interpretation methods have been proved useful in many situations and are nowadays well-established verification tools for proving properties of programs. Variants of interpretation have been used for example to prove the termination of term rewriting systems [31, 26], to obtain sound approximations of program behaviors useful to static analysis [13] and to obtain implicit characterizations of complexity classes [7, 32].

One variation of particular interest for implicit computational complexity is the notion of quasi-interpretation [7]. A quasi-interpretation maps program constructions to functions over real numbers. The mapping is chosen in such a way that the function obtained as the interpretation of a program describes an upper bound on the size of the computed values with respect to the size of input values. Thanks to this, quasi-interpretations are particularly adapted to study program complexity in an elegant way.

Another important property of quasi-interpretation is that the problem of finding a quasi-interpretation of a given program for some restricted class of polynomials is decidable [2, 9]. This suggests that quasi-interpretations can be used as a concrete tool to analyze the complexity of functional programs.

An important new issue is whether interpretations can be used in order to infer such properties on programs computing over infinite data. Here we approach this problem by considering lazy programs over stream data.

Contribution. In this paper, we consider a simple first-order lazy language and we start a systematic study of *space properties* of programs working on streams by means of interpretation methods.

In many stream applications one is interested in processing data in a fast and memory-safe way. In order to do this, one can think to improve space-efficiency by using some buffering operations to memorize only the part of the stream involved in the actual computation. Following this intuition, it becomes natural to study space properties of programs working on streams in a more abstract way. We study two classes of space properties:

- *Stream Upper Bounds*: these are properties about the size of each stream element produced by a program. They correspond to properties about the elements memorized in the buffer.

- *Bounded Input/Output Properties*: these are properties about the number of stream elements produced by a program. They correspond to properties about the number of elements produced on the output wrt to the number of elements read on the input.

These properties analyze two "dimensions" of programs working on streams. The combination of these properties allows one to study a reasonable class of programs and to obtain the information needed in order to improve the memory management process of programs working on streams.

The results presented in this paper have been originally developed in [18] and [19]. In [19], we mainly studied the space upper bounds properties while in [18] we studied the bounded input/output properties. The present paper generalizes and extends these works, in particular, to a pure functional programming style. Indeed previous works were restricted to term rewrite systems and the adaptation of interpretation methods to pure functional programs is a new non-trivial feature. Consequently, new proofs but also more illustrating diagrams and examples have been provided. Finally, a deeper comparison with the state of the art on stream properties (productivity, complexity, ...) and related works has been provided in Section 7.

Stream Upper Bounds. In order to process stream data in a memory-efficient way it is useful to obtain an estimate of the memory needed to store the elements produced by a stream program.

In some situations, an estimate can be obtained by considering in a *global* way the greatest size of the elements produced by the program as outputs. In other situations, however, there is no such a maximal element with respect to the size measure and so only an estimate considering the *local* position of the element in the stream can be given. Consider the following stream definitions:

$$\begin{array}{ll} \texttt{ones} :: [\texttt{Nat}] & \texttt{nats} :: \texttt{Nat} \to [\texttt{Nat}] \\ \texttt{ones} \doteq \underline{1} : \texttt{ones} & \texttt{nats}\ \texttt{x} \doteq \texttt{x} : (\texttt{nats}\ (\texttt{x} + 1)) \end{array}$$

In both cases, it is easy to obtain such estimates. Indeed, in the stream definition of `ones` all the elements have the same size, while in the definition of `nats` every element has a size depending on its position in the stream.

However, when more complex stream programs are considered, deeper analyses are needed. In this paper, we will use interpretations to define two criteria useful to compute both kinds of space estimates.

Consider the following stream program:

$$\begin{array}{ll} \texttt{repeat} :: \texttt{Nat} \to [\texttt{Nat}] & \texttt{zip} :: [a] \to [a] \to [a] \\ \texttt{repeat}\ \texttt{x} \doteq \texttt{x} : (\texttt{repeat}\ \texttt{x}) & \texttt{zip}\ (\texttt{x} : \texttt{xs})\ \texttt{ys} \doteq \texttt{x} : (\texttt{zip}\ \texttt{ys}\ \texttt{xs}) \end{array}$$

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 n, i.e. the maximal natural number encoding $\underline{\texttt{n}}$ in a subterm `repeat` $\underline{\texttt{n}}$ in `s`. In particular, it means that every stream `s` built only using `repeat` and `zip` is globally bounded by a constant n. In order to generalize this informal analysis, we study a *Global Upper Bound* (GUB) criterion 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:

$$\begin{array}{ll} \texttt{nats} :: \texttt{Nat} \to [\texttt{Nat}] & \texttt{sadd} :: [\texttt{Nat}] \to [\texttt{Nat}] \to [\texttt{Nat}] \\ \texttt{nats}\ \texttt{x} \doteq \texttt{x} : (\texttt{nats}\ (\texttt{x} + 1)) & \texttt{sadd}\ (\texttt{x} : \texttt{xs})\ (\texttt{y} : \texttt{ys}) \doteq (\texttt{add}\ \texttt{x}\ \texttt{y}) : (\texttt{sadd}\ \texttt{xs}\ \texttt{ys}) \end{array}$$

Every stream `s` built using `nats` and `sadd` is not globally bound. Nevertheless it is easy for every such an `s` to compute a function f such that every element of `s` in the *local* position n has a size bounded by $f(n)$. In order to generalize this informal argument, we study a *Local Upper Bound* (LUB) criterion ensuring that the size of the n-th eval-

uated element of a stream is bounded by a function in its index n and the maximal size of the input. All the productive stream functions have a local upper bound, however in order to establish a criteria ensuring it we need an extension of the usual notion of interpretation. For this reason we introduce the notion of *parametrized interpretation*, i.e. an interpretation where functions depend on external parameters.

Bounded Input/Output Properties. Another information that is useful to obtain in order to improve memory-efficiency is an estimate of the number of elements produced by a stream program when fed with only a portion of the input stream. Indeed if one think to online streaming, these properties consist in bounding the speed-up that might occur in the network during communication.

In some situations, such an estimate can be obtained by considering only the *length* of the portion of the input stream. In other situations, however, this is not sufficient and so in order to obtain the estimate one needs to consider also the *size* of the elements in the portion. Consider the following definitions:

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

$$\texttt{dup} :: [a] \to [a]$$
$$\texttt{dup}\ (\texttt{x} : \texttt{xs}) \doteq \texttt{x} : (\texttt{x} : (\texttt{dup}\ \texttt{xs}))$$

It is easy to verify that each stream expression built using only `merge` and `dup` will only generate a number of output elements that depends on the number of input read elements; e.g. the expression $\texttt{dup}\ (\texttt{merge}\ (\texttt{dup}\ \texttt{s})\ (\texttt{dup}\ \texttt{s}))$ for each read element of the input stream `s` produces four elements of the type $a \times a$. In order to generalize this informal argument, we study a *Length-Based Upper Bound* (LBUB) criterion ensuring that the number m of output stream elements is bounded by a function in the number n of stream elements in input.

Many stream functions have a length-based upper bound. However, there are stream functions that generate a number of output elements that does not depend only on the number of input read elements. Consider the following definitions:

$$\texttt{app} :: [a] \to [a] \to [a]$$
$$\texttt{app}\ (\texttt{x} : \texttt{xs})\ \texttt{ys} \doteq \texttt{x} : (\texttt{app}\ \texttt{xs}\ \texttt{ys})$$
$$\texttt{app}\ \texttt{nil}\ \texttt{ys} \doteq \texttt{ys}$$

$$\texttt{upto} :: \texttt{Nat} \to [\texttt{Nat}]$$
$$\texttt{upto}\ \texttt{0} \doteq \texttt{nil}$$
$$\texttt{upto}\ (\texttt{x} + 1) \doteq (\texttt{x} + 1) : (\texttt{upto}\ \texttt{x})$$

$$\texttt{extendupto} :: [\texttt{Nat}] \to [\texttt{Nat}]$$
$$\texttt{extendupto}\ (\texttt{x} : \texttt{xs}) \doteq \texttt{app}\ (\texttt{upto}\ \texttt{x})\ (\texttt{extendupto}\ \texttt{xs})$$

It is easy to verify that every stream expression built using only `upto` and `extendupto` will generate a number of output elements that is related to both the number and the size of input read elements; e.g. the expression: $\texttt{extendupto}\ (\texttt{extendupto}\ \texttt{s})$ for each natural number $\underline{\texttt{n}}$ in the input stream `s` outputs $\sum_{i=1}^{|\underline{\texttt{n}}|} i$ elements. In order to generalize this informal argument, we study a *Size-Based Upper Bound* (SBUB) criterion ensuring that the number m of output stream elements is bounded by a function in the number and the size of the stream elements in input.

Other Technical Contributions. Besides the study of stream program properties, this paper contains two other technical contributions:

- a definition of interpretations for a lazy first-order programming language
- a definition of a new kind of interpretations, named *parametrized interpretations*

Interpretations have been so far presented as tools dealing with properties about rewriting systems. Here instead, we are interested in programs of a first-order lazy functional program. A possible approach could have been to translate programs in a term rewriting system and analyze them using the standard interpretation framework. Instead, we have adapted the interpretation tools to our case. This choice is due on the one hand to the desire to have a treatment as close as possible to the programming language, on the other hand this is also due to the desire of understanding the flexibility and the adaptability of the interpretation tools.

Parametrized interpretations extend standard interpretations by means of an external parameter. In the parametrized interpretations, all the functions appearing in the assignments can depend on external parameters. However, the parameter has a different status with respect to the other arguments of the functions. Thanks to this extension, we are able to deal with properties about stream local positions as required by the Local Upper Bound property.

Outline of the paper. In Section 2, we introduce the language, named SFL, and some notations. In Section 3, we introduce interpretations and parametrized interpretations. In Section 4, we study the space upper bound properties and the semantic interpretation criteria to ensure them. In Section 5, we consider the bounded input/output upper bound properties and how to ensure them through interpretation criteria. In section 6, we discuss the problem of computing program interpretations. In Section 7, we present the related works. In Section 8, we draw some conclusions.

2. The SFL language

In the present section, we introduce the syntax and the operational semantics of the language that will be used all along this paper. The language is dubbed SFL, acronym for *Stream First-order Lazy language*. This is an Haskell-like lazy first-order language computing on simple *stream* data.

We consider programs of SFL to be well-typed closed expressions of base type. Programs can be evaluated thanks to a lazy big step semantics where by *lazy* we mean that the evaluation does not go under a constructor. This permits to deal with streams and infinite computations in a natural way. Indeed, analogously to what happens in Haskell, we can prove program properties by equational reasoning. However, our operational semantics differs from the Haskell one since we do not consider sharing.

2.1. Syntax and Types

Let $\mathcal{X}$, $\mathcal{C}$ and $\mathcal{F}$ be three disjoint sets representing the set of *variables*, the set of *constructor symbols* (or constructors) and the set of *function symbols* respectively. In the sequel, $\mathtt{x}, \mathtt{c}, \mathtt{f}$ and $\mathtt{t}$ denote symbols in $\mathcal{X}$, $\mathcal{C}$, $\mathcal{F}$ and $\mathcal{C} \cup \mathcal{F}$, respectively.

Definition 1. *The syntax of the* SFL *language is described by the following grammar:*

$$
\begin{array}{lll}
\mathtt{p} & ::= \mathtt{x} \mid \mathtt{c}(\mathtt{x}, \ldots, \mathtt{x}) & \text{(Patterns)} \\
\mathtt{e} & ::= \mathtt{x} \mid \mathtt{c}(\mathtt{e}, \ldots, \mathtt{e}) \mid \mathtt{f}(\mathtt{e}, \ldots, \mathtt{e}) \mid \mathtt{LetRec\ d\ in\ e} \mid & \text{(Expressions)} \\
 & \quad \mathtt{Case\ e\ of\ p} \to \mathtt{e}, \ldots, \mathtt{p} \to \mathtt{e} & \\
\mathtt{d} & ::= \mathtt{f}(\mathtt{x}, \ldots, \mathtt{x}) \doteq \mathtt{e} & \text{(Definitions)} \\
\mathtt{v} & ::= \mathtt{c}(\mathtt{e}, \ldots, \mathtt{e}) & \text{(Lazy Values)} \\
\underline{\mathtt{v}} & ::= \mathtt{c}(\underline{\mathtt{v}}, \ldots, \underline{\mathtt{v}}) & \text{(Strict Values)}
\end{array}
$$

In the examples presented in the sequel, the set of constructor symbols will include the usual constructors for natural numbers (i.e. $\mathtt{0}$, $+\,1$), lists (i.e. $\mathtt{nil}$, $:$) and pairs (i.e. $\langle \cdot, \cdot \rangle$) and it may also include other standard algebraic data types. Besides, we assume the set of constructors to contain also a constructor `Err` that will be used to track pattern matching failures.

We consider *patterns* that are either a variable or a constructor possibly applied to other variables. For simplicity, we assume that a variable can appear at most once in a pattern and that the patterns are non-overlapping.

Expressions can be built using variables, constructors, function symbols, the `LetRec` construction and the `Case` construction. We consider a grammar where constructors and functions symbols do not appear partially applied in an expression, e.g a function symbol `f` of arity two will only appear in the form $\mathtt{f}(\mathtt{e}_1, \mathtt{e}_2)$, for some expressions $\mathtt{e}_1$ and $\mathtt{e}_2$.

The `Case` constructor as usual allows one to perform pattern matching. Note that even if the patterns are built by using (at most) one constructor at a time, by using nested `Case` more complex patterns can be explored. The `LetRec` construction is used to locally define recursive functions. In particular, a construction like `LetRec` $\mathtt{d_f}$ `in` `e` has two parameters: a *function definition* $\mathtt{d_f}$ and an expression `e`. The function definition $\mathtt{d_f}$ is the actual place where a recursive definition is assigned to the function symbol `f`. The expression `e` is the scope of that definition.

We distinguish two kinds of *values*: *lazy* and *strict values*. The semantics in the next subsection evaluates programs to lazy values. Strict values are specific lazy values that will be used to define the program analyses presented in the following sections. In particular, later in this section, we will show how to define an `eval` program forcing the evaluation of a program to a strict value.

Free and *bound* variables are defined as usual. However, free variables in expressions can be also explicitly bound in *definitions*, that is: given a definition `d` of the shape $\mathtt{f}(\mathtt{x}_1, \cdots, \mathtt{x}_n) \doteq \mathtt{e}$, the bound variables of `d` are the ones of `e` and $\mathtt{x}_1, \ldots, \mathtt{x}_n$. Note also that a `LetRec` construction can bind function symbols. That is, the function symbol `f` is bound in $\mathtt{e}'$ in an expression of the shape $\mathtt{LetRec\ f}(\mathtt{x}_1, \cdots, \mathtt{x}_n) \doteq \mathtt{e\ in\ e}'$. For simplicity, we assume that all the bound variables and function symbols have distinct names so that name clashes are avoided.

As outlined above, we are mainly concerned with stream programs properties related to space. So, we need to introduce a notion of *size* for expressions and programs.

Definition 2 (Size)**.** *The* size *of an expression* `e`*, denoted* $|\mathtt{e}|$*, is defined as*

- $|\mathtt{x}| = 1$

- $|\mathtt{t}(\mathtt{e}_1, \cdots, \mathtt{e}_n)| = \begin{cases} 0 & \textit{if } \mathtt{t} \textit{ is of arity } 0 \\ \sum_{1 \leq i \leq n} |\mathtt{e}_i| + 1 & \textit{if } \mathtt{t} \textit{ is of arity } n \geq 1 \end{cases}$
- $|\mathtt{LetRec\ d\ in\ e}| = |\mathtt{e}|$
- $|\mathtt{Case\ e\ of\ p}_1 \rightarrow \mathtt{e}_1, \ldots, \mathtt{p}_n \rightarrow \mathtt{e}_n| = |\mathtt{e}| + \max_{1 \leq i \leq n} |\mathtt{e}_i|$

Note that if we take N to be the set of natural number expressions inductively defined using the constructors 0 and $+1$ then for each $\underline{\mathtt{n}} \in \mathtt{N}$ we have $|\underline{\mathtt{n}}| = n$, i.e. the size of a natural number expression is equal to the value it represents.

In the sequel only expressions that are well-formed and well-typed will be considered.

Definition 3.

- *A definition* $\mathtt{f}(\mathtt{x}_1, \cdots, \mathtt{x}_n) \doteq \mathtt{e}$ *is* well-formed *if and only if all the free variables of* e *are among* $\mathtt{x}_1 \cdots \mathtt{x}_n$*, i.e. the definition has no free variables.*
- *An expression* e *is* well-formed *if and only if for every function symbol* f *there is exactly one well-formed function definition* d *defining it, i.e. of the shape* $\mathtt{f}(\mathtt{x}_1, \cdots, \mathtt{x}_n) \doteq \mathtt{e}'$.

We conclude this part by describing some of the notations we will use in the sequel. In presenting the examples we adopt the standard applicative convention for the parenthesis (as in Haskell), e.g. we use $\mathtt{f}\ (\mathtt{x} + 1)\ 0$ to denote $\mathtt{f}(\mathtt{x} + 1, 0)$. We use the vector notation $\vec{\mathtt{e}}$ as a shorthand for a sequence of expressions as $\mathtt{e}_1, \ldots, \mathtt{e}_n$. So, for instance the expression $\mathtt{t}(\mathtt{e}_1, \ldots, \mathtt{e}_n)$ could be also written as $\mathtt{t}\ \vec{\mathtt{e}}$. Finally, given a sequence of expressions $\vec{\mathtt{e}}$ and a function F on expressions, we use $F(\vec{\mathtt{e}})$ to denote $F(\mathtt{e}_1), \ldots, F(\mathtt{e}_n)$, i.e. the componentwise application of F to the sequence $\vec{\mathtt{e}}$. For instance, given a sequence $\vec{\mathtt{e}} = \mathtt{e}_1, \cdots, \mathtt{e}_n$, we use $|\vec{\mathtt{e}}|$ as a notation for $|\mathtt{e}_1|, \ldots, |\mathtt{e}_n|$.

Type system. As stressed before, we want to consider only expressions that are well-typed. Here we introduce the type system that assigns types to all the syntactic constructions of the SFL language. As usual, the type system ensures that a program does not go wrong. Roughly speaking, a *wrong computation* happens when a program cannot be evaluated to a value because of some stuck computation. Note however that this does not prevent a program from either diverging or evaluating to Err. Indeed, in our setting, this fact is important both for making the pattern matching working properly and also for using some program analysis techniques presented in the following sections.

In order to make simpler our analyses, we only consider well-typed first-order programs dealing with lists that do not contain other lists. This is because we want to prevent object like streams of streams that cannot be analyzed in a proper way by the methods that we will present in the sequel. We assure this property by a typing restriction similar to the one of [17]. The following type definition reflects this and the fact that we restrict our attention only to first-order programs.

Definition 4. *The* SFL *types are defined by the following grammar:*

$$\frac{\Gamma(\mathtt{x}) = \mathtt{A}}{\Gamma;\Delta \vdash \mathtt{x} :: \mathtt{A}}\ (\mathtt{Var}) \qquad \frac{}{\Gamma;\Delta \vdash \mathtt{Err} :: \mathtt{A}}\ (\mathtt{A-Err})$$

$$\frac{\mathtt{c} :: \mathtt{A}_1 \to \cdots \to \mathtt{A}_n \to \mathtt{A} \qquad \Gamma;\Delta \vdash \mathtt{e}_i :: \mathtt{A}_i\ (1 \leq i \leq n)}{\Gamma;\Delta \vdash \mathtt{c}(\mathtt{e}_1, \ldots, \mathtt{e}_n) :: \mathtt{A}}\ (\mathtt{Con})$$

$$\frac{\Delta(\mathtt{f}) = \mathtt{A}_1 \to \cdots \to \mathtt{A}_n \to \mathtt{A} \qquad \Gamma;\Delta \vdash \mathtt{e}_i :: \mathtt{A}_i\ (1 \leq i \leq n)}{\Gamma;\Delta \vdash \mathtt{f}(\mathtt{e}_1, \ldots, \mathtt{e}_n) :: \mathtt{A}}\ (\mathtt{Fun})$$

$$\frac{\Gamma;\Delta \vdash \mathtt{e} :: \mathtt{A} \qquad \Gamma_i;\emptyset \vdash \mathtt{p}_i :: \mathtt{A} \quad \Gamma,\Gamma_i;\Delta \vdash \mathtt{e}_i :: \mathtt{B}\ (1 \leq i \leq m)}{\Gamma;\Delta \vdash \mathtt{Case\ e\ of\ p}_1 \to \mathtt{e}_1, \ldots, \mathtt{p}_m \to \mathtt{e}_m :: \mathtt{B}}\ (\mathtt{Case})$$

$$\frac{\Gamma, \mathtt{x}_1 :: \mathtt{A}_1, \ldots, \mathtt{x}_n :: \mathtt{A}_n, ;\Delta, \mathtt{f} : \mathtt{A}_1 \to \cdots \to \mathtt{A}_n \to \mathtt{A} \vdash \mathtt{e} :: \mathtt{A}}{\Gamma;\Delta \vdash \mathtt{f}(\mathtt{x}_1, \ldots, \mathtt{x}_n) \doteq \mathtt{e} :: \mathtt{A}_1 \to \cdots \to \mathtt{A}_n \to \mathtt{A}}\ (\mathtt{Def})$$

$$\frac{\Gamma;\Delta \vdash \mathtt{f}(\mathtt{x}_1, \ldots, \mathtt{x}_n) \doteq \mathtt{e} :: \phi \quad \Gamma;\Delta, \mathtt{f} :: \phi \vdash \mathtt{e}_1 :: \mathtt{A}}{\Gamma;\Delta \vdash \mathtt{LetRec\ f}(\mathtt{x}_1, \ldots, \mathtt{x}_n) \doteq \mathtt{e\ in\ e}_1\ :: \mathtt{A}}\ (\mathtt{Letrec})$$

Table 1: SFL type system

$$\begin{array}{lcll} \sigma & ::= & a \mid \mathtt{Nat} \mid \sigma \times \sigma & \textit{(basic types)} \\ \mathtt{A} & ::= & \alpha \mid \sigma \mid \mathtt{A} \times \mathtt{A} \mid [\sigma] & \textit{(base types)} \\ \phi & ::= & \mathtt{A} \mid \mathtt{A} \to \phi & \textit{(types)} \end{array}$$

where a is a basic type variable*, α is a* type variable*,* Nat *is a* constant type *representing natural numbers, $\times$ and $[\,]$ are* base type constructors *for* pairs *and (finite and infinite lists)* streams *respectively.*

As stressed above, it is worth noticing that the above definition can be extended to other algebraic data types. In the sequel, we use a, b to denote basic type variables, α, β to denote type variables, σ, τ for basic data types, $\mathtt{A}, \mathtt{B}$ to denote base types and ϕ for types. We will tacitly use restricted polymorphism, i.e. a basic type variable a and a type variable α will represent every basic and base type respectively.

For notational convenience, we will use the vector notation $\overrightarrow{\mathtt{A}} \to \mathtt{B}$ as an abbreviation for $\mathtt{A}_1 \to \cdots \to \mathtt{A}_n \to \mathtt{B}$.

The type system proves two kinds of typing judgments: $\Gamma;\Delta \vdash \mathtt{e} :: \mathtt{A}$ for expressions, and $\Gamma;\Delta \vdash \mathtt{f}(\mathtt{x}_1, \ldots, \mathtt{x}_n) \doteq \mathtt{e} :: \phi$ for function definitions. In particular, the judgments for expressions assign a *base type* to an expression, while the judgments for function definitions assign a *type* to a function definition. The symbols Γ and Δ denote variables and function symbols *contexts* respectively; that is, partial functions assigning types to variables and function symbols respectively. Note that we do not consider constants symbols in contexts but instead we assume that they come with a fixed type signature.

Definition 5. *Well-typed expressions and function definitions are defined using the type*

system in Table 1.

It is worth noticing that the symbol `Err` can be typed with each base type `A`. This is essential in order to get type preservation in the evaluation mechanism. Note also that the functional types can be assigned to constructor and function symbols, but only base types can be assigned to expressions. Consequently, our language only allows programs with first-order function definitions.

Definition 6. *A* SFL program *is a well-formed expression* $\mathtt{e}$ *that is typable through a type judgment of the shape* $\emptyset;\emptyset \vdash \mathtt{e} :: \mathtt{A}$ *such that* $\mathtt{A}$ *does not contain free type variables.*

While the above definition could seem a bit odd, it is easy to verify that this corresponds to the usual notion of programs considered as closed terms of observable types.

Notations for the examples. The language we introduced above makes the interpretation definitions we will provide in the following section more formal. In contrast, concrete examples can be cumbersome. So, in the remainder of the paper we will use some *syntactic sugar* to improve readability. Let us start to show that we can use general forms of pattern matching in our examples. Note that we have introduced patterns following the grammar:

$$\mathtt{p} ::= \mathtt{x} \mid \mathtt{c}(\mathtt{x}, \ldots, \mathtt{x})$$

So, in particular we do not have patterns for nested constructors. However, more complex pattern matching can be easily simulated through the use of combined `Case` constructions. As an example consider a function `f` that we want to define by pattern matching on expressions of the shape $(\mathtt{x}+1)+1$. This can be defined as follow:

$$\mathtt{f}\ \mathtt{y}_1 = \mathtt{Case}\ \mathtt{y}_1\ \mathtt{of}\ \mathtt{y}_2 + 1 \to \mathtt{Case}\ \mathtt{y}_2\ \mathtt{of}\ \ \mathtt{x} + 1 \to \mathtt{e}$$

Instead of writing this in full form, we will simply write it as:

$$\mathtt{f}\ ((\mathtt{x}+1)+1) \doteq \mathtt{e}$$

More generally, we will use the notation:

$$\begin{array}{l} \mathtt{f}\ \mathtt{p}_1^1\ \cdots \mathtt{p}_n^1 \doteq \mathtt{e}_1 \\ \vdots \\ \mathtt{f}\ \mathtt{p}_1^k\ \cdots \mathtt{p}_n^k \doteq \mathtt{e}_k \end{array}$$

as syntactic sugar for a function definition of the shape:

$$\begin{array}{l} \mathtt{f}(\mathtt{x}_1, \cdots, \mathtt{x}_n) \doteq \mathtt{Case}\ \mathtt{x}_1\ \mathtt{of}\ \mathtt{p}_1^1 \to \ldots \mathtt{Case}\ \mathtt{x}_n\ \mathtt{of}\ \mathtt{p}_n^1 \to \mathtt{e}_1 \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathtt{p}_1^k \to \ldots \mathtt{Case}\ \mathtt{x}_n\ \mathtt{of}\ \mathtt{p}_n^k \to \mathtt{e}_k \end{array}$$

More complex examples consisting of function definitions with several distinct function symbols will be treated analogously by juxtaposition of their syntactic sugars. We

then assume these definitions to be bound by a LetRec for some particular expression under consideration. That is, we will usually consider an expression e in isolation but this has to be considered as a program of the shape:

$$\texttt{LetRec}\ \texttt{d}_1, \ldots, \texttt{d}_n\ \texttt{in}\ \texttt{e}$$

where $\texttt{d}_1, \ldots, \texttt{d}_n$ are all the function definitions for the function symbols in e.

Stream terminology. The program analysis methods we will present in the following sections are specific to the study of stream program properties. This means that we will pay particular attention to programs working on the type [A], the type of both finite and infinite lists of type A.

We distinguish two classes of functions symbols useful to work with streams. Following the terminology of [15], we have:

Definition 7.

- *A function symbol* f *is a* stream function *if* $\texttt{f} :: [\sigma_1] \to \cdots \to [\sigma_n] \to \overrightarrow{\tau} \to [\sigma]$, *with* $n > 0$. *Conversely, a function symbol* f *is a* stream constructor *if* $\texttt{f} :: \overrightarrow{\tau} \to [\sigma]$.
- *A function definition* $\texttt{d}_\texttt{f}$ *such that* $\texttt{f}(\texttt{x}_1, \ldots, \texttt{x}_n) = \texttt{e}$ *is a* stream function definition *if* f *is a stream function. Conversely, if* f *is a stream constructor we say that* $\texttt{d}_\texttt{f}$ *is a* stream definition.

Intuitively, we call stream functions those functions that transform and combine input streams to produce an output stream. Analogously, we call stream constructors those functions that can be used to actually produce new output streams from scratch.

Example 1. *Consider the following definitions:*

$$\begin{array}{ll}
\texttt{odd} :: [a] \to [a] & \texttt{nats} :: \texttt{Nat} \to [\texttt{Nat}] \\
\texttt{odd}\ (\texttt{x} : \texttt{y} : \texttt{xs}) \doteq \texttt{x} : (\texttt{odd}\ \texttt{xs}) & \texttt{nats}\ \texttt{x} \doteq \texttt{x} : (\texttt{nats}\ (\texttt{x} + 1)) \\
\\
\texttt{zip} :: [a] \to [a] \to [a] & \texttt{nodd} :: [\texttt{Nat}] \\
\texttt{zip}\ (\texttt{x} : \texttt{xs})\ \texttt{ys} \doteq \texttt{x} : (\texttt{zip}\ \texttt{ys}\ \texttt{xs}) & \texttt{nodd} \doteq \texttt{odd}\ (\texttt{nats}\ (0 + 1))
\end{array}$$

We have that odd *and* zip *are two stream functions of one and two arguments respectively, while both* nats *and* nodd *are stream constructors. Note that the fact of being a stream constructor does not impose limitations on the kind of functions that can be used in the right-hand side of the definition. Indeed, in the* nodd *example, we use both a stream function (i.e* odd*) and a stream constructor (i.e.* nats*).*

2.2. Lazy operational semantics

In this section, we describe the SFL operational semantics. As outlined above, the operational semantics can be described by means of a *lazy* big-step semantics. With the term *lazy*, in the tradition of [33, 1], we identify a semantics that does not evaluate

$$\frac{\mathtt{c} \in \mathcal{C}}{\mathcal{H}; \mathtt{c}(\mathtt{e}_1, \cdots, \mathtt{e}_n) \Downarrow \mathtt{c}(\mathtt{e}_1, \cdots, \mathtt{e}_n)}\ (\text{val}) \qquad \frac{\mathcal{H} \cup \{\mathtt{d}\}; \mathtt{M} \Downarrow \mathtt{v}}{\mathcal{H}; \mathtt{LetRec\ d\ in\ M} \Downarrow \mathtt{v}}\ (\text{rec})$$

$$\frac{\mathcal{H}; \mathtt{e}\{\mathtt{e}_1/\mathtt{x}_1, \cdots, \mathtt{e}_n/\mathtt{x}_n\} \Downarrow \mathtt{v} \quad (\mathtt{f}\ \mathtt{x}_1\ \cdots\ \mathtt{x}_n \doteq \mathtt{e}) \in \mathcal{H}}{\mathcal{H}; \mathtt{f}(\mathtt{e}_1, \cdots, \mathtt{e}_n)\ \Downarrow \mathtt{v}}\ (\text{fun})$$

$$\frac{\mathcal{H}; \mathtt{e} \Downarrow \mathtt{c}(\mathtt{e}'_1, \cdots, \mathtt{e}'_m) \quad \mathtt{p}_i = \mathtt{c}(\mathtt{x}_1, \cdots, \mathtt{x}_m) \quad \mathcal{H}; \mathtt{e}_i\{\mathtt{e}'_1/\mathtt{x}_1, \cdots, \mathtt{e}'_m/\mathtt{x}_m\} \Downarrow \mathtt{v}}{\mathcal{H}; \mathtt{Case\ e\ of\ p}_1 \to \mathtt{e}_1, \ldots, \mathtt{p}_n \to \mathtt{e}_n \Downarrow \mathtt{v}}\ (\text{pm})$$

$$\frac{\mathcal{H}; \mathtt{e} \Downarrow \mathtt{c}(\mathtt{e}'_1, \cdots, \mathtt{e}'_m) \quad \forall i \leq n, \mathtt{p}_i \neq \mathtt{c}(\mathtt{x}_1, \cdots, \mathtt{x}_m)}{\mathcal{H}; \mathtt{Case\ e\ of\ p}_1 \to \mathtt{e}_1, \ldots, \mathtt{p}_n \to \mathtt{e}_n \Downarrow \mathtt{Err}}\ (\text{pm}_e)$$

Table 2: SFL lazy operational semantics

under the constructors. So in particular, we do not consider the sharing issue that is studied in other lazy and call-by-need semantics [27, 3].

In order to describe the semantics, we need two additional components: *substitutions* and *environments*. A *substitution* $\{\mathtt{e}_1/\mathtt{x}_1, \ldots, \mathtt{e}_n/\mathtt{x}_n\}$ is a partial function mapping variables to expressions. As usual we denote $\mathtt{e}\{\mathtt{e}_1/\mathtt{x}_1, \ldots, \mathtt{e}_n/\mathtt{x}_n\}$ the result of the application of the substitution $\{\mathtt{e}_1/\mathtt{x}_1, \ldots, \mathtt{e}_n/\mathtt{x}_n\}$ to the free variables of $\mathtt{e}$. An *environment* is simply a set of well-formed function definitions. We will use the letter $\mathcal{H}$ to denote environments.

Definition 8. *The* operational evaluation relation $\Downarrow$ *is the relation between environments, expressions and lazy values inductively defined by the rules in Table 2.*

Intuitively, the judgment $\mathcal{H}; \mathtt{e} \Downarrow \mathtt{v}$ means that the expression $\mathtt{e}$ can be evaluated to the lazy value $\mathtt{v}$ using the rules of the semantics and the function definitions contained in the environment $\mathcal{H}$. For notational convenience, we simply write $\mathtt{e} \Downarrow \mathtt{v}$ for $\mathcal{H}; \mathtt{e} \Downarrow \mathtt{v}$ when $\mathcal{H} = \emptyset$. Moreover, in the sequel when we write $\mathcal{H}; \mathtt{e} \Downarrow \mathtt{v}$ we implicitly assume that $\mathcal{H}$ contains the function definitions for all the function symbols in $\mathtt{e}$.

It is worth noticing that as usual in lazy semantics the abstract machine does not explore the entire result but stops once the requested information is found; this is why the axiom rule (val) only refers to lazy values. Moreover, as anticipated in the previous subsection, we use the constructor `Err` to deal with pattern matching errors. This should not be confused with the errors that can be generated by programs that go wrong. Indeed, to prevent such situations types are sufficient, as usual.

Strict evaluation. We have introduced the operational semantics of our language SFL in the previous paragraph. We have defined it to be lazy since our main concern is to deal with infinite computations in a natural way. However, another concern of our work is to describe program analysis techniques using only finitary operational tools without making reference to infinite abstract domains. For this reason, sometimes we will need to consider the complete evaluation of values. This is why we have introduced the category of *strict values* in the grammar definition in Definition 1.

In order to evaluate programs to strict values, we can define a particular function $\texttt{eval}_{\texttt{A}}$ for every base type $\texttt{A}$ as follows:

$$\begin{array}{l} \texttt{eval}_{\texttt{A}} :: \texttt{A} \to \texttt{A} \\ \texttt{eval}_{\texttt{A}} \quad (\texttt{c}\ \texttt{x}_1\ \cdots\ \texttt{x}_n) \quad \doteq \quad \hat{\texttt{C}}\ (\texttt{eval}_{\texttt{A}_1}\ \texttt{x}_1)\ \cdots\ (\texttt{eval}_{\texttt{A}_n}\ \texttt{x}_n) \end{array}$$

where $\hat{\texttt{C}}$ is a function symbol representing the *strict* version of the primitive constructor $\texttt{c}$. For instance in the case where $\texttt{c}$ is $+1$ we can define $\hat{\texttt{C}}$ to be the function $\texttt{succ} :: \texttt{Nat} \to \texttt{Nat}$ defined as:

$$\begin{array}{lrcl} \texttt{succ} & \texttt{Err} & \doteq & \texttt{Err} + 1 \\ \texttt{succ} & \texttt{0} & \doteq & \texttt{0} + 1 \\ \texttt{succ} & (\texttt{x} + 1) & \doteq & (\texttt{x} + 1) + 1 \end{array}$$

When we want to stress that an expression $\texttt{e}$ is completely evaluated (i.e. that besides being an expression, it is also a strict value) we use the notation $\underline{\texttt{e}}$. A relevant set of completely evaluated expressions is the set $\texttt{N}$ of *canonical numerals* defined as:

$$\texttt{N} = \{\underline{\texttt{n}} \mid \underline{\texttt{n}} = \underbrace{(\cdots(0+1)\cdots)+1)}_{\texttt{n times}} \text{ and } \underline{\texttt{n}} :: \texttt{Nat}\}$$

A concrete example of computation by strict evaluation can be found in Appendix B.

More notations. For notational convenience, in the sequel we will use the notation:

$$\mathcal{H}, \texttt{e} \Downarrow_v \underline{\texttt{v}}$$

to denote the judgment:

$$\mathcal{H}, \texttt{eval}_{\texttt{A}}\ \texttt{e} \Downarrow \underline{\texttt{v}}$$

assuming that the type $\texttt{A}$ is made clear by the context. Moreover, we introduce some notation for some well-established functions that we will use in the following sections.

We use the notation $\texttt{e}_{\underline{\texttt{n}}}$ as a shorthand for the expression $\texttt{e}\ !!\ \underline{\texttt{n}}$ where $!!$ is the usual indexing function returning the n-th element of a list. That is:

$$\begin{array}{llll} !! :: [a] \to \texttt{Nat} \to a & & & \\ \texttt{nil} \quad !! \quad \texttt{y} & \doteq & \texttt{Err} & \\ (\texttt{x} : \texttt{xs}) \quad !! \quad 0 & \doteq & \texttt{x} & \\ (\texttt{x} : \texttt{xs}) \quad !! \quad (\texttt{y} + 1) & \doteq & \texttt{xs}\ !!\ \texttt{y} & \end{array}$$

We use the shorthand $\texttt{e}\lceil_{\underline{\texttt{n}}}$ to denote the expression $\texttt{take}\ \underline{\texttt{n}}\ \texttt{e}$ where $\texttt{take}$ is the usual function which returns the first n elements of a list:

$$\begin{array}{llll} \texttt{take} :: \texttt{Nat} \to [a] \to [a] & & & \\ \texttt{take} \quad 0 \quad \texttt{s} & \doteq & \texttt{nil} & \\ \texttt{take} \quad (\texttt{x} + 1) \quad \texttt{nil} & \doteq & \texttt{Err} & \\ \texttt{take} \quad (\texttt{x} + 1) \quad (\texttt{y} : \texttt{ys}) & \doteq & \texttt{y} : (\texttt{take}\ \texttt{x}\ \texttt{ys}) & \end{array}$$

Finally, we use $\texttt{lg}$ to denote the function that returns the number of elements in a finite partial list:

$$
\begin{array}{lrl}
\mathtt{lg} :: [a] \to \mathtt{Nat} & & \\
\mathtt{lg} \quad \mathtt{nil} & \doteq & \underline{0} \\
\mathtt{lg} \quad \mathtt{Err} & \doteq & \underline{0} \\
\mathtt{lg} \quad (\mathtt{x} : \mathtt{xs}) & \doteq & (\mathtt{lg}\ \mathtt{xs}) + 1
\end{array}
$$

In the sequel, we tacitly assume that the above definitions are contained in all the environments $\mathcal{H}$ that we will consider.

3. Interpretation

The program analyses that we will introduce in Sections 4 and 5 will be based on the notion of *interpretation*. Intuitively, an interpretation consists of an assignment mapping each symbol of a program to a function over non-negative real numbers. Thanks to the real numbers ordering, such a peculiar assignment combined with some additional criteria permits to prove program properties.

This kind of reasoning is inspired by the notion of *polynomial interpretation* [31, 26, 6], developed in the field of program termination, and by the notions of *quasi-interpretation* [8] and *sup-interpretation* [32], developed more recently in the field of implicit computational complexity.

We now stress the main distinctions between the notion of interpretation presented in this section and the standard notion of interpretations on Term Rewrite Systems (see the survey [7]). In Subsections 3.1 and 3.2, we define the notions of *assignment* and *interpretation*. These definitions are similar to the one on TRS ([7]). The only distinction is that these notions are adapted to the presented functional language (the case construct is treated). The notions of *additive* and *monotonic assignments* are also standard. The only new notion is the notion of *almost-additive* (see Definition 3) allowing to deal with stream construct in a more flexible manner. All the results relating the size of a value and its interpretation (e.g. Corollary 1) or the interpretations of a term and its evaluation (e.g. Proposition 1) are fairly standard so an expert reader may go directly to Subsection 3.3 where a new notion of *parametrized interpretation* is defined. This notion will be useful for the *Local Upper Bound* (LUB) criterion.

3.1. Assignment

In the following, an *assignment* is used as a method to map in a canonical way programs to non-negative real numbers (i.e. elements of $\mathbb{R}^+$) in such a way that a comparison of programs is possible thanks to the usual ordering on real numbers. In order to do this, an assignment maps program components either to non-negative real numbers or to functions over non-negative real numbers.

Definition 9 (Assignment)**.**

- *A* variable assignment*, denoted ρ is a map associating to each $\mathtt{x} \in \mathcal{X}$ a value r in $\mathbb{R}^+$.*
- *A* symbol assignment*, denoted ξ is a map associating to each symbol $\mathtt{t} \in \mathcal{C} \cup \mathcal{F}$ a function $F : \mathbb{R}^+ \times \ldots \times \mathbb{R}^+ \to \mathbb{R}^+$ of the same arity.*

- *Given a variable assignment ρ and a symbol assignment ξ, an* assignment *is the extension of ρ and ξ to expressions defined as follows:*
  - $(\!|\texttt{Err}|\!)_{\rho,\xi} = 0$
  - $(\!|\texttt{x}|\!)_{\rho,\xi} = \rho(x)$
  - $(\!|\texttt{t}(\texttt{e}_1, \ldots, \texttt{e}_n)|\!)_{\rho,\xi} = \xi(\texttt{t})((\!|\texttt{e}_1|\!)_{\rho,\xi}, \ldots, (\!|\texttt{e}_n|\!)_{\rho,\xi})$
  - $(\!|\texttt{LetRec d in e}|\!)_{\rho,\xi} = (\!|\texttt{e}|\!)_{\rho,\xi}$
  - $(\!|\texttt{Case e of } \texttt{c}_1(\vec{\texttt{x}}_1) \to \texttt{e}_1, \ldots, \texttt{c}_m(\vec{\texttt{x}}_m) \to \texttt{e}_m|\!)_{\rho,\xi}$
    $= \max_{1 \le i \le m} \{(\!|\texttt{e}_i|\!)_{\rho\{\vec{\texttt{x}}_i = \vec{r}_i\},\xi} \mid \vec{r}_i \in \mathbb{R}^+ \textit{ and } (\!|\texttt{e}|\!)_{\rho,\xi} \ge (\!|\texttt{c}_i(\vec{\texttt{x}}_i)|\!)_{\rho\{\vec{\texttt{x}}_i = \vec{r}_i\},\xi}\}$

We consider variable and symbol assignments as total functions over program variables and program symbols, respectively. We write $\vec{r} \in \mathbb{R}^+$ as a shorthand for $\forall r \in \vec{r}, r \in \mathbb{R}^+$, and we write $\rho\{\texttt{x} := r\}$ for the variable assignment defined as ρ except for the variable $\texttt{x}$ to which it assigns the value r. We often abbreviate $\rho\{\texttt{x}_1 := r_1\} \cdots \{\texttt{x}_n := r_n\}$ by $\rho\{\vec{\texttt{x}} = \vec{r}\}$ (as for instance in the definition above). Note that we consider the constructor `Err` differently from the other constructors. This because as we will see later we want that interpretations behave well with respect to pattern matching.

The definition of assignment for the `Case` construction requires the existence of a maximal element $(\!|\texttt{e}_i|\!)_{\rho\{\vec{\texttt{x}}_i = \vec{r}_i\},\xi}$ for $1 \le i \le m$ and for $\vec{r}$ ranging over values in $\mathbb{R}^+$. The existence of such an element (or equivalently a bound on the search space) is ensured by the side condition $(\!|\texttt{e}|\!)_{\rho,\xi} \ge (\!|\texttt{p}_i|\!)_{\rho\{\vec{\texttt{x}}_i = \vec{r}_i\},\xi}$ and by the fact that $\texttt{e}$ does not contain the variables $\vec{\texttt{x}}$.

Example 2. *Consider the following function definitions:*

$$\texttt{add} :: \texttt{Nat} \to \texttt{Nat} \to \texttt{Nat}$$
$$\texttt{add 0 y} \doteq \texttt{y}$$
$$\texttt{add (z} + 1\texttt{) y} \doteq (\texttt{add z y}) + 1$$

$$\texttt{sadd} :: [\texttt{Nat}] \to [\texttt{Nat}] \to [\texttt{Nat}]$$
$$\texttt{sadd (x}' : \texttt{xs) (y}' : \texttt{ys)} \doteq (\texttt{add x}' \texttt{ y}') : (\texttt{sadd xs ys})$$

Concretely, in SFL *they can be computed by the environment $\mathcal{H}$ including the following definitions:*

$$\texttt{add x y} \doteq \texttt{Case x of 0} \to \texttt{y},\ \texttt{z} + 1 \to (\texttt{add z y}) + 1$$
$$\texttt{sadd x y} \doteq \texttt{Case x of x}' : \texttt{xs} \to \texttt{Case y of y}' : \texttt{ys} \to (\texttt{add x}' \texttt{ y}') : (\texttt{sadd xs ys})$$

For each variable assignment $\rho = \{\texttt{x} := r, \texttt{y} := s\}$ and symbol assignment ξ such that $\xi(\texttt{0}) = 0$, $\xi(:)(X, Y) = X + Y + 1$, $\xi(+1)(X) = X + 1$ and $\xi(\texttt{add})(X, Y) = \xi(\texttt{sadd})(X, Y) = X + Y$, we compute the assignment of the expression `add x y` *as follows:*

$$\begin{aligned}(\!|\texttt{add x y}|\!)_{\rho,\xi} &= \xi(\texttt{add})((\!|\texttt{x}|\!)_{\rho,\xi}, (\!|\texttt{y}|\!)_{\rho,\xi}) \\ &= \rho(\texttt{x}) + \rho(\texttt{y}) \\ &= r + s\end{aligned}$$

We compute the assignment of $\texttt{Case x of 0} \to \texttt{y},\ \texttt{z} + 1 \to (\texttt{add z y}) + 1$ *in a similar way:*

$$
\begin{aligned}
&(\!|\texttt{Case x of 0} \to \texttt{y},\ \texttt{z} + 1 \to (\texttt{add z y}) + 1|\!)_{\rho,\xi} \\
&= \max(\max\{(\!|\texttt{y}|\!)_{\rho,\xi} \mid (\!|\texttt{x}|\!)_{\rho,\xi} \geq (\!|\texttt{0}|\!)_{\rho,\xi}\}, \\
&\qquad \max\{(\!|(\texttt{add z y}) + 1|\!)_{\rho\{\texttt{z}:=t\},\xi} \mid t \in \mathbb{R}^+ \textit{ and } (\!|\texttt{x}|\!)_{\rho,\xi} \geq (\!|\texttt{z} + 1|\!)_{\rho\{\texttt{z}:=t\},\xi}\}) \\
&= \max(\rho(\texttt{y}), \max\{\rho(\texttt{y}) + \rho\{\texttt{z} := t\}(\texttt{z}) + 1 \mid t \in \mathbb{R}^+ \textit{ and } \rho(\texttt{x}) \geq \rho\{\texttt{z} := t\}(\texttt{z}) + 1\}) \\
&= \max(s, \max_{r \geq t+1}\{s + t + 1\}) \quad \textit{as } \rho(\texttt{x}) = r \textit{ and } \rho(\texttt{y}) = s \\
&= \max(s, s + r) = r + s
\end{aligned}
$$

The usual notion of assignment used in the context of interpretations does not distinguish between variable and symbol assignments. In our context, we prefer to keep this distinction because it highlights the extension of assignments to the `Case` construction and because, as we will see later, an interpretation will fix only the symbol assignments.

The following property shows that assignments internalize the substitution mechanism.

Lemma 1 (Assignment Substitution)**.** *Given an assignment* $(\!|-|\!)_{\rho,\xi}$ *and an expression* $\Gamma, \texttt{x} :: \texttt{A}; \Delta \vdash \texttt{e} :: \texttt{B}$*, for every expression* $\Gamma; \Delta \vdash \texttt{e}' :: \texttt{A}$ *we have:*

$$(\!|\texttt{e}\{\texttt{e}'/\texttt{x}\}|\!)_{\rho,\xi} = (\!|\texttt{e}|\!)_{\rho\{\texttt{x}:=(\!|\texttt{e}'|\!)_{\rho,\xi}\},\xi}$$

Proof. By induction on the structure of `e`. In the case where `e` is the variable `x` then the conclusion follows immediately. The cases where `e` is either `Err` or a variable distinct from `x` are trivial. The case where `e` is a `LetRec` follows directly by induction hypothesis.

Consider now the case $\texttt{e} = \texttt{t}(\texttt{e}_1, \ldots, \texttt{e}_n)$, by definition we have $(\!|\texttt{e}\{\texttt{e}'/\texttt{x}\}|\!)_{\rho,\xi} = \xi(\texttt{t})((\!|\texttt{e}_1\{\texttt{e}'/\texttt{x}\}|\!)_{\rho,\xi}, \ldots, (\!|\texttt{e}_n\{\texttt{e}'/\texttt{x}\}|\!)_{\rho,\xi})$. By induction hypothesis, $(\!|\texttt{e}_i\{\texttt{e}'/\texttt{x}\}|\!)_{\rho,\xi} = (\!|\texttt{e}_i|\!)_{\rho\{\texttt{x}:=(\!|\texttt{e}'|\!)_{\rho,\xi}\},\xi}$ for each $1 \leq i \leq n$. So, we can conclude:

$$
\begin{aligned}
(\!|\texttt{e}\{\texttt{e}'/\texttt{x}\}|\!)_{\rho,\xi} &= \xi(\texttt{t})((\!|\texttt{e}_1\{\texttt{e}'/\texttt{x}\}|\!)_{\rho,\xi}, \ldots, (\!|\texttt{e}_n\{\texttt{e}'/\texttt{x}\}|\!)_{\rho,\xi}) = \\
&\qquad \xi(\texttt{t})((\!|\texttt{e}_1|\!)_{\rho\{\texttt{x}:=(\!|\texttt{e}'|\!)_{\rho,\xi}\},\xi}, \ldots, (\!|\texttt{e}_n|\!)_{\rho\{\texttt{x}:=(\!|\texttt{e}'|\!)_{\rho,\xi}\},\xi}) = (\!|\texttt{e}|\!)_{\rho\{\texttt{x}:=(\!|\texttt{e}'|\!)_{\rho,\xi}\},\xi}
\end{aligned}
$$

Consider the case $\texttt{e} = \texttt{Case e}'' \texttt{ of } \texttt{c}_1(\vec{\texttt{y}}_1) \to \texttt{e}_1, \ldots, \texttt{c}_n(\vec{\texttt{y}}_n) \to \texttt{e}_n$, then by definition, for $\texttt{x} \notin \vec{\texttt{y}}_1, \ldots, \vec{\texttt{y}}_n$ (because `x` is supposed to be free in `e`), we have:

$$(\!|\texttt{e}\{\texttt{e}'/\texttt{x}\}|\!)_{\rho,\xi} = \max_{1 \leq i \leq m}\{(\!|\texttt{e}_i\{\texttt{e}'/\texttt{x}\}|\!)_{\rho\{\vec{\texttt{y}}_i:=\vec{r}_i\},\xi} \mid (\!|\texttt{e}''\{\texttt{e}'/\texttt{x}\}|\!)_{\rho,\xi} \geq (\!|\texttt{c}_i(\vec{\texttt{y}}_i)|\!)_{\rho\{\vec{\texttt{y}}_i:=\vec{r}_i\},\xi}\}$$

By induction hypothesis, we obtain:

$$
\begin{aligned}
&\{(\!|\texttt{e}_i\{\texttt{e}'/\texttt{x}\}|\!)_{\rho\{\vec{\texttt{y}}_i:=\vec{r}_i\},\xi} \mid (\!|\texttt{e}''\{\texttt{e}'/\texttt{x}\}|\!)_{\rho,\xi} \geq (\!|\texttt{c}_i(\vec{\texttt{y}}_i)\{\texttt{e}'/\texttt{x}\}|\!)_{\rho\{\vec{\texttt{y}}_i:=\vec{r}_i\},\xi}\} = \\
&\qquad \{(\!|\texttt{e}_i|\!)_{\rho\{\vec{\texttt{y}}:=\vec{r},\texttt{x}:=(\!|\texttt{e}'|\!)_{\rho,\xi}\},\xi} \mid (\!|\texttt{e}''|\!)_{\rho\{\texttt{x}:=(\!|\texttt{e}'|\!)_{\rho,\xi}\},\xi} \geq (\!|\texttt{c}_i(\vec{\texttt{y}}_i)|\!)_{\rho\{\vec{\texttt{y}}:=\vec{r},\texttt{x}:=(\!|\texttt{e}'|\!)_{\rho,\xi}\},\xi}\}
\end{aligned}
$$

and so we can conclude $(\!|\texttt{e}\{\texttt{e}'/\texttt{x}\}|\!)_{\rho,\xi} = (\!|\texttt{e}|\!)_{\rho\{\texttt{x}:=(\!|\texttt{e}'|\!)_{\rho,\xi}\},\xi}$. □

In this paper we will only deal with assignments that are *monotonic* where the monotonicity condition is defined as follows.

Definition 10 (Monotonic Assignment)**.**

- *A symbol assignment ξ is* monotonic *if for any $\mathtt{t} \in \mathcal{C} \cup \mathcal{F}$, $\xi(\mathtt{t})$ is a monotonic function, i.e. $\forall r, s \in \mathbb{R}^+$ s.t. $r \geq s$:*

$$\xi(\mathtt{t})(\ldots, r, \ldots) \geq \xi(\mathtt{t})(\ldots, s, \ldots)$$

- *An assignment $(\!|-|\!)_{\rho,\xi}$ is* monotonic *if the symbol assignment ξ is monotonic.*

Notice that the above definition of monotonicity concerns only the constants and the function symbols. This does not imply that all the functions used in an assignment are monotonic. In particular, the `Case` construction can be interpreted as a function $(\!|\mathtt{Case\ e\ of\ p}_1 \to \mathtt{e}_1, \ldots, \mathtt{p}_m \to \mathtt{e}_m|\!)_{\rho,\xi}$ that is monotonic in $(\!|\mathtt{e}|\!)_{\rho,\xi}$ and $(\!|\mathtt{e}_i|\!)_{\rho,\xi}$ but not in $(\!|\mathtt{p}_i|\!)_{\rho,\xi}$.

Other classes of assignments that will be useful in the sequel are the class of *almost-additive* and *additive* assignments.

Definition 11 (Almost-additive and Additive Assignment)**.**

- *The symbol assignment ξ is* almost-additive *if $\forall \mathtt{c} \in \mathcal{C}$ of arity n but the stream constructor* :*, we have:*

$$\xi(\mathtt{c})(r_1, \cdots, r_n) = \sum_{i=1}^{n} r_i + \alpha_{\mathtt{c}}, \text{ for some constant } \alpha_{\mathtt{c}} \geq 1, \text{ whenever } n > 0.$$

$$\xi(\mathtt{c}) = 0, \text{ otherwise.}$$

  The symbol assignment ξ is additive *if it is almost-additive and*

$$\xi(:)(r_1, r_2) = r_1 + r_2 + \alpha, \text{ for some constant } \alpha \geq 1$$

- *An assignment $(\!|-|\!)_{\rho,\xi}$ is an* almost-additive *(resp.* additive*) assignment if the symbol assignment ξ is almost-additive (resp. additive).*

The fact that an assignment is *additive* is useful in order to relate the interpretation of a strict value to its size. In particular, the following lemma shows that they are linearly related.

Lemma 2. *Given an additive assignment $(\!|-|\!)_{\rho,\xi}$, there is a constant α such that for each strict value $\vdash \underline{\mathtt{v}} :: \mathtt{A}$ we have:*

$$|\underline{\mathtt{v}}| \leq (\!|\underline{\mathtt{v}}|\!)_{\rho,\xi} \leq \alpha \times |\underline{\mathtt{v}}|$$

Proof. We consider $\alpha = \max_{\mathtt{c} \in \mathcal{C}} \alpha_{\mathtt{c}}$ and we prove the lemma by induction on the structure of $\underline{\mathtt{v}}$.

In the case $\underline{\mathtt{v}}$ is a constructor $\mathtt{c}$ of arity 0, by definition we have $|\mathtt{c}| = 0 = (\!|\mathtt{c}|\!)_{\rho,\xi}$, so the conclusion follows trivially.

Consider now the case $\underline{\mathtt{v}} = \mathtt{c}(\underline{\mathtt{v}}_1, \ldots, \underline{\mathtt{v}}_n)$. By induction hypothesis for $1 \leq i \leq n$ we have:

$$|\underline{\mathtt{v}}_i| \leq (\!|\underline{\mathtt{v}}_i|\!)_{\rho,\xi} \leq \alpha \times |\underline{\mathtt{v}}_i|$$

So, since by definition we also have:

$$|\underline{\mathtt{v}}| = (\sum_{1 \leq i \leq n} |\underline{\mathtt{v}}_i|) + 1 \quad \text{and} \quad (\!|\underline{\mathtt{v}}|\!)_{\rho,\xi} = (\sum_{1 \leq i \leq n} (\!|\underline{\mathtt{v}}_i|\!)_{\rho,\xi}) + \alpha_{\mathtt{c}}$$

and since $\alpha \geq \alpha_{\mathtt{c}} \geq 1$, using induction hypothesis we can conclude:

$$\sum_{1 \leq i \leq n} |\underline{\mathtt{v}}_i| + 1 \leq \sum_{1 \leq i \leq n} (\!|\underline{\mathtt{v}}_i|\!)_{\rho,\xi} + \alpha_{\mathtt{c}} \leq \sum_{1 \leq i \leq n} (\alpha \times |\underline{\mathtt{v}}_i|) + \alpha_{\mathtt{c}} \leq \alpha \times (\sum_{1 \leq i \leq n} |\underline{\mathtt{v}}_i| + 1)$$

□

A similar result can be obtained for almost-additive assignments if we restrict the attention to values that are not streams.

Corollary 1. *Given an assignment $(\!|-|\!)_{\rho,\xi}$ such that the symbol assignment ξ is almost-additive, there is a constant α such that for every strict value $\vdash \underline{\mathtt{v}} :: \sigma$ we have:*

$$|\underline{\mathtt{v}}| \leq (\!|\underline{\mathtt{v}}|\!)_{\rho,\xi} \leq \alpha \times |\underline{\mathtt{v}}|$$

3.2. Interpretations

Now, we are ready to define the main tool that will be used in the next sections: *interpretations*.

Definition 12 (Interpretation)**.** *An expression $\Gamma; \Delta \vdash \mathtt{e} :: \mathtt{A}$ admits an* interpretation *$(\!|-|\!)_\xi$ if for each variable assignment ρ, the assignment $(\!|-|\!)_{\rho,\xi}$ is monotonic and such that for each function definition $\mathtt{f}(\mathtt{x}_1, \ldots, \mathtt{x}_n) \doteq \mathtt{e}'$ the following holds:*

$$(\!|\mathtt{f}(\mathtt{x}_1, \ldots, \mathtt{x}_n)|\!)_{\rho,\xi} \geq (\!|\mathtt{e}'|\!)_{\rho,\xi}$$

The quantification on all variable assignments allows us to reason in general terms about values assigned to variables. For this reason, in the sequel we will usually write $X, Y, Z, \ldots$ to denote variables ranging over real numbers; e.g. we will write $(\!|\mathtt{f}|\!)_\xi(X_1, \ldots, X_n)$ for $(\!|\mathtt{f}(\mathtt{x}_1, \ldots, \mathtt{x}_n)|\!)_\xi$. Analogously, we will write $(\!|\mathtt{e}|\!)_\xi \geq (\!|\mathtt{e}'|\!)_\xi$ as a shorthands for: $\forall \rho, (\!|\mathtt{e}|\!)_{\rho,\xi} \geq (\!|\mathtt{e}'|\!)_{\rho,\xi}$.

In the sequel, we will need the following substitution property for interpretations.

Lemma 3 (Interpretation Substitution)**.** *Let $\Gamma; \Delta \vdash \mathtt{e} :: \mathtt{A}$ and $\Gamma, \mathtt{x} :: \mathtt{A}; \Delta \vdash \mathtt{e}_1, \mathtt{e}_2 :: \mathtt{B}$ be expressions admitting the interpretation $(\!|-|\!)_\xi$ and such that $(\!|\mathtt{e}_1|\!)_\xi \geq (\!|\mathtt{e}_2|\!)_\xi$. Then:*

$$(\!|\mathtt{e}_1\{\mathtt{e}/\mathtt{x}\}|\!)_\xi \geq (\!|\mathtt{e}_2\{\mathtt{e}/\mathtt{x}\}|\!)_\xi$$

Proof. By definition of interpretation we have $\forall \rho,\ (\!|\mathtt{e}_1|\!)_{\rho,\xi} \geq (\!|\mathtt{e}_2|\!)_{\rho,\xi}$. Thanks to the quantification over all the variable assignments we also have $\forall \rho,\ (\!|\mathtt{e}_1|\!)_{\rho\{\mathtt{x}:=(\!|\mathtt{e}|\!)_{\rho,\xi}\},\xi} \geq (\!|\mathtt{e}_2|\!)_{\rho\{\mathtt{x}:=(\!|\mathtt{e}|\!)_{\rho,\xi}\},\xi}$. So, by applying Lemma 1, we can conclude $\forall \rho,\ (\!|\mathtt{e}_1\{\mathtt{e}/\mathtt{x}\}|\!)_{\rho,\xi} \geq (\!|\mathtt{e}_2\{\mathtt{e}/\mathtt{x}\}|\!)_{\rho,\xi}$. □

The inequality conditions required by the definition of interpretation can be naturally inherited by the results of an evaluation. In order to show this we need to extend interpretations to environments.

Definition 13. *An environment $\mathcal{H}$ admits the interpretation $(\!|-|\!)_\xi$ if for every variable assignment ρ and every function definition $\mathtt{f}(\mathtt{x}_1, \ldots, \mathtt{x}_n) = \mathtt{e}$ in it, the following holds:*

$$(\!|\mathtt{f}(\mathtt{x}_1, \ldots, \mathtt{x}_n)|\!)_{\rho,\xi} \geq (\!|\mathtt{e}|\!)_{\rho,\xi}$$

We can now show some examples.

Example 3. *Consider again the environment $\mathcal{H}$ of Example 2. This environment admits an interpretation $(\!|-|\!)_\xi$ if the assignment $(\!|-|\!)_\xi$ satisfies the following inequalities:*

$$(\!|\mathtt{add\ x\ y}|\!)_\xi \geq (\!|\mathtt{Case\ x\ of\ 0} \to \mathtt{y},\ \mathtt{z}+1 \to (\mathtt{add\ z\ y})+1|\!)_\xi$$
$$(\!|\mathtt{sadd\ x\ y}|\!)_\xi \geq (\!|\mathtt{Case\ x\ of\ x'} : \mathtt{xs} \to \mathtt{Case\ y\ of\ y'} : \mathtt{ys} \to (\mathtt{add\ x'\ y'}) : (\mathtt{sadd\ xs\ ys})|\!)_\xi$$

Consider an additive assignment $(\!|-|\!)_\xi$ such that $(\!|\mathtt{0}|\!)_{\rho,\xi} = 0$, $(\!|:|\!)_{\rho,\xi}(X,Y) = X + Y + 1$ and $(\!|+1|\!)_{\rho,\xi}(X) = X + 1$. Now, we are interested in finding an interpretation for the symbols add *and* sadd *such that $\rho = \{\mathtt{x} := r, \mathtt{y} := s\}$ satisfies the following inequalities:*

$$(\!|\mathtt{add}|\!)_{\rho,\xi}(r,s) \geq \max(\max\{s \mid r \geq 0\}, \max\{1 + (\!|\mathtt{add}|\!)_{\rho,\xi}(t,s) \mid t \in \mathbb{R}^+ \text{ and } r \geq t+1\})$$
$$(\!|\mathtt{sadd}|\!)_{\rho,\xi}(r,s) \geq \max\{(\!|\mathtt{Case\ y\ of\ y'} : \mathtt{ys} \to (\mathtt{add\ x'\ y'}) : (\mathtt{sadd\ xs\ ys})|\!)_{\rho\{\mathtt{x'}:=v,\mathtt{xs}:=x\},\xi} \mid x, v \in \mathbb{R}^+ \text{ and } r \geq v + x + 1\}$$

The above can be reformulated as follows:

$$(\!|\mathtt{add}|\!)_{\rho,\xi}(r,s) \geq \max_{\{t \in \mathbb{R}^+ \mid r \geq t+1\}} (s, 1 + (\!|\mathtt{add}|\!)_{\rho,\xi}(t,s))$$
$$\begin{aligned}(\!|\mathtt{sadd}|\!)_{\rho,\xi}(r,s) &\geq \max\{(\!|(\mathtt{add\ x'\ y'}) : (\mathtt{sadd\ xs\ ys})|\!)_{\rho\{\mathtt{x'}:=v,\mathtt{xs}:=x,\mathtt{y'}:=w,\mathtt{ys}:=y\},\xi} \\ &\quad \mid x, y, v, w \in \mathbb{R}^+ \text{ and } r \geq v + x + 1 \text{ and } s \geq w + y + 1\} \\ &\geq \max\{(\!|\mathtt{add}|\!)_{\rho,\xi}(v,w) + (\!|\mathtt{sadd}|\!)_{\rho,\xi}(x,y) + 1 \\ &\quad \mid x, y, v, w \in \mathbb{R}^+ \text{ and } r \geq v + x + 1 \text{ and } s \geq w + y + 1\}\end{aligned}$$

So, for instance we can choose $\xi(\mathtt{add})(X,Y) = \xi(\mathtt{sadd})(X,Y) = X + Y$ (the first inequality is indeed proved to be an equality in Example 2). With this function symbol assignment ξ, since it is monotonic, we have that $(\!|-|\!)_\xi$ is an interpretation.

Example 4. *Consider the function symbol $\mathtt{eval}_\mathtt{A}$ for strict evaluation. The environment $\mathcal{H}$ that contains definitions of the shape:*

$$\mathtt{eval}_\mathtt{A}\ \mathtt{x} \doteq \mathtt{Case\ x\ of}\ (\mathtt{c}\ \mathtt{x}_1\ \cdots\ \mathtt{x}_n) \to \hat{\mathtt{C}}\ (\mathtt{eval}_{\mathtt{A}_1}\ \mathtt{x}_1)\ \cdots\ (\mathtt{eval}_{\mathtt{A}_n}\ \mathtt{x}_n)$$

will admit the interpretation $(\!|-|\!)_\xi$ defined by $\xi(\mathtt{c})(X_1, \cdots, X_n) = \xi(\hat{\mathtt{C}})(X_1, \cdots, X_n) = \sum_{i=1}^n X_i + \alpha_\mathtt{c}$ and $\forall \mathtt{A},\ \xi(\mathtt{eval}_\mathtt{A})(X) = X$.
Indeed, consider the assignment $(\!|-|\!)_{\rho,\xi}$ for the environment ρ such that $\rho(\mathtt{x}) = r$. On

the one hand, we have $(\!|\mathtt{eval}_{\mathtt{A}}(\mathtt{x})|\!)_{\rho,\xi} = \xi(\mathtt{eval}_{\mathtt{A}})((\!|\mathtt{x}|\!)_{\rho,\xi}) = (\!|\mathtt{x}|\!)_{\rho,\xi} = \rho(\mathtt{x}) = r$. *On the other hand, setting* $\vec{\mathtt{x}} = \mathtt{x}_1, \cdots, \mathtt{x}_n$ *and* $\vec{r} = r_1, \cdots, r_n$*:*

$$
\begin{aligned}
&(\!|\mathtt{Case\ x\ of\ c\ x}_1\ \cdots\ \mathtt{x}_n \to \hat{\mathtt{C}}\ (\mathtt{eval}_{\mathtt{A}_1}\ \mathtt{x}_1)\ \cdots\ (\mathtt{eval}_{\mathtt{A}_n}\ \mathtt{x}_n)|\!)_{\rho,\xi} \\
&= \max\{(\!|\hat{\mathtt{C}}(\mathtt{eval}_{\mathtt{A}_1}\ \mathtt{x}_1)\cdots(\mathtt{eval}_{\mathtt{A}_n}\ \mathtt{x}_n)|\!)_{\rho\{\vec{\mathtt{x}}=\vec{r}\},\xi} \mid (\!|\mathtt{x}|\!)_{\rho,\xi} \geq (\!|\mathtt{c\ x}_1\ \cdots\ \mathtt{x}_n|\!)_{\rho\{\vec{\mathtt{x}}=\vec{r}\},\xi}\} \\
&= \max\{\xi(\hat{\mathtt{C}})((\!|\mathtt{eval}_{\mathtt{A}_1}\ \mathtt{x}_1|\!)_{\rho\{\vec{\mathtt{x}}=\vec{r}\},\xi}, \cdots, (\!|\mathtt{eval}_{\mathtt{A}_n}\ \mathtt{x}_n|\!)_{\rho\{\vec{\mathtt{x}}=\vec{r}\},\xi}) \\
&\qquad\qquad \mid r \geq \xi(\mathtt{c})((\!|\mathtt{x}_1|\!)_{\rho\{\vec{\mathtt{x}}=\vec{r}\},\xi}, \cdots, (\!|\mathtt{x}_n|\!)_{\rho\{\vec{\mathtt{x}}=\vec{r}\},\xi})\} \\
&= \max\{\sum_{i=1}^{n} r_i + \alpha_{\hat{\mathtt{C}}} \mid r \geq \sum_{i=1}^{n} r_i + \alpha_{\hat{\mathtt{C}}}\} \leq r
\end{aligned}
$$

This inequality holds for an arbitrary value r *and, consequently, for every environment* ρ*. So,* $(\!|-|\!)_\xi$ *is an interpretation of* $\mathcal{H}$*.*

Throughout the paper, we will fix the assignment of the $\mathtt{eval}_{\mathtt{A}}$ function symbol by setting $\forall \mathtt{A},\ \xi(\mathtt{eval}_{\mathtt{A}})(X) = X$. As illustrated by the above example, such an assignment is a reasonable choice.

Now we can show that the result of an evaluation inherits the property of the interpretation: the interpretation of an expression is an upper bound on the interpretation of its computed value.

Proposition 1. *Let* $\mathcal{H}$ *and* $\emptyset; \Delta \vdash \mathtt{e} :: \mathtt{A}$ *be an environment and an expression admitting both the interpretation* $(\!|-|\!)_\xi$*. Then:*

$$\mathcal{H}, \mathtt{e} \Downarrow \mathtt{v} \quad \textit{implies} \quad (\!|\mathtt{e}|\!)_\xi \geq (\!|\mathtt{v}|\!)_\xi$$

Proof. By induction on the derivation proving $\mathcal{H}, \mathtt{e} \Downarrow \mathtt{v}$. The base case where the derivation consists only in an application of the rule (val) is trivial. The case the derivation ends with an application of the rule (rec) follows directly by induction hypothesis.

Let us consider the case where the derivation ends with:

$$\frac{\mathcal{H}; \mathtt{e}\{\mathtt{e}_1/\mathtt{x}_1, \cdots, \mathtt{e}_n/\mathtt{x}_n\} \Downarrow \mathtt{v} \quad (\mathtt{f\ x}_1\ \cdots\ \mathtt{x}_n = \mathtt{e}) \in \mathcal{H}}{\mathcal{H}; \mathtt{f}(\mathtt{e}_1, \cdots, \mathtt{e}_n)\ \Downarrow \mathtt{v}}$$

By induction hypothesis, we have $(\!|\mathtt{e}\{\mathtt{e}_1/\mathtt{x}_1, \cdots, \mathtt{e}_n/\mathtt{x}_n\}|\!)_\xi \geq (\!|\mathtt{v}|\!)_\xi$ and by assumption we have $(\!|\mathtt{f}(\mathtt{x}_1, \ldots, \mathtt{x}_n)|\!)_\xi \geq (\!|\mathtt{e}|\!)_\xi$. So, by several applications of Lemma 3 we obtain $(\!|\mathtt{f}(\mathtt{e}_1, \ldots, \mathtt{e}_n)|\!)_\xi \geq (\!|\mathtt{e}\{\mathtt{e}_1/\mathtt{x}_1, \cdots, \mathtt{e}_n/\mathtt{x}_n\}|\!)_\xi$ and by transitivity the conclusion follows.

Let us consider the case where the derivation ends with:

$$\frac{\mathcal{H}; \mathtt{e} \Downarrow \mathtt{c}(\mathtt{e}'_1, \cdots, \mathtt{e}'_m) \quad \mathtt{p}_i = \mathtt{c}(\mathtt{x}_1, \cdots, \mathtt{x}_m) \quad \mathcal{H}; \mathtt{e}_i\{\mathtt{e}'_1/\mathtt{x}_1, \cdots, \mathtt{e}'_m/\mathtt{x}_m\} \Downarrow \mathtt{v}}{\mathcal{H}; \mathtt{Case\ e\ of\ p}_1 \to \mathtt{e}_1, \ldots, \mathtt{p}_n \to \mathtt{e}_n \Downarrow \mathtt{v}}$$

By induction hypothesis, we have both $(\!|\mathtt{e}_i\{\mathtt{e}'_1/\mathtt{x}_1, \cdots, \mathtt{e}'_m/\mathtt{x}_m\}|\!)_\xi \geq (\!|\mathtt{v}|\!)_\xi$ and $(\!|\mathtt{e}|\!)_\xi \geq (\!|\mathtt{c}(\mathtt{e}'_1, \cdots, \mathtt{e}'_m)|\!)_\xi$. Consider an arbitrary variable assignment ρ. Applying Lemma 1 several times, we have $(\!|\mathtt{c}(\mathtt{e}'_1, \cdots, \mathtt{e}'_m)|\!)_{\rho,\xi} = (\!|\mathtt{c}(\mathtt{x}_1, \cdots, \mathtt{x}_m)|\!)_{\rho',\xi}$ for $\rho' = \rho\{\mathtt{x}_1 := (\!|\mathtt{e}'_1|\!)_{\rho,\xi}, \cdots, \mathtt{x}_m := (\!|\mathtt{e}'_m|\!)_{\rho,\xi}\}$. By definition of assignment we have:

$$(\!|\mathtt{Case\ e\ of\ p}_1 \to \mathtt{e}_1, \ldots, \mathtt{p}_n \to \mathtt{e}_n|\!)_{\rho,\xi} \geq (\!|\mathtt{e}_i|\!)_{\rho',\xi}$$

and by some applications of Lemma 1:

$$(\!|\texttt{Case e of } \texttt{p}_1 \to \texttt{e}_1, \ldots, \texttt{p}_n \to \texttt{e}_n|\!)_{\rho,\xi} \geq (\!|\texttt{e}_i|\!)_{\rho',\xi} = (\!|\texttt{e}_i\{\texttt{e}'_1/\texttt{x}_1, \cdots, \texttt{e}'_m/\texttt{x}_m\}|\!)_{\rho,\xi}$$

Since this holds for every ρ, the conclusion easily follows by the definition of interpretation. □

The previous result can be easily extended to strict evaluation: the interpretation of an expression is an upper bound on the interpretation of its computed strict value.

Corollary 2. *Let $\mathcal{H}$ and $\emptyset; \Delta \vdash \texttt{e} :: \texttt{A}$ be an environment and an expression admitting both the interpretation $(\!|-|\!)_\xi$. Then:*

$$\mathcal{H}, \texttt{e} \Downarrow_v \underline{\texttt{v}} \quad \textit{implies} \quad (\!|\texttt{e}|\!)_\xi \geq (\!|\underline{\texttt{v}}|\!)_\xi$$

Proof. The notation $\mathcal{H}, \texttt{e} \Downarrow_v \underline{\texttt{v}}$ is just a shorthand for $\mathcal{H}, \texttt{eval}_\texttt{A}\ \texttt{e} \Downarrow \underline{\texttt{v}}$, so the conclusion follows directly using Proposition 1 and the fact that we consider assignments such that $\xi(\texttt{eval}_\texttt{A})(X) = X$. □

The last important property of interpretations that will be used in the sequel relates the size of an expression with its interpretation.

Lemma 4. *Let $(\!|-|\!)_\xi$ be an interpretation. Then, there exists a function $F : \mathbb{R}^+ \to \mathbb{R}^+$ such that for every program $\texttt{e} :: \texttt{A}$ admitting $(\!|-|\!)_\xi$:*

$$(\!|\texttt{e}|\!)_\xi \leq F(|\texttt{e}|)$$

Proof. By induction on the shape of e. □

The proof of Lemma 4 proceeds in essentially the same way as the one of the subsequent Lemma 6. So, for convenience we detail only the proof of the latter.

3.3. Parametrized Interpretations

For the analyses that we will present in the next section it is convenient to extend the notion of interpretations in a parametric way. This notion allows us to obtain more precise analyses on stream programs.

The idea behind a parametrized interpretation is that the interpretations now become of the shape $(\!|-|\!)^l_\xi$ where $l \in \mathbb{R}$ is a parameter that can be used to refer to a particular element of a stream. In order to obtain this, we need to parametrize all the previous definitions.

Definition 14 (Parametrized Assignment)**.**

- *A* parametrized symbol assignment*, denoted ξ^l is a map associating to each symbol $\texttt{t} \in \mathcal{C} \cup \mathcal{F}$ and $l \in \mathbb{R}$ a function $F_l : \mathbb{R}^+ \times \ldots \times \mathbb{R}^+ \to \mathbb{R}^+$ of the same arity.*
- *Given a variable assignment ρ, a parametrized symbol assignment ξ^l , a* parame-trized assignment *$(\!|-|\!)^l_\xi$ is the extension of ρ and ξ^l to expressions defined as follows:*

- $(\!|\texttt{Err}|\!)^l_{\rho,\xi} = 0$
- $(\!|\texttt{x}|\!)^l_{\rho,\xi} = \rho(x)$
- $(\!|\texttt{e}_1 : \texttt{e}_2|\!)^l_{\rho,\xi} = \xi^l(:)((\!|\texttt{e}_1|\!)^l_{\rho,\xi}, (\!|\texttt{e}_2|\!)^{l-1}_{\rho,\xi})$
- $(\!|\texttt{t}(\texttt{e}_1, \ldots, \texttt{e}_n)|\!)^l_{\rho,\xi} = \xi^l(\texttt{t})((\!|\texttt{e}_1|\!)^l_{\rho,\xi}, \ldots, (\!|\texttt{e}_n|\!)^l_{\rho,\xi}) \qquad \textit{for } \texttt{t} \neq :$
- $(\!|\texttt{LetRec d in e}|\!)^l_{\rho,\xi} = (\!|\texttt{e}|\!)^l_{\rho,\xi}$
- $(\!|\texttt{Case e of } \texttt{c}_1(\vec{\texttt{x}}_1) \to \texttt{e}_1, \ldots, \texttt{c}_m(\vec{\texttt{x}}_m) \to \texttt{e}_m|\!)^l_{\rho,\xi}$
  $= \max_{1 \leq i \leq m} \{(\!|\texttt{e}_i|\!)^l_{\rho\{\vec{\texttt{x}}_i = \vec{r}_i\},\xi} \mid \vec{r}_i \in \mathbb{R}^+ \textit{ and } (\!|\texttt{e}|\!)^l_{\rho,\xi} \geq (\!|\texttt{c}_i(\vec{\texttt{x}}_i)|\!)^l_{\rho\{\vec{\texttt{x}}_i = \vec{r}_i\},\xi}\}$

The definitions given for interpretations can be easily adapted to the case of parametrized interpretations.

Definition 15 (Monotonic Parametrized Assignment)**.** *A parametrized assignment is monotonic if for any* $\texttt{t} \in \mathcal{C} \cup \mathcal{F}$*,* $\xi(\texttt{t})$ *is a monotonic function, i.e.* $\forall r, s \in \mathbb{R}^+$ *s.t.* $r \geq s$ *and* $\forall l, l' \in \mathbb{R}$ *s.t.* $l \geq l'$*:*

$$\xi^l(\texttt{t})(\ldots, r, \ldots) \geq \xi^{l'}(\texttt{t})(\ldots, s, \ldots)$$

Definition 16 (Almost-additive and Additive Parametrized Assignment)**.**

- *The parametrized symbol assignment* ξ^l *is* almost-additive *if* $\forall \texttt{c} \in \mathcal{C}$ *of arity* n *but the stream constructor* $:$ *we have:*

  $$\xi^l(\texttt{c})(r_1, \cdots, r_n) = \sum_{i=1}^{n} r_i + \alpha_{\texttt{c}}, \textit{ for some constant } \alpha_{\texttt{c}} \geq 1, \textit{ whenever } n > 0.$$

  $$\xi^l(\texttt{c}) = 0, \textit{ otherwise.}$$

  The parametrized symbol assignment ξ^l *is* additive *if it is almost-additive and*

  $$\xi^l(:)(r_1, r_2) = r_1 + r_2 + \alpha, \textit{ for some constant } \alpha \geq 1$$

- *A parametrized assignment* $(\!|-|\!)^l_{\rho,\xi}$ *is an* almost-additive *(resp.* additive*) assignment if the parametrized symbol assignment* ξ^l *is almost-additive (resp. additive).*

Differently from what happens in the case of assignments, parametrized assignments do not internalize the substitution mechanism. However, for monotonic parametrized assignment we have the following important property.

Lemma 5 (Parametrized Assignment Substitution)**.** *Given a monotonic parametrized assignment* $(\!|-|\!)^l_{\rho,\xi}$ *and an expression* $\Gamma, \texttt{x} :: \texttt{A}; \Delta \vdash \texttt{e} :: \texttt{B}$*, for every expression* $\Gamma; \Delta \vdash \texttt{e}' :: \texttt{A}$ *and every* $l \in \mathbb{R}$ *we have:*

$$(\!|\texttt{e}|\!)^l_{\rho\{\texttt{x} := (\!|\texttt{e}'|\!)^l_{\rho,\xi}\},\xi} \geq (\!|\texttt{e}\{\texttt{e}'/\texttt{x}\}|\!)^l_{\rho,\xi}$$

Proof. By induction on the expression $\mathtt{e}$. We consider just the two most interesting cases.

Consider the case $\mathtt{e} = \mathtt{e}_1 : \mathtt{e}_2$. By definition we have:

$$(\!|\mathtt{e}_1 : \mathtt{e}_2|\!)^l_{\rho\{\mathtt{x}:=(\!|\mathtt{e}'|\!)^l_{\rho,\xi}\},\xi} = \xi^l(:)((\!|\mathtt{e}_1|\!)^l_{\rho\{\mathtt{x}:=(\!|\mathtt{e}'|\!)^l_{\rho,\xi}\},\xi}, (\!|\mathtt{e}_2|\!)^{l-1}_{\rho\{\mathtt{x}:=(\!|\mathtt{e}'|\!)^l_{\rho,\xi}\},\xi})$$

but by monotonicity we have:

$$\xi^l(:)((\!|\mathtt{e}_1|\!)^l_{\rho\{\mathtt{x}:=(\!|\mathtt{e}'|\!)^l_{\rho,\xi}\},\xi}, (\!|\mathtt{e}_2|\!)^{l-1}_{\rho\{\mathtt{x}:=(\!|\mathtt{e}'|\!)^l_{\rho,\xi}\},\xi}) \geq \xi^l(:)((\!|\mathtt{e}_1|\!)^l_{\rho\{\mathtt{x}:=(\!|\mathtt{e}'|\!)^l_{\rho,\xi}\},\xi}, (\!|\mathtt{e}_2|\!)^{l-1}_{\rho\{\mathtt{x}:=(\!|\mathtt{e}'|\!)^{l-1}_{\rho,\xi}\},\xi})$$

Since by induction hypothesis we have:

$$\xi^l(:)((\!|\mathtt{e}_1|\!)^l_{\rho\{\mathtt{x}:=(\!|\mathtt{e}'|\!)^l_{\rho,\xi}\},\xi}, (\!|\mathtt{e}_2|\!)^{l-1}_{\rho\{\mathtt{x}:=(\!|\mathtt{e}'|\!)^{l-1}_{\rho,\xi}\},\xi}) \geq \xi^l(:)((\!|\mathtt{e}_1\{\mathtt{e}'/\mathtt{x}\}|\!)^l_{\rho,\xi}, (\!|\mathtt{e}_2\{\mathtt{e}'/\mathtt{x}\}|\!)^{l-1}_{\rho,\xi})$$

and since by definition:

$$\xi^l(:)((\!|\mathtt{e}_1\{\mathtt{e}'/\mathtt{x}\}|\!)^l_{\rho,\xi}, (\!|\mathtt{e}_2\{\mathtt{e}'/\mathtt{x}\}|\!)^{l-1}_{\rho,\xi}) = (\!|(\mathtt{e}_1 : \mathtt{e}_2)\{\mathtt{e}'/\mathtt{x}\}|\!)^l_{\rho,\xi}$$

the conclusion follows.

Consider now the case $\mathtt{e} = \mathtt{Case}\ \mathtt{e}''\ \mathtt{of}\ \mathtt{c}_1(\vec{\mathtt{x}}_1) \to \mathtt{e}_1, \ldots, \mathtt{c}_m(\vec{\mathtt{x}}_m) \to \mathtt{e}_m$. Then, by definition we have:

$$(\!|\mathtt{e}|\!)^l_{\rho\{\mathtt{x}:=(\!|\mathtt{e}'|\!)^l_{\rho,\xi}\},\xi} = \max_{1\leq i\leq m}\{(\!|\mathtt{e}_i|\!)^l_{\rho\{\vec{\mathtt{x}}_i=\vec{r}_i\}\{\mathtt{x}:=(\!|\mathtt{e}'|\!)^l_{\rho,\xi}\},\xi} \mid (\!|\mathtt{e}''|\!)^l_{\rho\{\mathtt{x}:=(\!|\mathtt{e}'|\!)^l_{\rho,\xi}\},\xi} \geq (\!|\mathtt{c}_i(\vec{\mathtt{x}}_i)|\!)^l_{\rho\{\vec{\mathtt{x}}_i=\vec{r}_i\},\xi}\}$$

since clearly $\mathtt{x}$ is disjoint from the variables in $\vec{\mathtt{x}}_i$. By induction hypothesis we can show that:

$$\max_{1\leq i\leq m}\{(\!|\mathtt{e}_i|\!)^l_{\rho\{\vec{\mathtt{x}}_i=\vec{r}_i\}\{\mathtt{x}:=(\!|\mathtt{e}'|\!)^l_{\rho,\xi}\},\xi} \mid (\!|\mathtt{e}''|\!)^l_{\rho\{\mathtt{x}:=(\!|\mathtt{e}'|\!)^l_{\rho,\xi}\},\xi} \geq (\!|\mathtt{c}_i(\vec{\mathtt{x}}_i)|\!)^l_{\rho\{\vec{\mathtt{x}}_i=\vec{r}_i\},\xi}\}$$
$$\geq \max_{1\leq i\leq m}\{(\!|\mathtt{e}_i\{\mathtt{e}'/\mathtt{x}\}|\!)^l_{\rho\{\vec{\mathtt{x}}_i=\vec{r}_i\},\xi} \mid (\!|\mathtt{e}''\{\mathtt{e}'/\mathtt{x}\}|\!)^l_{\rho,\xi} \geq (\!|\mathtt{c}_i(\vec{\mathtt{x}}_i)|\!)^l_{\rho\{\vec{\mathtt{x}}_i=\vec{r}_i\},\xi}\}$$

and so the conclusion follows. The other cases can be obtained similarly. □

We are now ready to define parametrized interpretations.

Definition 17 (Parametrized Interpretation)**.** *An expression* $\Gamma;\Delta \vdash \mathtt{e} : \mathtt{A}$ *admits a* parametrized interpretation $(\!|-|\!)^l_\xi$ *if for each variable assignment* ρ*, the assignment* $(\!|-|\!)^l_{\rho,\xi}$ *is monotonic and such that for each function definition* $\mathtt{f}(\mathtt{x}_1, \ldots, \mathtt{x}_n) \doteq \mathtt{e}'$ *and for each* $l \in \mathbb{R}$ *the following holds:*

$$(\!|\mathtt{f}(\mathtt{x}_1, \ldots, \mathtt{x}_n)|\!)^l_{\rho,\xi} \geq (\!|\mathtt{e}'|\!)^l_{\rho,\xi}$$

Parametrized interpretations can be extended to environment as expected and thanks to this extension it is easy to verify that parametrized interpretations behave similarly to usual interpretations with respect to program evaluation. Analogously, we will write $(\!|\mathtt{e}|\!)^l_\xi \geq (\!|\mathtt{e}'|\!)^l_\xi$ as a shorthand for $\forall \rho,\ (\!|\mathtt{e}|\!)^l_{\rho,\xi} \geq (\!|\mathtt{e}'|\!)^l_{\rho,\xi}$.

Similarly to the case of interpretation, we want to relate parametrized interpretations to the evaluation of programs. However, in order to do this we need to introduce a new evaluation relation counting the number of pattern matchings on stream data. Let $\mathcal{H}, \mathtt{e} \Downarrow^k \mathtt{v}$ be the relation defined in Figure 3. $\mathcal{H}, \mathtt{e} \Downarrow^k \mathtt{v}$ means that $\mathcal{H}, \mathtt{e} \Downarrow \mathtt{v}$ holds using exactly k pattern matching rules on streams for producing $\mathtt{v}$. We define $\Downarrow^k_v$ in the same manner: $\mathcal{H}, \mathtt{e} \Downarrow^k_v \mathtt{v}$ if $\mathcal{H}, \mathtt{eval}_\mathtt{A}\ \mathtt{e} \Downarrow^k \mathtt{v}$.

$$\frac{\mathtt{c} \in \mathcal{C}}{\mathcal{H}; \mathtt{c}(\mathtt{e}_1, \cdots, \mathtt{e}_n) \Downarrow^0 \mathtt{c}(\mathtt{e}_1, \cdots, \mathtt{e}_n)} \qquad \frac{\mathcal{H} \cup \{\mathtt{d}\}; \mathtt{M} \Downarrow^k \mathtt{v}}{\mathcal{H}; \mathtt{LetRec\ d\ in\ M} \Downarrow^k \mathtt{v}}$$

$$\frac{\mathcal{H}; \mathtt{e}\{\mathtt{e}_1/\mathtt{x}_1, \cdots, \mathtt{e}_n/\mathtt{x}_n\} \Downarrow^k \mathtt{v} \quad (\mathtt{f}\ \mathtt{x}_1\ \cdots\ \mathtt{x}_n \doteq \mathtt{e}) \in \mathcal{H}}{\mathcal{H}; \mathtt{f}(\mathtt{e}_1, \cdots, \mathtt{e}_n)\ \Downarrow^k \mathtt{v}}$$

$$\frac{\mathcal{H}; \mathtt{e} \Downarrow^k \mathtt{c}(\mathtt{e}'_1, \cdots, \mathtt{e}'_m) \quad \mathtt{p}_i = \mathtt{c}(\mathtt{x}_1, \cdots, \mathtt{x}_m) \quad \mathcal{H}; \mathtt{e}_i\{\mathtt{e}'_1/\mathtt{x}_1, \cdots, \mathtt{e}'_m/\mathtt{x}_m\} \Downarrow^{k'} \mathtt{v}}{\mathcal{H}; \mathtt{Case\ e\ of\ p}_1 \to \mathtt{e}_1, \ldots, \mathtt{p}_n \to \mathtt{e}_n \Downarrow^{k+k'} \mathtt{v}}$$

$$\frac{\mathcal{H}; \mathtt{e} \Downarrow^k \mathtt{e}'_1 : \mathtt{e}'_2 \qquad \mathcal{H}; \mathtt{e}_i\{\mathtt{e}'_1/\mathtt{x}_1, \mathtt{e}'_2/\mathtt{x}_2\} \Downarrow^{k'} \mathtt{v}}{\mathcal{H}; \mathtt{Case\ e\ of\ x}_1 : \mathtt{x}_2 \to \mathtt{e}_1, \mathtt{nil} \to \mathtt{e}_2 \Downarrow^{k+k'+1} \mathtt{v}}$$

$$\frac{\mathcal{H}; \mathtt{e} \Downarrow^k \mathtt{c}(\mathtt{e}_1, \cdots, \mathtt{e}_m) \quad \forall i \leq n,\ \mathtt{p}_i \neq \mathtt{c}(\mathtt{x}_1, \cdots, \mathtt{x}_m)}{\mathcal{H}; \mathtt{Case\ e\ of\ p}_1 \to \mathtt{e}_1, \ldots, \mathtt{p}_n \to \mathtt{e}_n \Downarrow^k \mathtt{Err}}$$

Table 3: Counting stream pattern matchings

Proposition 2. *Let $\mathcal{H}$ and $\emptyset; \Delta \vdash \mathtt{e} :: \mathtt{A}$ be an environment and an expression admitting both the parametrized interpretation $(\!|-|\!)^l_\xi$. Then:*

$$\mathcal{H}, \mathtt{e} \Downarrow^k \mathtt{v} \quad \textit{implies} \quad \forall l \in \mathbb{R},\ (\!|\mathtt{e}|\!)^l_\xi \geq (\!|\mathtt{v}|\!)^{l-k}_\xi$$

Proof. The proof is analogous to the proof of Proposition 1. The only interesting case is the one where the derivation ends with:

$$\frac{\mathcal{H}; \mathtt{e} \Downarrow^k \mathtt{e}'_1 : \mathtt{e}'_2 \qquad \mathcal{H}; \mathtt{e}_1\{\mathtt{e}'_1/\mathtt{x}_1, \mathtt{e}'_2/\mathtt{x}_2\} \Downarrow^{k'} \mathtt{v}}{\mathcal{H}; \mathtt{Case\ e\ of\ x}_1 : \mathtt{x}_2 \to \mathtt{e}_1, \mathtt{nil} \to \mathtt{e}_2 \Downarrow^{k+k'+1} \mathtt{v}}$$

By induction hypothesis, we have both $(\!|\mathtt{e}_1\{\mathtt{e}'_1/\mathtt{x}_1, \mathtt{e}'_2/\mathtt{x}_2\}|\!)^{l-(k+1)}_\xi \geq (\!|\mathtt{v}|\!)^{l-(k+1)-k'}_\xi$

and $(\!|\mathtt{e}|\!)^l_\xi \geq (\!|\mathtt{e}'_1 : \mathtt{e}'_2|\!)^{l-k}_\xi$. By definition we have:

$$(\!|\mathtt{Case\ e\ of\ x_1 : x_2 \to e_1, nil \to e_2}|\!)^l_{\rho,\xi} \geq \max\{(\!|\mathtt{e}_1|\!)^l_{\rho\{\mathtt{x}_1=r_1,\mathtt{x}_2=r_2\},\xi} \mid r_1, r_2 \in \mathbb{R}^+ \text{ and } (\!|\mathtt{e}|\!)^l_{\rho,\xi} \geq (\!|\mathtt{x}_1 : \mathtt{x}_2|\!)^l_{\rho\{\mathtt{x}_1=r_1,\mathtt{x}_2=r_2\},\xi}\}$$

By induction hypothesis, we can satisfy the side condition in the $\max$ by setting $r_1 = (\!|\mathtt{e}'_1|\!)^{l-k}_{\rho,\xi}$ and $r_2 = (\!|\mathtt{e}'_2|\!)^{l-k-1}_{\rho,\xi}$ since by definition of parametrized interpretation $(\!|\mathtt{e}|\!)^l_\xi \geq (\!|\mathtt{e}'_1 : \mathtt{e}'_2|\!)^{l-k}_\xi = \xi^l(:)((\!|\mathtt{e}'_1|\!)^{l-k}_{\rho,\xi}, (\!|\mathtt{e}'_2|\!)^{l-k-1}_{\rho,\xi})$. So we have:

$$\max\{(\!|\mathtt{e}_1|\!)^l_{\rho\{\mathtt{x}_1=r_1,\mathtt{x}_2=r_2\},\xi} \mid (\!|\mathtt{e}|\!)^l_{\rho,\xi} \geq (\!|\mathtt{x}_1 : \mathtt{x}_2|\!)^l_{\rho\{\mathtt{x}_1=r_1,\mathtt{x}_2=r_2\},\xi}\} \geq (\!|\mathtt{e}_1|\!)^l_{\rho\{\mathtt{x}_1=(\!|\mathtt{e}'_1|\!)^{l-k}_{\rho,\xi},\mathtt{x}_2=(\!|\mathtt{e}'_2|\!)^{l-k-1}_{\rho,\xi}\},\xi}$$

By monotonicity we have:

$$(\!|\mathtt{e}_1|\!)^l_{\rho\{\mathtt{x}_1=(\!|\mathtt{e}'_1|\!)^{l-k}_{\rho,\xi},\mathtt{x}_2=(\!|\mathtt{e}'_2|\!)^{l-k-1}_{\rho,\xi}\},\xi} \geq (\!|\mathtt{e}_1|\!)^{l-k-1}_{\rho\{\mathtt{x}_1=(\!|\mathtt{e}'_1|\!)^{l-k-1}_{\rho,\xi},\mathtt{x}_2=(\!|\mathtt{e}'_2|\!)^{l-k-1}_{\rho,\xi}\},\xi}$$

Finally, by applying Lemma 5 twice we have:

$$(\!|\mathtt{e}_1|\!)^{l-k-1}_{\rho\{\mathtt{x}_1=(\!|\mathtt{e}'_1|\!)^{l-k-1}_{\rho,\xi},\mathtt{x}_2=(\!|\mathtt{e}'_2|\!)^{l-k-1}_{\rho,\xi}\},\xi} \geq (\!|\mathtt{e}_1\{\mathtt{e}'_1/\mathtt{x}_1, \mathtt{e}'_2/\mathtt{x}_2\}|\!)^{l-(k+1)}_{\rho,\xi}$$

By induction hypothesis we have $(\!|\mathtt{e}_1\{\mathtt{e}'_1/\mathtt{x}_1, \mathtt{e}'_2/\mathtt{x}_2\}|\!)^{l-(k+1)}_\xi \geq (\!|\mathtt{v}|\!)^{l-(k+1)-k'}_\xi$, so the conclusion follows.

Since this holds for every ρ and every $l \in \mathbb{R}$, the conclusion easily follows by definition of parametrized interpretation. □

Corollary 3. *Let $\mathcal{H}$ and $\emptyset; \Delta \vdash \mathtt{e} :: \mathtt{A}$ be an environment and an expression both admitting the parametrized interpretation $(\!|-|\!)^l_\xi$. Then:*

$$\mathcal{H}, \mathtt{e} \Downarrow^k_v \underline{\mathtt{v}} \quad \textit{implies} \quad (\!|\mathtt{e}|\!)^l_\xi \geq (\!|\underline{\mathtt{v}}|\!)^{l-k}_\xi$$

Proof. Just check that we can define a parametrized interpretation of $\mathtt{eval_A}$ by setting $\forall l \in \mathbb{R},\ \forall \mathtt{A},\ \xi^l(\mathtt{eval_A})(X) = X$ as in Corollary 2. □

Lemma 6. *Let $(\!|-|\!)^l_\xi$ be a parametrized interpretation. Then, there exists a function $G : \mathbb{R}^+ \times \mathbb{R}^+ \to \mathbb{R}^+$ such that for every program $\mathtt{e} :: \mathtt{A}$ admitting $(\!|-|\!)^l_\xi$ and every $l \in \mathbb{R}^+$:*

$$(\!|\mathtt{e}|\!)^l_\xi \leq G(|\mathtt{e}|, l)$$

Proof. Define:

$$F(X, L) = \max(\max_{\mathtt{t} \in \mathcal{C} \cup \mathcal{F}} (\!|\mathtt{t}|\!)^L_\xi(X, \ldots, X), X)$$

and $F^{n+1}(X, L) = F(F^n(X, L), L)$ and $F^0(X, L) = F(X, L)$. It can be shown by induction on the structure of $\mathtt{e}$ that $(\!|\mathtt{e}|\!)^l_\xi \leq F^{|\mathtt{e}|}(|\mathtt{e}|, l)$. If $\mathtt{e}$ is a variable, a constructor or a function symbol of arity 0, then conclusion follows directly by definition of F, i.e $(\!|\mathtt{e}|\!)^l_\xi \leq F(|\mathtt{e}|, l)$. Now, consider $\mathtt{e} = \mathtt{t\ d_1} \cdots \mathtt{d}_n$ and suppose $|\mathtt{d}_j| = \max_{i=1}^n |\mathtt{d}_i|$. By

induction hypothesis, $(\!|\mathtt{d}_i|\!)^l_\xi \leq F^{|\mathtt{d}_i|}(|\mathtt{d}_i|, l)$. There are two possibilities depending on the shape of $\mathtt{t}$. If $\mathtt{t} \neq :$, that is $\mathtt{e} \neq \mathtt{e}_1 : \mathtt{e}_2$, then by induction hypothesis, definition and monotonicity of F we have:

$$\begin{aligned}(\!|\mathtt{e}|\!)^l_\xi &= \xi^l(\mathtt{t})((\!|\mathtt{d}_1|\!)^l_\xi, \ldots, (\!|\mathtt{d}_n|\!)^l_\xi) \leq \xi^l(\mathtt{t})(F^{|\mathtt{d}_1|}(|\mathtt{d}_1|, l), \ldots, F^{|\mathtt{d}_n|}(|\mathtt{d}_n|, l)) \\ &\leq \xi^l(\mathtt{t})(F^{|\mathtt{d}_j|}(|\mathtt{d}_j|, l), \ldots, F^{|\mathtt{d}_j|}(|\mathtt{d}_j|, l)) \leq F(F^{|\mathtt{d}_j|}(|\mathtt{d}_j|, l), l) \\ &\leq F^{|\mathtt{d}_j|+1}(|\mathtt{d}_j|, l) \leq F^{|\mathtt{e}|}(|\mathtt{e}|, l)\end{aligned}$$

In the case where $\mathtt{t} = :$, and so $\mathtt{e} = \mathtt{e}_1 : \mathtt{e}_2$, by definition of parametrized interpretation, induction hypothesis, definition and monotonicity of F we have:

$$\begin{aligned}(\!|\mathtt{e}|\!)^l_\xi = (\!|\mathtt{e}_1 : \mathtt{e}_2|\!)^l_\xi &= \xi^l(:)((\!|\mathtt{e}_1|\!)^l_\xi, (\!|\mathtt{e}_2|\!)^{l-1}_\xi) \leq \xi^l(:)(F^{|\mathtt{e}_1|}(|\mathtt{e}_1|, l), F^{|\mathtt{e}_2|}(|\mathtt{e}_2|, l-1)) \\ &\leq \xi^l(:)(F^{|\mathtt{e}_1|}(|\mathtt{e}_1|, l), F^{|\mathtt{e}_2|}(|\mathtt{e}_2|, l)) \leq F^{|\mathtt{e}|}(|\mathtt{e}|, l)\end{aligned}$$

We let the reader checking the other cases of the induction including the technical but simple case where $\mathtt{e} = \mathtt{Case}\ \mathtt{e}'\ \mathtt{of}\ \mathtt{c}_1(\vec{\mathtt{x}}_1) \to \mathtt{e}_1, \ldots, \mathtt{c}_m(\vec{\mathtt{x}}_m) \to \mathtt{e}_m$. Now the conclusion follows easily by taking $G(X, L) = F^X(X, L)$. □

4. Space Upper Bounds

4.1. Motivations

In several situations it is useful to have an estimate of the space needed to store the elements produced by a stream program. In some cases, this estimate can be obtained by considering the size of the greatest element produced as an output by the program. In other situations, unfortunately this cannot be done because there is no such a maximal element. However, an interesting estimate can be given by considering the position of the element in the stream. In functional programming, the full evaluation of a stream is never expected. A programmer will evaluate only 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 output index n of the element we want to reach. For example, we know that the size of the complete evaluation of the expression (`nats` $\underline{0}$) !! $\underline{\mathtt{n}}$, using the function symbol `nats` of Example 1, is bounded by the size of $\underline{\mathtt{n}}$. Note that such a measure always exists when a stream is productive since it only consists in providing the size of the n-th output value, for each integer n.

In this section, we will show how to use interpretations to define two criteria useful to compute space estimates similar to the ones described above. The first criterion, named Local Upper Bound (LUB), will ensure that programs admitting a particular interpretation compute streams where the n-th element is bounded by a function f in n and in the size of the inputs. Thanks to Lemma 6 the criterion will provide an estimate of such an f. This criterion is named "local" because the bound relies also on the output index n. The second criterion, named Global Upper Bound (GUB), is a special case of LUB in which the output does not depend on the index n. It will ensure that programs admitting a particular interpretation compute stream elements bounded by a function f in the size of the inputs, independently of the index. Again, thanks to Lemma 4 the criterion will provide an estimate of such an f. This criterion is named

"global" because the bound holds for all the output stream elements. This situation can be illustrated by the following figure:

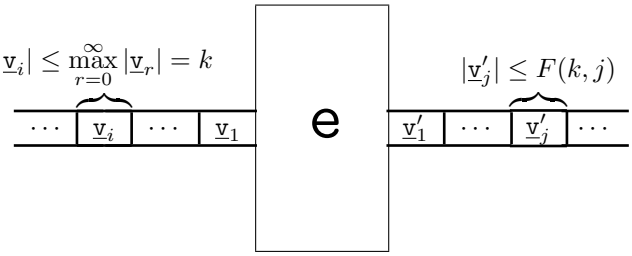

Figure 1: Local Upper Bound

A program e can be viewed as a box connecting a left tape representing the input stream and a right tape representing the output stream (we do not assume any synchrony between input and output). The program e has a *local upper bound* if each element $\underline{\mathtt{v}}'_j$ of the output stream has bounded size. Such a bound can depend not only on the size k of the maximal input stream element (in the case such a maximal element exists) but also on the the output element index j. In the particular case where this upper bound is independent of the index j, we say that e has a *global upper bound*. The aim of the LUB and GUB criteria we will present below is to exhibit a function $F : \mathbb{R}^+ \to \mathbb{R}^+$ witnessing these bounds.

4.2. Illustrating examples

A typical situation where programs have a local upper bound is described in the following example:

Example 5. *Consider expressions built using the following definitions already presented in Example 2:*

$$\begin{array}{l}
\mathtt{add} \;::\; \mathtt{Nat} \to \mathtt{Nat} \to \mathtt{Nat} \\
\mathtt{add}\ \mathtt{0}\ \mathtt{y} \doteq \mathtt{y} \\
\mathtt{add}\ (\mathtt{z}+1)\ \mathtt{y} \doteq (\mathtt{add}\ \mathtt{z}\ \mathtt{y}) + 1 \\
\\
\mathtt{sadd} \;::\; [\mathtt{Nat}] \to [\mathtt{Nat}] \to [\mathtt{Nat}] \\
\mathtt{sadd}\ (\mathtt{x}' : \mathtt{xs})\ (\mathtt{y}' : \mathtt{ys}) \doteq (\mathtt{add}\ \mathtt{x}'\ \mathtt{y}') : (\mathtt{sadd}\ \mathtt{xs}\ \mathtt{ys})
\end{array}$$

Each program will only generate output stream elements whose size is bounded by a function in their index. For example, the program:

$$\mathtt{e} = \mathtt{sadd}\ (\mathtt{nats}\ \underline{3})\ (\mathtt{sadd}\ (\mathtt{nats}\ \underline{5})\ (\mathtt{nats}\ \underline{4}))$$

is not globally bounded but if we evaluate the n-th output stream element as $\mathcal{H}; \mathtt{e}_{\underline{\mathtt{n}}} \Downarrow_v \underline{\mathtt{v}}$, then we know that the size of $\underline{\mathtt{v}}$ is bounded by $12 + 3 \times n$. Consequently, this program has a local upper bound given by the function $F(X) = 12 + 3 \times X$ applied to the index n.

A typical situation where programs have a global upper bound is described in the following example.

Example 6. *Consider expressions built using the following function definitions:*

$$\begin{array}{l}
\texttt{repeat} :: \texttt{Nat} \to [\texttt{Nat}] \\
\texttt{repeat x} \doteq \texttt{x} : (\texttt{repeat x}) \\
\texttt{zip} :: [a] \to [a] \to [a] \\
\texttt{zip}\ (\texttt{x} : \texttt{xs})\ \texttt{ys} \doteq \texttt{x} : (\texttt{zip ys xs}) \\
\texttt{square} :: [\texttt{Nat}] \to [\texttt{Nat}] \\
\texttt{square}\ (\texttt{x} : \texttt{xs}) \doteq (\texttt{mul x x}) : (\texttt{square xs}) \\
\texttt{mul} :: \texttt{Nat} \to \texttt{Nat} \to \texttt{Nat} \\
\texttt{mul}\ (\texttt{x} + 1)\ \texttt{y} \doteq \texttt{add y}\ (\texttt{mul x y}) \\
\texttt{mul}\ 0\ \texttt{y} \doteq 0
\end{array}$$

Each program will only produce output stream elements whose size is bounded by some constant k. For instance, the program:

$$\texttt{square}\ (\texttt{zip}\ (\texttt{repeat}\ \underline{5})\ (\texttt{square}\ (\texttt{zip}\ (\texttt{repeat}\ \underline{7})\ (\texttt{repeat}\ \underline{4}))))$$

computes stream elements whose size is bounded by $k = 2401 = 7^4$. So, its global upper bound is given by the constant $k = 2401$. Now, consider the following generalization of the above program:

$$\texttt{square}\ (\texttt{zip}\ (\texttt{repeat}\ \underline{5})\ (\texttt{square}\ (\texttt{zip x}\ (\texttt{repeat}\ \underline{4})))) \tag{1}$$

It is easy to verify that when x *is substituted by a stream* s *having element sizes bounded by a constant k, then the output stream will have only elements whose sizes are bounded either by 4^4 or by k^4. So this expression has a global upper bound given by the function F defined by $F(X) = \max(256, X^4)$.*

The above examples clearly illustrate that a global upper bound implies a local upper bound but that the converse does not hold.

4.3. Definition

More formally, we can describe the situations outlined above using the formal framework presented in Section 2 as follows.

Definition 18 (Local and Global Upper Bounds)**.** *A stream program* $\texttt{e} :: [\sigma]$ *has a* local upper bound *if there is a function $F \in \mathbb{R}^+ \to \mathbb{R}^+$ such that:*

$$\forall \underline{\texttt{n}} \in \texttt{N},\ \textit{if}\ \mathcal{H}; \texttt{e}_{\underline{\texttt{n}}} \Downarrow_v \underline{\texttt{v}}\ \textit{then}\ F(|\underline{\texttt{n}}|) \geq |\underline{\texttt{v}}|$$

In the case the function F is constant, then $\texttt{e} :: [\sigma]$ *has also a* global upper bound.

Note that the above definition can be used to analyze a stream program e containing both stream functions and stream definitions (programs with stream functions are provided in Examples 11 and 12). Now consider the following examples in order to illustrate this definition.

Example 7. *Consider again the stream definition of* nats*:*

$$\texttt{nats} :: \texttt{Nat} \to [\texttt{Nat}]$$
$$\texttt{nats}\ \texttt{x} \doteq \texttt{x} : (\texttt{nats}\ (\texttt{x} + 1))$$

Clearly, nats e *produces a stream whose elements are of unbounded size. However it is easy to verify that* $\forall \underline{\texttt{n}} \in \texttt{N}$*, if* $\mathcal{H}; (\texttt{nats}\ \texttt{e})_{\underline{\texttt{n}}} \Downarrow \texttt{v}$ *then* $\texttt{v} = ((\texttt{e} \underbrace{+1) + \cdots) + 1}_{n \text{ times}}$.
Consequently, by taking $F(X) = 2 \times X$*, the following inequalities are satisfied* $\forall \underline{\texttt{n}} \in \texttt{N}$*:*

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

Example 8. *Consider the program* ones *defined as:*

$$\texttt{ones} \doteq \underline{1} : \texttt{ones}$$

Clearly, it has an obvious global upper bound ($K = 1$*). Analogously, consider the program* repeat $\underline{\texttt{n}}$ *for every* $\underline{\texttt{n}} \in \texttt{Nat}$ *where:*

$$\texttt{repeat}\ \texttt{x} \doteq \texttt{x} : (\texttt{repeat}\ \texttt{x})$$

Clearly, repeat $\underline{\texttt{n}}$ *has a global upper bound that can be given by the function* $F(X) = X$*. That is, for every* $\underline{\texttt{n}}$ *we have a constant* $K_{\underline{\texttt{n}}} = F(|\underline{\texttt{n}}|) = |\underline{\texttt{n}}|$*. In the same spirit, going back to Equation 1 of Example 6, the function* $F(X) = \max(256, X^4)$ *provides for every input stream* s *of data of size bounded by* k *a constant* $K_{\texttt{s}} = \max(256, k^4)$*.*

4.4. LUB *and* GUB *criteria*

To ensure a Local Upper Bound we present a combined criterion consisting in a semantic condition on programs and in a semantic condition on interpretations.

Concerning the criterion on programs, we need to identify a restricted class of programs that we dub *linear programs*. These are programs that produce outputs with only a linear number of reads (stream pattern matchings in our concern). We can define them formally as follows.

Definition 19 (Linear program)**.** *Let* $\downarrow^k$ *and* $\downarrow^k_v$ *be the relations defined by* $\mathcal{H}; \texttt{e} \downarrow^k \texttt{v}$ *if there exists* $k' \leq k$ *such that* $\mathcal{H}; \texttt{e} \Downarrow^{k'} \texttt{v}$ *and* $\mathcal{H}; \texttt{e} \downarrow^k_v \texttt{v}$ *if there exists* $k' \leq k$ *such that* $\mathcal{H}; \texttt{e} \Downarrow^{k'}_v \texttt{v}$*, respectively.*
A program $\texttt{e} :: [\sigma]$ *is linear if there is a* $k \geq 1$ *such that for all* $\underline{\texttt{n}} \in \texttt{N}$*,* $\mathcal{H}; \texttt{e}_{\underline{\texttt{n}}} \downarrow^{k \times (|\underline{\texttt{n}}|+1)}_v \underline{\texttt{v}}$ *holds. The constant* k *is called the linearity constant.*

Now we are ready to define our criterion.

Definition 20 (LUB Criterion)**.** *A program* $\texttt{e} :: [\sigma]$ *is* LUB *if it is linear and it admits a parametrized interpretation* $(\!|-|\!)^l_\xi$ *that is almost-additive and such that:*

$$\xi^l(:)(X, Y) = \max(X, Y)$$

Now we can provide a similar criterion for Global Upper Bound:

Definition 21 (GUB Criterion)**.** *A program* $\texttt{e} :: [\sigma]$ *is* GUB *if it admits an interpretation* $(\!|-|\!)_\xi$ *that is almost-additive and such that:*

$$\xi(:)(X, Y) = \max(X, Y)$$

4.5. Soundness

Now we want to show that if a given program $\mathtt{e} :: [\sigma]$ is LUB (resp. GUB) then it has a local (resp. global) upper bound. For that purpose, we first show an intermediate technical lemma.

Lemma 7. *Given an expression* $\mathtt{e} :: [\sigma]$*:*

1. *If* $\mathtt{e}$ *is a* GUB *program then* $\forall \underline{\mathtt{n}} \in \mathtt{N},$ *s.t.* $\mathcal{H}; \mathtt{e}_{\underline{\mathtt{n}}} \Downarrow_v \underline{\mathtt{v}}$ *we have:*

$$(\!|\mathtt{e}|\!)_\xi \geq (\!|\underline{\mathtt{v}}|\!)_\xi$$

2. *If* $\mathtt{e}$ *is a* LUB *program of linearity constant* k *then* $\forall \underline{\mathtt{n}} \in \mathtt{N}$*, s.t.* $\mathcal{H}; \mathtt{e}_{\underline{\mathtt{n}}} \Downarrow_v \underline{\mathtt{v}}$ *we have:*

$$(\!|\mathtt{e}|\!)_\xi^{k\times(|\underline{\mathtt{n}}|+1)} \geq (\!|\underline{\mathtt{v}}|\!)_\xi^0$$

Proof. (1) We proceed by induction on $\underline{\mathtt{n}} \in \mathtt{N}$.
Let $\underline{\mathtt{n}} = \underline{\mathtt{0}}$ and $\mathcal{H}; \mathtt{e}_{\underline{\mathtt{0}}} \Downarrow_v \underline{\mathtt{v}}$. Then, necessarily we have a value $\mathtt{v}'$ such that $\mathcal{H}; \mathtt{e} \Downarrow \mathtt{v}'$. We have three cases, either $\mathtt{v}' = \mathtt{Err}$ or $\mathtt{v}' = \mathtt{nil}$ or $\mathtt{v}' = \mathtt{e}' : \mathtt{e}''$. The former two cases are trivial. For the latter, by definition of GUB and by Proposition 1 we have:

$$(\!|\mathtt{e}|\!)_\xi \geq (\!|\mathtt{e}' : \mathtt{e}''|\!)_\xi = (\!|:|\!)_\xi((\!|\mathtt{e}'|\!)_\xi, (\!|\mathtt{e}''|\!)_\xi) \geq (\!|\mathtt{e}'|\!)_\xi$$

Since we clearly have $\mathcal{H}; \mathtt{e}' \Downarrow_v \underline{\mathtt{v}}$, by applying Corollary 1 we obtain $(\!|\mathtt{e}'|\!)_\xi \geq (\!|\underline{\mathtt{v}}|\!)_\xi$. So by transitivity we can conclude $(\!|\mathtt{e}|\!)_\xi \geq (\!|\underline{\mathtt{v}}|\!)_\xi$.
Now, let $\underline{\mathtt{n}} = \underline{\mathtt{n}}' + 1$ and $\mathcal{H}; \mathtt{e}_{(\underline{\mathtt{n}}'+1)} \Downarrow \underline{\mathtt{v}}$. Again we have a value $\mathtt{v}'$ such that $\mathcal{H}; \mathtt{e} \Downarrow \mathtt{v}'$. The case $\mathtt{v}' = \mathtt{Err}$ is trivial. So, consider the case $\mathtt{v}' = \mathtt{e}' : \mathtt{e}''$. Again by definition of GUB and by Proposition 1 we have:

$$(\!|\mathtt{e}|\!)_\xi \geq (\!|\mathtt{e}' : \mathtt{e}''|\!)_\xi = (\!|:|\!)_\xi((\!|\mathtt{e}'|\!)_\xi, (\!|\mathtt{e}''|\!)_\xi) \geq (\!|\mathtt{e}''|\!)_\xi$$

Moreover $\mathcal{H}; \mathtt{e}_{(\underline{\mathtt{n}}'+1)} \Downarrow_v \underline{\mathtt{v}}$ implies by definition that $\mathcal{H}; \mathtt{e}''_{\underline{\mathtt{n}}'} \Downarrow_v \underline{\mathtt{v}}$ and by induction hypothesis we have $(\!|\mathtt{e}''|\!)_\xi \geq (\!|\underline{\mathtt{v}}|\!)_\xi$. So we can conclude $(\!|\mathtt{e}|\!)_\xi \geq (\!|\underline{\mathtt{v}}|\!)_\xi$.
(2) Assume that $\mathtt{e} :: [\sigma]$ is a LUB program of linearity constant k. We proceed by induction on $\underline{\mathtt{n}} \in \mathtt{N}$.
Consider the base case where $\underline{\mathtt{n}} = \underline{\mathtt{0}}$. By assumption, we have $\mathcal{H}; \mathtt{e}_{\underline{\mathtt{0}}} \Downarrow_v \underline{\mathtt{v}}$ and, necessarily we have a value $\mathtt{v}'$ such that $\mathcal{H}; \mathtt{e} \Downarrow \mathtt{v}'$. We have three cases, either $\mathtt{v}' = \mathtt{Err}$ or $\mathtt{v}' = \mathtt{nil}$ or $\mathtt{v}' = \mathtt{e}' : \mathtt{e}''$. The former two cases are trivial. For the latter, by definition of linear program with linearity constant k, we know that $\mathcal{H}; \mathtt{e} \Downarrow^{k'} \mathtt{e}' : \mathtt{e}''$, for some $\mathtt{e}'$ such that $\mathcal{H}; \mathtt{e}' \Downarrow_v^{k''} \underline{\mathtt{v}}$ with $k' + k'' \leq k$. Consequently, by Proposition 2 we have:

$$(\!|\mathtt{e}|\!)_\xi^{k\times(|\mathtt{0}|+1)} = (\!|\mathtt{e}|\!)_\xi^k \geq (\!|\mathtt{e}' : \mathtt{e}''|\!)_\xi^{k-k'}$$

By definition it is easy to verify that:

$$(\!|\mathtt{e}' : \mathtt{e}''|\!)_\xi^{k-k'} \geq (\!|\mathtt{e}'|\!)_\xi^{k-k'}$$

and by Corollary 3 and monotonicity (since $k' + k'' \leq k$) we obtain:

$$(\!|\mathtt{e}'|\!)_\xi^{k-k'} \geq (\!|\underline{\mathtt{v}}|\!)_\xi^{k-k'-k''} \geq (\!|\underline{\mathtt{v}}|\!)_\xi^0$$

Now we prove the induction step for $\underline{\mathtt{n}}' + 1 = \underline{\mathtt{n}}$. Suppose that $\mathcal{H}; \mathsf{e}_{\underline{\mathtt{n}}} \downarrow_v^{k\times(|\underline{\mathtt{n}}|+1)} \underline{\mathtt{v}}$. It is easy to verify that necessarily $\mathcal{H}; \mathsf{e} \Downarrow^j \mathsf{e}' : \mathsf{e}''$ and $\mathcal{H}; \mathsf{e}''_{\underline{\mathtt{n}}'} \Downarrow_v \underline{\mathtt{v}}$ for some $j,\ j < k$. By applying Proposition 2 we have:

$$(\!|\mathsf{e}|\!)_\xi^{(|\underline{\mathtt{n}}|+1)\times k} \geq (\!|\mathsf{e}' : \mathsf{e}''|\!)_\xi^{(|\underline{\mathtt{n}}|+1)\times k-j}$$

By definition it is easy to verify that:

$$(\!|\mathsf{e}' : \mathsf{e}''|\!)_\xi^{(|\underline{\mathtt{n}}|+1)\times k-j} \geq (\!|\mathsf{e}''|\!)_\xi^{(|\underline{\mathtt{n}}|+1)\times k-(j+1)}$$

and since $j + 1 \leq k$, by monotonicity:

$$(\!|\mathsf{e}''|\!)_\xi^{(|\underline{\mathtt{n}}|+1)\times k-(j+1)} \geq (\!|\mathsf{e}''|\!)_\xi^{k\times(|\underline{\mathtt{n}}'|+1)}$$

Now, applying induction hypothesis we have:

$$(\!|\mathsf{e}''|\!)_\xi^{k\times(|\underline{\mathtt{n}}'|+1)} \geq (\!|\underline{\mathtt{v}}|\!)_\xi^0$$

and so the conclusion follows. □

We can now prove the main result of this section.

Theorem 1. *If a program is* LUB *(*GUB*) then it admits a local (global) upper bound.*

Proof. Consider a LUB program $\mathsf{e} :: [\sigma]$ wrt the parametrized interpretation $(\!|-|\!)_\xi^l$. By Lemma 6, there is a function $G : \mathbb{R}^+ \times \mathbb{R}^+ \to \mathbb{R}^+$ such that $\forall l \in \mathbb{R}^+,\ G(|\mathsf{e}|, l) \geq (\!|\mathsf{e}|\!)_\xi^l$. Let us take $F(X) = G(|\mathsf{e}|, k\times(X+1))$, k being the linearity constant of e, and assume for $\underline{\mathtt{n}} \in \mathtt{Nat}$ that $\mathsf{e}_{\underline{\mathtt{n}}} \Downarrow_v \underline{\mathtt{v}}$. By Lemma 7(2), we have $(\!|\mathsf{e}|\!)_\xi^{k\times(|\underline{\mathtt{n}}|+1)} \geq (\!|\underline{\mathtt{v}}|\!)_\xi^0$. Moreover, since $(\!|-|\!)_\xi^0$ has a fixed parameter, it corresponds to an almost-additive interpretation. So, by Corollary 1 we have $(\!|\underline{\mathtt{v}}|\!)_\xi^0 \geq |\underline{\mathtt{v}}|$. Summing up, we have:

$$F(|\underline{\mathtt{n}}|) = G(|\mathsf{e}|, k \times (|\underline{\mathtt{n}}| + 1)) \geq (\!|\mathsf{e}|\!)_\xi^{k\times(|\underline{\mathtt{n}}|+1)} \geq (\!|\underline{\mathtt{v}}|\!)_\xi^0 \geq |\underline{\mathtt{v}}|$$

and so the conclusion follows.
Now consider a GUB program $\mathsf{e} :: [\sigma]$. By Lemma 4 there is a function $F : \mathbb{R}^+ \to \mathbb{R}^+$ such that $F(|\mathsf{e}|) \geq (\!|\mathsf{e}|\!)_\xi$. Let us take $K = F(|\mathsf{e}|)$ and assume for $\underline{\mathtt{n}} \in \mathtt{Nat}$ that $\mathcal{H}; \mathsf{e}_{\underline{\mathtt{n}}} \Downarrow_v \underline{\mathtt{v}}$. By Lemma 7(1) we have $(\!|\mathsf{e}|\!)_\xi \geq (\!|\underline{\mathtt{v}}|\!)_\xi$. Moreover, by Corollary 1 we have $(\!|\underline{\mathtt{v}}|\!)_\xi \geq |\underline{\mathtt{v}}|$, since such that ξ is an almost-additive symbol assignment. So, summing up, we have:

$$K = F(|\mathsf{e}|) \geq (\!|\mathsf{e}|\!)_\xi \geq (\!|\underline{\mathtt{v}}|\!)_\xi \geq |\underline{\mathtt{v}}|$$

and so the conclusion follows. □

Thanks to the above theorem, if we can find a LUB interpretation for a program e, then we also have a local upper bound. Let us consider some examples.

Example 9. *Consider again the stream definition of* nats *of Example 1:*

$$\begin{aligned}&\mathtt{nats} :: \mathtt{Nat} \to [\mathtt{Nat}]\\ &\mathtt{nats}\ \mathtt{x} \doteq \mathtt{x} : (\mathtt{nats}\ (\mathtt{x}+1))\end{aligned}$$

We want to show that nats *is* LUB. *First, notice that the program* nats *is linear with linearity constant $k = 1$. Indeed, the definition of* nats *does not involve pattern matching on stream data so the only pattern matchings correspond to the* !! *definition where one read is needed to produce on output. Now, consider the parametrized interpretation $(\!|-|\!)^l_{\rho,\xi}$ defined by: $\xi^l(\mathtt{nats})(X) = X + l$, $\xi^l(+1)(X) = X + 1$, $\xi^l(\mathtt{0}) = 0$ and $\xi^l(:)(X,Y) = \max(X,Y)$. We check that $\forall l \in \mathbb{R}$:*

$$\begin{aligned}(\!|\mathtt{nats}\ \mathtt{x}|\!)^l_{\rho,\xi} &= (\!|\mathtt{nats}|\!)^l_{\rho,\xi}((\!|\mathtt{x}|\!)^l_{\rho,\xi}) = \rho(\mathtt{x}) + l\\ &\geq \max(\rho(\mathtt{x}), (\rho(\mathtt{x})+1) + (l-1))\\ &= \max((\!|\mathtt{x}|\!)^l_{\rho,\xi}, (\!|\mathtt{nats}(\mathtt{x}+1)|\!)^{l-1}_{\rho,\xi}) = (\!|\mathtt{x} : (\mathtt{nats}\ (\mathtt{x}+1))|\!)^l_{\rho,\xi}\end{aligned}$$

The interpretation $(\!|-|\!)^l_{\rho,\xi}$ clearly respects the required criterion for nats *to be* LUB. *That is, it is almost-additive and it is defined on : as $\xi^l(:)(X,Y) = \max(X,Y)$.*
So, nats *admits a local upper bound. We obtain the required bound by setting $F(X) = (\!|\mathtt{nats}(\underline{\mathtt{m}})|\!)^X_{\rho,\xi} = X + (\!|\underline{\mathtt{m}}|\!)_{\rho,\xi} = \mathtt{X} + |\underline{\mathtt{m}}|$, for all canonical numerals $\underline{\mathtt{m}}$, $\underline{\mathtt{n}} \in \mathtt{N}$ such that $(\mathtt{nats}\ \underline{\mathtt{m}})\ !!\ \underline{\mathtt{n}} \Downarrow_v \underline{\mathtt{v}_{\underline{\mathtt{n}}}}$, the following holds $F(|\underline{\mathtt{n}}|) \geq |\underline{\mathtt{n}}| + |\underline{\mathtt{m}}| \geq |\underline{\mathtt{v}_{\underline{\mathtt{n}}}}|$ (Indeed for all $\underline{\mathtt{n}}$, $\underline{\mathtt{v}_{\underline{\mathtt{n}}}} = \underline{\mathtt{m}+\mathtt{n}}$).*

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

$$\begin{aligned}&\mathtt{tail} :: [a] \to a\\ &\mathtt{tail}\ \mathtt{x} : \mathtt{xs} \doteq \mathtt{xs}\\ &\\ &\mathtt{fib} :: [\mathtt{Nat}]\\ &\mathtt{fib} \doteq \underline{\mathtt{0}} : (\underline{\mathtt{1}} : (\mathtt{sadd}\ \mathtt{fib}\ (\mathtt{tail}\ \mathtt{fib})))\end{aligned}$$

We want to show that this program is LUB. *First, notice that* fib *is linear with linearity constant $k = 4$. Now, consider the parametrized interpretation $(\!|-|\!)_{\rho,\xi}$ defined by: $\xi^l(0) = 0$, $\xi^l(+1)(X) = X + 1$, $\xi^l(:)(X,Y) = \max(X,Y)$, $\xi^l(\mathtt{sadd})(X,Y) = \xi^l(\mathtt{add})(X,Y) = X + Y$, $\xi^l(\mathtt{tail})(X) = X$ and $\xi^l(\mathtt{fib}) = 2^l$. For the first rule and for each $l \in \mathbb{R}$, the following inequalities are satisfied:*

$$\begin{aligned}(\!|\mathtt{fib}|\!)^l_\xi = 2^l &\geq \max(0, 1, 2 \times 2^{l-2})\\ &= \max((\!|\underline{\mathtt{0}}|\!)^l_\xi, \max((\!|\underline{\mathtt{1}}|\!)^{l-1}_\xi, 2 \times (\!|\mathtt{fib}|\!)^{l-2}_\xi))\\ &= \max((\!|\underline{\mathtt{0}}|\!)^l_\xi, \max((\!|\underline{\mathtt{1}}|\!)^{l-1}_\xi, (\!|\mathtt{sadd}\ \mathtt{fib}\ (\mathtt{tail}\ \mathtt{fib})|\!)^{l-2}_\xi))\\ &= \max((\!|\underline{\mathtt{0}}|\!)^l_\xi, (\!|\underline{\mathtt{1}} : \mathtt{sadd}\ \mathtt{fib}\ (\mathtt{tail}\ \mathtt{fib})|\!)^{l-1}_\xi)\\ &= (\!|\underline{\mathtt{0}} : (\underline{\mathtt{1}} : \mathtt{sadd}\ \mathtt{fib}\ (\mathtt{tail}\ \mathtt{fib}))|\!)^l_\xi\end{aligned}$$

We let the reader check the inequalities for the other definitions. So, we have that $(\!|-|\!)^l_{\rho,\xi}$ respects the required criterion for fib *to be* LUB. *That is, it is almost-additive and it is defined on : as $\xi^l(:)(X,Y) = \max(X,Y)$. Consequently,* fib *admits a local upper*

bound. The function 2^l is a parametrized upper bound on the Fibonacci sequence: for each canonical numeral $\underline{\mathtt{n}} \in \mathtt{N}$ s.t. $\mathtt{fib}\ !!\ \underline{\mathtt{n}} \Downarrow_v \underline{\mathtt{v}_{\mathtt{n}}}$, the inequality $2^{4\times(|\mathtt{n}|+1)} \geq |\underline{\mathtt{v}_{\mathtt{n}}}|$ is satisfied.

In the same way, if we can find a GUB interpretation for a program e, then we also have a global upper bound. Let us consider some examples.

Example 11 (Thue-Morse sequence)**.** *The following* morse *program computes the Thue-Morse sequence:*

$$\begin{array}{l}
\mathtt{morse} :: [\mathtt{Nat}] \\
\mathtt{morse} \doteq \underline{\mathtt{0}} : (\mathtt{zip}\ (\mathtt{inv}\ \mathtt{morse})\ (\mathtt{tail}\ \mathtt{morse})) \\
\mathtt{tail} :: [a] \\
\mathtt{tail}\ \mathtt{x} : \mathtt{xs} \doteq \mathtt{xs} \\
\mathtt{inv} :: [\mathtt{Nat}] \to [\mathtt{Nat}] \\
\mathtt{inv}\ \ \underline{\mathtt{0}} : \mathtt{xs} \ \doteq\ \underline{\mathtt{1}} : \mathtt{xs} \\
\mathtt{inv}\ \ \underline{\mathtt{1}} : \mathtt{xs} \ \doteq\ \underline{\mathtt{0}} : \mathtt{xs}
\end{array}$$

The morse *program is* GUB *with respect to the following interpretation:* $(\!|\mathtt{0}|\!)_\xi = 0$, $(\!|{+1}|\!)_\xi(X) = 1 + X$, $(\!|:|\!)_\xi(X,Y) = (\!|\mathtt{zip}|\!)_\xi = \max(X,Y)$, $(\!|\mathtt{inv}|\!)_\xi(X) = \max(1,X)$, $(\!|\mathtt{tail}|\!)_\xi(X) = X$ *and* $(\!|\mathtt{morse}|\!)_\xi = 1$. *For instance, for the first rule the following inequality is satisfied:*

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

We let the reader check the inequalities for the other definitions.

Note that the Thue-Morse program can be easily checked to have a global upper bound by looking to the (finite) output range even without using the criterion. However, this example shows that the analysis can be done in a modular way. Moreover it shows that even if only simple functions are used in interpretations some interesting examples can be captured.

Example 12. *The program of Example 6 is GUB with respect to the interpretation* $(\!|-|\!)_{\rho,\xi}$ *defined by:*

$$\begin{array}{rl}
(\!|\mathtt{0}|\!)_\xi &= 0 \\
(\!|{+1}|\!)_\xi(X) &= X + 1 \\
(\!|\mathtt{add}|\!)_\xi(X,Y) &= X + Y \\
(\!|\mathtt{zip}|\!)_\xi(X,Y) &= \xi(:)(X,Y) = \max(X,Y) \\
(\!|\mathtt{square}|\!)_\xi(X) &= X^2 \\
(\!|\mathtt{mul}|\!)_\xi(X,Y) &= X \times Y \\
(\!|\mathtt{repeat}|\!)_\xi(X) &= X
\end{array}$$

Indeed, we can check the inequalities of the interpretation is satisfied for every definition and for every variable assignment ρ. For the definition of `repeat` *we have:*

$$\begin{aligned}
(\!|\texttt{repeat x}|\!)_{\rho,\xi} &= (\!|\texttt{repeat}|\!)_{\xi}((\!|\texttt{x}|\!)_{\rho,\xi}) \\
&= \rho(\texttt{x}) \\
&\geq \max(\rho(\texttt{x}), \rho(\texttt{x})) \\
&= \xi(:)((\!|\texttt{x}|\!)_{\rho,\xi}, (\!|\texttt{repeat x}|\!)_{\rho,\xi}) \\
&= (\!|\texttt{x} : (\texttt{repeat x})|\!)_{\rho,\xi}
\end{aligned}$$

For the definition of `zip`*:*

$$\texttt{zip z ys} \doteq \texttt{Case z of } (\texttt{x} : \texttt{xs}) \rightarrow \texttt{x} : (\texttt{zip ys xs})$$

we check the following inequality:

$$\begin{aligned}
(\!|\texttt{zip z ys}|\!)_{\rho,\xi} &= \max(\rho(\texttt{z}), \rho(\texttt{ys})) \\
&\geq \max\{\max(u, v, \rho(\texttt{ys})) \\
&\quad \mid \forall u, v \in \mathbb{R}^+ \; s.t. \; \rho(\texttt{z}) \geq \max(u, v)\} \\
&= \max\{(\!|\texttt{x} : (\texttt{zip ys xs})|\!)_{\rho[\texttt{x}=u,\texttt{xs}=v],\xi} \\
&\quad \mid \forall u, v \in \mathbb{R}^+ \; s.t. \; (\!|\texttt{z}|\!)_{\xi} \geq (\!|\texttt{x} : \texttt{xs}|\!)_{\rho[\texttt{x}=u,\texttt{xs}=v],\xi}\} \\
&= (\!|\texttt{Case z of } (\texttt{x} : \texttt{xs}) \rightarrow \texttt{x} : (\texttt{zip ys xs})|\!)_{\rho,\xi}
\end{aligned}$$

We let the reader check that the inequalities are also satisfied for the remaining definitions. Consequently, the program of Example 6 admits a global upper bound. In this particular setting, the program:

$$\texttt{e} = \texttt{square } (\texttt{zip } (\texttt{repeat } \underline{5}) \; (\texttt{square } (\texttt{zip } (\texttt{repeat } \underline{7}) \; (\texttt{repeat } \underline{4}))))$$

admits a global upper bound that is equal to:

$$\begin{aligned}
(\!|\texttt{e}|\!)_{\rho,\xi} &= (\!|\texttt{zip } (\texttt{repeat } \underline{5}) \; (\texttt{square } (\texttt{zip } (\texttt{repeat } \underline{7}) \; (\texttt{repeat } \underline{4}))))|\!)^2_{\rho,\xi} \\
&= (\max((\!|\texttt{repeat } \underline{5}|\!)_{\rho,\xi}, (\!|\texttt{square } (\texttt{zip } (\texttt{repeat } \underline{7}) \; (\texttt{repeat } \underline{4}))|\!)_{\rho,\xi}))^2 \\
&= (\max((\!|\underline{5}|\!)_{\rho,\xi}, (\max((\!|\underline{7}|\!)_{\rho,\xi}, (\!|\underline{4}|\!)_{\rho,\xi}))^2))^2 \\
&= (\!|\underline{7}|\!)^4_{\rho,\xi} \\
&= 7^4
\end{aligned}$$

and we obtain that for all $\underline{\texttt{n}} \in \texttt{N}$ *such that* $\mathcal{H}, \texttt{e}_{\underline{\texttt{n}}} \Downarrow_v \underline{\texttt{v}}_{\underline{\texttt{n}}}$, $7^4 \geq |\underline{\texttt{v}}_{\underline{\texttt{n}}}|$.

5. Bounded Input/Output Properties

5.1. Motivations

In this section, we show how interpretations can be used to ensure stream properties relating input reads to output writes. In particular, we are interested in estimating the

ability of a program to return a certain finite amount of elements in the output stream when fed with some (finite part of the) input stream. Giving an upper bound on the quantities of information needed can be particularly useful to implement the program in an efficient way with respect to the needed memory.

Since we are interested in the dependencies with respect to the inputs, the properties we will analyze in this section mainly concern stream functions (whereas the properties presented in the previous sections also concern stream definitions). We will concentrate on two properties of stream functions:

- The length based I/O upper bound that provides an upper bound on the number of written output stream elements in the number of read input elements.
- The size based I/O upper bound, a more precise and general notion, that provides an upper bound on the number of written output stream elements in both the number and the size of read input elements.

The size based I/O upper bound is illustrated in Figure 2.

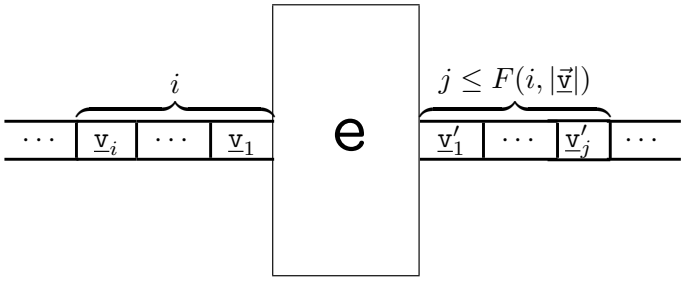

Figure 2: Size based Input/Output upper bound

Here a stream function $\mathtt{e}$ after reading i elements from the input produces j elements of the output. What we want is to obtain a relation linking j to i. In particular we want a function F describing an upper bound $F(i, |\vec{\underline{\mathtt{v}}}|)$ on j with respect to i and $|\vec{\underline{\mathtt{v}}}|$. We dub this kind of bound *size based* because the upper bound may depend also on the size of the input elements. In the particular case, where the function F is independent from $|\vec{\underline{\mathtt{v}}}|$, we obtain a length based upper bound. We dub this kind of bound *length based* because it only relies on the length i of the input (i.e. the number of input reads).

These properties are of finite nature. For this reason, we will define them in terms of finite stream fragments (i.e. finite lists) in a way that is a reminiscent of Bird and Wadler's *Take Lemma* [5].

Notation for contexts. For notational convenience, let $\mathtt{e}(\mathtt{x})$ be a notation for the expression $\mathtt{e}$ where the free variable $\mathtt{x}$ is explicitly mentioned. Let $\mathtt{e}(\underline{\mathtt{v}})$ be a notation for $\mathtt{e}(\mathtt{x})\{\underline{\mathtt{v}}/\mathtt{x}\}$ and, finally, define $(\!|\mathtt{e}|\!)_{\rho,\xi}(r)$ by $(\!|\mathtt{e}|\!)_{\rho,\xi}(r) = (\!|\mathtt{e}(\mathtt{x})|\!)_{\rho\{\mathtt{x}=r\},\xi}$.

5.2. Illustrating examples

A typical situation where programs have a length based Input/Output upper bound is described in the following example:

Example 13. *Consider stream expressions defined in terms of the following definitions:*

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

$$\texttt{dup} :: [a] \to [a]$$
$$\texttt{dup}\ (\texttt{x} : \texttt{xs}) \doteq \texttt{x} : (\texttt{x} : (\texttt{dup}\ \texttt{xs}))$$

It is easy to verify that each such an expression will only generate a number of output elements related to the number of input read elements. For example, the expression:

$$\texttt{dup}\ (\texttt{merge}\ (\texttt{dup}\ \texttt{s})\ (\texttt{dup}\ \texttt{s}))$$

for each read element of the input stream $\texttt{s}$ *computes a number of output elements bounded by* $k = 4$*. Consequently,* $F(i) = 4 \times i$ *in this particular case.*

A situation where programs have a size based Input/Output upper bound is described in the following example:

Example 14. *Consider stream expressions defined in terms of the following definitions:*

$$\texttt{app} :: [a] \to [a] \to [a]$$
$$\texttt{app}\ (\texttt{x} : \texttt{xs})\ \texttt{ys} \doteq \texttt{x} : (\texttt{app}\ \texttt{xs}\ \texttt{ys})$$
$$\texttt{app}\ \texttt{nil}\ \texttt{ys} \doteq \texttt{ys}$$

$$\texttt{upto} :: \texttt{Nat} \to [\texttt{Nat}]$$
$$\texttt{upto}\ \texttt{0} \doteq \texttt{nil}$$
$$\texttt{upto}\ (\texttt{x} + 1) \doteq (\texttt{x} + 1) : (\texttt{upto}\ \texttt{x})$$

$$\texttt{extendupto} :: [\texttt{Nat}] \to [\texttt{Nat}]$$
$$\texttt{extendupto}\ (\texttt{x} : \texttt{xs}) \doteq \texttt{app}\ (\texttt{upto}\ \texttt{x})\ (\texttt{extendupto}\ \texttt{xs})$$

It is easy to verify that such expressions will only generate a number of output elements related to the number and the size of input read elements. For example, the expression:

$$\texttt{extendupto}\ (\texttt{extendupto}\ \texttt{s})$$

it performs $\sum_{i=1}^{|\underline{\texttt{n}}|} i$ *output writes for each number* $\underline{\texttt{n}}$ *it reads on the input stream* $\texttt{s}$*. Consequently, it has no length based Input/Output upper bound.*

5.3. Definition

More formally, we can describe the situations outlined above as follows:

Definition 22 (Length and Size Based I/O Upper Bounds)**.**

- *A stream function* $\texttt{x} :: [\sigma] \vdash \texttt{e(x)} :: [\tau]$ *has a* length based I/O upper bound *if there is a function* $F : \mathbb{R}^+ \to \mathbb{R}^+$ *such that for every expression* $\texttt{s} :: [\sigma]$ *we have:*

$$\forall\ \underline{\texttt{n}} \in \texttt{N},\ \textit{if}\ \mathcal{H}; \texttt{s}\lceil_{\underline{\texttt{n}}} \Downarrow_v \underline{\texttt{v}}\ \textit{and}\ \mathcal{H}; \texttt{lg}\ \texttt{e}(\underline{\texttt{v}}) \Downarrow_v \underline{\texttt{m}}\ \textit{then}\ F(|\underline{\texttt{n}}|) \geq |\underline{\texttt{m}}|$$

- *A stream function* $\mathtt{x} :: [\sigma] \vdash \mathtt{e(x)} :: [\tau]$ *has a* size based I/O upper bound *if there is a function* $F : \mathbb{R}^+ \to \mathbb{R}^+$ *such that for every expression* $\mathtt{s} :: [\sigma]$ *we have:*

$$\forall\, \underline{\mathtt{n}} \in \mathtt{N},\ \textit{if}\ \mathcal{H}; \mathtt{s}\lceil_{\underline{\mathtt{n}}} \Downarrow_v \underline{\mathtt{v}}\ \textit{and}\ \mathcal{H}; \mathtt{lg\ e}(\underline{\mathtt{v}}) \Downarrow_v \underline{\mathtt{m}}\ \textit{then}\ F(|\underline{\mathtt{v}}|) \geq |\underline{\mathtt{m}}|$$

Note that for simplicity we have defined the length and size based I/O upper bounds only in the case of a unary function. However, the above definition can be easily extended to the case of functions with multiple arguments. Notice that the size based I/O property informally generalizes the length based 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. However, in some situations it is preferable to have the uniformity given by the length based I/O upper bound.

5.4. LBUB *and* SBUB *criteria*

We now want to introduce two criteria ensuring that a stream function has a length and size based I/O upper bounds. The definitions of length and size based I/O upper bounds above have been given in terms of the `lg` and `take` function symbols. So, for simplicity in what follows we suppose that the considered stream functions do not use neither `lg` symbol nor `take` symbol.

Definition 23 (LBUB and SBUB Interpretations)**.**

- *An interpretation* $(\!|-|\!)_{\rho,\xi}$ *is* LBUB *if* $(\!|-|\!)_{\rho,\xi}$ *is almost-additive with* $\xi(+1)(X) = X + 1$ *and such that:*

$$\xi(:)(X, Y) = Y + 1\ \textit{(resp.}\ X + Y + 1\textit{)}$$

- *An interpretation* $(\!|-|\!)_{\rho,\xi}$ *is* SBUB *if* $(\!|-|\!)_{\rho,\xi}$ *is additive with* $\xi(+1)(X) = X + 1$ *and:*

$$\xi(:)(X, Y) = X + Y + 1$$

- *An expression* $\mathtt{x} :: [\sigma] \vdash \mathtt{e(x)} :: [\tau]$ *is* LBUB *(resp.* SBUB*) if it admits an interpretation* $(\!|-|\!)_{\xi}$ *such that* $(\!|-|\!)_{\rho,\xi}$ *is* LBUB *(resp.* SBUB*).*

5.5. Soundness

LBUB assignments have some basic properties useful to deal with length based upper bounds. In particular, they give precise measures on natural numbers and lists as shown by the following two lemmas. Note that SBUB assignments behaves similarly on natural numbers.

Lemma 8. *Given a* LBUB *or* SBUB *assignment* $(\!|-|\!)_{\rho,\xi}$*, for every* $\underline{\mathtt{n}} :: \mathtt{Nat}$ *we have* $(\!|\underline{\mathtt{n}}|\!)_{\rho,\xi} = |\underline{\mathtt{n}}|$.

Proof. By induction on the length of $\underline{\mathtt{n}}$. The base case follows easily by additivity of $(\!|-|\!)_{\rho,\xi}$. That is $|\mathtt{0}| = (\!|\mathtt{0}|\!)_{\rho,\xi} = 0$.
Consider now the case $\underline{\mathtt{n}} + 1$. We have $(\!|\underline{\mathtt{n}} + 1|\!)_{\rho,\xi} = (\!|+1|\!)_{\rho,\xi}((\!|\underline{\mathtt{n}}|\!)_{\rho,\xi}) = (\!|\underline{\mathtt{n}}|\!)_{\rho,\xi} + 1$. By induction hypothesis $(\!|\underline{\mathtt{n}}|\!)_{\rho,\xi} = |\underline{\mathtt{n}}|$ and so the conclusion follows. □

Second, the interpretation of a finite list is equal to its length:

Lemma 9. *Given a* LBUB *assignment* $(\!|-|\!)_{\rho,\xi}$*, for every* $\underline{\mathtt{v}} :: [\sigma]$ *we have if* $\mathcal{H}; \mathtt{lg}\ \underline{\mathtt{v}} \Downarrow_v \underline{\mathtt{m}}$ *then* $(\!|\underline{\mathtt{v}}|\!)_{\rho,\xi} = |\underline{\mathtt{m}}|$.

Proof. By induction on the length of $\underline{\mathtt{v}}$. The base case follows easily by almost-additivity of $(\!|-|\!)_{\rho,\xi}$. That is $|\mathtt{0}| = (\!|\mathtt{nil}|\!)_{\rho,\xi} = 0$ since $\mathcal{H}; \mathtt{lg\ nil} \Downarrow_v \mathtt{0}$.
Consider now the case $\underline{\mathtt{v}}_1 : \underline{\mathtt{v}}_2$ and suppose that $\mathcal{H}; \mathtt{lg}\ (\underline{\mathtt{v}}_1 : \underline{\mathtt{v}}_2) \Downarrow_v \underline{\mathtt{m}} + 1$. We have $(\!|\underline{\mathtt{v}}_1 : \underline{\mathtt{v}}_2|\!)_{\rho,\xi} = \xi(:)((\!|\underline{\mathtt{v}}_1|\!)_{\rho,\xi}, (\!|\underline{\mathtt{v}}_2|\!)_{\rho,\xi}) = (\!|\underline{\mathtt{v}}_2|\!)_{\rho,\xi} + 1$. Moreover, $\mathcal{H}; \mathtt{lg}\ \underline{\mathtt{v}}_2 \Downarrow_v \underline{\mathtt{m}}$ and, by induction hypothesis, $(\!|\underline{\mathtt{v}}_2|\!)_{\rho,\xi} = |\underline{\mathtt{m}}|$ and so the conclusion follows. □

For simplicity we have assumed that the program expressions of this section do not contain the function symbol `lg`. However, since our criteria are defined in terms of it, in order to prove their soundess we need to have an interpretation for it. This can be done easily. Indeed, every LBUB or SBUB assignment can be extended to accommodate an interpretation for the `lg` function symbol as follows.

Lemma 10. *Suppose that the environment* $\mathcal{H}$ *admits the interpretation* $(\!|-|\!)_{\rho,\xi}$ *and that* $(\!|-|\!)_{\rho,\xi}$ *is a* LBUB *or* SBUB *assignment. Then,* $(\!|-|\!)_{\rho,\xi}$ *can be extended to the function symbol* `lg` *by setting* $\xi(\mathtt{lg})(X) = X$ *in such a way that the environment* $\{\mathtt{lg\ y} = \mathtt{Case\ y\ of\ nil} \to \underline{\mathtt{0}}, \mathtt{Err} \to \underline{\mathtt{0}}, \mathtt{x : xs} \to (\mathtt{lg\ xs}) + 1\} \cup \mathcal{H}$ *admits the interpretation* $(\!|-|\!)_{\rho,\xi}$.

Proof. Consider the definition $\{\mathtt{lg\ y} \doteq \mathtt{Case\ y\ of\ nil} \to \underline{\mathtt{0}}, \mathtt{Err} \to \underline{\mathtt{0}}, \mathtt{x : xs} \to (\mathtt{lg\ xs}) + 1\}$. We have to check that the extension of $(\!|-|\!)_{\rho,\xi}$ still satisfies the inequalities of an interpretation for this definition:

$$\begin{aligned}
(\!|\mathtt{lg\ y}|\!)_{\rho,\xi} &= \rho(\mathtt{y}) = r \\
&\geq \max(0, r) \\
&= \max(\max\{(\!|\underline{\mathtt{0}}|\!)_{\rho,\xi} \mid r \geq (\!|\mathtt{nil}|\!)_{\rho,\xi}\}, \\
&\qquad \max\{(\!|\underline{\mathtt{0}}|\!)_{\rho,\xi} \mid r \geq (\!|\mathtt{Err}|\!)_{\rho,\xi}\}, \\
&\qquad \max\{(\!|(\mathtt{lg\ xs}) + 1|\!)_{\rho[\mathtt{x}:=u,\mathtt{xs}:=v],\xi} \mid \forall u, v\ s.t.\ r \geq v + 1\}) \\
&= (\!|\mathtt{Case\ y\ of\ nil} \to \underline{\mathtt{0}}, \mathtt{Err} \to \underline{\mathtt{0}}, \mathtt{x : xs} \to (\mathtt{lg\ xs}) + 1|\!)_{\rho,\xi}
\end{aligned}$$

This inequality holds for every ρ such that $\rho(\mathtt{y}) = r$, for an arbitrary $r \geq 0$, and so the conclusion. □

We are now ready to prove that the LBUB (resp. SBUB) is a criterion to ensure the length (resp. size) based I/O upper bound.

Theorem 2. *If an expression* $\mathtt{x} :: [\sigma] \vdash \mathtt{e(x)} :: [\tau]$ *is* LBUB *(resp.* SBUB*) then it has also a length (resp. size) based I/O upper bound.*

Proof. Given a LBUB expression $\mathtt{e(x)}$, suppose $\mathcal{H}; \mathtt{s}\restriction_{\underline{\mathtt{n}}} \Downarrow_v \underline{\mathtt{v}}$ and $\mathcal{H}; \mathtt{lg\ e}(\underline{\mathtt{v}}) \Downarrow_v \underline{\mathtt{m}}$.

By Corollary 2, we have $(\!|\mathtt{lg\ e}(\underline{\mathtt{v}})|\!)_{\rho,\xi} \geq (\!|\underline{\mathtt{m}}|\!)_{\rho,\xi}$ and, by Lemma 10, we have $(\!|\mathtt{e}(\underline{\mathtt{v}})|\!)_{\xi} \geq (\!|\underline{\mathtt{m}}|\!)_{\xi}$.

By Lemma 9, we have $(\!|\underline{\mathtt{v}}|\!)_{\rho,\xi} = |\underline{\mathtt{n}}|$. Applying Lemma 8, we have $(\!|\underline{\mathtt{m}}|\!)_{\rho,\xi} = |\underline{\mathtt{m}}|$. So, we obtain $(\!|\mathtt{e}|\!)_{\xi}(|\underline{\mathtt{n}}|) \geq |\underline{\mathtt{m}}|$ and by taking $F = (\!|\mathtt{e}|\!)_{\rho,\xi}$ the conclusion follows.

Now suppose that $\mathtt{e}(\mathtt{x})$ is SBUB, that $\mathcal{H}; \mathtt{s}\restriction_{\underline{\mathtt{n}}} \Downarrow \underline{\mathtt{v}}$ and that $\mathcal{H}; \mathtt{lg}\ \mathtt{e}(\underline{\mathtt{v}}) \Downarrow_v \underline{\mathtt{m}}$.

By Corollary 2, we have $(\!|\mathtt{lg}\ \mathtt{e}(\underline{\mathtt{v}})|\!)_\xi \geq (\!|\underline{\mathtt{m}}|\!)_\xi$ and by, Lemmata 8 and 10, we have $(\!|\mathtt{e}(\underline{\mathtt{v}})|\!)_\xi \geq (\!|\underline{\mathtt{m}}|\!)_\xi = |\underline{\mathtt{m}}|$.

By definition, we obtain $(\!|\mathtt{e}(\underline{\mathtt{v}})|\!)_{\rho,\xi} = (\!|\mathtt{e}|\!)_{\rho,\xi}((\!|\underline{\mathtt{v}}|\!)_{\rho,\xi})$. Moreover, by Lemma 2, since $(\!|-|\!)_\xi$ is additive we have a constant α such that $(\!|\underline{\mathtt{v}}|\!)_{\rho,\xi} \leq |\underline{\mathtt{v}}| \times \alpha$. So, taking $F(X) = (\!|\mathtt{e}|\!)_{\rho,\xi}(X \times \alpha)$, the conclusion follows. □

We end this section by showing that the situation presented in Example 13 can be captured using the LBUB criterion while the one in Example 14 can be captured using the SBUB criterion.

Example 15. *Consider again the stream definitions presented in Example 13:*

$$\begin{array}{l}\mathtt{merge} :: [a] \to [a] \to [a \times a] \\ \mathtt{merge}\ (\mathtt{x} : \mathtt{xs})\ (\mathtt{y} : \mathtt{ys}) \doteq (\mathtt{x}, \mathtt{y}) : (\mathtt{merge}\ \mathtt{ys}\ \mathtt{xs})\end{array}$$

$$\begin{array}{l}\mathtt{dup} :: [a] \to [a] \\ \mathtt{dup}\ (\mathtt{x} : \mathtt{xs}) \doteq \mathtt{x} : (\mathtt{x} : (\mathtt{dup}\ \mathtt{xs}))\end{array}$$

To verify that every expression built using these definitions has a length based I/O upper bound it is sufficient to verify that it admits the following LBUB *interpretation:*

$$\xi(\mathtt{merge})(X, Y) = \max(X, Y) \quad \xi(\mathtt{dup})(X) = 2X$$

As an example, for each $\mathtt{s}$ *we have:*

$$(\!|\mathtt{dup}\ (\mathtt{merge}\ (\mathtt{dup}\ \mathtt{s})\ (\mathtt{dup}\ \mathtt{s}))|\!)_{\rho,\xi} = 4(\!|\mathtt{s}|\!)_{\rho,\xi}$$

and this gives a length based I/O upper bound.

Example 16. *Consider again the stream definitions of Example 14:*

$$\begin{array}{l}\mathtt{app} :: [a] \to [a] \to [a] \\ \mathtt{app}\ (\mathtt{x} : \mathtt{xs})\ \mathtt{ys} \doteq \mathtt{x} : (\mathtt{app}\ \mathtt{xs}\ \mathtt{ys}) \\ \mathtt{app}\ \mathtt{nil}\ \mathtt{ys} \doteq \mathtt{ys}\end{array}$$

$$\begin{array}{l}\mathtt{upto} :: \mathtt{Nat} \to [\mathtt{Nat}] \\ \mathtt{upto}\ 0 \doteq \mathtt{nil} \\ \mathtt{upto}\ (\mathtt{x} + 1) \doteq (\mathtt{x} + 1) : (\mathtt{upto}\ \mathtt{x})\end{array}$$

$$\begin{array}{l}\mathtt{extendupto} :: [\mathtt{Nat}] \to [\mathtt{Nat}] \\ \mathtt{extendupto}\ (\mathtt{x} : \mathtt{xs}) \doteq \mathtt{app}\ (\mathtt{upto}\ \mathtt{x})\ (\mathtt{extendupto}\ \mathtt{xs})\end{array}$$

To show that every expression built using these definitions has a size based I/O upper bound it is enough to show that the following interpretation is SBUB*:*

$$\xi(\mathtt{nil}) = (\!|0|\!)_\xi = 0 \quad \xi(+1)(X) = X + 1 \quad \xi(:)(X, Y) = X + Y + 1$$

$$\xi(\mathtt{app})(X, Y) = X + Y \quad \xi(\mathtt{upto})(X) = \xi(\mathtt{extendupto})(X) = 2 \times X^2$$

In particular, by taking $F(X) = (\!|\mathtt{extendupto}|\!)_\xi((\!|\mathtt{extendupto}|\!)_\xi(X))$ *we obtain a size based upper bound for the expression:*

$$\mathtt{extendupto}\ (\mathtt{extendupto}\ \mathtt{s})$$

That is $F(X) = 8 \times X^4$ is an upper bound on the number of output elements. Notice that the bound is less tight than what one would have expected. Indeed, in this example F gives a bound also on the size of produced elements.

6. Computing an Interpretation

One of the aspects of interpretation methods that makes them of practical interest is that they do not only provide techniques to *ensure the existence* of particular upper bounds but in several cases of practical interest they also actually provide a tool to effectively *compute* those upper bounds. Indeed, Lemma 4 says that if we have an interpretation of a given program `M`, then we can compute an upper bound on the size of the result of its evaluation. So it is natural to consider the corresponding *interpretation synthesis* problem that can be formulated as follows:

> **Interpretation synthesis problem:** given a program `M` and a class of functions $\mathfrak{F}$, is an interpretation for `M` using only functions in $\mathfrak{F}$ computable?

The ability of being able to compute an interpretation clearly depends on the class of functions $\mathfrak{F}$ that one wants to consider. In particular as a consequence of the Rice's theorem the interpretation synthesis problem is undecidable for an unrestricted class of computable functions. More interestingly this problem becomes decidable when restricted class of functions are considered. See [34] for a survey.

Even if in this paper we do not concentrate on the synthesis problem for the different criteria we have introduced, the possibility of having efficient procedures to compute the studied bounds is a strong motivation of our work. In particular, the criteria studied here become particularly useful when the interpretations use only small classes of functions as codomain (e.g. polynomials, logarithmic or linear functions). We leave this important study for future works.

Another related problem that we have not addressed here is the complexity of the interpretation synthesis problem. This problem has been previously studied for functions coming from max-plus and max-poly algebras over integers and reals and the results obtained can be adapted to our framework (see [34]).

It is worth noting that this problem has also been studied by the rewriting community both from theoretical and practical points of view [30, 25].

7. Related works

In the data processing scenario, a greater attention is paid to the so called *streaming algorithms*. These are algorithms working with restricted computational power on huge amount of data (not necessarily infinite). Usually they have only limited access to the inputs and they have only a little amount of available memory. Moreover, they may also satisfy some timing constraints. Moreover, the criteria we have designed so far are inspired by the constraints that streaming algorithms should usually satisfy.

It is not surprising that when one wants to implement programs working on streams, one needs to pay attention to memory management. It is indeed not difficult to find examples of stream programs generating subtle buffering or overflow errors. In this

perspective, in [17] Frankau and Mycroft have proposed a framework that, starting from programs written in a first-order functional language, extracts stream program implementations avoiding unbounded buffering. In order to achieve this goal, their programs have to obey some specific linearity and stability typing disciplines. Analogously, Hughes et al. in their paper introducing *sized types* [22] show how the latter can be used to prevent errors related to memory leaks and buffer overflows of embedded programs. Even if sized types have been mainly introduced to prove termination properties of reactive systems (corresponding to productivity of stream programs) they have also found several applications in complexity analysis. For instance in [37], the authors show how to exploit sized types in a system designed to verify the resource usage of strict first-order programs working on lists. The programming language they are able to analyze through their type system is similar to the language we consider in the present paper, a key distinction being that their analysis is restricted to finite lists.

It is quite surprising that in the implicit computational complexity domain only few works have been carried so far on programs computing over infinite data structures. This is even more surprising if one thinks that usual tools of complexity theory, well behaving on finite data types, cannot be directly applied neither to streams nor to other infinite data structures. In [10] Burrell et al. have developed a sound and complete polynomial time complexity programming language, dubbed Pola, based on a type system with restrictions inspired by safe recursion. Interestingly, Pola permits the programmer to deal with polynomial time functional programs working both on inductive and coinductive data types.

In [29], Leivant and Ramyaa have proposed a framework based on equational programs and intrinsic theories, previously introduced by Leivant in [28], that is useful to reason about programs over inductive and co-inductive types. They used such a framework to obtain an implicit characterization of primitive corecurrence (a weak form of productivity). More recently, in [36] they have also shown that a ramified version of co-recurrence gives an implicit characterization of the class of functions over streams working in logarithmic space. In contrast to our approach considering functions working on data types, their characterization deals with functions working on streams of digits; however, they consider the complexity of a stream program as a function of the output. That is the space needed to compute the n-th element of the stream. With this respect, their characterization uses an approach similar to the one we follow for the Local Upper Bound property and for the Bounded Input/Output properties.

Using an approach similar to the one presented in this paper, Férée et al. [16] show that interpretations can be used on stream programs also to characterize type 2 polynomial time functions. In particular, they extend interpretations in order to use second order polynomials to characterize the set of functions computable in polynomial time by Oracle Turing Machines and by their unitary version. Thanks to this they obtain an implicit characterization of the class of the Basic Feasible Functionals of Cook and Urquhart [11].

Recently, Baillot and Dal Lago [4] have developed a technique inspired by quasi-interpretations to study the complexity of higher-order rewriting programs. In their framework infinite data are first class citizens in the form of higher order functions. However, they do not consider programs working on declarative infinite data structures as streams that are instead the focus of our work.

8. Conclusion

In this paper, we have studied some complexity properties of programs working on streams. In order to do this, in a first step we have adapted the interpretation methods to a functional programming languages able to express in a natural way stream programs. This has required to customize the definitions of interpretations, developed so far in the context of first-order term rewriting, to the more specific case of a first-order lazy functional programming language. As a byproduct, this has shown the flexibility of the interpretation tools in dealing with different object languages and in dealing with streams and infinite data types.

In a second step, we have exploited the use of interpretation methods by defining several criteria for the study of different space properties. These criteria correspond to resource static analyses useful to the programmer that would be able to control the complexity of the stream program he writes. They fall in two main categories:

- The first category, that includes local and global upper bounds, provides an upper bound on the size of each computed stream element. This upper bound can be a constant in the case of the global upper bound or a function of the output element position in the case of the local upper bound. The stream program that can be analyzed with respect to these criteria gives the guarantee that no one of its output elements will provoke a *memory overflow*.

- The second category, that includes the size and length based input and output upper bounds, provides an upper bound on the number of output elements with respect to the number or size of input reads.

The two categories above deal with two different dimensions of streams. By combining together the different criteria one can study several memory management aspects of programs working on streams.

[1] Abramsky, S., Ong, C.-H. L., 1993. Full abstraction in the lazy lambda calculus. Inf. Comput. 105 (2), 159–267.

[2] Amadio, R., 2005. Synthesis of max-plus quasi-interpretations. Fundamenta Informaticae 65(1–2).

[3] Ariola, Z. M., Felleisen, M., 1997. The call-by-need lambda calculus. J. Funct. Program. 7 (3), 265–301.

[4] Baillot, P., Dal Lago, U., 2012. Higher-Order Interpretations and Program Complexity. In: Proceedings of CSL'12. Vol. 16 of Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp. 62–76.

[5] Bird, R., Wadler, P., 1988. Introduction to Functional Programming. Prentice Hall.

[6] Bonfante, G., Cichon, A., Marion, J.-Y., Touzet, H., 2001. Algorithms with polynomial interpretation termination proof. Journal of Functional Programming 11 (1), 33–53.

[7] Bonfante, G., Marion, J., Moyen, J., 2011. Quasi-interpretations a way to control resources. Theoretical Computer Science.

[8] Bonfante, G., Marion, J.-Y., Moyen, J.-Y., Jul 2001. On lexicographic termination ordering with space bound certifications. In: PSI. Vol. 2244 of LNCS. Springer.

[9] Bonfante, G., Marion, J.-Y., Moyen, J.-Y., Péchoux, R., 2005. Synthesis of quasi-interpretations. LCC2005, LICS Workshop.

[10] Burrell, M. J., Cockett, R., Redmond, B. F., August 2009. Pola: a language for PTIME programming. In: Tenth International Workshop on Logic and Computational Complexity. Los Angeles, USA.

[11] Cook, S., Urquhart, A., 1989. Functional interpretations of feasibly constructive arithmetic. In: Proceedings of the twenty-first annual ACM symposium on Theory of computing. STOC '89. ACM, New York, NY, USA, pp. 107–112.

[12] Coquand, T., 1994. Infinite objects in type theory. Vol. 806 of LNCS. Springer, pp. 62–78.

[13] Cousot, P., Cousot, R., 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. Proceedings of ACM POPL'77, 238–252.

[14] Dijkstra, E.-W., 1980. On the productivity of recursive definitions, EWD749. URL `http://www.cs.utexas.edu/users/EWD/ewd07xx/EWD749.PDF`

[15] Endrullis, J., Grabmayer, C., Hendriks, D., Isihara, A., Klop, J., 2007. Productivity of Stream Definitions. Vol. 4639 of LNCS. Springer, p. 274.

[16] Férée, H., Hainry, E., Hoyrup, M., Péchoux, R., 2010. Interpretation of stream programs: characterizing type 2 polynomial time complexity. Springer, pp. 291–303.

[17] Frankau, S., Mycroft, A., 2003. Stream processing hardware from functional language specifications. Proceeding of IEEE HICSS-36.

[18] Gaboardi, M., Péchoux, R., 2009. Upper bounds on stream I/O using semantic interpretations. In: CSL 2009. Vol. 5771 of LNCS. Springer, pp. 271–286.

[19] Gaboardi, M., Péchoux, R., 2010. Global and local space properties of stream programs. In: 1st International Workshop on Foundational and Practical Aspects of Resource Analysis – FOPARA'09. Vol. 6324 of LNCS. pp. 51 – 66.

[20] Gordon, A., 1999. Bisimilarity as a theory of functional programming. TCS 228.

[21] Henderson, P., Morris, J., 1976. A lazy evaluator. Proceedings of POPL'76, 95–103.

[22] Hughes, J., Pareto, L., Sabry, A., 1996. Proving the correctness of reactive systems using sized types. Proceedings of ACM POPL'96, 410–423.

[23] Kennaway, J., Klop, J., Sleep, M., de Vries, F., 1989. Transfinite Reductions in Orthogonal Term Rewriting Systems. Vol. 488 of LNCS. pp. 1–12.

[24] Kennaway, J., Klop, J., Sleep, M., de Vries, F., 1997. Infinitary lambda calculus. TCS 175 (1), 93–125.

[25] Korp, M., Sternagel, C., Zankl, H., Middeldorp, A., 2009. Tyrolean termination tool 2. In: RTA. Springer, pp. 295–304.

[26] Lankford, D., 1979. On proving term rewriting systems are noetherien, tech. rep.

[27] Launchbury, J., 1993. A natural semantics for lazy evaluation. Proceedings of POPL'93, 144–154.

[28] Leivant, D., 1995. Intrinsic theories and computational complexity. In: Leivant, D. (Ed.), Logic and Computational Complexity. Vol. 960 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 177–194.

[29] Leivant, D., Ramyaa, R., 2012. Implicit complexity for coinductive data: a characterization of corecurrence. In: Marion, J.-Y. (Ed.), Second Workshop on Developments in Implicit Computational Complexity. Vol. 75 of EPTCS. Open Publishing Association, pp. 1–14.

[30] Lucas, S., 2005. Polynomials over the reals in proofs of termination: from theory to practice. RAIRO Theoretical Informatics and Applications 39 (3), 547–586.

[31] Manna, Z., Ness, S., 1970. On the termination of Markov algorithms. In: Third hawaii international conference on system science. pp. 789–792.

[32] Marion, J.-Y., Péchoux, R., 2009. Sup-interpretations, a semantic method for static analysis of program resources. ACM Transactions on Computational Logic (TOCL) 10 (4), 27.

[33] Ong, C.-H. L., 1992. Lazy lambda calculus: Theories, models and local structure characterization (extended abstract). In: ICALP. pp. 487–498.

[34] Péchoux, R., 2013. Synthesis of sup-interpretations: A survey. Theor. Comput. Sci. 467, 30–52.

[35] Pitts, A. M., 1997. Operationally-based theories of program equivalence. In: Semantics and Logics of Computation. Cambridge University Press.

[36] Ramyaa, R., Leivant, D., 2011. Ramified corecurrence and logspace. ENTCS 276 (0), 247 – 261, Twenty-seventh Conference on the Mathematical Foundations of Programming Semantics (MFPS XXVII).

[37] Shkaravska, O., van Eekelen, M., Tamalet, A., July 2009. Collected Size Semantics for Functional Programs over Polymorphic Nested Lists. Tech. Rep. ICIS–R09003, Radboud University Nijmegen.

[38] Sijtsma, B., 1989. On the productivity of recursive list definitions. ACM TOPLAS 11 (4), 633–649.

[39] Stephens, R., 1997. A survey of stream processing. Acta Informatica 34 (7), 491–541.

[40] Weihrauch, K., 1997. A foundation for computable analysis. Vol. 1337 of LNCS. Springer, pp. 185–200.

Appendix A. Comparison with quasi-interpretation

Quasi-interpretations introduced in [7] are a previous formulation of interpretations that require inequalities of the shape $(\!|l|\!)_\xi \geq (\!|r|\!)_\xi$ for each rule $l \to r$ of a TRS. Note that the new definition suggested in this paper is a generalization of these previous works on TRS to functional programs since the interpretation of a definition $\mathtt{f\ x} \doteq \mathtt{Case\ x\ of\ c}_1(\vec{\mathtt{x}}_1) \to \mathtt{e}_1, \ldots, \mathtt{c}_m(\vec{\mathtt{x}}_m) \to \mathtt{e}_m$ generates the following inequality:

$$(\!|\mathtt{f\ x}|\!)_{\rho,\xi} \geq \max_{i\in\{1,m\}} \{(\!|\mathtt{e}_i|\!)_{\rho\{\vec{\mathtt{x}}_i:=\vec{r}_i\},\xi} \mid \forall \vec{r}_i \ s.t.\ (\!|\mathtt{x}|\!)_{\rho,\xi} \geq (\!|\mathtt{c}_i(\vec{\mathtt{x}}_i)|\!)_{\rho\{\vec{\mathtt{x}}_i:=\vec{r}_i\},\xi}\}$$

which implies, using Lemma 3, $\forall i \in \{1, m\}$:

$$\begin{array}{l}
(\!|\mathtt{f\ c}_i(\vec{\mathtt{x}}_i)|\!)_{\rho,\xi} \\
= (\!|\mathtt{f\ x}\{\mathtt{c}_i(\vec{\mathtt{x}}_i)/\mathtt{x}\}|\!)_{\rho,\xi} \\
\geq \max\limits_{i\in\{1,m\}} \{(\!|\mathtt{e}_i|\!)_{\rho\{\vec{\mathtt{x}}_i:=\vec{r}_i\},\xi} \mid \forall \vec{r}_i \ s.t.\ (\!|\mathtt{x}\{\mathtt{c}_i(\vec{\mathtt{x}}_i)/\mathtt{x}\}|\!)_{\rho,\xi} \geq (\!|\mathtt{c}_i(\vec{\mathtt{x}}_i)|\!)_{\rho\{\vec{\mathtt{x}}_i:=\vec{r}_i\},\xi}\} \\
\geq \max\{(\!|\mathtt{e}_i|\!)_{\rho\{\vec{\mathtt{x}}_i:=\vec{r}_i\},\xi} \mid \forall \vec{r}_i \ s.t.\ (\!|\mathtt{c}_i(\vec{\mathtt{x}}_i)|\!)_{\rho,\xi} \geq (\!|\mathtt{c}_i(\vec{\mathtt{x}}_i)|\!)_{\rho\{\vec{\mathtt{x}}_i:=\vec{r}_i\},\xi}\} \\
\geq (\!|\mathtt{e}_i|\!)_{\rho,\xi}
\end{array}$$

The last inequality is obtained by taking the particular values $r_i = \rho(\mathtt{x}_i)$ since, in this particular case, $\rho\{\vec{\mathtt{x}}_i := \vec{r}_i\} = \rho\{\vec{\mathtt{x}}_i := \rho(\vec{\mathtt{x}}_i)\} = \rho$ and, consequently, the inequality $(\!|\mathtt{c}_i(\vec{\mathtt{x}}_i)|\!)_{\rho,\xi} \geq (\!|\mathtt{c}_i(\vec{\mathtt{x}}_i)|\!)_{\rho\{\vec{\mathtt{x}}:=\vec{r}_i\},\xi}$ is satisfied. To conclude, we obtain $\forall i \in \{1, m\}$, $(\!|\mathtt{f\ c}_i(\vec{\mathtt{x}}_i)|\!)_\xi \geq (\!|\mathtt{e}_i|\!)_\xi$ which corresponds exactly to the notion of quasi-interpretation applied to the equivalent TRS defined by the rules $\mathtt{f\ c}_1(\vec{\mathtt{x}}_1) \to \mathtt{e}_1, \ldots, \mathtt{f\ c}_m(\vec{\mathtt{x}}_m) \to \mathtt{e}_m$.

Appendix B. An example of Strict Evaluation

Consider an environment $\mathcal{H}$ containing the functions defined in Example 1. As already stressed, the lazy evaluation produces the first bit of information. So, for instance we have:

$$\dfrac{\dfrac{}{\mathcal{H}; \mathtt{0} : (\mathtt{nats}\ (\mathtt{0}+\mathtt{1})) \ \Downarrow\ \mathtt{0} : (\mathtt{nats}\ (\mathtt{0}+\mathtt{1}))}\ (\text{val})}{\mathcal{H}; \mathtt{nats\ 0} \ \Downarrow\ \mathtt{0} : (\mathtt{nats}\ (\mathtt{0}+\mathtt{1})))}\ (\text{fun})$$

When a pattern matching error occurs this is traced by the `Err` constant. So, since the definition of `zip` without syntactic sugar is:

$$\mathtt{zip\ x\ y} \doteq \mathtt{Case\ x\ of\ z : w} \to \mathtt{z} : (\mathtt{zip\ y\ w})$$

then we clearly have the following evaluation:

$$\dfrac{\dfrac{\dfrac{}{\mathcal{H};\mathtt{nil} \Downarrow \mathtt{nil}}\ (\mathrm{val}) \quad \mathtt{nil} \neq \mathtt{z} : \mathtt{w}}{\mathcal{H};\mathtt{Case\ nil\ of\ z} : \mathtt{w} \to \mathtt{z} : (\mathtt{zip\ nil\ w}) \Downarrow \mathtt{Err}}\ (\mathrm{pm}_e)}{\mathcal{H};\mathtt{zip\ nil\ nil} \Downarrow \mathtt{Err}}\ (\mathrm{fun})$$

Note that the following evaluation does not produce a pattern matching error:

$$\dfrac{\dfrac{\dfrac{}{\mathcal{H};\mathtt{0} : \mathtt{nil} \Downarrow \mathtt{0} : \mathtt{nil}} \quad \dfrac{}{\mathcal{H};\mathtt{0} : (\mathtt{zip\ nil\ nil}) \Downarrow \mathtt{0} : (\mathtt{zip\ nil\ nil})}}{\mathcal{H};\mathtt{Case}\ (\mathtt{0} : \mathtt{nil})\ \mathtt{of\ z} : \mathtt{w} \to \mathtt{z} : (\mathtt{zip\ nil\ w}) \Downarrow \mathtt{0} : (\mathtt{zip\ nil\ nil})}}{\mathcal{H};\mathtt{zip}\ (\mathtt{0} : \mathtt{nil})\ \mathtt{nil} \Downarrow \mathtt{0} : (\mathtt{zip\ nil\ nil})}$$

Now, let us consider the environment $\mathcal{H}'$ obtained by adding to $\mathcal{H}$ the functions definitions defining $\mathtt{eval}_{[\mathtt{Nat}]}$ on lists of numerals:

$$\begin{array}{lrcl}
\mathtt{eval}_{[\mathtt{Nat}]} & :: [\mathtt{Nat}] & \to & [\mathtt{Nat}] \\
\mathtt{eval}_{[\mathtt{Nat}]} & \mathtt{Err} & \doteq & \mathtt{Err} \\
\mathtt{eval}_{[\mathtt{Nat}]} & \mathtt{nil} & \doteq & \mathtt{nil} \\
\mathtt{eval}_{[\mathtt{Nat}]} & (\mathtt{x} : \mathtt{y}) & \doteq & \mathtt{cons}\ (\mathtt{eval}_{\mathtt{Nat}}\ \mathtt{x})\ (\mathtt{eval}_{[\mathtt{Nat}]}\ \mathtt{y})
\end{array}$$

where $\mathtt{eval}_{\mathtt{Nat}}$ is defined analogously using the strict function `succ` defined above and `cons` is the function symbol for the strict version of the list of natural numbers constructor:

$$\begin{array}{llcl}
\mathtt{cons} & \mathtt{Err} & \mathtt{Err} & \doteq & \mathtt{Err} : \mathtt{Err} \\
\mathtt{cons} & \mathtt{Err} & \mathtt{nil} & \doteq & \mathtt{Err} : \mathtt{nil} \\
\mathtt{cons} & \mathtt{0} & \mathtt{nil} & \doteq & \mathtt{0} : \mathtt{nil} \\
\mathtt{cons} & (\mathtt{x} + 1) & \mathtt{nil} & \doteq & (\mathtt{x} + 1) : \mathtt{nil} \\
\mathtt{cons} & \mathtt{Err} & (\mathtt{y} : \mathtt{z}) & \doteq & \mathtt{Err} : (\mathtt{y} : \mathtt{z}) \\
\mathtt{cons} & \mathtt{0} & (\mathtt{y} : \mathtt{z}) & \doteq & \mathtt{0} : (\mathtt{y} : \mathtt{z}) \\
\mathtt{cons} & (\mathtt{x} + 1) & (\mathtt{y} : \mathtt{z}) & \doteq & (\mathtt{x} + 1) : (\mathtt{y} : \mathtt{z})
\end{array}$$

By evaluating the expression $\mathtt{eval}_{[\mathtt{Nat}]}(\mathtt{zip}\ (\mathtt{0} : \mathtt{nil})\ \mathtt{nil})$, we obtain a result distinct from the result obtained without using the $\mathtt{eval}_{[\mathtt{Nat}]}$ function symbol. Indeed, this allows us to explore the entire result as shown by the following derivation where some steps are omitted for clarity:

$$\dfrac{\dfrac{\vdots}{\mathcal{H}';\mathtt{zip\ nil\ nil} \Downarrow \mathtt{Err}}}{\dfrac{\dfrac{}{\mathcal{H}';\mathtt{eval}_{\mathtt{Nat}}\mathtt{0} \Downarrow \mathtt{0}} \quad \dfrac{\vdots}{\mathcal{H}';\mathtt{eval}_{[\mathtt{Nat}]}(\mathtt{zip\ nil\ nil}) \Downarrow \mathtt{Err}}}{\mathcal{H}';\mathtt{cons}\ (\mathtt{eval}_{\mathtt{Nat}}\mathtt{0})\ (\mathtt{eval}_{[\mathtt{Nat}]}(\mathtt{zip\ nil\ nil})) \Downarrow \mathtt{0} : \mathtt{Err}}}{\mathcal{H}';\mathtt{eval}_{[\mathtt{Nat}]}(\mathtt{zip}\ (\mathtt{0} : \mathtt{nil})\ \mathtt{nil}) \Downarrow \mathtt{0} : \mathtt{Err}}$$